Summer 9-1-2014

Real-Time Wireless Sensor-Actuator Networks for Cyber-Physical Systems

Abusayeed Saifullah
Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Chenyang Lu, Chair
Kunal Agrawal
Yixin Chen
Christopher Gill
Humberto Gonzalez
Jie Liu

Real-Time Wireless Sensor-Actuator Networks for Cyber-Physical Systems

by

Abusayeed Saifullah

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

August 2014
Saint Louis, Missouri

Contents

List of Figures

List of Tables

Acknowledgments

From the beginning of my PhD to the end, I owe an immense debt of gratitude to my advisor, Prof. Chenyang Lu, for his invaluable advice and careful guidance. Without his propitious co-operation, time, ideas, and advice, I doubt my PhD should ever have seen the end.

My heartiest gratitude goes to Prof. Yixin Chen, Prof. Kunal Agrawal, and Prof. Christopher Gill whose collaboration has made my PhD experience productive and stimulating. I extend my deepest appreciation to Dr. Jie Liu for supervising me during my internship at Microsoft Research, and for continued collaboration on data center and white space sensor networking research. This work has become an important part of the dissertation. My heartiest gratitude goes to Dr. Ranveer Chandra and Dr. Bodhi Priyantha from Microsoft Research, and Sriram Sankar from Microsoft Corporation who have contributed significantly in this collaborative work. I extend my thanks to Eric Rotvold from Emerson, Inc. for providing us with valuable industrial insights that have motivated my research.

I would be remiss without mentioning the friendly hands of my fellow labmates in the CPS Lab who have contributed immensely to my personal and professional time at Wash U through stimulating discussions and collaboration. I would like to thank my parents foremost who contributed to enlightening the way of my learning with endless love. Special thanks go to my wife, Farida Akter, who was always with me through the good times and bad.

Finally, I acknowledge the financial support from Prof. Chenyang Lu and the Department of Computer Science and Engineering in the form of research assistantship through NSF grants. I am greatly indebted also to all the faculty members and the office staff of the department whose helpful and efficient hands were with me in every step along the journey. Last but not the least, I thank all committee members for their time, suggestions, and service.

Abusayeed Saifullah

Washington University in Saint Louis
August 2014

Dedicated to my parents.

ABSTRACT OF THE DISSERTATION

Real-Time Wireless Sensor-Actuator Networks for Cyber-Physical Systems

by

Abusayeed Saifullah

Doctor of Philosophy in Computer Science

Washington University in St. Louis, August 2014

Professor Chenyang Lu, Chair

A *cyber-physical system (CPS)* employs tight integration of, and coordination between computational, networking, and physical elements. Wireless sensor-actuator networks provide a new communication technology for a broad range of CPS applications such as process control, smart manufacturing, and data center management. Sensing and control in these systems need to meet stringent real-time performance requirements on communication latency in challenging environments. There have been limited results on real-time scheduling theory for wireless sensor-actuator networks. Real-time transmission scheduling and analysis for wireless sensor-actuator networks requires new methodologies to deal with unique characteristics of wireless communication. Furthermore, the performance of a wireless control involves intricate interactions between real-time communication and control. This thesis research tackles these challenges and make a series of contributions to the theory and system for wireless CPS. (1) We establish a new real-time scheduling theory for wireless sensor-actuator networks. (2) We develop a scheduling-control co-design approach for holistic optimization of control performance in a wireless control system. (3) We design and implement a wireless

sensor-actuator network for CPS in data center power management. (4) We expand our research to develop scheduling algorithms and analyses for real-time parallel computing to support computation-intensive CPS.

Chapter 1

Introduction

A *cyber-physical system (CPS)* employs a tight integration of, and coordination between the system's computational, networking, and physical elements. Wireless sensor-actuator networks (WSANs) provide a new communication technology for a broad range of CPS applications such as process control, smart manufacturing, and data center management. A WSAN involves feedback control loops between sensors and actuators through a wireless mesh network. The sensors measure process variables, and deliver to a controller through the network. The controller sends control commands to the actuators, which then operate the control and safety components to adjust physical processes so the system's performance is optimized for efficiency and safety. Sensing and control in these systems need to meet stringent real-time performance requirements on communication latency in challenging environments. Violation of these requirements may result in plant shutdown or accidents causing deaths or significant economic or environmental cost.

While the reliability and real-time requirements are critical for wireless control applications, industry settings pose a harsh environment for wireless communication due to unpredictable channel conditions, limited bandwidth, physical obstacle, multi-path fading, and interference from coexisting wireless devices, causing frequent transmission failures [166]. Addressing this limitation, industrial wireless standards such as WirelessHART [22] mitigate frequent transmission failures through multi-channel communication and *graph routing* where a packet is transmitted through multiple paths and multiple channels. Real-time communication in these wireless networks pose new and important challenges. Unlike real-time wired networks, there have been limited results on real-time scheduling theory for wireless networks. Real-time transmission scheduling and analysis for wireless sensor-actuator networks requires new

methodologies to deal with unique characteristics of wireless communication. In addition, the performance of a wireless control system induces a complicated problem involving multiple interrelated objectives (e.g., reliability, real-time performance, control performance) and interdependent decision variables (e.g., transmission schedule, routes, sampling rates), requiring a scheduling-control co-design approach that has witnessed little progress for WSANs till date due to its inherent challenges and interdisciplinary nature.

In this thesis research, we tackle the above challenges and make a series of contributions to the theory and system for wireless CPS. First, we establish a new real-time scheduling theory for WSANs by bridging wireless mesh network and real-time scheduling domain. Second, we develop a scheduling-control co-design approach for holistic optimization of control performance in a wireless control system. Our technical approach hinges on a novel integration of real-time scheduling theory, wireless networking, optimization theory, and control theory in a unified framework. Third, we design and implement a wireless sensor-actuator network for CPS in data center power management. Finally, we expand our research to develop scheduling algorithms and analyses for real-time parallel computing to enable the forthcoming generation of computation-intensive CPS.

The thesis is organized as follows. Chapters 2, 3, 4, and 5 concentrate to developing real-time scheduling theories for WSANs by bridging real-time scheduling theory and wireless networking. Chapter 6 presents the proposed scheduling-control co-design approach for holistic optimization in multi-hop wireless control systems. Chapter 7 presents a set of distributed channel allocation algorithms for wireless sensor networks. Chapter 8 presents the design and implementation of a wireless sensor-actuator network for CPS in data center power management. Chapters 9 and 10 extend our research and present scheduling algorithms and analyses for real-time parallel computing to support computation-intensive CPS.

Chapter 2

Real-Time Wireless: Dynamic Scheduling

WirelessHART is an open wireless sensor-actuator network standard for industrial process monitoring and control that requires real-time data communication between sensor and actuator devices. Salient features of a WirelessHART network include a centralized network management architecture, multi-channel TDMA transmission, redundant routes, and avoidance of spatial reuse of channels for enhanced reliability and real-time performance. This paper makes several key contributions to real-time transmission scheduling in WirelessHART networks: (1) formulation of the end-to-end real-time transmission scheduling problem based on the characteristics of WirelessHART; (2) proof of NP-hardness of the problem; (3) an optimal branch-and-bound scheduling algorithm based on a necessary condition for schedulability; and (4) an efficient and practical heuristic-based scheduling algorithm called Conflict-aware Least Laxity First (C-LLF). Extensive simulations based on both random topologies and real network topologies of a physical testbed demonstrate that C-LLF is highly effective in meeting end-to-end deadlines in WirelessHART networks, and significantly outperforms common real-time scheduling policies.

2.1 Introduction

Wireless Sensor-Actuator Networks (WSANs) are emerging as a new generation of communication infrastructure for industrial process monitoring and control [56]. Feedback control loops in industrial environments impose stringent end-to-end latency requirements on data

communication. To support a feedback control loop, the network periodically delivers data from sensors to a controller and then delivers its control input data to the actuators within an end-to-end deadline. The direct effects of deadline misses in data communication may range from production inefficiency, equipment destruction to irreparable financial and environmental impacts. For instance, real-time monitoring of level measurement and control are required to avoid overfilling of oil tanks that may lead to serious economic loss and environmental threats. Moreover, stringent regulations for Health, Safety, and the Environment (HSE) are now being enforced in many countries [1]. HSE regulations require continuous monitoring of safety (shower, corrosive chemicals, and safety instrumentation) for workers around the plant so that help can be dispatched on time.

WirelessHART [22] has recently been developed as an open standard for WSANs for process industries. The standard has been instrumental in the adoption and deployment of wireless network technology in the field of process monitoring and control [167]. Drawing upon the insights and lessons learned from real-world industrial applications, WirelessHART has the following salient features specifically designed to meet the stringent real-time and reliability requirements of process monitoring and control: centralized network management architecture, multi-channel Time Division Multiple Access (TDMA), avoidance of spatial reuse of channels [56], and redundant routes. The unique characteristics of WirelessHART introduce a challenging real-time transmission scheduling problem.

In this paper, we study the real-time transmission scheduling problem of a set of periodic data flows with end-to-end deadlines from sensors to actuators in a WirelessHART network. This paper makes the following key contributions to address this problem:

- We formulate the real-time transmission scheduling problem based on the characteristics of WirelessHART networks and prove that it is NP-hard.
- We derive a necessary condition for schedulability in WirelessHART networks which can be used to effectively prune the search space for an optimal solution as well as to provide the insight for an efficient heuristic-based solution.
- We propose an optimal scheduling algorithm based on a branch-and-bound technique.

- We design a practical heuristic-based algorithm called **C**onflict-aware **L**east **L**axity **F**irst (C-LLF) that is efficient and, hence, can be used to handle dynamic changes of network topology and workloads.

The algorithms are evaluated using extensive simulations based on both random network topologies and real network topologies of a physical indoor testbed. Our results demonstrate that C-LLF is highly effective in meeting end-to-end communication deadlines in WirelessHART networks, while significantly outperforming the existing real-time scheduling policies. Moreover, it incurs minimal computational overhead and buffer space in the field devices, thereby making it a practical and effective solution for real-time transmission scheduling in WirelessHART networks.

The rest of the paper is structured as follows. The WirelessHART network model is presented in Section 2.2. Section 2.3 presents the problem formulation and the proof of NP-hardness. We derive the necessary condition for schedulability in Section 2.4. Section 2.5 presents the optimal scheduling based on branch-and-bound. C-LLF scheduling algorithm is presented in Section 2.6. Section 2.7 shows the simulation results. Section 2.8 discusses related work. Section 2.9 is the conclusion.

2.2 WirelessHART Network Model

We consider a WirelessHART network consisting of field devices, one gateway, and a centralized network manager. The gateway provides the host system with access to network devices. Scheduling of transmissions is performed centrally at the network manager connected to the gateway which uses the network routing information in combination with communication requirements of the devices and applications. The network manager, then, distributes the schedules among the devices. The salient features of WirelessHART which make it particularly suitable for process industries are as follows:

Limiting Network Size. Experiences in industrial environments have shown daunting challenges in deploying large-scale WSANs. Typically, 80-100 field devices comprise a WirelessHART network with one gateway. The limit on the network size for a WSAN makes the

centralized management practical and desirable, and enhances the reliability and real-time performance. Large-scale networks can be organized using multiple gateways or as hierarchical networks that connect small WSANs through traditional resource-rich networks such as Ethernet and 802.11 networks.

Time Division Multiple Access (TDMA). Compared to CSMA/CA mechanism, TDMA protocols can provide predictable communication latencies making them an attractive approach for real-time communication. In WirelessHART networks, time is synchronized and slotted, and the length of a time slot allows exactly one transmission and its associated acknowledgement between a device pair.

Route and Spectrum Diversity. Spatial diversity of routes allows messages to be routed through multiple paths in order to mitigate physical obstacles, broken links, and interference. Spectrum diversity gives the network access to all 16 channels defined in IEEE 802.15.4 physical layer and allows per time slot channel hopping in order to avoid jamming and mitigate interference from coexisting wireless systems. Besides, any channel that suffers from persistent external interference is blacklisted and not used. The combination of spectrum and route diversity allows a packet to be transmitted multiple times, over different channels over different paths, thereby handling the challenges of network dynamics in harsh and variable environments at the cost of redundant transmissions and scheduling complexity.

Handling Internal Interference. Due to difficulty in detecting interference between nodes and the variability of interference patterns, WirelessHART allows only one transmission in each channel in a time slot across the entire network, effectively avoiding the spatial reuse of channels [56] to avoid transmission failure due to interference between concurrent transmissions. Thus, the maximum number of concurrent transmissions in the entire network at any slot cannot exceed the number of available channels [56]. This design decision improves the reliability at the potential cost of reduced throughput. The potential loss in throughput is also mitigated due to the small size of network.

Based on the above features, a WirelessHART network forms a mesh network modeled as a graph $G = (V, E)$, where the nodes V represent the network devices and E is the set of edges between the devices. That is, the set V consists of the gateway and the field devices. Every field device is either a sensor node, or an actuator, or both. An edge $e = (u, v)$ exists in the graph if and only if nodes u and v can communicate reliably with each other. Each

$u \in V$ is able to send and receive packets, and to route packets for other network devices. For a transmission $\tau_i = \vec{uv}$ happening along an edge (u, v), device u is designated as the *sender* and device v the *receiver*.

A device cannot both transmit and receive at the same time slot. Two transmissions with the same intended receiver at a slot interfere each other. Hence, two transmissions $\tau_i = \vec{uv}$ and $\tau_j = \vec{wz}$ are *conflicting* and cannot be scheduled in the same slot, if $(u = w) \vee (u = z) \vee (v = w) \vee (v = z)$. A set of transmissions is *mutually exclusive* if every pair of transmissions in the set is conflicting.

2.3 Problem Formulation

In the *real-time scheduling for a WirelessHART network* $G = (V, E)$, we consider N end-to-end flows $F = \{F_1, F_2, \cdots, F_N\}$. Each flow $F_i \in F$ periodically generates a packet that originates at a network device $u \in V$, called the *source* of the packet, passes through the gateway, and ends at a network device $v \in V - \{u\}$, called the *destination* of the packet, within a deadline. The source and destination are characterized to be a sensor node and an actuator, respectively. From a source to a destination, there may exist more than one route, and the packet is delivered to the destination through each of these routes. The *release time* of a packet is the earliest time slot when it is ready to be scheduled. For a packet released at slot k and delivered to a destination at slot j through a route, its *end-to-end latency* through this route is $j - k + 1$. For flow F_i, its *end-to-end latency* L_i is the maximum end-to-end latency among all the packets generated by F_i.

Each flow F_i is, thus, characterized by a period P_i, a deadline D_i where $D_i \leq P_i$, and a set of routes Φ_i. A packet generated by F_i is routed through each $\phi \in \Phi_i$ that connects the source node to a destination node through the gateway. Thus, given the set of flows F, our objective is to schedule all transmissions in m channels such that $L_i \leq D_i, \forall F_i \in F$.

For the above problem, a scheduling algorithm $\mathbb{A}$ is called *optimal*, if $\mathbb{A}$ can schedule all transmissions whenever a feasible schedule (where no deadline is missed) exists. In the following, we prove that the problem is NP-hard by proving that its decision version is NP-complete.

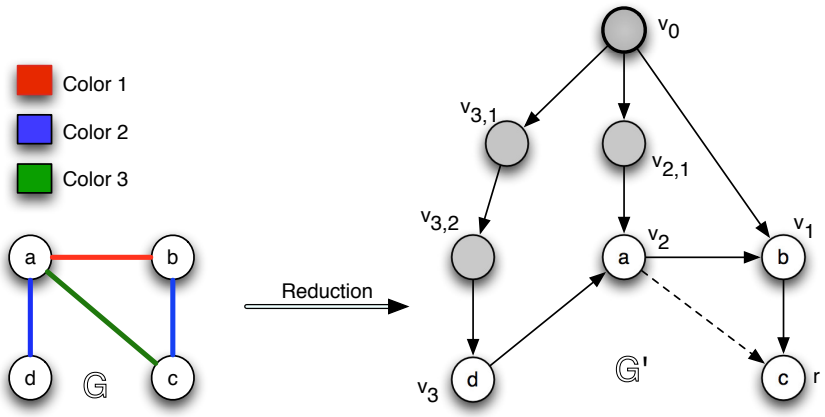


Figure 2.1: Reduction from edge-coloring

Theorem 1. *Given a real-time scheduling problem for a WirelessHART network, it is NP-complete to decide whether it is schedulable or not.*

Proof. Given an instance of the real-time scheduling problem for a WirelessHART network with N flows, we can verify in $O(N)$ time whether all the flows meet their deadlines. Hence, the problem is in NP. To prove NP-hardness, we reduce an arbitrary instance $< \mathbb{G}, k >$ of the *graph edge-coloring* problem to an instance $\mathbb{S}$ of the real-time scheduling for a WirelessHART network and show that graph $\mathbb{G}$ is k edge-colorable if and only if $\mathbb{S}$ is schedulable (Figure 2.1).

Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ has n nodes. We create a depth-first search tree of $\mathbb{G}$ rooted at an arbitrary node $r \in \mathbb{V}$. For every $u \in \mathbb{V} - \{r\}$, a tree edge is directed from u to its *parent*; and zero or more ancestors connected by a non-tree edge directed from u are its *virtual parents*. Every node in $\mathbb{V} - \{r\}$ is given a unique label v_i, where $1 \leq i \leq n-1$. Create a node v_0. For every node v_i, $1 \leq i \leq n-1$, add $i-1$ additional nodes $v_{i,1}, v_{i,2}, \cdots, v_{i,i-1}$ and connect v_0 to v_i through these nodes (i.e., create $v_0 - v_{i,1} - v_{i,2} - \cdots - v_{i,i-1} - v_i$ path). Now, following is an instance $\mathbb{S}$ of the real-time scheduling for a WirelessHART network. The reduced graph $\mathbb{G}' = (\mathbb{V}', \mathbb{E}')$ is a network with $n + 1 + \frac{(n-2)(n-1)}{2}$ nodes. Node v_0 is the gateway. The parent and the virtual parents of every node v_i, $1 \leq i \leq n-1$, are the destination nodes, and v_0 is a source node. For every v_i, $1 \leq i \leq n-1$, a flow F_i periodically generates a packet starting at $(n-i)$-th slot at v_0 and follows the route $v_0 - v_{i,1} - v_{i,2} - \cdots - v_i$ and is, then, forwarded by v_i to its parent and every virtual parent. For simplicity, we consider only the first packet

of every flow F_i. For F_i, the release time and the absolute deadline of this packet are $n-i$ and $n-1+k$, respectively. All flows have the same period $\geq n-1+k$. The number of channels is $n-1$. This reduction runs in $O(n^2)$ time.

Let $\mathbb{G}$ is edge-colorable using k colors. Let Q be the set of all last one-hop transmissions in $\mathbb{G}'$. These transmissions involve edges $\mathbb{E} \subset \mathbb{E}'$, one transmission per edge. Using all $n-1$ channels, we can complete all transmissions in $\mathbb{G}'$ except those in Q in first $n-1$ slots. Since the transmissions along the edges having the same color can be scheduled on the same slot, all transmissions in Q can be scheduled in next k slots. Hence, all packets meet the deadline. Now, let $\mathbb{S}$ is schedulable by an algorithm $\mathbb{A}$. If $\mathbb{A}$ uses all channels, then all but the transmissions in Q are completed in first $n-1$ slots. Hence, all transmissions in Q are schedulable using next k slots. For transmissions that happen on the same slot, the corresponding edges can be given the same color. Hence, graph $\mathbb{G}$ is k edge-colorable. If $\mathbb{A}$ does not use all channels, then no transmission in Q can happen in first $n-1$ slots. Let there are t slots starting from the earliest slot at which some transmission in Q can be scheduled to the latest slot by which all transmissions in Q must be scheduled. Since all packets meet the deadline, $t \leq k$. The value of t is the smallest when we can schedule all non-conflicting transmissions in Q on the same slot. That is, the smallest value of t is the edge chromatic number χ of $\mathbb{G}$. Thus, $\chi \leq t \leq k$. Since $\mathbb{G}$ is χ edge-colorable, it is k edge-colorable also. □

2.4 Necessary Condition for Schedulability

In this section, we establish a necessary condition for schedulability. This condition can be used to effectively prune the search space in a branch-and-bound algorithm. It also provides the key insights for efficient heuristic scheduling policies. In a WirelessHART network, the conflicting transmissions play a major role in the communication delays and the schedulability of the flows. The delays caused by conflicting transmissions are especially high near the gateway where all the flows converge creating a hot spot.

We first define some terminologies used in the necessary condition analysis. For the given set of flows, let T be the *hyper-period*, i.e., the *least common multiple* of the periods of flows. It is sufficient to find a schedule for transmissions of packets generated no later than slot T.

We use $p_{i,j}$ to denote the j-th packet, $0 \leq j < T/P_i$, generated by flow F_i. For packet $p_{i,j}$, its *release time* $R_{i,j} = P_i * j + 1$, and the *absolute deadline* $D_{i,j} = R_{i,j} + D_i - 1$.

From the release time and deadline of a packet, we can also derive a deadline and an anticipated release time for every transmission of the packet. For packet $p_{i,j}$, let $\tau_k = \vec{uv}$ be a transmission of $p_{i,j}$ through a route connecting its source to a destination such that the destination is $post_k$ hops away from node v. Since $D_{i,j}$ is the deadline of packet $p_{i,j}$, transmission τ_k needs to happen no later than slot $D_{i,j} - post_k$. Therefore, we can define the *deadline* of transmission τ_k as $d_k = D_{i,j} - post_k$. At a time slot s, let packet $p_{i,j}$ requires $pre_{k,s}$ transmissions before transmission $\tau_k = \vec{uv}$ can happen. That is, packet $p_{i,j}$ is $pre_{k,s}$ hops away from node u on its route at slot s. At slot s, a transmission is said to be *released* and, hence, is ready to be scheduled, if its preceding transmission is already scheduled before slot s. Therefore, unlike a packet, the exact release time of a transmission cannot be determined in advance (except for a packet's first hop transmission). Instead, at time slot s, we define the *anticipated release time* of transmission τ_k as $r_k = R_{i,j} + pre_{k,s} + max(s - R_{i,j}, 0)$.

Now we analyze the time demand of a packet for scheduling its transmissions in different time windows. For transmission τ_k of packet $p_{i,j}$, we call the time window $[r_k, d_k]$ the *lifetime* of transmission τ_k meaning that τ_k can happen no earlier than slot r_k and no later than slot d_k. Therefore, in window $[r_k, d_k]$, packet $p_{i,j}$ must need at least one time slot. If $pre_{k,s} > 0$ at slot s, then the lifetime of $p_{i,j}$'s transmission that must precede τ_k is $[r_k - 1, d_k - 1]$. Similarly, if $post_k > 0$, then the lifetime of $p_{i,j}$'s transmission that is preceded by τ_k is $[r_k + 1, d_k + 1]$. Thus, at any time slot s, we can conclude that packet $p_{i,j}$ needs at least: *(a)* 1 slot in window $[r_k, d_k]$; *(b)* 2 slots in window $[r_k - 1, d_k]$, if $pre_{k,s} > 0$; *(c)* 2 slots in window $[r_k, d_k + 1]$, if $post_k > 0$; *(d)* 3 slots in window $[r_k - 1, d_k + 1]$, if $pre_{k,s} > 0$ and $post_k > 0$. For τ_k, these time windows are denoted by:

$$\Omega(\tau_k) = \{[r_k - \beta_1, d_k + \beta_2] \mid 0 \leq \beta_1, \beta_2 \leq 1\} \tag{2.1}$$

Let Γ be the set of transmissions of all the packets released no later than slot T (hyperperiod). At slot s, $\Gamma_s \subseteq \Gamma$ denotes the set of unscheduled transmissions. Considering a transmission $\tau_k \in \Gamma_s$ of packet $p_{i,j}$, we know a lower bound of the time demand of $p_{i,j}$ in every window in $\Omega(\tau_k)$ from the above analysis. Again, a window $[a, b] \in \Omega(\tau_k)$ may contain another window $[a', b'] \in \Omega(\tau_{k'})$ (i.e., $a \leq a'$ and $b' \leq b$) of another transmission $\tau_{k'}$. Taking

into account the lower bounds of time demand of every packet in window $[a, b] \in \Omega(\tau_k)$ yields a tighter lower bound of the number of transmissions by the packets in window $[a, b]$. Since, the total number of transmissions that can be accommodated in a time window is limited by the conflicting transmissions as well as the number of available channels, this time window analysis leads to a necessary condition for the schedulability. Intuitively, analyzing the lower bound of time demands in all windows $[x, y],\ 1 \leq x \leq y \leq T$ will lead to a strong necessary condition. But there are $O(T^2)$ such windows, thereby making the analysis computationally very expensive. However, a time window that does not contain any transmission's lifetime is useless in the analysis. Besides, the number of transmissions in a window is finite. As a result, the larger the numbers β_1 and β_2 are, the less effective the window $[r_k - \beta_1, d_k + \beta_2]$ is for necessary condition analysis. To balance between the complexity and effectiveness, we limit our analysis to $\beta_1 \leq 1$ and $\beta_2 \leq 1$ (in Equation 2.1).

We now derive the necessary condition for schedulability. Let $\psi^k_{a,b}$ be the number of transmissions in the largest set of mutually exclusive transmissions containing transmission τ_k such that the lifetime of each of these transmissions is contained in window $[a, b]$. Let $q_{a,b}$ be the total number of transmissions whose lifetimes are contained in $[a, b]$. For window$[a, b]$ and a transmission τ_k whose lifetime is contained in $[a, b]$, we define $\Delta^k_{a,b}$ as follows:

$$\Delta^k_{a,b} = (b - a + 1) - max(\psi^k_{a,b}, \left\lceil \frac{q_{a,b}}{m} \right\rceil) \tag{2.2}$$

Let $\mu(\tau_k)$ be the minimum $\Delta^k_{a,b}$ among all $[a, b] \in \Omega(\tau_k)$.

$$\mu(\tau_k) = min(\{\Delta^k_{a,b} \mid [a, b] \in \Omega(\tau_k)\}) \tag{2.3}$$

Based on above time window analysis, Theorem 2 establishes a strong necessary condition for schedulability.

Theorem 2. *For a set of flows F, let Γ_s be the set of unscheduled transmissions at slot s. If these transmissions are schedulable, then $min(\{\mu(\tau_k)|\tau_k \in \Gamma_s\}) \geq 0$.*

Proof. Let $\mathbb{S}$ be a feasible schedule of these transmissions where all the flows meet their deadlines. Time window $[a, b]$ can accommodate at most $b - a + 1$ mutually exclusive transmissions, irrespective of how many channels are available. Again, time window $[a, b]$ can

accommodate at most $m * (b - a + 1)$ transmissions in total. But, for $q_{a,b}$ transmissions that must happen in window $[a, b]$, at least $\lceil \frac{q_{a,b}}{m} \rceil$ time slots are required. Again, for every transmission τ_k among these, there are $\psi^k_{a,b}$ transmissions each of which must be scheduled on a different time slot. That is, at least $max(\psi^k_{a,b}, \lceil \frac{q_{a,b}}{m} \rceil)$ time slots are required to accommodate the transmissions in window $[a, b]$. At any time slot s, the *laxity* of a packet $p_{i,j}$ can be defined as $(D_{i,j} - s + 1) - h_{i,j}$, where $h_{i,j}$ is the remaining number of transmissions of $p_{i,j}$ through its route. The *Laxity of schedule* $\mathbb{S}$ is the minimum laxity among all packets. The value $\mu(\tau_k) = min(\{\Delta^k_{a,b} | \ [a, b] \in \Omega(\tau_k)\})$ is an upper bound of the schedule laxity of $\mathbb{S}$. Thus, $min(\{\mu(\tau_k) | \tau_k \in \Gamma_s\})$ indicates a tighter upper bound. Since $\mathbb{S}$ is a feasible schedule, $min(\{\mu(\tau_k) | \tau_k \in \Gamma_s\}) \geq 0$. □

2.5 Optimal Branch-and-Bound Scheduling

In this section, we present a scheduling algorithm based on branch-and-bound (B&B). Our B&B scheduling algorithm exploits the necessary condition established in Theorem 2 to effectively discard infeasible branches in the search space. It is optimal and complete in that it guarantees to find a schedule whenever a feasible one exists. The optimal B&B uses a search tree, where every node corresponds to a partial schedule that may or may not lead to a complete feasible schedule. For decision making at every node, the algorithm estimates an upper bound of the laxity of the schedule that the node may lead to. The *laxity of a packet* is its remaining time slots minus its remaining number of transmissions, and the *laxity of a schedule* is the minimum laxity among all packets. According to Theorem 2, for transmissions Γ_s to be scheduled on or after slot s, following is an upper bound (UB) of its schedule's laxity:

$$UB = \min(\{\mu(\tau_k) | \tau_k \in \Gamma_s\}) \tag{2.4}$$

The search globally maintains a lower bound (LB) of schedule laxity as 0. Computing UB at a node (using Equation 2.4) gives one of these two decisions: *unschedulable* or *may be schedulable*. Specifically, if $UB < LB$ at a node, it is guaranteed that this node will not lead to any feasible schedule and, hence, is discarded without further consideration. In contrast, if $UB \geq LB$, then this node may lead to a feasible solution and, hence, is expanded further. The algorithm terminates as soon as it finds a feasible complete schedule that meets all

Algorithm 1: Optimal B&B Scheduling Algorithm

Step 0. Γ= set of all transmissions. $LB \leftarrow 0$;

Step 1. Compute UB for Γ. If $UB < LB$ then stop since the given instance is unschedulable. Otherwise, create an empty schedule. Call this node the parent node. Find the released transmissions.

Step 2. For every valid subschedule of released transmissions, create a new child node. For each node, append the subschedule to the parent schedule and create new set of released transmissions. Compute UB for this node.

Step 3. If steps 4 and 5 have been performed for all childnodes then close the parent and go to step 6, otherwise, select the next child.

Step 4. If no unscheduled transmission is left, then stop, a feasible solution has been found.

Step 5. If $UB < LB$, then close this child node. Otherwise, create next released set of transmissions. Go to Step 2.

Step 6. Select a node among the open nodes. Call this node the parent node and go to step 2.

deadlines. If the original problem is infeasible, the algorithm will also terminate as soon as it determines that this is the case.

The search tree has as its root node an empty schedule along with all unscheduled transmissions. If it turns out that $UB < LB$ at the root, then we terminate immediately with *unschedulable* decision. Otherwise, we determine the released transmissions at the first slot. For every valid subschedule of the released transmissions, we create a successor node that appends its subschedule to the schedule determined at the parent. By a *valid subschedule* we mean a subset of released transmissions that can be scheduled in current slot. Considering all unscheduled transmissions, the algorithm computes UB at this node to decide whether they are unschedulable or may be schedulable. If $UB < LB$, then this child is closed. Otherwise, we calculate the transmissions that are to be released for the next slot and the node is expanded further. We continue to create new nodes in the search tree until we either find a feasible solution, or until there exists no unexpanded node for which $UB \geq LB$. In the latter case, no feasible valid solution exists. The steps of our optimal B&B are presented as Algorithm 1.

2.6 Conflict-aware Least Laxity First

While the B&B algorithm presented in Section 2.5 is optimal, its execution time may limit its applicability to dynamic environments where network topology changes frequently requiring the schedule to be recomputed quickly. In this section, we present a simple and efficient scheduling policy that is suitable for dynamic environments.

While the traditional real-time scheduling policies such as Least Laxity First (LLF) have been effective in end-to-end real-time scheduling over wired networks, such traditional policies do not deal with conflicts between transmissions in wireless networks. Since conflicting transmissions must be scheduled in different time slots, transmission conflicts contribute significantly to the communication delays in wireless networks. In WirelessHART networks, transmission conflicts can play a major role in schedulability even for moderate workloads due to the high degree of conflicts near the gateway. Moreover, different nodes experience different degree of conflicts as different nodes have different number of neighbors in a routing graph. The gateway and the nodes with high connectivity in the routing graph tend to experience significantly higher degrees of conflicts. *Hence, scheduling algorithms for WirelessHART networks must be cognizant of conflicts between transmissions.*

Based on this key insight into the WirelessHART networks, we present an efficient scheduling policy called *Conflict-aware Least Laxity First* (C-LLF). It uses *conflict-aware laxity* of every released transmission as the decision variable. The *conflict-aware laxity* of a transmission is determined by considering the length of time windows in which the transmission must be scheduled as well as the potential conflicts that the transmission may experience in these windows. That is, the approach combines LLF and the degree of conflicts associated with a transmission. Thus, it can schedule a transmission while the remaining ones are likely to retain the necessary condition established in Theorem 2. Specifically, the algorithm identifies some critical time windows in which too many conflicting transmissions have to be scheduled, thereby determining the criticality of each released transmission. Criticality of a transmission is quantified by its conflict-aware laxity. Transmissions exhibiting lower conflict-aware laxity are assessed to be more critical. C-LLF gives the highest priority to the transmissions exhibiting lower conflict-aware laxity.

Now, for a transmission $\tau_k = \vec{uv}$ at slot s, we derive an expression (Equation 6.11) to compute its conflict-aware laxity denoted by λ_k^s. The transmissions whose lifetimes intersect with τ_k's lifetime are the potential sources of conflict while trying to schedule τ_k. We consider a subset of these transmissions to compute λ_k^s efficiently and effectively for transmission $\tau_k = \vec{uv}$. The subset consists of the transmissions that involve node u. Since we have to consider these transmissions until they are scheduled or until their deadlines are past, we consider the time windows that start at current slot and ends at their deadlines. For transmission $\tau_k = \vec{uv}$, let Λ_k^u be the set of deadlines of transmissions that involve node u and whose lifetimes intersect with the lifetime of τ_k, i.e.,

$$\Lambda_k^u = \{d_j | \tau_j = \vec{uz} \text{ or } \vec{zu},\ r_k \leq r_j \leq d_k\} \tag{2.5}$$

Since conflicting transmissions have to be scheduled on different slots, at slot s, we assess the criticality of window $[s, b]$, for every $b \in \Lambda_k^u$, by the difference $\delta_{s,b}$ between its length and the considered number of conflicts in it. That is,

$$\delta_{s,b} = (b - s + 1) - \sigma_{s,b}^u,\ \text{for } b \in \Lambda_k^u \tag{2.6}$$

with $\sigma_{s,b}^u$ being the number of transmissions that involve node u in time window $[s, b]$. That is, $\sigma_{s,b}^u$ counts every transmission $\tau_j = \vec{uz}$ or $\vec{zu}$ with $s \leq r_j$ and $d_j \leq b$, where $[r_j, d_j]$ is the lifetime of τ_j.

Now according to Equation 2.6, a smaller value of $\delta_{s,b}$ indicates that transmission τ_k will be conflicting with too many transmissions in a short time window. Therefore, the *conflict-aware laxity* λ_k^s of transmission $\tau_k = \vec{uv}$ at slot s is defined as the minimum $\delta_{s,b}$ over all $b \in \Lambda_k^u$ as follows:

$$\lambda_k^s = \min(\{\delta_{s,b} | b \in \Lambda_k^u\}) \tag{2.7}$$

Therefore, the smaller the value of λ_k^s is, the more critical transmission τ_k is.

At every time slot s, C-LLF computes the conflict-aware laxity λ_k^s for every released transmission τ_k. Once it is calculated for every released transmission, the transmission with the smallest conflict-aware laxity is scheduled first. If there is a tie, then the transmission (among those having the smallest conflict-aware laxity) that has the earliest deadline is selected to

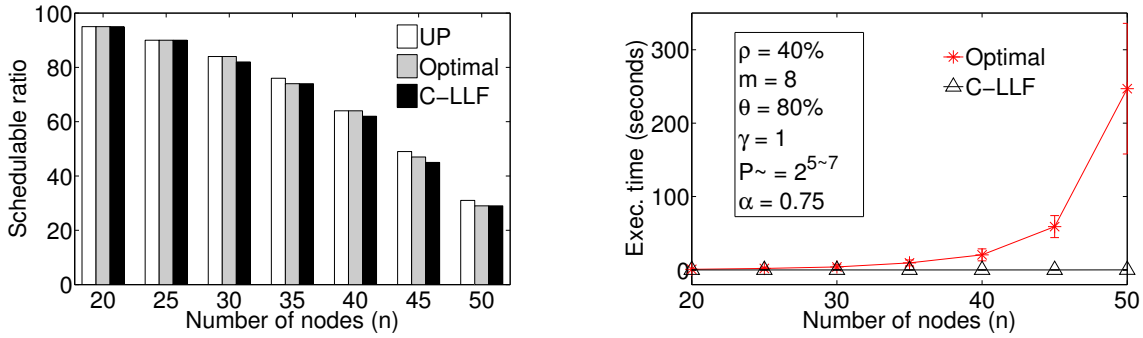
schedule on an available channel. Any further tie is broken arbitrarily. Every released transmission that conflicts with the scheduled one cannot be scheduled in this slot. If there remains an unassigned channel, then the transmission with the next smallest conflict-aware laxity among the remaining released transmissions is picked. Similarly, the tie is broken first by the earliest deadline and then arbitrarily, and the transmissions conflicting with the scheduled one are no more considered for current slot. For the current slot, the same thing is repeated until no free channel is available, or no ready transmission is conflict-free with the scheduled ones in this slot, or there is no released transmission unscheduled. Then the schedule is performed for the next slot in the same way.

Algorithm 2: C-LLF Scheduling Algorithm

Input: $\Gamma \leftarrow$ transmissions of packets released no later than hyper-period T ; $m \leftarrow$ total channels;
Output: $S[1 \cdots T][0 \cdots m-1]$; /* schedule */
$s \leftarrow 1$; /* initialize time slot */
$\Gamma_s \leftarrow \Gamma$; /* Unscheduled transmissions */
while $(\Gamma_s \neq \emptyset)$ **do**
  $Released(s) \leftarrow$ set of released transmissions at slot s;
  $ch \leftarrow 0$; /* initialize channel offset */
  for $(each\ \tau_k \in Released(s))$ **do** Compute λ_k^s ;
  while $(ch < m)$ **do**
    $B \leftarrow$ transmissions with the smallest λ in $Released(s)$;
    $\tau^* \leftarrow$ a transmission with the shortest deadline among B;
    if *(τ^* misses deadline)* **then return** *unschedulable;* ;
    $S[s][ch] \leftarrow \tau^*$; $\Gamma_s = \Gamma_s - \{\tau^*\}$; $ch \leftarrow ch + 1$;
    Remove from $Released(s)$ every transmission conflicting with τ^*;
  end
  $s \leftarrow s + 1$; /* go to next slot */
end

As shown in the pseudo code (Algorithm 2), C-LLF outputs the schedule as a 2-dimensional array $S[1 \cdots T][0 \cdots m-1]$. The algorithm terminates with the *unschedulable* decision if, for a transmission τ_k with deadline d_k, it determines that $s > d_k$ at any slot s. When a transmission τ^* is assigned slot s and a channel offset ch, $0 \leq ch < m$, the schedule is recorded as $S[s][ch] = \tau^*$. In WirelessHART, the channel offset is then mapped to a physical channel for slot s.

Complexity analysis. C-LLF is a pseudo-polynomial time algorithm as analyzed below. There can be at most $O(N)$ released transmissions at a time slot considering a constant number of routes for every flow. At slot s, to calculate the conflict-aware laxity λ_k^s for a transmission τ_k, we need to consider all packets released within window $[r_k, d_k]$ (lifetime of



(a) Schedulable ratio

(b) Execution time

Figure 2.2: Scheduling with the B&B and C-LLF under varying network sizes

τ_k). Therefore, the number of transmissions that should be considered for calculating λ_k^s is upper bounded by $O(N.H.D/P)$, where H is the maximum length among all routes, D is the maximum relative deadline and P is the minimum period among all flows. Since at most two transmissions of a packet can involve a node in any direction (sensor to gateway or gateway to actuator) on a route, calculating λ_k^s takes $O(N.H.D/P)$ time. The time for calculating the conflict-aware laxities for all released transmissions and sorting them is $O(N^2.H.D/P)$. Since the schedule has to be calculated only up to the hyper-period T, the complexity of C-LLF is thus $O(N^2.T.H.D/P)$.

n :	Number of nodes of a network
m :	Number of channels
ρ :	Edge-density of a network
θ :	Fraction of sources and destinations
γ :	Number of routes between every source and destination
$P_\sim$:	Period range
α :	Maximum route length$\leq$deadline$\leq \alpha *$period

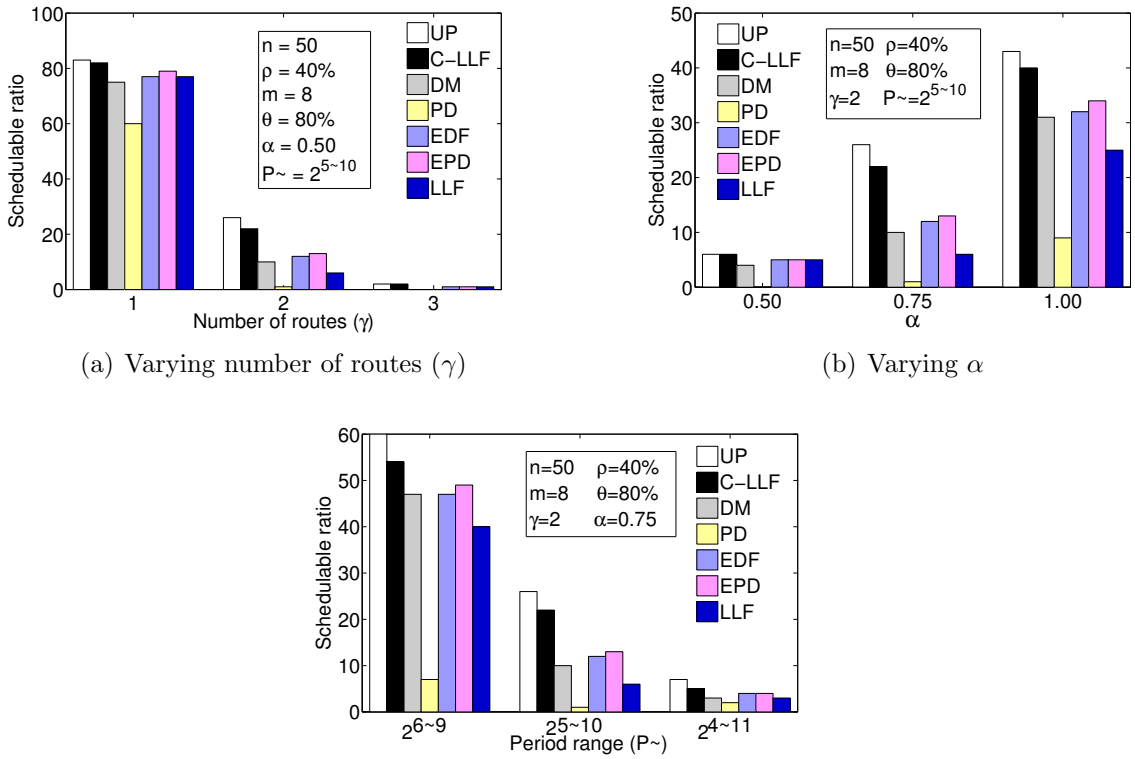
Table 2.1: Notations

2.7 Evaluation

Baselines. We compare our algorithms against several well-known real-time scheduling policies: *(a) Deadline Monotonic (DM)* schedules using the relative deadline of the flows; transmission that belongs to the flow with the smallest relative deadline being scheduled first; *(b) Earliest Deadline First (EDF)* schedules a transmission based on its packet's absolute deadline; *(c) Proportional Deadline monotonic (PD)* schedules a transmission based on its packet's *relative subdeadline* defined as the relative deadline (of its flow) divided by the total number of transmissions along its route; *(d) Earliest Proportional Deadline first (EPD)* schedules a transmissiont based on its packet's *absolute subdeadline* defined, at every slot, as its remaining time divided by the remaining number of transmissions; *(e) Least Laxity First (LLF)* schedules a transmission based on its packet's *laxity* defined as its remaining time minus its remaining number of transmissions.

Metrics. Following metrics are used for performance analysis. *(a) Schedulable ratio* is measured as the percentage of test cases for which an algorithm is able to find a feasible schedule. *(b) Buffer size* is the maximum number of packets buffered at a node when transmissions are scheduled. *(c) Execution time* is the total time required to create a schedule for packets generated within the hyper-period. We plot the average execution time (along with the 95% confidence interval) of the schedulable cases out of 100 runs.

Simulation Setup. A fraction (θ) of nodes is used as sources and destinations of the flows. The sets of sources and destinations are disjoint. The node with the highest number of neighbors is the gateway. The *reliability* of a link is represented by the *packet reception ratio (PRR)* along it. The most reliable route connecting a source to a destination is determined. For additional routes, we choose the next most reliable route that excludes the links of any existing route between the same source and destination. The periods of flows are harmonic and are generated randomly in a given range denoted by $P_{\sim} = 2^{i \sim j}$, $i \leq j$. The relative deadline of a flow F_i with period P_i is generated randomly in the range between H_i and $\alpha * P_i$, for $0 < \alpha \leq 1$, with H_i being the maximum length among the routes associated with F_i. In every figure, we show the parameter setups of the corresponding experiment. The algorithms have been written in C and the tests have been performed on a Mac OS X



(a) Varying number of routes (γ)

(b) Varying α

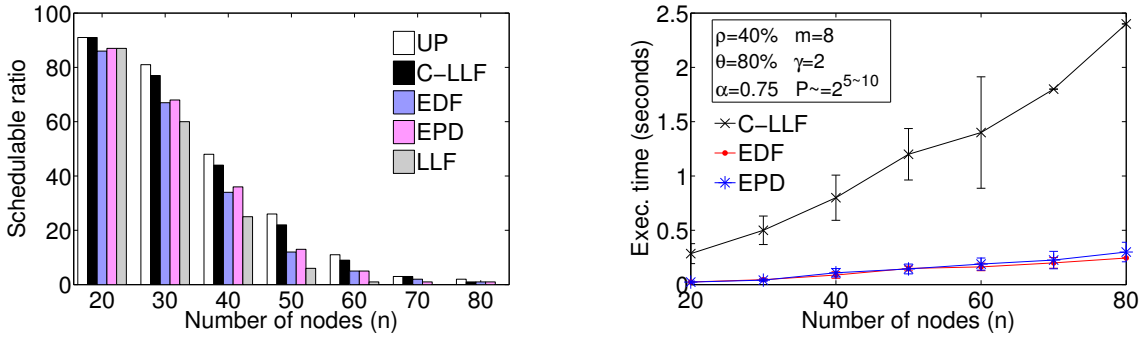
(c) Varying periods ($P_{\sim}$)

Figure 2.3: Schedulable ratio of C-LLF and baselines

machine with 2.4 GHz Intel Core 2 Duo processor. The notations used in this section are summarized in Table 2.1.

2.7.1 Simulations with Random Topologies

Generating networks. Given the number of nodes (n) and edge-density (ρ), we generate random networks. A network with n nodes and i% edge-density has a total of $(n(n-1) * i)/(2*100)$ bidirectional edges. The edges are chosen randomly and assigned PRR randomly in the range $0.80 \sim 1.0$. We keep regenerating a network until the required number of routes, denoted by γ, between every source and destination pair are found.



(a) Schedulable ratio (b) Execution time

Figure 2.4: Comparison under varying network sizes

Optimal B&B and C-LLF. We evaluate the tightness of our necessary condition based on the percentage of test cases that pass the necessary condition but are found to be unschedulable under the B&B. The percentage of test cases that pass the condition is denoted by UP and it indicates an upper bound of schedulable ratio. Figure 2.2 shows the performance of the B&B, C-LLF, and the tightness of our necessary condition using 8 channels in different sized networks of 40% edge-density with $\gamma = 1$ and θ=80%. θ=80% implies that 40% of the total nodes are sources while another 40% are destinations of flows. Figure 2.2(a) shows that the number of test cases that satisfy necessary condition but are not schedulable by the B&B is less than 3%. It indicates that the necessary condition in Theorem 2 is highly effective for pruning the search space in the B&B. Figure 2.2(a) also shows that C-LLF is highly competitive against the B&B in terms of schedulable ratio. As shown in Figure 2.2(b), while the B&B incurs reasonable execution times for 20 and 30 nodes, its execution time increases dramatically as the number of nodes increases. Its average execution time is 247 seconds for 50 nodes, making it less desirable for relatively larger networks with frequent topology changes. In contrast, C-LLF remains highly efficient for varying network sizes and maintains an average execution time of 0.06 seconds for 50-node networks. This result indicates that C-LLF can be used as an effective online scheduling algorithm in face of dynamic network topologies.

Comparison with real-time heuristics. Now we compare our algorithm with the baselines. As the B&B has significantly longer execution time, we exclude the B&B in this set of simulations. As the last set of simulations showed that the necessary condition is fairly

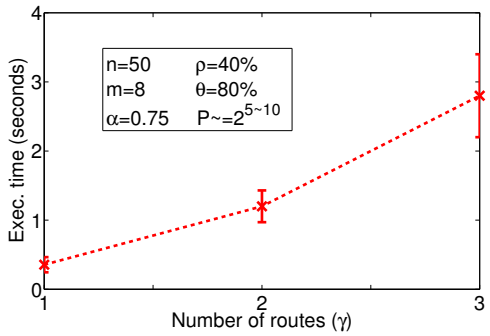
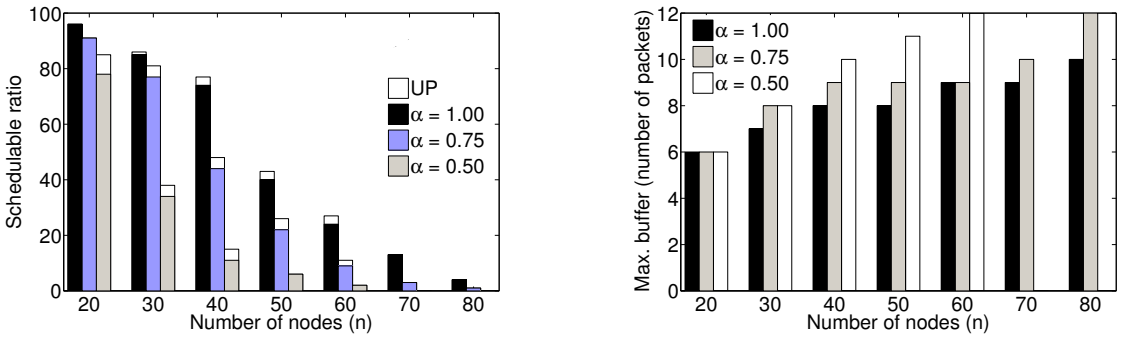


Figure 2.5: Execution time of C-LLF under varying number of routes (γ)

tight in practice, we plot UP as a conservative upper bound for the schedulable ratio under any scheduling algorithm. Figures 2.3(a), 2.3(b), and 2.3(c) show the schedulable ratios achieved by C-LLF and the baselines under varying γ, α, and periods, respectively, in 50-node networks. C-LLF consistently outperforms all baselines under all tested configurations. Moreover, its schedulable ratio remains close to UP.

Since the performances of PD and DM are less competitive, we no more present them. Figure 2.4 shows the performance of C-LLF against the baselines under varying number of nodes in the network. Figure 2.4(a) indicates that the schedulable ratio of C-LLF is higher than those of the baselines and is close to UP even when the number of nodes is 80. Figure 2.4(b) shows that the baselines are much faster than C-LLF. However, for 80 nodes, the average time of C-LLF is less than 2.5 seconds which is a reasonable time for computing schedules for WirelessHART networks.

Scalability of C-LLF. Figure 2.5 shows that the execution time of C-LLF increases sharply with the increase of γ. This is reasonable since increasing γ increases the workload significantly. However, in less than 3 seconds, can it complete scheduling when $\gamma = 3$. No feasible solution is found if we further increase γ. Figure 2.6 shows the scalability of C-LLF under increasing number of nodes and with different values of α when $\gamma = 2$. The Figure 2.6(a) shows that the schedulable ratio of C-LLF remains close to UP even with tighter deadlines (i.e., with lower α) in different sized networks. For 20, 70, and 80 nodes, C-LLF performs like an optimal algorithm as it achieves schedulable ratio equal to UP when $\alpha \geq 0.75$. Figure 2.6(b) shows the maximum buffer size required at a node when transmissions are scheduled under C-LLF. The maximum number packets buffered at a



(a) Schedulable ratio

(b) Maximum buffer

Figure 2.6: Scheduling by C-LLF under varying network sizes

node is 12 for a 80-node network. It indicate that buffer size required at a node does not dramatically increase with the increase of network size.

2.7.2 Simulations with Testbed Topologies

Network Topology. We evaluate our algorithms on the network topologies of a physical indoor testbed [2] (in Bryan Hall of Washington University in St Louis) consisting of 45 TelosB motes equipped with Chipcon CC2420 radios which are compliant with the IEEE 802.15.4 standard. At each transmission power level, every node broadcasts 50 packets while its neighbors record the sequence numbers of the packets they receive. After a node completes sending its 400 packets, the next sending node is selected in a round-robin fashion. This cycle is repeated giving each node 5 rounds to transmit 400 packets in each round. Figure 2.7 shows the network topology with transmission power of 0 dBm. Every link with a higher than 80% packet reception ratio is considered a reliable link and drawn in Figure 2.7. We test the algorithms on the topologies at 4 different power levels.

Scheduling performance. For the network topology at 0 dBm, Figure 3.7 shows the performances of C-LLF, the B&B, and the baseline heuristics under varying θ. In Figure 2.8(a), we see that the schedulable ratio of C-LLF is close to that of the B&B and is better than those of the baselines. It also shows that UP and the schedulable ratio of B&B are very close. Figure 2.8(b) shows that the average execution time of the B&B is 33 seconds while it is close to 0 seconds for C-LLF and the baselines when $\theta = 80\%$. Figure 2.9 shows the

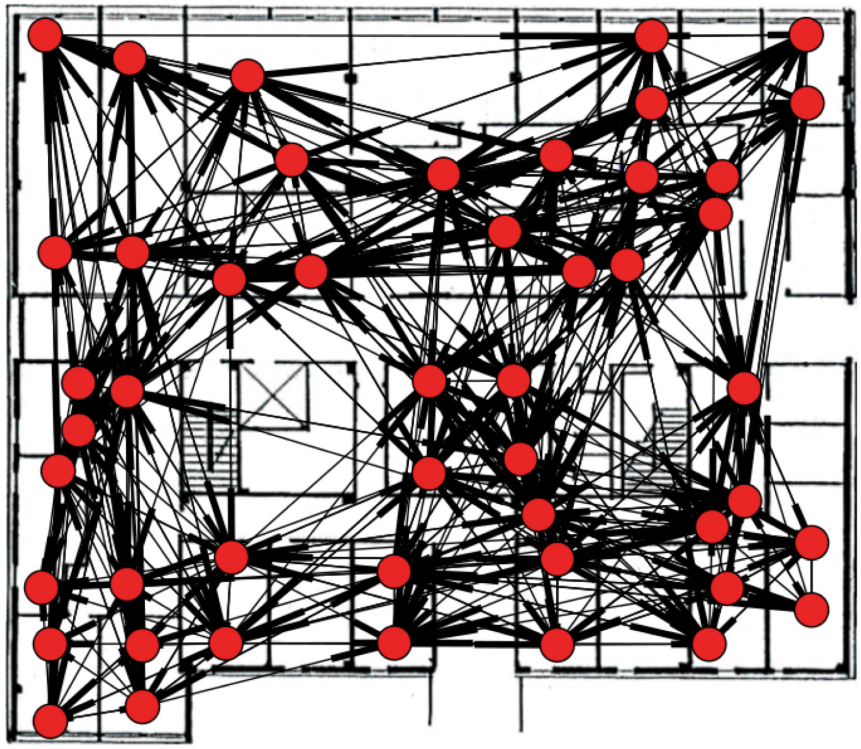
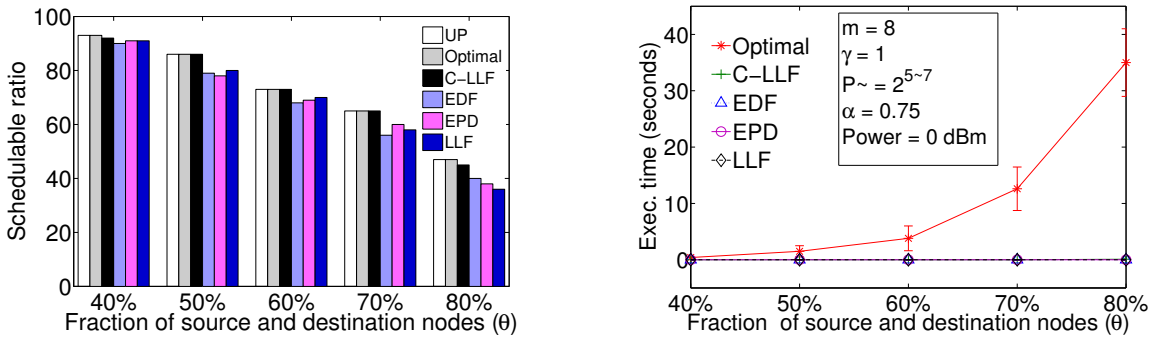


Figure 2.7: The testbed topology with a transmission power of 0 dBm



(a) Schedulable ratio

(b) Execution time

Figure 2.8: Scheduling with the B&B, C-LLF, and baselines under varying number of sources and destinations

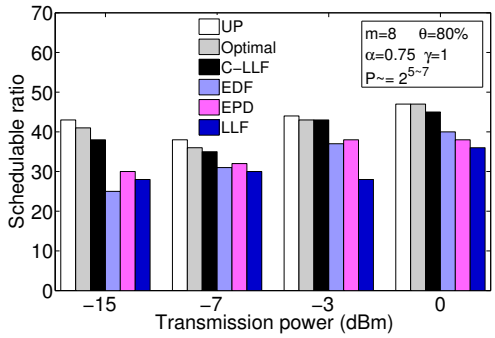


Figure 2.9: Schedulable ratio under different power level

performance comparison under different power levels. As expected, the schedulable ratios of all algorithms slightly decrease when the transmission power level decreases. However, at every power level, we can see that the schedulable ratio of C-LLF is close to that of the B&B as well as UP which demonstrates the effectiveness of C-LLF in meeting deadlines in WirelessHART networks.

2.8 Related Works

Although real-time transmission scheduling in wireless networks has been studied in the literature [170], few of previous works are applicable to WirelessHART networks. Several papers [101, 102, 116, 128, 176] proposed scheduling based on CSMA/CA MAC protocols. In contrast, WirelessHART adopts a TDMA-based approach to achieve predictable latency bounds. Several others developed TDMA scheduling algorithms [55, 60, 87, 123], but did not consider multi-channel communication supported by WirelessHART.

Scheduling for WirelessHART networks has been investigated recently since the standard was ratified in 2007. Convergecast scheduling has been studied for simplified network models such as linear [189] and tree networks [165] with depth no greater than the number of channels. There are several fundamental differences between our work and these previous studies. First, we consider the general WirelessHART network model. Our algorithms support multi-path routing, whereas the previous research only considered a single route for each node. Besides, our algorithms can deal with arbitrary network topologies without any

constraint on the length of routes. Second, our scheduling algorithms support real-time flows for feedback control loops, whereas previous research only considered data collection to the gateway. Finally and importantly, our algorithms aim to meet end-to-end deadlines which may differ based on the requirements of flows, while previous research focused on minimizing data collection latencies. Our research, therefore, addresses a more complicated scheduling problem suitable for process control in WirelessHART networks.

2.9 Summary

In this paper, we make key contributions to real-time transmission scheduling in WirelessHART networks: (1) formulation of the end-to-end real-time transmission scheduling problem based on the characteristics of WirelessHART, (2) proof of NP-hardness of the problem, (3) an optimal branch-and-bound scheduling algorithm based on a strong necessary condition, and (4) an efficient and practical heuristic-based algorithm called Conflict-aware Least Laxity First (C-LLF). The key insight underlying C-LLF is that it is important to incorporate transmission conflicts in scheduling policies for WirelessHART networks. Simulations based on both random and real network topologies demonstrate that C-LLF significantly outperforms the traditional real-time scheduling policies and that it is highly effective in meeting end-to-end communication deadlines.

Acknowledgement

This research was supported by NSF under grants CNS-0448554 (CAREER), CNS-0627126 (NeTS-NOSS), CNS-1017701 (NeTS), and CNS-0708460 (CRI).

Chapter 3

Real-Time Wireless: Delay Analysis for Fixed Priority Scheduling

WirelessHART is a new standard specifically designed for real-time and reliable communication between sensor and actuator devices for industrial process monitoring and control applications. End-to-end communication delay analysis for WirelessHART networks is required to determine the schedulability of real-time data flows from sensors to actuators for the purpose of acceptance test or workload adjustment in response to network dynamics. In this paper, we consider a network model based on WirelessHART, and map the scheduling of real-time periodic data flows in the network to real-time multiprocessor scheduling. We then exploit the response time analysis for multiprocessor scheduling and propose a novel method for the delay analysis that establishes an upper bound of the end-to-end communication delay of each real-time flow in the network. Simulation studies based on both random topologies and real network topologies of a 74-node physical wireless sensor network testbed demonstrate that our analysis provides safe and reasonably tight upper bounds of the end-to-end delays of real-time flows, and hence enables effective schedulability tests for WirelessHART networks.

3.1 Introduction

Wireless Sensor-Actuator Networks (WSANs) are an emerging communication infrastructure for monitoring and control applications in process industries. In a feedback control system where the networked control loops are closed through a WSAN, the sensor devices

periodically send data to the controllers, and the control input data are then delivered to the actuators through the network. To maintain the stability and control performance, industrial monitoring and control applications impose stringent end-to-end delay requirements on data communication between sensors and actuators [56]. Real-time communication is critical for process monitoring and control since missing a deadline may lead to production inefficiency, equipment destruction, and severe economic and/or environmental threats. For example, in oil refineries, spilling of oil tanks is avoided by monitoring and control of level measurement in real-time.

WirelessHART [22] has been designed as an open WSAN standard to address the challenges in industrial monitoring and control. To meet the stringent real-time and reliability requirements in harsh and unfriendly industrial environments, the standard features a centralized network management architecture, multi-channel Time Division Multiple Access (TDMA), redundant routes, and channel hopping [56]. These unique characteristics introduce unique challenges in end-to-end delay analysis for process monitoring and control in WirelessHART networks.

In this paper, we address the problem of end-to-end delay analysis for periodic real-time flows from sensors to actuators in a network that is modeled based on WirelessHART (simply named WirelessHART network throughout the paper). We derive upper bounds of the end-to-end delays of the flows under fixed priority scheduling where the transmissions associated with each flow are scheduled based on the fixed priority of the flow. Fixed priority scheduling is a common class of real-time scheduling policies in practice.

Analytical delay bounds can be used to test, both at design time and for online admission control, whether a set of real-time flows can meet all their deadlines. Compared to extensive testing and simulations, an end-to-end delay analysis is highly desirable in process monitoring and control applications that require real-time performance guarantees. It can also be used for adjusting the workload in response to network dynamics. For example, when a channel is blacklisted or some routes are recalculated, the delay analysis can be used to promptly decide whether some flow has to be removed or some rate has to be updated.

A key insight underlying our analysis is to map the real-time transmission scheduling in WirelessHART networks to real-time multiprocessor scheduling. This mapping allows us to provide a delay analysis of the real-time flows in WirelessHART networks by taking an

analysis approach similar to that for multiprocessor scheduling. By incorporating the unique characteristics of WirelessHART networks into the state-of-the-art worst case response time analysis for multiprocessor scheduling [88], we propose a novel end-to-end delay analysis for fixed priority transmission scheduling in WirelessHART networks. The proposed analysis calculates a safe and tight upper bound of the end-to-end delay of every real-time periodic data flow in pseudo polynomial time. Furthermore, we extend the pseudo polynomial time analysis to a polynomial time method that provides slightly looser bounds but can calculate the bounds more quickly.

We evaluate our analysis through simulations based on both random network topologies and the real network topologies of a wireless sensor network testbed consisting of 74 TelosB motes. The simulation results show that our delay bounds are safe and reasonably tight. The proposed analysis, hence, enables an effective schedulability test for WirelessHART networks.

In the rest of the paper, Section 3.2 reviews related works. Section 3.3 presents the network model. Section 3.4 defines the scheduling problem. Section 3.5 presents the mapping and the end-to-end delay analysis. Section 3.6 extends our delay analysis to a polynomial time method. Section 3.7 shows how our analysis can be extended for graph routing. Section 3.8 presents evaluation results. Section 3.9 concludes the paper.

3.2 Related Works

Real-time transmission scheduling in wireless networks has been widely studied in previous works [170]. However, very few of those are applicable to WirelessHART networks. Scheduling based on CSMA/CA protocols has been studied in [94,100–102,116,128,176]. In contrast, WirelessHART adopts a TDMA-based protocol to achieve predictable latency bounds. Although TDMA-based scheduling has been studied in [55, 87, 123], these works do not focus on schedulability or delay analysis. The authors in [25] propose a schedulability analysis for wireless sensor networks (WSNs) by upper bounding the real-time capacity of the network. However, in their model, taking the advantage of TDMA or frequency division has no effect. The schedulability analysis for WSNs has also been pursued in [60, 99, 159]. But these are

designed only for data collection through a routing tree using single channel, and do not address multi-channel communication or multi-path routing supported by WirelessHART.

For WirelessHART networks, routing [93], schedule modeling [32], real-time transmission scheduling [156, 157], and rate selection [154] have been studied recently. Our work in [157] proves the NP-hardness of the optimal real-time transmission scheduling in a WirelessHART network. It also presents an optimal scheduling algorithm based on branch-and-bound and a heuristic policy. Neither algorithm employed fixed priority. Moreover, no efficient worst-case delay analysis was provided for either algorithm. We studied priority assignment in [156] and rate selection methods in [154] for real-time flows in WirelessHART networks, both of which leverages worst-case delay analysis which is the focus of this paper. To summarize, none of our previous works addresses worst-case delay analysis. In contrast, this paper presents an end-to-end delay analysis that is suitable for any fixed priority scheduling policy. *Instead of devising a new real-time transmission scheduling algorithm, the key contribution of this paper is an efficient analysis for deriving the worst case delay bounds for real-time flows that are scheduled based on fixed priority.* An efficient delay analysis is particularly useful for online admission control and adaptation (e.g., when network route or topology changes) so that the network manager is able to quickly reassess the schedulability of the flows.

3.3 Network Model

We consider a network model inspired by WirelessHART. A WirelessHART network consisting of a set of field devices and one gateway. These devices form a mesh network that can be modeled as a graph $G = (V, E)$, where V is the set of nodes (i.e., field devices and the gateway), and E is the set of communication links between the nodes. A *field device* is either a sensor node, an actuator or both, and is usually connected to process or plant equipment. The *gateway* connects the WirelessHART network to the plant automation system, and provides the host system with access to the network devices. For any link $e = (u, v)$ in E, devices $u \in V$ and $v \in V$ can communicate with each other. For a transmission, denoted by $\vec{uv}$, that happens along link (u, v), device u is designated as the *sender* and device v the *receiver*. All network devices (i.e., field devices and the gateway) are able to send, receive, and route packets.

For process control, the controllers are installed in control hosts connected to the gateway through the plant automation network. The sensor devices deliver their sensor data to the gateway. The control messages from the gateway are then delivered to the actuators through the wireless mesh network. The unique features that make WirelessHART particularly suitable for industrial process control are as follows [22, 56].

Centralized Management. A WirelessHART network is managed by a *centralized network manager* installed in the gateway. The network manager collects the network topology information, and determines the routes. It then creates the schedule of transmissions, and distributes the schedules among the devices. Large-scale networks can be organized using multiple gateways or as hierarchical networks.

Time Division Multiple Access (TDMA). In WirelessHART networks, time is synchronized, and communication is TDMA-based. A time slot is 10ms long, and allows exactly one transmission and its associated acknowledgement between a device pair. For transmission between a receiver and its senders, a time slot can be either dedicated or shared. In a *dedicated time slot*, only one sender is allowed to transmit to the receiver. In a *shared slot*, more than one sender can attempt to transmit to the same receiver. Since collisions may occur within a shared slot, a transmission within a shared slot may be successful only when other senders do not need to send.

Route Diversity. To enhance the end-to-end reliability, both upstream and downstream communications are scheduled based on graph routing. A *routing graph* between two devices is a directed list of paths that connect two devices, thereby providing redundant paths between them. On one path from the source to the destination, the scheduler allocates a dedicated slot for each en-route device starting from the source, followed by allocating a second dedicated slot on the same path to handle a retransmission. Then, to offset failure of both transmissions along a primary link, the scheduler again allocates a third shared slot on a separate path to handle another retry.

Spectrum Diversity. Spectrum diversity gives the network access to all 16 channels defined in IEEE 802.15.4 and allows per time slot channel hopping in order to avoid jamming and mitigate interference from coexisting wireless systems. Besides, any channel that suffers

from persistent external interference is *blacklisted* and not used. Due to difficulty in detecting interference between nodes and the variability of interference patterns, WirelessHART networks typically avoid spatial reuse of a channel within the same time slot. Thus all transmissions in a time slot use different channels. This strategy effectively avoids transmission failure due to interference between concurrent transmissions, thereby providing a high degree of reliability for critical process monitoring and control applications. Henceforth we assume there is no spatial reuse of channels in this work.

Each device is equipped with a half-duplex omnidirectional radio transceiver and, hence, cannot both transmit and receive in the same time slot. In addition, two transmissions that have the same intended receiver interfere each other. Therefore, two transmissions $\vec{uv}$ and $\vec{ab}$ are *conflicting* and, hence, are not scheduled in the same slot if $(u = a) \vee (u = b) \vee (v = a) \vee (v = b)$. Since different nodes experience different degrees of conflict during communication, transmission conflicts play a major role in analyzing the end-to-end delays in the network.

Simplifying assumptions. As the first step toward a real-time schedulability analysis for WirelessHART networks, we make some simplifying assumptions on routing. Instead of a general graph routing, we assume a multi-path routing between every source and destination pair where the number of routes between each pair is a small constant (typically 1 or 2). To simplify the analysis further, we also assume that the packets are scheduled using dedicated slots only. The simplifying assumption facilitates the development of the first end-to-end delay analysis based on real-time scheduling theory. While our analysis leverages these simplified assumptions, it provides fundamental building blocks for the analysis based on general graph routing. Section 3.7 shows how our results can be extended to the analysis for general graph routing.

3.4 End-to-End Scheduling Problem

We consider a WirelessHART network $G = (V, E)$ with a set of end-to-end flows denoted by $\mathbb{F}$. Each flow $\mathbb{F}_j \in \mathbb{F}$ is characterized by a period P_j, a deadline D_j where $D_j \leq P_j$, and a set of one or more routes Φ_j. Each $\phi \in \Phi_j$ is a route from a network device $Source_j \in V$, called the *source of* $\mathbb{F}_j$, to another network device $Destination_j \in V$, called the *destination*

of $\mathbb{F}_j$, through the gateway. Each flow $\mathbb{F}_j$ periodically generates a packet at period P_j which originates at $Source_j$ and has to be delivered to $Destination_j$ within deadline D_j. For flow $\mathbb{F}_j$, if a packet generated at slot r is delivered to $Destination_j$ at slot f through a route $\phi \in \Phi_j$, its *end-to-end delay* through ϕ is defined as $L_j(\phi) = f - r + 1$.

A flow $\mathbb{F}_j$ may need to deliver its packet through more than one route in Φ_j. If the delivery through a route fails, the packet can still be delivered through another route in Φ_j. Therefore, in a TDMA schedule, for a flow $\mathbb{F}_j$, time slots must be reserved for transmissions through each route in Φ_j for redundancy. Hence, for end-to-end delay analysis purpose, through each of its routes flow $\mathbb{F}_j$ is treated as an individual flow F_i with deadline and period equal to $\mathbb{F}_j$'s deadline and period, respectively. That is, $\mathbb{F}_j$ is now considered $|\Phi_j|$ individual flows, each with a single route. Therefore, from now onward the term 'flow' will refer to an individual flow through a route. We denote this set of flows by $F = \{F_1, F_2, \cdots, F_N\}$. Thus, associated with each flow $F_i, 1 \leq i \leq N$, are a period P_i, a deadline D_i, a source node $Source_i$, a destination node $Destination_i$, and a route ϕ_i from $Source_i$ to $Destination_i$. For each flow F_i, if every transmission is repeated χ times to handle retransmission on a single route, then the number of transmissions required to deliver a packet from $Source_i$ to $Destination_i$ through its route ϕ_i is $C_i = length(\phi_i) * \chi$, where $length(\phi_i)$ is the number of links on ϕ_i. Thus, C_i is the number of time slots required by flow F_i.

Fixed priority scheduling. For fixed priority scheduling, each flow F_i has a fixed priority. We assume that all flows are ordered by priorities. Flow F_i has higher priority than flow F_j if and only if $i < j$. We use $hp(F_i)$ to denote the set of flows whose priorities are higher than that of flow F_i. That is, $hp(F_i) = \{F_1, F_2, \cdots, F_{i-1}\}$. In practice, priorities may be assigned based on deadlines, rates, or the criticality of the real-time flows. Priority assignment policies are not the focus of this paper, and our delay analysis can be applied to any fixed priority assignment. Under a *fixed priority scheduling policy*, the transmissions of the flows are scheduled in the following way. Starting from the highest priority flow F_1, the following procedure is repeated for every flow F_i in decreasing order of priority. For current priority flow F_i, the network manager schedules its transmissions along its route (starting from the source) on earliest available time slots and on available channels. A time slot is *available* if no conflicting transmission is already scheduled in that slot. In a WirelessHART network, the complete schedule is divided into superframes. A *superframe* represents transmissions

in a series of time slots that repeat infinitely and represent the communication pattern of a group of devices.

Problem formulation. Transmissions are scheduled using m channels. The set of flows F is called *schedulable* under a scheduling algorithm $\mathbb{A}$, if $\mathbb{A}$ is able to schedule all transmissions in m channels such that no deadline is missed, i.e., $L_i \leq D_i, \forall F_i \in F$, with L_i being the end-to-end delay of F_i. For $\mathbb{A}$, a schedulability test $\mathbb{S}$ is *sufficient* if any set of flows deemed schedulable by $\mathbb{S}$ is indeed schedulable by $\mathbb{A}$. To determine schedulability of a set of flows, it is sufficient to show that, for every flow, an upper bound of its worst case end-to-end delay is no greater than its deadline. Thus, given the flows F and a fixed priority algorithm $\mathbb{A}$, our objective is to decide schedulability of F based on end-to-end delay analysis. As proven in [157] that it is NP-complete to decide the schedulability of a set of periodic real-time flows in a WirelessHART network for both dynamic priority and fixed priority scheduling, an exact schedulability analysis (i.e., both sufficient and necessary) is an NP-hard problem. We therefore pursue an end-to-end delay analysis which serves only as a sufficient condition for schedulability.

Uses of a sufficient analysis. An end-to-end delay analysis can be used as a schedulability test for real-time flows in practice. Since the delay analysis provides upper bounds of the delays, a set of flows is guaranteed to be schedulable if their delay bounds meet the respective deadlines. On the other hand, since the bounds can be pessimistic, it is possible for a set of flows to be actually schedulable when their delay bounds exceed the deadlines. In this case, a conserve but safe approach is not to admit all the flows since a schedulability guarantee is not assured. Since creating a complete schedule for all flows requires exponential time in general (as it has to be created up to the hyper-period of all flows), such a sufficient schedulability analysis is more efficient and suitable for networks that need to decide schedulability quickly, e.g., in response to network changes at run time. For real-time flows in industrial process control applications that require hard real-time guarantees, a sufficient analysis can thus be used for online admission control and to adjust workload in response to network dynamics. For example, when a channel is blacklisted or some routes are recalculated, the network manager can execute our sufficient analysis to verify whether the current set of flows remain schedulable. If the analysis cannot guarantee the schedulability of all the flows, the network manager may remove a subset of the flows (e.g., based on criticality) or reduce the data rates of some of the flows so that the new set of flows becomes schedulable under our analysis.

3.5 End-to-end Delay Analysis

In this section, we present an end-to-end delay analysis for the real-time flows in a WirelessHART network. An efficient end-to-end delay analysis is particularly useful for online admission control and adaptation to network dynamics so that the network manager is able to quickly reassess the schedulability of the flows (e.g., when network route or topology changes, or some channel is blacklisted). In analyzing the end-to-end delays, we observe two reasons that contribute to the delay of a flow. A lower priority flow can be delayed by higher priority flows (a) due to *channel contention* (when all channels are assigned to transmissions of higher priority flows in a time slot), and (b) due to *transmission conflicts* (when a transmission of the flow and a transmission of a higher priority flow involve a common node). At first, we analyze each delay separately. We, then, incorporate both types of delays into our analysis and end up with an upper bound of the end-to-end delay for every flow. A holistic approach that can analyze two types of delays combining into a single step might lead to tighter delay bound, but we opt for the divide-and-conquer approach to simplify the theoretical analysis of the safety of the bound. If every transmission is repeated χ times to handle retransmission on a single route, then every time slot is simply multiplied by χ in delay calculation. For simplicity of presentation we use retransmission parameter $\chi = 1$.

3.5.1 Delay due to Channel Contention

Observations between Transmission Scheduling and Multiprocessor CPU Scheduling

A key insight in this work is that we can map the multi-channel fixed priority transmission scheduling problem for WirelessHART networks to the fixed priority real-time CPU scheduling on a global multiprocessor platform. Towards this direction, we make the following observations between these two domains.

In a WirelessHART network, each channel can accommodate one transmission in a time slot across the entire network. Thus, a flow executing for one time unit on a CPU of a multiprocessor system is equivalent to a packet transmission on a channel which takes exactly one time slot in a WirelessHART network. That one flow cannot be scheduled on

different processors at the same time is similar to the fact that one flow cannot be scheduled on different channels at the same time. In addition, flows executing on multiprocessor platform are considered independent while the flows being scheduled in a WirelessHART network are also independent. Again, execution of flows on a global multiprocessor platform is equivalent to switching of a packet to different channels at different time slots due to channel hopping[1]. Finally, completing the execution of a flow on a CPU is equivalent to completing all transmissions of a packet from the source to the destination of the flow.

Thus, in absence of conflicts, the worst case response time of a flow in a multiprocessor platform is equivalent to the upper bound of its end-to-end delay in a WirelessHART network. Therefore, to analyze the delay due to channel contention, we can map the transmission scheduling in a WirelessHART network to global multiprocessor CPU scheduling.

Mapping to Multiprocessor CPU Scheduling

Based on the observations discussed above, the mapping from multi-channel transmission scheduling in a WirelessHART network to multiprocessor CPU scheduling is as follows.

- Each channel is mapped to a *processor*. Thus, m channels correspond to m processors.
- Each flow $F_i \in F$, is mapped to a *task* that executes on multiprocessor with period P_i, deadline D_i, execution time C_i, and priority equal to the priority of flow F_i.

While the proposed mapping allows us to potentially leverage the rich body of literature on real-time CPU scheduling, the end-to-end delay analysis for WirelessHART networks remains an open problem. An important observation is that we must consider transmission conflicts in the delay analysis. Note that transmission conflict is a distinct feature of wireless networks that does not exist in traditional real-time CPU scheduling problems. A key contribution of our work, therefore, is to incorporate the delays caused by transmission conflicts into the end-to-end delay analysis. By incorporating the delay due to these conflicts into the

[1]Since a flow is not restricted to be scheduled on a specific channel, partitioned or semi-partitioned approach does not fit for our network model. Instead, global scheduling is a suitable real-time scheduling approach for the transmission scheduling problem.

multiprocessor real-time schedulability analysis, we establish an upper bound of the end-to-end delay of every flow in a WirelessHART network.

In the proposed end-to-end delay analysis, we first analyze the delay due to channel contention between the flows. Whenever there is a channel contention between two flows, the lower priority flow is delayed by the higher priority one. Based on the above mapping, the analysis for the worst case delay that a lower priority flow experiences from the higher priority flows due to channel contention in a WirelessHART network is similar to that when the flows are scheduled on a multiprocessor platform. Therefore, instead of establishing a completely new analysis for the delay due to channel contention, the proposed mapping allows us to exploit the results of the state-of-the-art response time analysis for multiprocessor scheduling [88].

Response Time Analysis for Multiprocessor

To make our paper self-contained, here we present the results of the state-of-the-art response time analysis for multiprocessor scheduling proposed by Guan et al. [88]. Assuming that the flows are executed on a multiprocessor platform, they have observed that a flow experiences the worst case delay when the earliest time instant after which all processors are occupied by the higher priority flows occurs just before its release time. Therefore, for flow F_k, a *level-k busy period* is defined as the maximum continuous time interval during which all processors are occupied by flows of priority higher than or equal to F_k's priority, until F_k finishes its active instance. We use the notation $\mathrm{BP}(k,t)$ to denote a level-k **b**usy **p**eriod of t slots. The delay that some higher priority flow $F_i \in hp(F_k)$ will cause to F_k depends on the workload of all instances of F_i during a $\mathrm{BP}(k,t)$. Flow F_i has *carry-in* workload in a $\mathrm{BP}(k,t)$, if it has one instance with release time earlier than the $\mathrm{BP}(k,t)$ and deadline in the $\mathrm{BP}(k,t)$. When F_i has no carry-in, an upper bound $W_k^{\mathrm{nc}}(F_i,t)$ of its workload in a $\mathrm{BP}(k,t)$, and an upper bound $I_k^{\mathrm{nc}}(F_i,t)$ of the delay it can cause to F_k are as follows.

$$W_k^{\mathrm{nc}}(F_i,t) = \left\lfloor \frac{t}{P_i} \right\rfloor . C_i + \min(t \bmod P_i,\ C_i) \tag{3.1}$$

$$I_k^{\mathrm{nc}}(F_i,t) = \min\left(W_k^{\mathrm{nc}}(F_i,t),\ t - C_k + 1\right) \tag{3.2}$$

When F_i has carry-in, an upper bound $W_k^{\text{ci}}(F_i, t)$ of its workload in a BP(k, t), and an upper bound $I_k^{\text{ci}}(F_i, t)$ of the delay that it can cause to F_k are as follows.

$$W_k^{\text{ci}}(F_i, t) = \left\lfloor \frac{\max(t - C_i, 0)}{P_i} \right\rfloor . C_i + C_i + \mu_i \tag{3.3}$$

$$I_k^{\text{ci}}(F_i, t) = \min\Big(W_k^{\text{ci}}(F_i, t), t - C_k + 1\Big) \tag{3.4}$$

where carry-in $\mu_i = \min\Big(\max\Big(\lambda - (P_i - R_i), 0\Big), C_i - 1\Big)$; $\lambda = \max(t - C_i, 0) \bmod P_i$; with R_i being the worst case response time of F_i.

With the observation that at most $m - 1$ higher priority flows can have carry-in, an upper bound $\Omega_k(t)$ of the total delay caused by all higher priority flows to an instance of F_k during a BP(k, t) is

$$\Omega_k(t) = X_k(t) + \sum_{F_i \in hp(F_k)} I_k^{\text{nc}}(F_i, t) \tag{3.5}$$

with $X_k(t)$ being the sum of the $\min(|hp(F_k)|, m - 1)$ largest values of the differences $I_k^{\text{ci}}(F_i, t) - I_k^{\text{nc}}(F_i, t)$ among all $F_i \in hp(F_k)$.

3.5.2 Delay due to Transmission Conflicts

Now we analyze the delay that a flow can experience due to transmission conflicts. Whenever two transmissions conflict, the transmission that belongs to the lower priority flow must be delayed, no matter how many channels are available. Since different transmissions experience different degrees of conflict during communication, these conflicts play a major role in analyzing the end-to-end delays in the network. In the following discussion, we derive an upper bound of the delay that a lower priority flow can experience from the higher priority ones due to conflicts.

Two flows F_k and F_i are said to be *conflicting* when a transmission of F_k conflicts with a transmission of F_i, i.e., their transmissions involve a common node. When F_k and $F_i \in hp(F_k)$ conflict, F_k has to be delayed due to having lower priority. Intuitively, the amount of delay depends on how their routes intersect. A transmission $\vec{uv}$ of F_k is delayed at most by ω slots by an instance of F_i, if F_i has ω transmissions that involve node u or v. For example, in

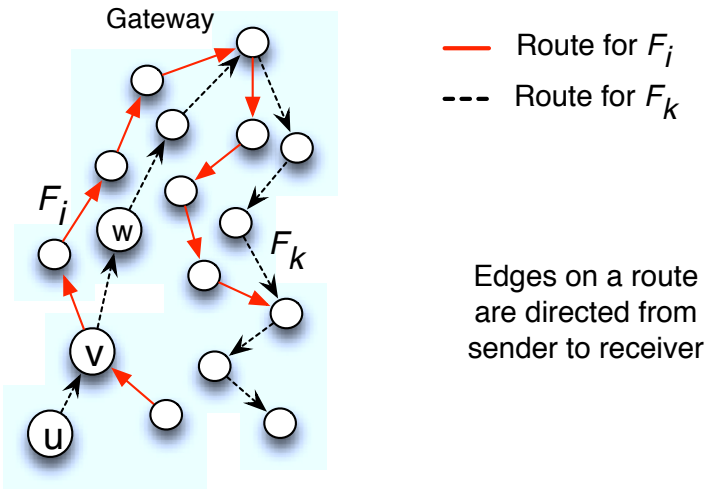
Figure 7.1, a transmission $\overrightarrow{uv}$ or $\overrightarrow{vw}$ of F_k has to be delayed at most by 2 slots by an instance of F_i. Let $Q(k,i)$ be the total number of F_i's transmissions that share nodes on F_k's route. Since two routes can intersect arbitrarily, in the worst case, flow F_k may conflict with each of these $Q(k,i)$ transmissions of F_i. As a result, $Q(k,i)$ represents an upper bound of the delay that F_k can experience from an instance of F_i due to conflicts.

$Q(k,i)$ often overestimates the delay because when there is "too much" overlap between the routes of F_i and F_k, F_i will not necessarily cause "too much" delay to F_k. We define $\Delta(k,i)$ as a more precise upper bound of the delay that F_k can experience from an instance of F_i due to transmission conflicts. In Figure 7.1, an instance of F_k can be delayed by an instance of F_i at most by 5 slots since $Q(k,i) = 5$, but in Figure 7.4, F_k can be delayed by an instance of F_i at most by 3 slots while $Q(k,i) = 8$. To obtain a value of $\Delta(k,i)$, we introduce the concept of a *maximal common path (MCP)* between F_k and F_i defined as a path $v_1 \to v_2 \to \cdots \to v_h$, where $v_l \neq v_q$ for $l \neq q$ (where $1 \leq l, q \leq h$), on F_i's route such that $v_1 \to v_2 \to \cdots \to v_h$ or $v_h \to v_{h-1} \to \cdots \to v_1$ is a path on F_k's route and it is maximal, i.e., no such longer path contains it (Figure 7.4). On an MCP between F_k and F_i, denoted by $M_j(k,i)$, F_k can be directly delayed by F_i at most by 3 slots, no matter how long the MCP is. For $M_j(k,i)$, we define its *length* $\beta_j(k,i)$ as the total number of F_i's transmissions along it. That is, for $M_j(k,i) = v_1 \to \cdots \to v_h$, if there exist $u, w \in V$ such that $u \to v_1 \to \cdots \to v_h \to w$ is also on F_i's route, then $\beta_j(k,i) = h+1$. If only u or only w exists, then $\beta_j(k,i) = h$. If neither u nor v does exist, then $\beta_j(k,i) = h-1$. During the time when F_i executes these transmissions (i.e., $\overrightarrow{uv_1}$, $\overrightarrow{v_1v_2}, \cdots, \overrightarrow{v_hw}$), it can cause delay to F_k at most by 3 of these transmissions. Thus, Lemma 3 establishes a value of $\Delta(k,i)$.

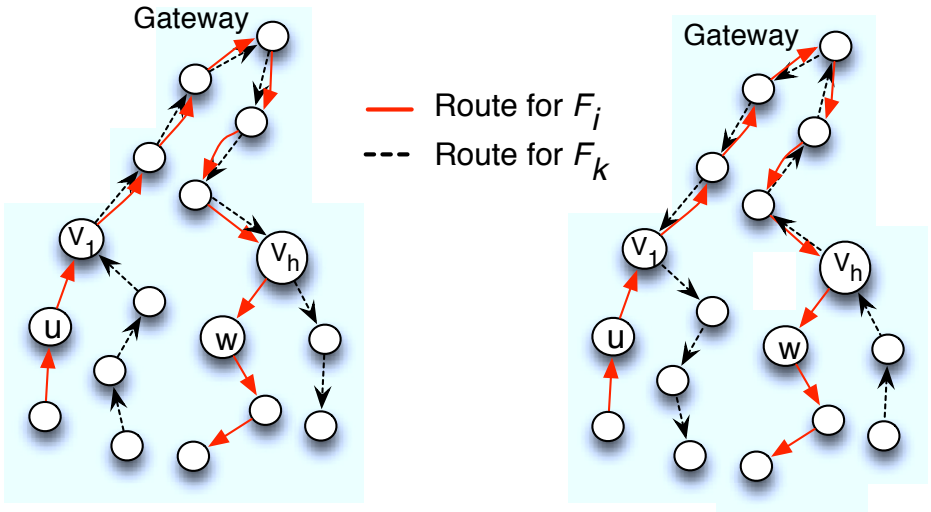
Lemma 3. *Let $\beta'_j(k,i)$ denote the length of an MCP $M'_j(k,i)$ between F_k and $F_i \in hp(F_k)$ with length at least 4. If there are total $\sigma(k,i)$ MCPs between F_k and F_i each with length at least 4, then*

$$\Delta(k,i) = Q(k,i) - \sum_{j=1}^{\sigma(k,i)} \Big(\beta'_j(k,i) - 3\Big) \tag{3.6}$$

Proof. Let an MCP $M'_j(k,i)$ be $v_1 \to \cdots \to v_h$. Let there exist u and w such that the path $u \to v_1 \to \cdots \to v_h \to w$ is on F_i's route. Now, either $v_1 \to \cdots \to v_h$ or $v_h \to \cdots \to v_1$ must lie on F_k's route (Figure 7.4). If $v_1 \to \cdots \to v_h$ is on F_k's route, then a transmission $v_l\overrightarrow{v_{l+1}}$, $1 \leq l < h$, of F_k on this path shares node with at most 3 transmissions of F_i on



(a) $Q(k,i) = 5$ and $\Delta(k,i) = 5$



(b) Two scenarios where routes of F_k and F_i overlap with $Q(k,i) = 8$, but $\Delta(k,i) \leq 3$

Figure 3.1: An example when F_k can be delayed by F_i

$u \rightarrow v_1 \rightarrow \cdots \rightarrow v_h \rightarrow w$. Similarly, if $v_h \rightarrow \cdots \rightarrow v_1$ is on F_k's route, then a transmission $\overrightarrow{v_l v_{l-1}}$, $1 < l \leq h$, of F_k on this path shares node with at most 3 transmissions of F_i on $u \rightarrow v_1 \rightarrow \cdots \rightarrow v_h \rightarrow w$. Therefore, in either case, a transmission of F_k on $M'_j(k,i)$ can be delayed by the transmissions of F_i on $M'_j(k,i)$ at most by 3 slots. Again, in either case, once the delayed transmission of F_k is scheduled, the subsequent transmissions of F_k and F_i on $M'_j(k,i)$ do not conflict and can happen in parallel. That is, for any $M'_j(k,i)$ with length at least 4, at least $\beta'_j(k,i) - 3$ transmissions will not cause delay to F_k. But $Q(k,i)$ counts every transmission of F_i on $M'_j(k,i)$. Therefore, $Q(k,i) - \sum_{j=1}^{\sigma(k,i)} \left(\beta'_j(k,i) - 3\right)$ represents the bound $\Delta(k,i)$. □

According to Lemma 3, we need to look for an MCP only if $Q(k,i) \geq 4$ and at least 4 consecutive transmissions of F_i share nodes on F_k's route. This is because in such cases looking for an MCP will no longer reduce the bound as the delay is (already) at most 3 (as $Q(k,i)$ is at most 3). Again, when $\beta'_j(k,i)$ is calculated for an $M'_j(k,i)$, we look for the next MCP only if $Q(k,i) - \beta'_j(k,i) \geq 4$.

The number of instances of flow $F_i \in hp(F_k)$ that contribute to the delay of an instance of flow F_k during a time interval of t slots is upper bounded by $\lceil \frac{t}{P_i} \rceil$. Hence, the total delay that an instance of F_k can experience from flow F_i is at most $\left\lceil \frac{t}{P_i} \right\rceil \Delta(k,i)$. An upper bound of the total delay that flow F_k can experience from all higher priority flows due to transmission conflicts during a time interval of t slots is denoted by $\Theta_k(t)$ and can thus be expressed as

$$\Theta_k(t) = \sum_{F_i \in hp(F_k)} \left\lceil \frac{t}{P_i} \right\rceil . \Delta(k,i) \tag{3.7}$$

3.5.3 A Tighter Bound on Conflict Delay

The upper bound derived in Equation 3.7 for the transmission conflict delay experienced by a flow is based on pessimistic assumptions that will result in overestimate of the end-to-end delay of the flow. In this subsection, we avoid the pessimistic assumptions, and establish a tighter bound on the delay of a flow that occurs due to transmission conflict.

To determine $\Theta_k(t)$ in Equation 3.7, we assumed that

1. The lower priority flow F_k is delayed by every instance of the higher priority flow F_i that is released within the time interval of t slots, and

2. The lower priority flow F_k is delayed by $\Delta(k,i)$ time slots by every instance of F_i.

In a real scheduling sequence, as we present in the next discussion, not every instance of a higher priority flow F_i can cause delay by $\Delta(k,i)$ time slots on F_k, thereby making the above assumptions highly pessimistic. The delay due to transmission conflicts plays a major role in the end-to-end delay of a flow. Overestimate in conflict delay may result in significant pessimism in the end-to-end delay analysis. In the rest of this subsection, we provide critical observations to avoid these pessimistic assumptions, and establish a more precise bound on conflict delay, that results in an improved schedulability test.

The pessimistic assumptions are due to the fact that the analysis for determining $\Theta_k(t)$ in the previous subsection does not exclude F_k's transmissions that have already been scheduled into the consideration for calculating the future delay on F_k. In other words, some transmissions of F_k that have already been scheduled are still considered to be subject to delay by F_i, which clearly should not be the case.

Since a flow is a chain of transmissions from a source to a destination, in considering the conflict delay caused by multiple instances of F_i on flow F_k, we observe that at the time when a transmission of F_k conflicts with some transmission of F_i, the preceding transmissions on F_k are already scheduled. These *already scheduled* transmissions of F_k are no longer subject to delay by the subsequent instances of F_i. For example, in Figure 7.1 let us consider that one instance of F_i is conflicting and causing delay on F_k's transmission $\vec{vw}$. This implies that F_k's transmission $\vec{uv}$ is already scheduled (since transmission $\vec{vw}$ can be ready only after transmission $\vec{uv}$ is scheduled). Hence, the next instance of F_i must not cause delay on transmission $\vec{uv}$ (since this transmission is already scheduled). That is, in calculating $\Theta_k(t)$ for F_k, only the transmissions that have not yet been scheduled should be considered for conflict delay by the subsequent instances of F_i (that will be released in future in the considered time interval). These observations lead to Lemma 4, and then to Theorem 5 to upperbound the total delay (due to transmission conflict) caused on F_k by all instances of F_i.

Lemma 4. *Let us consider any two instances of a higher priority flow F_i such that each causes conflict delay on a lower priority flow F_k in a time interval. Then, there is at most one common transmission on F_k that can be delayed by both instances.*

Proof. Let these two instances of F_i be denoted by $F_{i,1}$ and $F_{i,2}$, where $F_{i,1}$ is released before $F_{i,2}$. Suppose to the contrary, both of these instances cause delay on two transmissions, say τ_j and τ_r, of the lower priority flow F_k. Without loss of generality, we assume that τ_j precedes τ_r on the route of flow F_k. $F_{i,1}$ causes delay on τ_r because τ_r is ready to be scheduled. This implies that τ_j has already been scheduled. Hence, $F_{i,2}$ which releases after $F_{i,1}$ cannot cause any delay on τ_j, thereby contradicting our assumption. □

Based on Lemma 4, we can now determine a tight upper bound of the conflict delay caused by multiple instances of F_i on F_k in any case. To do so, we introduce the notion of a *bottleneck transmission* (of F_k with respect to F_i) which is the transmission of F_k that may face the maximum conflict delay from F_i. An upper bound of the conflict delay caused by one instance of F_i on F_k's bottleneck transmission is denoted by $\delta(k,i)$, and is determined in the following way. For every transmission τ of F_k, we count the total number of F_i's transmissions that share a node with τ. Then, the maximum of these values (among all transmissions of F_k) is determined as $\delta(k,i)$. In other words, there are at most $\delta(k,i)$ transmissions of (one instance of) F_i such that each of them share a node (and hence may conflict) with the same transmission of F_k. By Lemma 4, for any two instances of F_i, F_k has at most one transmission on which both instances can cause delay. In the worst case, the bottleneck transmission of F_k can be delayed by multiple instances of F_i. Hence, the value of $\delta(k,i)$ plays a major role in determining the delay caused by F_i on F_k as shown in Theorem 5.

Theorem 5. *In a time interval of t slots, the worst case conflict delay caused by a higher priority flow F_i on a lower priority flow F_k is upper bounded by*

$$\Delta(k,i) + \Big(\Big\lfloor \frac{t}{P_i} \Big\rfloor - 1 \Big).\delta(k,i) + \min\Big(\delta(k,i), t \bmod P_i\Big)$$

Proof. For the case when $t < P_i$, there is at most one instance of F_i in a time interval of t slots. Hence, the total conflict delay caused by F_i on F_k is at most $\Delta(k,i)$ which clearly follows the theorem. We consider the case with $t \geq P_i$ for the rest of the proof.

There are at most $\lceil \frac{t}{P_i} \rceil$ instances of F_i in a time interval of t slots. Let the set of transmissions of F_i which cause conflict delay on F_k be denoted by Γ. When one instance $F_{i,1}$ of F_i causes conflict delay on F_k, a subset Γ_1 of Γ causes the delay. Now consider a second instance $F_{i,2}$ of F_i. For $F_{i,2}$, another subset Γ_2 of Γ causes delay on F_k. When all subsets $\Gamma_1, \Gamma_2, \cdots, \Gamma_{\lceil \frac{t}{P_i} \rceil}$ are mutually disjoint, by the definition of $\Delta(k,i)$, the conflict delay caused by Γ on F_k is at most $\Delta(k,i)$. Hence, the total conflict delay caused by all $\Gamma_1, \Gamma_2, \cdots, \Gamma_{\lceil \frac{t}{P_i} \rceil}$ in this case is at most $\Delta(k,i)$. That is, the total conflict delay on F_k caused by F_i is at most $\Delta(k,i)$.

Now let us consider the case when the subsets $\Gamma_1, \Gamma_2, \cdots, \Gamma_{\lceil \frac{t}{P_i} \rceil}$ are not mutually disjoint, i.e., there is at least one pair Γ_j, Γ_h such that $\Gamma_j \cap \Gamma_h \neq \emptyset$, where $1 \leq j, h \leq \lceil \frac{t}{P_i} \rceil$. Let the total delay caused by all instances of F_i on F_k is $\Delta(k,i) + Z(k,i)$, i.e., the delay is higher than $\Delta(k,i)$ by $Z(k,i)$ time slots. The additional delay (beyond $\Delta(k,i)$) happens because the transmissions that are common between Γ_j and Γ_h cause both instances of F_i to create delay on F_i. By Lemma 4, for any two instances of F_i, F_k has at most one transmission on which both instances can cause delay. If there is no transmission of F_k that is delayed by both the p-th instance and the $p+1$-th instance of F_i, then no transmission of F_k is delayed by both the p-th instance and the q-th instance of F_i, for any $q > p+1$, where $1 \leq p < \lceil \frac{t}{P_i} \rceil$. Thus, $Z(k,i)$ is maximum when for each pair of consecutive instances (say, the p-th instance and $p+1$-th instance, for each p, $1 \leq p < \lceil \frac{t}{P_i} \rceil$) of F_i, there is a transmission of F_k that is delayed by both instances. Hence, at most $\lceil \frac{t}{P_i} \rceil - 1$ instances contribute to this additional delay $Z(k,i)$, each instance causing some additional delay on a transmission. Since one instance of F_i can cause delay on a transmission of F_k at most by $\delta(k,i)$ slots, $Z(k,i) \leq (\lceil \frac{t}{P_i} \rceil - 1)\delta(k,i)$. Since the last instance may finish after the considered time window of t slots, the delay caused by it is at most $\min(\delta(k,i), t \bmod P_i)$ slots. Taking this into consideration, $Z(k,i) \leq (\lfloor \frac{t}{P_i} \rfloor - 1)\delta(k,i) + \min(\delta(k,i), t \bmod P_i)$. Thus, the total delay caused on F_k by all instances of F_i is at most

$$\Delta(k,i) + Z(k,i) \leq \Delta(k,i) + (\lfloor \frac{t}{P_i} \rfloor - 1).\delta(k,i) + \min(\delta(k,i), t \bmod P_i)$$

□

From Theorem 5, now $\Theta_k(t)$ (i.e., an upper bound of the total delay flow F_k can experience from all higher priority flows due to transmission conflicts during a time interval of t slots)

is calculated as follows.

$$\Theta_k(t) = \sum_{F_i \in hp(F_k)} \left(\Delta(k,i) + \left(\left\lfloor \frac{t}{P_i} \right\rfloor - 1 \right) . \delta(k,i) + \min\left(\delta(k,i), t \bmod P_i \right) \right) \tag{3.8}$$

Since usually $\delta(k,i) \ll \Delta(k,i)$, the above value of $\Theta_k(t)$ is significantly smaller than that derived in Equation 3.7. Our simulation results (in Section 3.8) also demonstrate that the above bound is a significant improvement over the bound derived in Equation 3.7.

3.5.4 End-to-End Delay Bound

Now we consider both types of delays together to develop an upper bound of the end-to-end delay of every flow. For a flow, we first derive an upper bound of its end-to-end delay assuming that it does not conflict with any higher priority flow. We then incorporate its worst case delay due to conflict into this upper bound. This is done for every flow in decreasing order of priority starting with the highest priority flow as explained below.

For F_k, we use $R_k^{\text{ch,con}}$ to denote an upper bound of the worst case end-to-end delay considering delays due to both **ch**annel contention and **con**flicts between flows. We use the following two steps to estimate $R_k^{\text{ch,con}}$ for every flow $F_k \in F$ in decreasing order of priority starting with the highest priority flow.

Step 1

First, we calculate a pseudo upper bound (i.e., not an actual upper bound), denoted by R_k^{ch}, of the worst case end-to-end delay of F_k assuming that F_k is delayed by the higher priority flows due to channel contention only. That is, we assume that F_k does not conflict with any higher priority flow. This calculation is based on the upper bounds $R^{\text{ch,con}}$ of the worst case end-to-end delays of the higher priority flows which are already calculated considering both types of delay. Based on our discussion in Subsection 3.5.1, to determine R_k^{ch}, the worst case delay that flow F_k will experience from the higher priority flows can be calculated using Equation 3.5. The amount of delay that a higher priority flow F_i will cause to F_k depends on F_i's workload during a BP(k,x) (i.e., a level-k busy period of x slots). Note that, in Equations 3.1 and 3.3, the workload bound of F_i was derived in absence of conflict between

the flows. Now we first analyze the workload bound of $F_i \in hp(F_k)$ in the network where both channel contention and transmission conflicts contributed to the worst case end-to-end delay of F_i.

From Equation 3.1, if flow F_i does not have carry-in, its workload $W_k^{\text{nc}}(F_i, x)$ during a BP(k, x) does not depend on its worst case end-to-end delay. Therefore, if F_i has no carry-in, $W_k^{\text{nc}}(F_i, x)$ during a BP(k, x) still can be calculated using Equation 3.1, no matter what the worst case end-to-end delay of F_i is. That is,

$$W_k^{\text{nc}}(F_i, x) = \left\lfloor \frac{x}{P_i} \right\rfloor . C_i + \min(x \bmod P_i, C_i) \tag{3.9}$$

Now $I_k^{\text{nc}}(F_i, x)$ is calculated using Equation 4.5.3 and is guaranteed to be an upper bound of the delay that $F_i \in hp(F_k)$ can cause to F_k due to channel contention.

From Equation 3.3, when flow F_i has carry-in, its workload $W_k^{\text{ci}}(F_i, x)$ during a BP(k, x) depends on its worst case response time R_i. Equation 3.3 also indicates that $W_k^{\text{ci}}(F_i, x)$ is monotonically nondecreasing in R_i. Now, in the WirelessHART network, an upper bound of the end-to-end delay of F_i must be no less than R_i since both channel contention and transmission conflicts contribute to its end-to-end delay. That is, $R_i^{\text{ch,con}} \geq R_i$. Therefore, if we replace R_i with $R_i^{\text{ch,con}}$ in Equation 3.3, $W_k^{\text{ci}}(F_i, x)$ is guaranteed to be an upper bound of F_i's workload during a BP(k, x). Thus,

$$W_k^{\text{ci}}(F_i, x) = \left\lfloor \frac{\max(x - C_i, 0)}{P_i} \right\rfloor . C_i + C_i + \mu_i \tag{3.10}$$

where $\mu_i = \min\Big(\max\Big(\lambda - (P_i - R_i^{\text{ch,con}}), 0\Big), C_i - 1\Big)$ and $\lambda = \max(x - C_i, 0) \bmod P_i$. Similarly, $I_k^{\text{ci}}(F_i, x)$ calculated using Equation 3.4 is guaranteed to be an upper bound of the delay that F_i can cause to F_k due to channel contention.

Once the bounds $I_k^{\text{nc}}(F_i, x)$ and $I_k^{\text{ci}}(F_i, x)$ of the delay from every higher priority flow $F_i \in hp(F_k)$ are calculated, the total delay $\Omega_k(x)$ that an instance of F_k experiences from all higher priority flows during a BP(k, x) due to channel contention is calculated using Equation 3.5. Now assuming that F_k does not conflict with any higher priority flow, an upper bound of its end-to-end delay can be found using the same iterative method that is used for multiprocessor scheduling [88]. Since there are m channels, the pseudo upper bound R_k^{ch} of the worst

case end-to-end delay of F_k can be obtained by finding the minimal value of x that solves Equation 3.11.

$$x = \left\lfloor \frac{\Omega_k(x)}{m} \right\rfloor + C_k \tag{3.11}$$

Equation 3.11 is solved using an iterative fixed-point algorithm starting with $x = C_k$. This algorithm either terminates at some fixed-point $x^* \leq D_k$ that represents the bound R_k^{ch} or x will exceed D_k eventually. In the latter case, this algorithm terminates and reports the instance as *"unschedulable"*.

Effect of Channel Hopping. To every transmission, the scheduler assigns a channel offset between 0 and $m-1$ instead of an actual channel, where m is the total number of channels. Any channel offset c (i.e., 1, 2, $\cdots, m-1$) is mapped to different channels at different time slots s as follows.

$$channel = (c + s) \bmod m$$

That is, although the physical channels used along a link changes (hops) in every time slot, the total number m of available channels is fixed. The scheduler only assigns a fixed channel index to a transmission which maps to different physical channels in different time slots, keeping the total number of available channels at m always, and scheduling each flow on at most one channel at any time. Hence, channel hopping does not have effect on channel contention delay.

Step 2

Once the value of R_k^{ch} is computed, we incorporate the transmission conflict delay into it to obtain the bound $R_k^{\text{ch,con}}$. Namely, for flow F_k, the bound R_k^{ch} has been derived in Step 1 by assuming that F_k does not conflict with any higher priority flow. Therefore, in this step, we take into account that F_k may conflict with the higher priority flows and, hence, can experience further delay from them. An upper bound $\Theta_k(y)$ of the total delay that an instance of F_k can experience due to conflicts with the higher priority flows during a time interval of y slots is calculated using Equation 3.8. Note that when F_k conflicts with some higher priority flow it must be delayed, no matter how many channels are available. Therefore, we add the delay $\Theta_k(y)$ to the pseudo upper bound R_k^{ch} to derive an upper bound

of F_k's worst case end-to-end delay. Thus, the minimal value of y that solves the following equation gives the bound $R_k^{\text{ch,con}}$ for F_k that includes both types of delay:

$$y = R_k^{\text{ch}} + \Theta_k(y) \tag{3.12}$$

Equation 3.12 is solved using an iterative fixed-point algorithm starting with $y = R_k^{\text{ch}}$. Like Step 1, this algorithm also either terminates at some fixed-point $y^* \leq D_k$ that is considered as the bound $R_k^{\text{ch,con}}$ or terminates with an *"unschedulable"* decision when $y > D_k$. Thus, termination of the algorithm is guaranteed.

Theorem 6. *For every flow $F_k \in F$, let R_k^{ch} be the minimal value of $x \geq C_k$ that solves Equation 3.11, and $R_k^{ch,con}$ be the minimal value of $y \geq R_k^{ch}$ that solves Equation 3.12. Then $R_k^{ch,con}$ is an upper bound of the worst case end-to-end delay of F_k.*

Proof. Flows are ordered according to their priorities as $F_1, F_2, \cdots, F_N$ with F_1 being the highest priority flow. We use mathematical induction on priority level k, $1 \leq k \leq N$. When $k = 1$, i.e., for the highest priority flow F_1, Equations 3.11 and 3.12 yield $R_1^{\text{ch,con}} = C_1$, where C_1 is the number of transmissions along F_1's route. Since no flow can delay the highest priority flow F_1, the end-to-end delay of F_1 is always C_1. Hence, the upper bound calculated using Equation 3.12 holds for $k = 1$. Now let the upper bound calculated using Equation 3.12 holds for flow F_k, for any k, $1 \leq k < N$. We have to prove that the upper bound calculated using it also holds for flow F_{k+1}.

To calculate $R_{k+1}^{\text{ch,con}}$ in Step 2, we initialize y (in Equation 3.12) to R_{k+1}^{ch}. Note that R_{k+1}^{ch} is computed in Step 1 for flow F_{k+1}. In Step 1, R_{k+1}^{ch} is computed considering upper bounds $R_h^{\text{ch,con}}$ of the worst case end-to-end delays of all F_h with $h < k+1$ which are already computed considering both types of delay. Equation 3.11 assumes that F_{k+1} does not conflict with any higher priority flow. This implies that the minimal solution of x, i.e., R_{k+1}^{ch} is an upper bound of the worst case end-to-end delay of F_{k+1}, if F_{k+1} is delayed by the higher priority flows due to channel contention only. If F_{k+1} conflicts with some higher priority flow, then it can be further delayed by the higher priority flows at most by $\sum_{F_h \in hp(F_{k+1})} \lceil \frac{y}{P_h} \rceil \Delta(k+1, h)$ slots during any time interval of length y. Equation 3.12 adds this delay to R_{k+1}^{ch} and establishes the recursive equation for y. Therefore, the minimal solution of y, i.e., $R_{k+1}^{\text{ch,con}}$ is guaranteed to be an upper bound of the worst case end-to-end delay of F_{k+1} that includes the worst case delays both due to channel contention and due to conflicts between flows. □

The end-to-end delay analysis procedure calculates $R_i^{\text{ch,con}}$, for $i = 1, 2, \cdots, N$ (in decreasing order of priority level), and decides the flow set to be schedulable if, for every $F_i \in F$, $R_i^{\text{ch,con}} \leq D_i$. According to Equations 3.11 and 3.12, each $R_i^{\text{ch,con}}$ can be calculated in pseudo polynomial time for every F_i. The correctness of this upper bound of the worst case end-to-end delay follows from Theorem 6.

Note that our above analysis has been derived considering the retransmission parameter $\chi = 1$. If every transmission is repeated χ times to handle retransmission on a single route, then every time slot is simply multiplied by χ in delay calculation. Hence, to adopt the above delay analysis for any general value of χ, we simply replace the values of $C_i, \Delta(k,i)$, and $\delta(k,i)$ with $C_i.\chi$, $\Delta(k,i).\chi$, and $\delta(k,i).\chi$, respectively. Our model is motivated by WirelessHART [22] that uses a fixed number of retransmissions for all links. It is trivial from the above analysis to handle varying the number of transmissions for different links based on link qualities. Specifically, instead of multiplying the above values by a uniform value of χ, we have to consider different values for different links.

3.6 Delay Analysis in Polynomial Time

We now extend the pseudo polynomial time analysis to a polynomial time method. While this may provide comparatively looser bounds, it can calculate the bounds more quickly, and hence is more suitable for online use when time efficiency is critical.

Exploiting the same mapping presented in Section 3.5, we can use the polynomial time response time analysis for global multiprocessor scheduling proposed in [45] to calculate the channel contention delays. In particular, using this analysis, the maximum channel contention delay, denoted by $\Omega_k(D_k)$, that a flow F_k can experience during its lifetime from the higher priority flows can be expressed as follows.

$$\Omega_k(D_k) = \sum_{F_i \in hp(F_k)} \min(W_k(i), D_k - C_k + 1), \tag{3.13}$$

$$\text{where } W_k(i) = \left\lfloor \frac{D_k + D_i - C_i}{P_i} \right\rfloor .C_i + \min\left(C_i, D_k + D_i - C_i - \left\lfloor \frac{D_k + D_i - C_i}{P_i} \right\rfloor .P_i\right)$$

Therefore, similar to Equation 3.11, R_k^{ch} of F_k (i.e., the worst case end-to-end delay of F_k assuming that it is delayed by the higher priority flows due to channel contention only) can be calculated as follows.

$$R_k^{\text{ch}} = \left\lfloor \frac{\Omega_k(D_k)}{m} \right\rfloor + C_k \tag{3.14}$$

To calculate the conflict delay of F_k in polynomial time, we can estimate the maximum delay in an interval of D_k slots from Equation 3.8 as follows.

$$\Theta_k(D_k) = \sum_{F_i \in hp(F_k)} \left(\Delta(k,i) + \left(\left\lfloor \frac{D_k}{P_i} \right\rfloor - 1 \right) . \delta(k,i) + \min\left(\delta(k,i), D_k \bmod P_i \right) \right) \tag{3.15}$$

Like Equation 3.12, the worst case end-to-end delay $R_k^{\text{ch,con}}$ of flow F_k considering both channel contention delay and transmission conflict delay is calculated as

$$R_k^{\text{ch,con}} = R_k^{\text{ch}} + \Theta_k(D_k) \tag{3.16}$$

The above analysis does not require calculating the worst case end-to-end delays of the flows in order of their priorities. Since the lower priority flows have the higher chances of missing deadlines, the above analysis allows us to calculate the end-to-end delays of the lower priority flows first, thereby getting a quicker decision on the schedulability of the flows.

3.7 Extending to Graph Routing Model

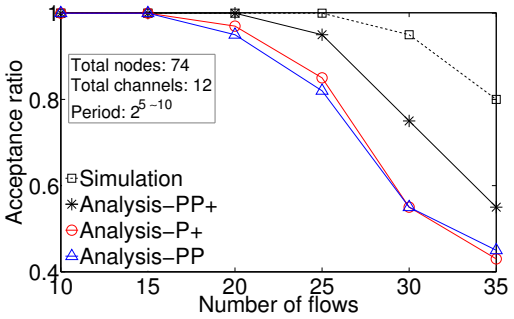
Now we show that our analysis based on simplified assumptions provides the theoretical foundation for more practical analysis based on graph routing. For end-to-end communication based on graph routing between a source and destination pair in a WirelessHART network, a packet is scheduled on each link on each path in the routing graph between the pair for redundancy. The convention is to allocate one link for each en-route device starting from the source, followed by allocating a second dedicated slot on the same path to handle a retransmission, and then to allocate a third shared slot on a separate path to handle another retry. For example, if a transmission is successful in the first attempt, two other redundant time slots allocated for this transmission remain unused. Hence, although the

packet is scheduled on each path, only one path will be chosen by the packet based on link conditions. In other words, the packet is always directed through one single route in the routing graph. Hence if we analyze the end-to-end delay for the packet through each single route in the routing graph, the maximum delay among these paths represents the packet's worst case end-to-end delay.

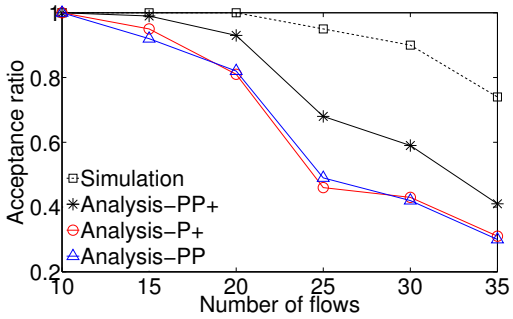
Based on the above observation, using our current analysis we can determine the end-to-end delay through each path in a routing graph. Note that, in graph routing, some links on a path may be scheduled using shared time slots. Since the links scheduled for shared slots do not need to wait for dedicated slots, the delay bound through these paths may be shorter than those determined by our current method for the same paths. As a result, our method will provide pessimistic upper bounds of end-to-end delays for graph routing. Another issue is the efficiency in terms of time complexity. A routing graph can have many single routes, and analyzing the delay through each single route may not be an efficient method. Leveraging our current result based on simplified assumptions, our future work will investigate efficient delay analysis based on graph routing.

3.8 Evaluation

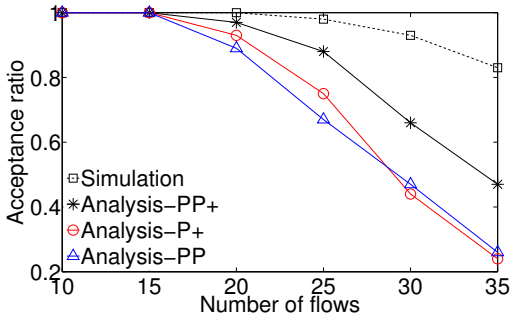
We evaluate our end-to-end delay analysis through simulations based on both random topologies and a real wireless sensor network testbed topologies. Evaluations are performed in terms of acceptance ratio and pessimism ratio. *Acceptance ratio* is the proportion of the number of test cases deemed schedulable by the delay analysis method to the total number of test cases. For each flow, *pessimism ratio* is quantified as the proportion of the analyzed theoretical bound to its maximum end-to-end delay observed in simulation. In particular, *pessimism ratio* quantifies our overestimate in the analytical delay bounds. Due to this overestimate in the delay bounds, some test case that is schedulable may be determined as unschedulable by our conservative delay analysis, and hence is rejected by admission control based on our analysis. The impact of a sufficient delay analysis on the pessimism of admission control is quantified by the *acceptance ratio* metric. The higher the acceptance ratio, the less pessimistic (i.e., more effective) the delay analysis.



(a) Topology at -1 dBm Tx power



(b) Topology at -3 dBm Tx power



(c) Topology at -5 dBm Tx power

Figure 3.2: Schedulability without retransmission on testbed topology

There is no baseline to compare the performance of our analysis which, to our knowledge, is the first delay analysis for real-time flows in WirelessHART networks. Hence, we evaluate the performance of our delay analysis by observing the delays through simulations of the complete schedule of all flows released within the hyper-period. In the figures in this section, **"Simulation"** denotes the fraction of test cases that have no deadline misses in the simulations. This fraction indicates an upper bound of acceptance ratio for any delay analysis method. The analyses evaluated in this section are named as follows.

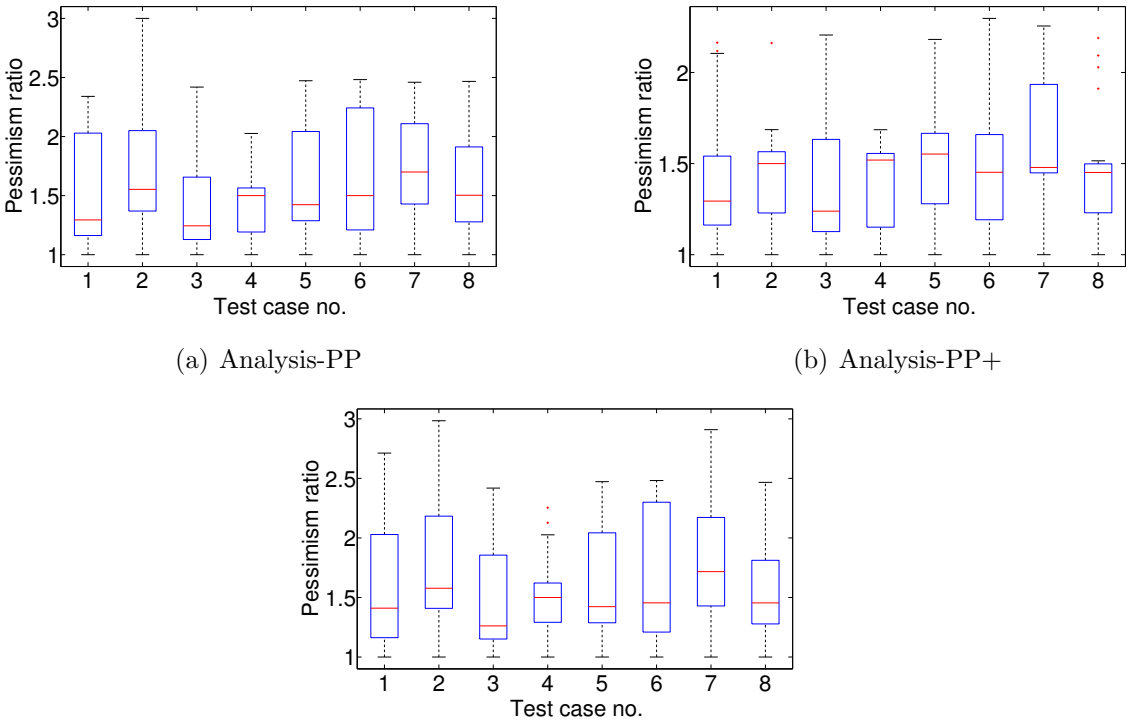
Analysis-PP is the pseudo polynomial time analysis without considering the improved conflict delay bound of Section 3.5.3. Namely, it calculates the end-to-end delay bound using Equation 3.12 where the conflict delay is calculated based on Equation 3.7.

Analysis-PP+ is the pseudo polynomial time analysis by considering the tighter conflict delay bound of Section 3.5.3. That is, Analysis-PP+ calculates the end-to-end delay bound using Equation 3.12 where the conflict delay is calculated based on Equation 3.8.

Analysis-P+ is the polynomial time analysis derived in Section 3.6. It calculates the delay bounds using Equation 3.16 based on the tighter conflict delay bound.

3.8.1 Simulation Setup

A fraction of nodes is considered as sources and destinations. The sets of sources and destinations are disjoint. The *reliability* of a link is represented by the *packet reception ratio (PRR)* along it. The node with the highest number of neighbors is designated as the gateway. Since all flows pass through the gateway, we determine routes between the sources and destinations that include the gateway. Routes are determined based on link reliabilities. The most reliable route connecting a source to a destination is determined as the primary route. For additional routes, we choose the next most reliable route that excludes the links of any existing route between the same source and destination. Each flow is assigned a harmonic period of the form 2^a time slots, where $a > 1$. The deadline of each flow is set equal to its period. The priorities of the flows are assigned based on *deadline monotonic* policy that assigns priorities according to relative deadlines.



(a) Analysis-PP

(b) Analysis-PP+

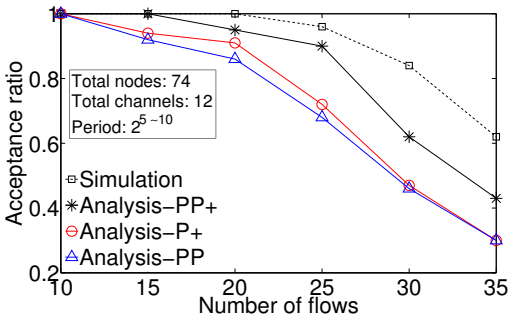
(c) Analysis-P+

Figure 3.3: Pessimism ratio without retransmission on testbed topology

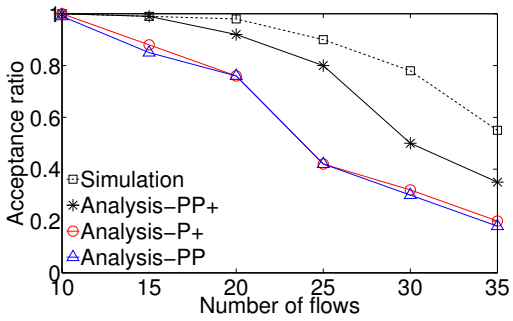
3.8.2 Simulations with Testbed Topologies

Due to large impact of transmission conflicts on the end-to-end delay of a flow, the delay analysis largely depends on the topology of the network since transmission conflicts depend on how the links or routes intersect (as seen in Sections 3.5.2 and 3.5.3). Therefore, first we conduct simulation results based on real network topologies. These are the topologies of a wireless sensor network testbed, and are generated using various transmission power levels of its nodes since the network connectivity (hence the topology) varies as we vary transmission powers. Our testbed consists of 74 TelosB motes each equipped with Chipcon CC2420 radios which are compliant with IEEE 802.15.4 (WirelessHART's physical layer is also based on IEEE 802.15.4). It is deployed in two buildings of Washington University [2]. Setting the same transmission (Tx) power at every node, each node (in a round-robin fashion) broadcasts 50 packets while its neighbors record the sequence numbers of the packets they receive. This cycle is repeated giving each node 5 rounds to transmit 50 packets in each round. Every link with a higher than 80% PRR is considered a reliable link to derive the topology of the testbed. We collected topologies at 3 different Tx power levels (-1 dBm, -3 dBm, -5 dBm). We generate different flows in these topologies by randomly selecting the sources and destinations. Their periods are randomly generated in the range $2^{5\sim10}$ time slots. We generate 100 test cases considering these topologies.

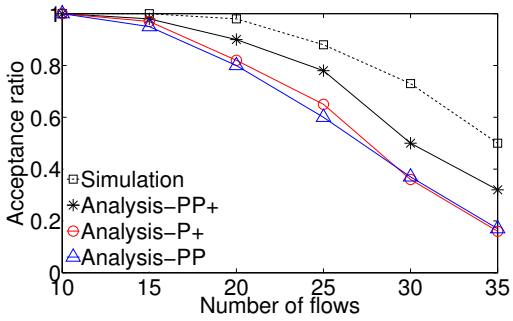
Figure 3.2 shows the acceptance ratios of our delay analysis methods without considering retransmission and without redundant routes. According to Figure 3.2(a), when the number of flows $N < 25$ in the topology with Tx power of -1 dBm, Analysis-PP+ has an acceptance ratio of 1.0, which means that all test cases that are indeed schedulable are also deemed schedulable by our analysis. When $N = 30$, the value of "Simulation" is 0.99 while the acceptance ratio of Analysis-PP+ is 0.95 which indicates that the analysis is highly efficient. After that, the acceptance ratios of our analysis decreases with the increase in N. However, the difference between its acceptance ratio and the value of "Simulation" always remains strictly less than 0.24. Therefore, the acceptance ratios are always tight for any (moderate or severe) overload in the testbed topology. Besides, the acceptance ratio of Analysis-PP+ is always a lot higher than that of Analysis-PP. This happens because the delay bounds calculated in Analysis-PP+ are significantly tighter than those in Analysis-PP. Analysis-P+ which determines looser bounds in polynomial time is highly competitive against Analysis-PP. This happens because Analysis-P+ determines the conflict delay based on the improvement made



(a) Topology at -1 dBm Tx power



(b) Topology at -3 dBm Tx power



(c) Topology at -5 dBm Tx power

Figure 3.4: Schedulability with retransmission on testbed topology

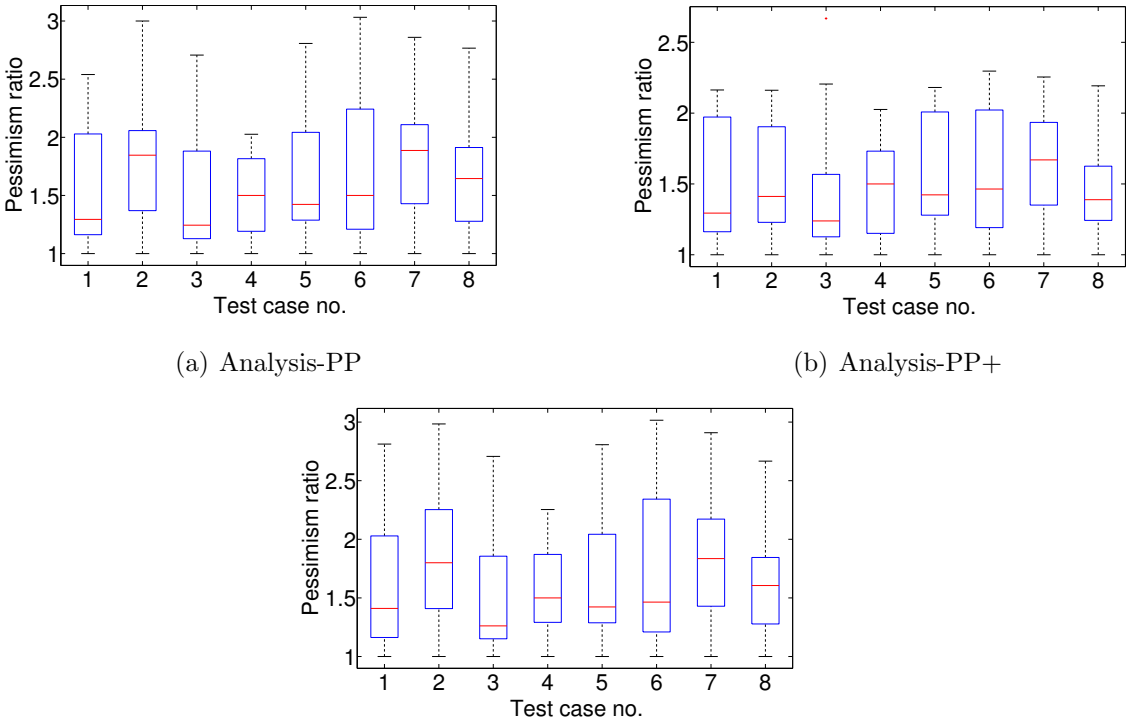
in Equation 3.8. Figures 3.2(b) and 3.2(c) show the similar results for the topology with Tx power of -3 dBm and -5 dBm, respectively.

Now we analyze our results to evaluate the tightness of the delay bounds in terms of pessimism ratios. Among 100 test cases, each consisting of 25 flows from the above experiment we randomly select 8 test cases that are schedulable under all 3 analyses, and plot the distributions of pessimism ratios as box plots in Figure 3.3. The figures indicate that the end-to-end delay bounds calculated in Analysis-PP+ are tighter than those calculated in Analysis-PP since the former uses a tighter bound of conflict delay. Specifically, the results show that the 75th percentiles of the pessimism ratios are no greater than 1.75, 2.2, and 2.25 for Analysis-PP+, Analysis-PP, and Analysis-P+, respectively. This indicates that the delay bounds derived in our new analysis (Analysis-PP+) are much tighter than those in the original analysis (Analysis-PP). Even the polynomial time analysis Analysis-P+ that uses our improved conflict delay analysis is highly competitive against Analysis-PP that

is a pseudo polynomial-time analysis. These results thus indicate that incorporating our improved conflict delay analysis into the original analysis significantly tightens the delay bounds. In addition, if we look back to Figure 3.2 for acceptance ratio, our algorithms are effective for admission control (in term of acceptance ratio) despite the (high) pessimism ratio.

Figure 3.4 shows the acceptance ratios by considering retransmissions but without redundant routes. In this case, the acceptance ratios are a lot lower than those in Figure 3.2. This is reasonable because we have to schedule each transmission in two time slots, and due to limited bandwidth the schedulable cases (observed in simulation) are also lower than those in Figure 3.2 (wthout retransmission). However, these results also indicate that the acceptance ratio of Analysis-PP+ is always higher than that of Analysis-PP. For the same 8 test cases selected in Figure 3.3, we now draw the pessimism ratios in Figure 3.5 considering retransmissions. Figure 3.5 indicates that the pessimism ratios increase in some cases but do not vary a lot compared to the case without retransmission. The pessimism ratios increase in some cases but do not vary significantly compared to the case without retransmission. Since both the analytical delay (x) and the delay observed in simulations (y) increase under retransmissions, the pessimism ratios ($\frac{x}{y}$) do not vary significantly compared to the case without retransmission.

We now determine the schedulability considering both retransmissions and redundant routes. That is, for each transmission along the primary route between a sources and destination is scheduled on 2 time slots. In addition, each packet is also scheduled along each redundant route. Figure 3.6 shows how the schedulability changes with the increase of number of routes considering 25 flows in the topology with -1 dBm Tx power. When there is no redundant route, the value of "Simulation" is 0.96 while the acceptance ratio under Analysis-PP+ is 0.9. As the number of redundant routes increases, the schedulable cases as well as acceptance ratios decrease sharply. However, at least 50% of the total schedulable cases are determined as schedulable by Analysis-PP+ as long as the number of redundant routes is no greater than 2. When there are 3 redundant routes, the value of "Simulation" is 0.15 and the acceptance ratio under Analysis-PP+ is 0.05. This decrease in acceptance ratio is because many redundant links need to be scheduled.



(a) Analysis-PP

(b) Analysis-PP+

(c) Analysis-P+

Figure 3.5: Pessimism ratio with retransmission on testbed topology

These results demonstrate that the improved analysis (derived in Subsection 3.5.3) of transmission conflict delay is highly effective in reducing the pessimism of the analysis. It also shows that the polynomial-time analysis is reasonably tight when compared against the original pseudo polynomial time analysis.

3.8.3 Simulations with Random Topologies

We test the scalability of our algorithms in terms of number of flows on random topologies of larger number of nodes. Given the number of nodes and edge-density, we generate random networks. A network with n nodes and ρ% edge-density has a total of $(n(n-1)*\rho)/(2*100)$ bidirectional edges. The edges are chosen randomly and assigned PRR randomly in the range $[0.80, 1.0]$. Then we generate different number of flows in 400-node networks of 40% edge-density. For every different number of flows, we generate 100 test cases. The periods

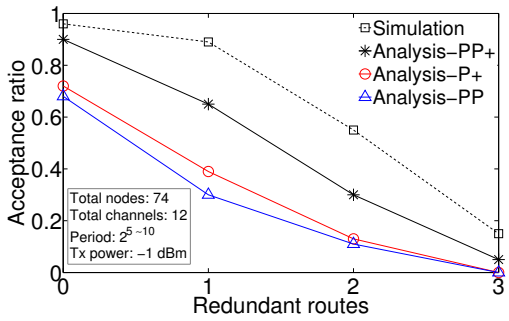
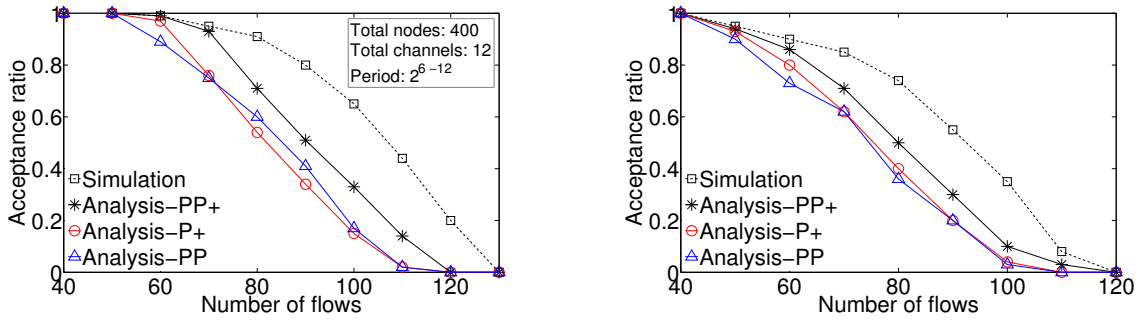


Figure 3.6: Schedulability with retransmission and redundant routes on testbed topology



(a) Without retransmission

(b) With retransmission

Figure 3.7: Schedulability on random topology

are considered harmonic and are randomly generated in the range $2^{6\sim12}$ time slots. Larger periods (compared to the case with testbed topologies) are used to accommodate large networks and a large number of flows.

The acceptance ratios of our analyses in 400-node network are shown in Figure 3.7. Figure 3.7(a) shows that without retransmission the acceptance ratio of Analysis-PP+ is equal to the value of "Simulation" as long as the number of flows is no greater than 60. As the number of flows increases, the difference between the acceptance ratios of Analysis-PP+ and the value of "Simulation" increases but always remains less than 0.33. Figure 3.7(b) shows the results considering retransmissions along the primary route but no redundant routes. In this case acceptance ratios in all methods are lower since the total number of actual schedulable cases are lower. However, the acceptance ratio of Analysis-PP+ is always higher than that of Analysis-PP, and Analysis-P+ is competitive against Analysis-PP.

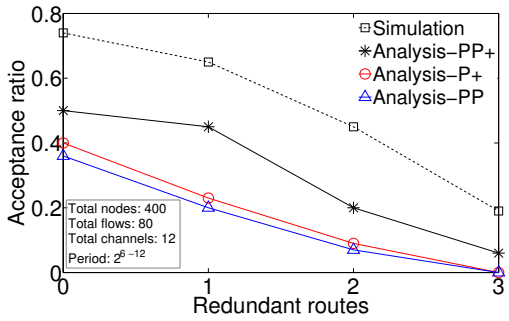


Figure 3.8: Schedulability with retransmission and redundant routes on random topology

Figure 3.8 shows the results for 80 flows in the 400-node network under retransmissions and varying number of redundant routes. Similar to our results with testbed topology here also we observe that both the value of "Simulation" and the acceptance ratios of our analyses decrease sharply with the increase in the number of redundant routes.

In every setup, we have observed that the acceptance ratios of our analysis are close to those of simulation which indicates that not many schedulable cases are rejected by our analysis. All test cases accepted by our analysis meet their deadlines in the simulations which demonstrates that the estimated bounds are safe. The results demonstrate that our analysis can be used as an acceptance test for real-time flows under various network configurations.

3.9 Summary

In this paper, we have mapped the transmission scheduling of real-time data flows between sensors and actuators in a WirelessHART network to real-time multiprocessor scheduling. Based on the mapping, we have presented an end-to-end delay analysis to determine the schedulability of real-time data flows in WirelessHART networks. Through simulation studies, we have demonstrated that our analysis enables effective schedulability tests for WirelessHART networks.

Chapter 4

Real-Time Wireless: Delay Analysis for Reliable Graph Routing

Wireless sensor-actuator networks are gaining ground as the communication infrastructure for process monitoring and control. Industrial control applications demand a high degree of reliability and real-time guarantees in communication between sensors and actuators. Because wireless communication is susceptible to transmission failures in industrial environments, industrial wireless standards such as WirelessHART adopt reliable graph routing to handle transmission failures through retransmissions in dedicated and shared time slots and route diversity. While these mechanisms are critical for reliable communication, they introduce substantial challenges in analyzing the schedulability of real-time flows. This paper presents the first worst-case end-to-end delay analysis for periodic real-time flows under reliable graph routing. The proposed delay analysis can be used to quickly assess the schedulability of real-time flows with stringent requirements on both high reliability and network latency. We have implemented and experimented our analytical results on a wireless testbed of 69 nodes. Both experimental results and simulations show that our delay bounds are safe and enable effective schedulability tests under reliable graph routing.

4.1 Introduction

Wireless sensor-actuator networks (WSANs) are gaining ground as the communication infrastructure for industrial process monitoring and control systems. To support monitoring and control, a WSAN periodically delivers data from sensors to a controller and then delivers

its control input data to the actuators through the multi-hop mesh network. Wireless control in process industries demands a high degree of reliability and real-time guarantees in communication [166]. Failures in wireless transmissions are prevalent in industrial environments due to channel noise, power failure, physical obstacle, multi-path fading, and interference from co-existing wireless systems. Industrial wireless standards such as WirelessHART [22] adopt a *reliable graph routing* approach to handle transmission failures through retransmissions and route diversity.

Reliable graph routing [22] employs the following mechanisms to recover from transmission failures. A routing graph is constructed as a directed list of paths between two devices, thereby providing redundant routes for real-time flows between sensors and actuators. For each flow, the network handles transmission failures by allocating a *dedicated time slot* (i.e., a time slot when at most one transmission is scheduled to a receiver) for each node on a path from the source, followed by allocating a second dedicated slot on the same path for a retransmission, and then by allocating a third *shared slot* (i.e., a time slot when multiple nodes may contend to send to a common receiver) on a separate path for another retransmission [22]. While highly effective in achieving reliable communication, this fault-tolerant mechanism introduces significant challenges in worst-case delay analysis for real-time flows in a WSAN. For industrial wireless control applications with stringent requirements on both high reliability and network latency, an efficient worst-case delay analysis is of utmost importance to quickly assess the schedulability of real-time flows for the purpose of online admission control or workload adjustment in response to network dynamics.

In this paper, we propose the first worst-case end-to-end delay analysis for periodic real-time flows under reliable graph routing. Specifically, we consider periodic real-time flows whose transmissions are scheduled based on fixed priority. In a *fixed priority scheduling* policy, all transmissions of a flow are scheduled based on the fixed priority of the flow. While delay analyses for single or independent routes have been proposed in the literature [25, 60, 99, 155, 159], an efficient delay analysis under reliable graph routing in a WSAN represents a challenging open problem. Since a routing graph for a flow can consist of an exponential (in number of nodes) number of routes between its source and destination, determining an effective delay bound for a flow by enumerating all of these paths is time consuming, making it unsuitable for WSANs subject to frequent changes to link and channel conditions

in industrial environments. We address this challenge and propose an end-to-end delay analysis *without* enumerating all the paths.

In a WSAN under multi-channel graph routing, a flow may be delayed by higher priority flows due to *channel contention* (when all channels are assigned to the higher priority flows in a time slot) and due to *transmission conflicts* (when a transmission of the lower priority flow and a transmission of a higher priority flow involve a common node). We use an efficient method based on depth-first search to determine an upper bound of transmission conflict delay of each flow. This bound holds for all paths in the flow's routing graph and is computed without requiring the enumeration of all the paths. We then analyze the properties of graph routing and exploit an observation that, unlike single or independent routes, transmission conflict may increase channel contention in graph routing. Through an analysis of the worst-case scenario for transmission conflict and the worst-case scenario for channel contention in the presence of transmission conflict, we determine the worst-case end-to-end delay bounds of the flows. Moreover, we propose a probabilistic end-to-end delay bound that represents a safe upper bound with high probability.

We have implemented and experimented graph routing and fixed priority transmission scheduling on a wireless testbed of 69 nodes, where we have seen that our worst case bounds are at most 2.68 times that observed in the experiments we have performed. We have also performed trace-driven simulations on real network topologies. Both experimental results and simulations show that our delay bounds are safe in practice and the probabilistic delay bounds represent safe upper bounds with probability ≥ 0.90. The worst-case and probabilistic bounds can be used in different application scenarios depending on the level of predictability required. Our analysis hence can be used for effective schedulability test and admission control of real-time flows in WSANs under reliable graph routing.

Section 4.2 reviews related work. Section 4.3 describes the system model and graph routing mechanisms. Section 4.4 provides an overview of fixed priority scheduling for real-time flows in a WSAN under graph routing. Section 4.5 presents the delay analysis under reliable graph routing. Section 4.6 presents the probabilistic delay analysis. Section 4.7 presents the experimental results. Section 4.8 concludes the paper.

4.2 Related Work

Real-time scheduling for wireless networks has been explored in many early [170] and recent works [55,87,94,100,101,116,123,128,139,176]. However, these works do not focus on efficient worst-case delay analysis in the network. Other works [25, 60, 99, 139, 159] have researched delay analysis in wireless sensor networks. These works focus on data collection through a routing tree [60, 159] and/or do not consider multiple channels [25, 60, 99]. In contrast, we consider a WSAN based on multiple channels and reliable graph routing of WirelessHART. Besides, our analysis is targeted for real-time flows between sensors and actuators for process control purposes, and is not limited to data collection towards a sink.

Real-time scheduling for WSANs based on WirelessHART has received considerable attention in recent works [93,154–157,195]. The works presented in [93] and [195] address graph routing algorithm and localization, respectively, in WirelessHART networks. None of these concerns delay analysis. Our earlier work proposed delay analysis [155]. As a first step in establishing a delay analysis for WSANs, this earlier effort is based on single-route routing instead of reliable graph routing, which are important for reliable communication in process control applications. We have also studied priority assignment policies in [156] and rate selection algorithms in [154] for real-time flows. Our work in [157,179] also considered dynamic priority scheduling. However, none of our earlier work considers delay analysis under reliable graph routing.

This paper presents the first delay analysis for WSANs under reliable graph routing. Since industrial applications impose stringent requirements on *both* real-time performance and reliability in hash environments with frequent transmission failures, the delay analysis represents an important contribution to real-time scheduling for real-world WSANs. Efficient delay analysis is particularly useful for online admission control and adaptation (e.g., when network route, topology, or channel condition change) so that the network manager can quickly reassess the schedulability of the flows.

4.3 System Model

4.3.1 Network Model

Because of the world-wide adoption of WirelessHART in process monitoring and control, we consider a wireless sensor-actuator network (WSAN) model inspired by the WirelessHART standard [22]. This WSAN is a multi-hop mesh network consisting of a Gateway, a set of field devices, and several access points. A centralized network manager and the controllers are connected to the *Gateway*. The *network manager* is responsible for managing the entire network such as routing and transmission scheduling. The *field devices* are wirelessly networked sensors and actuators. *Access points* are wired to the Gateway to provide redundant paths between the wireless network and the Gateway. The sensor devices periodically deliver sample data to the controllers (through the access points wired to the Gateway), and the control messages are then delivered to the actuators. The network manager creates the routes and schedules of transmissions.

To achieve high reliability, WirelessHART employs a number of mechanisms to handle transmission failures. Transmissions are scheduled based on multi-channel TDMA (Time Division Multiple Access). Each time slot is of fixed length (10 ms), and each transmission needs one time slot. A transmission and its acknowledgement (ACK) are scheduled in the same slot using the same channel. For transmission between a receiver and its sender, a time slot can be either dedicated or shared for the link between the sender and the receiver, and the link is called a *dedicated link* or a *shared link*, respectively. In a time slot, when a link is used as a *dedicated link*, only one sender is allowed to transmit to the receiver. In a time slot, a *shared link* associated with a receiver indicates that multiple senders can attempt to send to the common receiver in that slot. The network uses the channels defined in IEEE 802.15.4, and adopts *channel hopping* in every time slot. Any excessively noisy channel is *blacklisted* not to be used. Each receiver uses a *distinct channel* for reception in any time slot. As a result, there are at most m successful transmissions in a time slot, where m is the total number of channels. This design decision prevents potential interference between concurrent transmissions in a dedicated slot and trades network throughput for a higher degree of predictability and reliability that is essential for industrial applications.

WirelessHART supports two types of routing approaches: source routing and graph routing. *Source routing* provides a single route for each flow. The delay analysis for source routing has been addressed in the literature [155]. The focus of this paper is to develop new delay analysis for graph routing that achieves a higher degree of reliability by providing multiple paths for each flow. In graph routing, a *routing graph* is a directed list of paths that connect two devices. Packets from all sensor nodes are routed to the Gateway through the *uplink graph*. For every actuator, there is a *downlink graph* from the Gateway through which the Gateway delivers control messages. The end-to-end communication between a source (sensor) and destination (actuator) pair happens in two phases. In the *sensing phase*, on one path from the source to the Gateway in the uplink graph, the scheduler allocates a dedicated slot for each device starting from the source, followed by allocating a second dedicated slot on the same path to handle a retransmission. The links on this path are dedicated links. Then, to offset failure of both transmissions along a primary link, the scheduler allocates a third shared slot on a separate path to handle another retry. The links on these paths are shared links. Then, in the *control phase*, using the same way, the dedicated links and shared links are scheduled in the downlink graph of the destination.

Each node is equipped with a *half-duplex* omnidirectional radio transceiver that cannot both transmit and receive at the same time and can receive from at most one sender at a time. Two transmissions along links $u \rightarrow v$ (u is the sender and v is the receiver) and $a \rightarrow b$ are called *conflicting* if $(u = a) \vee (u = b) \vee (v = a) \vee (v = b)$. Two conflicting transmissions cannot be scheduled in the same dedicated slot. However, two transmissions having the same receiver can be scheduled in a shared slot even though they are conflicting. Therefore, collision may occur in a shared slot (although with a low probability because a shared link is used only if the corresponding dedicated link fails in two dedicated slots). The transmitters employ a CCA (Clear Channel Assessment) check before transmitting in a shared slot to avoid/reduce conflict.

4.3.2 Flow Model

A periodic end-to-end communication between a source (sensor) and a destination (actuator) is called a *flow*. We consider there are n real-time flows denoted by $F_1, F_2, \cdots, F_n$ in the network. The *source* and the *destination* of flow F_i are denoted by s_i and d_i, respectively.

The subgraph of the uplink graph that s_i uses to deliver sensor data to the Gateway is denoted by UG_i. The downlink graph for d_i is denoted by DG_i. The graph consisting of UG_i and DG_i is the *routing graph* of F_i, and is denoted by G_i. The period and the deadline of flow F_i are denoted by T_i and D_i, respectively. Time slots are used as time units. We assume $D_i \leq T_i, \ \forall i$.

Each flow F_i, $1 \leq i \leq n$, has a fixed priority. Transmissions of a flow are scheduled based on the priority of the flow. In practice, flows may be prioritized based on deadlines, rates, or criticality. We assume that the priorities are already assigned using any algorithm, and that $F_1, F_2, \ \cdots, \ F_n$ are ordered by priorities. Flow F_h has higher priority than flow F_i if and only if $h < i$. All notations are summarized in Table 4.1.

4.4 Fixed Priority Scheduling

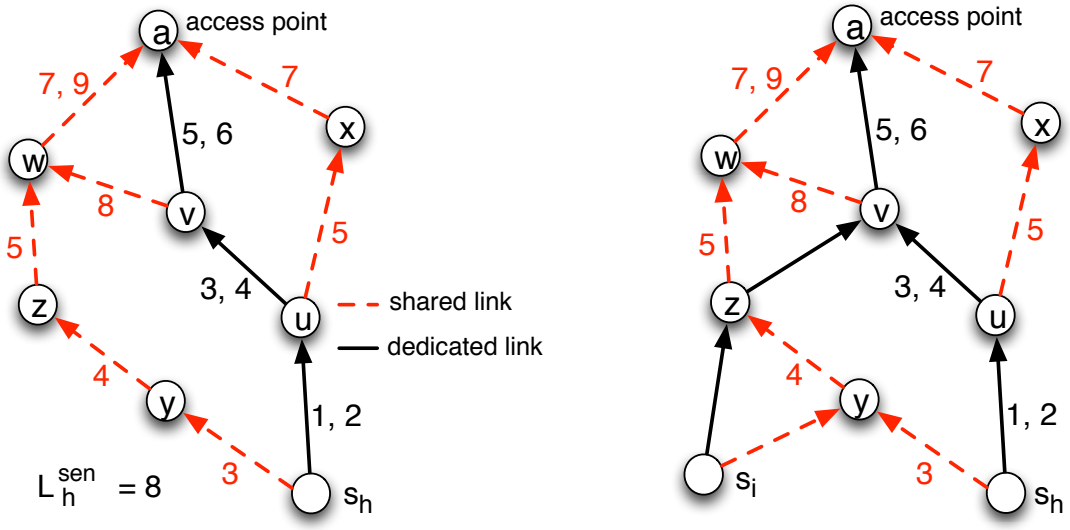
In this section, we provide an overview of the fixed priority transmission scheduling algorithm under reliable graph routing for which our delay analysis is developed. Due to its simplicity, fixed-priority scheduling is a commonly adopted policy in practice for real-time CPU scheduling, Control-Area Networks, and also for WirelessHART networks. In a *fixed priority scheduling policy*, each flow has a fixed priority, and its transmissions are scheduled based on this priority. The schedule is created by resolving the transmission conflicts and considering the limited number of channels. The complete schedule is split into superframes. A *superframe* is a series of time slots that repeat at a constant rate and represents the communication pattern of a set of flows.

We first describe how transmissions are scheduled using graph routing to account for failures. Figure 4.1(a) shows UG_h (the subgraph of the uplink graph used by F_h) for flow F_h. In the figure, the dedicated links used by F_h in the sensing phase are shown in solid lines while the dotted lines indicate the shared links used by F_h. Considering that F_h is not delayed by any other flow, the time slots in which a link is activated are shown beside the links (starting from slot 1). The first (starting from the source node s_h) dedicated link $s_h \rightarrow u$ is scheduled first at slot 1. Then to handle the transmission failure of slot 1, time slot 2 is also allocated for this link. Then, the next dedicated link $u \rightarrow v$ is allocated time slots 3 and 4. Similarly, the next dedicated link $v \rightarrow a$ is allocated time slots 5 and 6. Thus, if the first transmission (scheduled

m	total number of available channels in the network
n	total number of flows
F_i	a flow with priority i
s_i	source (sensor) of F_i
d_i	destination (actuator) of F_i
T_i	period of F_i
D_i	deadline of F_i
R_i	an upper bound of end-to-end delay of F_i
UG_i	subgraph of the uplink graph used by F_i
DG_i	downlink graph of F_i
G_i	routing graph of F_i (consists of UG_i and DG_i)
E_i	total number of dedicated links of flow F_i
S_i	total number of shared links of flow F_i
L_i^{sen}	worst-case time requirement of F_i in sensing phase
L_i^{con}	worst-case time requirement of F_i in control phase
L_i	worst-case time requirement of F_i, i.e., $L_i^{\mathrm{sen}} + L_i^{\mathrm{con}}$
$\Omega_i(x)$	channel contention delay suffered by F_i in an interval of x time slots
$\lambda_i^{h,\mathrm{sen}}$	maximum conflict delay caused by one instance of F_h along the bottleneck sensing path of F_i
$\lambda_i^{h,\mathrm{con}}$	maximum conflict delay caused by one instance of F_h along the bottleneck control path of F_i
δ_i^h	maximum conflict delay caused by one instance of F_h along the bottleneck link of F_i
Δ_i^h	maximum conflict delay that one instance of higher priority flow F_h can cause on F_i's bottleneck path
δ'^h_i	maximum conflict delay caused by one instance of F_h along the bottleneck link of F_i's dedicated path
Δ'^h_i	maximum conflict delay that one instance of higher priority flow F_h can cause on F_i's dedicated path

Table 4.1: Notations

on slot 5) along $v \to a$ succeeds (given at least one transmission along $s_h \to u$ and at least one transmission along $u \to v$ succeeded), then the packet will reach the access point a in 5 time slots. If the first transmission (scheduled on slot 5) along $v \to a$ fails but the second one (scheduled on slot 6) along that link succeeds, then the packet will reach the access point a in 6 time slots. For every link starting from the source, to handle failure of both transmissions along the link, the scheduler again allocates a third shared slot on a separate path to handle another retry. There can be situations when the second transmission on a dedicated link, say $s_h \to u$, succeeds but the ACK gets lost. As a result, s_h retransmits the packet along the shared link $s_h \to y$ on the third slot (as s_h is unaware of the successful transmission on



(a) UG_h for F_h (b) UG_i and UG_h

Figure 4.1: Routing in the sensing phase of F_i and F_h (the numbers beside each link indicate the time slots allocated to the link.)

the dedicated link) while the packet at u is transmitted through the subsequent links. Thus a packet can be duplicated and delivered through multiple routes. We call this problem *ACK-lost problem*. To handle ACK-lost problem, we avoid conflicts among the duplicated packets while scheduling on a routing graph, except the case that transmissions along the shared links having the same receiver are allowed to schedule in the same slot. Thus, the links on paths $s_h \to y \to z \to w \to a$ are scheduled on slots 3, 4, 5, and 7. Then the links on path $u \to x \to a$ are scheduled on slots 5 and 7. Then the links on path $v \to w \to a$ are scheduled on slots 8 and 9. Thus the packet can take at most 9 slots to reach the access point (along $s_h \to u \to v \to w \to a$).

Under fixed priority scheduling, the transmissions of the flows are scheduled in the following way. Starting from the highest priority flow F_1, the following procedure is repeated for every flow F_i in decreasing order of priority. For current priority flow F_i, the network manager schedules its dedicated links and shared links on UG_i in its sensing phase on earliest available time slots and on available channels. It then schedules the dedicated links and shared links on DG_i in the control phase following the same way. A time slot is *available* if no conflicting transmission is already scheduled in that slot except the case that transmissions along the

shared links having the same receiver are allowed to schedule in the same slot. Thus a packet is scheduled on multiple paths along the routing graph. When there is no ACK-lost problem, a packet is delivered through one path in the routing graph. In presence of ACK-lost problem, a packet can be duplicated and thus delivered through multiple paths.

Note that we do not propose any new algorithm for real-time transmission scheduling or any new fault tolerance mechanism for WirelessHART networks. Instead, the key contribution of our work is an efficient analysis for deriving the worst case delay bounds in a WSAN under graph routing, which is applicable for any existing fixed-priority scheduling policy for real-time flows in WSANs. The delay bound provided by our analysis is applicable only to the packets that are successfully delivered to the destination using existing graph routing mechanisms in WirelessHART.

4.5 Delay Analysis under Reliable Graph Routing

We first formulate the problem of worst-case delay analysis for real-time flows in a WSAN. We then present the delay analysis for any given fixed priority scheduling policy.

4.5.1 Problem Formulation

For each flow F_i, the sensor (s_i) periodically generates data at a period of T_i which has to be delivered to the Gateway (through an access point) in the sensing phase, and then the control message has to be delivered to the actuator (d_i) in the control phase. The total communication delay in two phases is called an *end-to-end delay* of F_i. The flows are called *schedulable* under a given fixed priority scheduling algorithm $\mathbb{A}$, if $\mathbb{A}$ is able to schedule the transmissions where no deadline will be missed. A schedulability test $\mathbb{S}$ is *sufficient* if any set of flows deemed schedulable by $\mathbb{S}$ is indeed schedulable. To determine the schedulability of a set of flows, it is sufficient to show that, for every flow, an upper bound of its worst case end-to-end delay is no greater than its deadline. Our objective is to determine an upper bound R_i of the end-to-end delay of each flow F_i. The end-to-end delay analysis will determine the flows to be *schedulable* if $R_i \leq D_i, \ \forall i$.

Note that creating a complete schedule for all flows requires an exponential time since the schedule has to be created up to the hyper-period of the flows. Compared to extensive testing and simulations of the entire schedule, analytical delay bounds are highly desirable in process monitoring and control applications that require real-time performance guarantees. An efficient end-to-end delay analysis can also be used for online admission control and to quickly adjust the workload in response to network dynamics. For example, when a channel is blacklisted or some routes are recalculated, the analysis can be used to promptly decide whether some flow has to be removed or some rate has to be updated to meet deadlines.

In transmission scheduling, a lower priority flow may be delayed by higher priority flows due to (a) *transmission conflicts* (when a transmission of the lower priority flow and a transmission of a higher priority flow involve a common node) and (b) *channel contention* (when all channels are assigned to the transmissions of higher priority flows in a time slot). For each kind of delay we first separately analyze how reliable graph routing in WSANs affect it. We then incorporate each component of the delays into one analysis that provides an upper bound of a flow's end-to-end delay taking into account the fault-tolerant mechanisms.

4.5.2 Transmission Conflict Delay under Graph Routing

First we analyze the delay that a flow can experience due to transmission conflicts only under graph routing. Whenever two transmissions conflict, the one that belongs to the lower priority flow needs to be delayed. The term 'delay' used in this subsection will refer to 'only transmission conflict delay'.

First we determine the conflict delay that one higher priority flow F_h may cause on a lower priority flow F_i. Under multi-path graph routing, a transmission of F_h along a link ℓ_h and a transmission of F_i along a link ℓ_i may be conflicting in 4 ways as follows when these two links involve a common node:

1. *Type 1:* ℓ_h is a dedicated link and ℓ_i is a shared or dedicated link.
2. *Type 2:* ℓ_h is a shared link and ℓ_i is a dedicated link.

3. *Type 3:* ℓ_h is a shared link and ℓ_i is a shared link, and the receiver nodes of the two links are different.

4. *Type 4:* ℓ_h is a shared link and ℓ_i is a shared link, and the receiver nodes of the two links are the same. In this case, the transmission of F_i is not delayed.

In the first 3 cases the transmission of F_i is delayed while for Type 4 conflict it will not be delayed. Therefore, the total delay caused by F_h on F_i depends on how their dedicated and shared links intersect in the routing graphs. Now we will first upperbound the conflict delay that one instance of a higher priority flow F_h may cause on F_i. To determine this, in the next discussion we limit our attention only to F_h and F_i.

In the routing graph G_i (consisting of UG_i and DG_i) of flow F_i which involves N_i nodes, there can be $O(N_i^2)$ directed end-to-end paths from its source s_i to destination d_i. If every node in the routing graph has to make the first two tries along a dedicated link and then to make a third retransmission along a shared link, then this number of paths can be 2^{N_i}. Among these end-to-end paths, the one that experiences the maximum conflict delay from F_h is called the *bottleneck path* with respect to F_h. The conflict delay caused by F_h along F_i's bottleneck path represents the upper bound of the conflict delay that F_h may cause on F_i. Let Δ_i^h be an *upper bound of conflict delay* that one instance of F_h may cause along the bottleneck path of F_i. We determine Δ_i^h in an efficient way *without requiring* to find the bottleneck path or without enumerating all end-to-end paths in G_i as described below.

Let us call the bottleneck path (with respect to F_h) in UG_i the *bottleneck sensing path* of F_i. Let an upper bound of conflict delay caused by F_h on F_i's bottleneck **sen**sing path be $\lambda_i^{h,\text{sen}}$. A value of $\lambda_i^{h,\text{sen}}$ can be efficiently calculated without enumerating all paths in UG_i as explained below. Let us consider a particular path p in UG_i. The total number of transmissions of (one instance of) F_h that may have Type 1, 2, or 3 conflict on p represents a value of conflict delay along p caused by one instance of F_h. To illustrate this, in Figure 4.1(b), with flow F_h, we also show UG_i for flow F_i. The figure shows links $s_i \rightarrow z$, $z \rightarrow v$, and $v \rightarrow a$ as dedicated links in UG_i while the corresponding shared links are $s_i \rightarrow y$, $z \rightarrow w$, and $v \rightarrow w$, respectively. In Figure 4.1(b), F_h has 9 transmissions that may cause delay along $p = s_i \rightarrow z \rightarrow w \rightarrow a$ of F_i. (Note that this is the delay along p considering links $s_i \rightarrow z$, $z \rightarrow v$, $z \rightarrow w$, and $w \rightarrow a$ of F_i. Link $z \rightarrow v$ is considered because $z \rightarrow w$ is scheduled only after scheduling $z \rightarrow v$.) Now the path in UG_i whose delay (calculated using the above

Algorithm 3: Finding conflict delay on F_i caused by F_h

```
Procedure FindConflict(UG_i, r)                          /* r is a node in UG_i */
  for each node u in UG_i do
    | status(u):=undiscovered; λ_i^h(u) := 0;
  end
  DFSearch(r);                                           /* start search at node r */
  return λ_i^h(r);                                       /* λ_i^h in subtree rooted at r */
end Procedure

Procedure DFSearch(r)
  status(r):=discovered;                                 /* node r is now discovered */
  for each v ∈ children(r, UG_i) do
    | if status(v)=undiscovered; then DFSearch(v);
  end
  λ_i^h(r) := max{λ_i^h(v)|v ∈ children(r, UG_i)};
  x(r) := new conflict delay on F_i by F_h observed at node r;
  λ_i^h(r) := λ_i^h(r) + x(r);
end Procedure
```

method) is maximum is the bottleneck sensing path, and its delay represents $\lambda_i^{h,\text{sen}}$. Such a value of $\lambda_i^{h,\text{sen}}$ is determined quickly by exploring each link on UG_i once based on a depth-first search on UG_i. The method is shown as Algorithm 3, and $\lambda_i^{h,\text{sen}}$ is determined by calling

$$\lambda_i^{h,\text{sen}} = \text{FindConflict}(\text{UG}_i,\ s_i);$$

In Algorithm 3, we use $children(u, UG_i)$ to denote the set of nodes to which node u transmits in UG_i for flow F_i. (For example, in Figure 4.1(a), node s_h has children u and y.) The search starts at node s_i. In this method, when the search backtracks at a node u, we use $\lambda_i^h(u)$ to denote the maximum conflict delay along a path among all the paths in the subtree (induced by depth first search) rooted at u. The value of $\lambda_i^h(u)$ is calculated by taking the maximum of the values from u's children and then by adding the new conflict delay that we observe at node u, when the search finishes node u. Note that we do not need to execute Algorithm 3 for every distinct F_h to determine $\lambda_i^{h,\text{sen}}$. Instead, we need to execute Algorithm 3 only once for all $h < i$ to determine $\lambda_i^{h,\text{sen}}$ for flow F_i, making our approach highly efficient.

Similarly, let $\lambda_i^{h,\text{con}}$ be the conflict delay along the bottleneck **con**trol path. The value of $\lambda_i^{h,\text{con}}$ is also determined using Algorithm 3 on DG_i and starting the search at an access point a, i.e., by calling

$$\lambda_i^{h,\text{con}} = \text{FindConflict}(\text{DG}_i,\ a);$$

Based on these values, Lemma 7 provides a bound of Δ_i^h.

Lemma 7. *For a higher priority flow F_h and a lower priority flow F_i, $\Delta_i^h \leq \lambda_i^{h,sen} + \lambda_i^{h,con}$.*

Proof. Since the control phase of F_i starts after its sensing phase is complete, the bottleneck path between s_i and d_i consists of its bottleneck sensing path and the bottleneck control path. Hence, $\lambda_i^{h,\text{sen}} + \lambda_i^{h,\text{con}}$ is an upper bound of conflict delay caused by one instance of F_h along F_i's bottleneck path. □

Note that Δ_i^h is an upper bound of delay that one instance of F_h can cause along F_i's bottleneck path. Now we will upperbound the total delay caused by all instances of F_h. In considering the delay caused by multiple instances, we observe that at the time when a transmission on a directed path p in G_i conflicts with some transmission of F_h, the preceding transmissions on p are already scheduled. These already scheduled transmissions on p are no more subject to delay by the subsequent instances of F_h. For example, in Figure 4.1(b) let us consider the path $s_i \to y \to z \to v \to w \to a$ in UG_i of F_i. If some instance of F_h conflicts and causes delay on F_i's transmission along $v \to w$, the next instance of F_h must not delay F_i's transmissions along links $s_i \to y$, $y \to z$, and $z \to v$ on this path since these are already scheduled. Thus only the transmissions that are not yet scheduled along path p will be considered for conflict delay by the subsequent instances of F_h. These observations lead to Lemma 8, and then to Theorem 9 to upperbound the total delay (due to transmission conflict) caused on F_i by all instances of F_h.

Lemma 8. *Let us consider any two instances of a higher priority flow F_h such that each causes conflict delay on a directed path p in G_i of a lower priority flow F_i in a time interval. Then, there is at most one common transmission on p that can be delayed by both instances.*

Proof. Let these two instances of F_h be denoted by $F_{h,1}$ and $F_{h,2}$, where $F_{h,1}$ is released before $F_{h,2}$. Suppose to the contrary, both of these instances cause delay on two transmissions, say τ_j and τ_r, on directed path p of F_i. Without loss of generality, we assume that τ_j precedes τ_r on p. $F_{h,1}$ causes delay on τ_r because τ_r is ready to be scheduled. This implies that τ_j has already been scheduled. Hence, $F_{h,2}$ which releases after $F_{h,1}$ cannot cause any delay on τ_j, thereby contradicting our assumption. □

By Lemma 8, for any two instances of F_h, any directed path in G_i of F_i has at most one transmission on which both instances can cause delay. Let the link on G_i that may have maximum conflict delay of Type 1, 2, or 3 with F_h be called the *bottleneck link* of F_i (with respect to F_h). That is, a transmission of F_i along this link may face the highest conflict with F_h. Let δ_i^h denote the *maximum conflict delay* along the bottleneck link. (For example, considering only UG_i in Figure 4.1(b), we can see that $\delta_i^h = 7$, since a link of F_i can have conflict with at most 7 transmissions of F_h. Here, $z \to v$ is F_i's bottleneck link.) In the worst case, the transmission along the bottleneck link of F_i (with respect to F_h) can be delayed by multiple instances of F_h. Hence, the value of δ_i^h plays a major role in determining the worst case delay caused by F_h on F_i as shown in Theorem 9.

Theorem 9. *In a time interval of t slots, the worst case conflict delay caused by a higher priority flow F_h on a lower priority flow F_i is upper bounded by*

$$\Delta_i^h + \Big(\Big\lfloor \frac{t}{T_h} \Big\rfloor - 1 \Big).\delta_i^h + \min \Big(\delta_i^h, t \bmod T_h \Big)$$

Proof. There are at most $\lceil \frac{t}{T_h} \rceil$ instances of F_h in a time interval of t slots. We consider a particular directed path p in G_i of F_i. Let the set of transmissions of F_h which cause conflict delay along p be denoted by Γ. When one instance $F_{h,1}$ of F_h causes conflict delay on p, a subset Γ_1 of Γ causes delay on p. Now consider a second instance $F_{h,2}$ of F_h. For $F_{h,2}$, another subset Γ_2 of Γ causes delay on p. When all subsets $\Gamma_1, \Gamma_2, \cdots, \Gamma_{\lceil \frac{t}{T_h} \rceil}$ are mutually disjoint, by the definition of Δ_i^h, the conflict delay caused by Γ on p is at most Δ_i^h. Hence, the total conflict delay caused by all $\Gamma_1, \Gamma_2, \cdots, \Gamma_{\lceil \frac{t}{T_h} \rceil}$ in this case is at most Δ_i^h. That is, the total conflict delay on p caused by F_h is at most Δ_i^h.

Now let us consider the case when the subsets $\Gamma_1, \Gamma_2, \cdots, \Gamma_{\lceil \frac{t}{T_h} \rceil}$ are not mutually disjoint, i.e., there is at least one pair Γ_j, Γ_k such that $\Gamma_j \cap \Gamma_k \neq \emptyset$, where $1 \leq j, k \leq \lceil \frac{t}{T_h} \rceil$. Let the total delay caused by all instances of F_h on p in such case be $\Delta_i^h + Z_i^h$, i.e., the delay is higher than Δ_i^h by Z_i^h time slots. The additional delay (beyond Δ_i^h) happens because the transmissions that are common between Γ_i and Γ_j cause both instances of F_h to create delay along p. By Lemma 8, for any two instances of F_h, p has at most one transmission on which both instances can cause delay. If there is no transmission of p that is delayed by both the k-th instance and the $(k+1)$-th instance of F_h, then no transmission of p is delayed by both the k-th instance and the q-th instance of F_h, for any $q > (k+1)$,

where $1 \leq k < \lceil \frac{t}{T_h} \rceil$. Thus, Z_i^h is maximum when for each pair of consecutive instances (say, the k-th instance and $k+1$-th instance, for each k, $1 \leq k < \lceil \frac{t}{T_h} \rceil$) of F_h, there is a transmission of p that is delayed by both instances. Hence, at most $\lceil \frac{t}{T_h} \rceil - 1$ instances contribute to this additional delay Z_i^h, each instance causing some additional delay on a transmission. Since one instance of F_h can cause delay on a transmission of p at most by δ_i^h slots, $Z_i^h \leq (\lceil \frac{t}{T_h} \rceil - 1)\delta_i^h$. Since the last instance may finish after the considered time window of t slots, the delay caused by it is at most $\min(\delta_i^h, t \bmod T_h)$ slots. Taking this into consideration, $Z_i^h \leq (\lfloor \frac{t}{T_h} \rfloor - 1)\delta_i^h + \min(\delta_i^h, t \bmod T_h)$. Thus, the total delay caused on p by all instances of F_h is at most

$$\Delta_i^h + Z_i^h \leq \Delta_i^h + (\left\lfloor \frac{t}{T_h} \right\rfloor - 1).\delta_i^h + \min(\delta_i^h, t \bmod T_h)$$

Since the above bound is true for any path in G_i (of F_i), it is true for the bottleneck path in G_i. Since the conflict delay along the bottleneck path represents the conflict delay caused on F_i by F_h, the theorem follows. □

From Theorem 9, now an upper bound of the total delay that flow F_i can experience from all higher priority flows due to transmission conflicts during a time interval of t slots is calculated as follows.

$$\sum_{h<i} \Big(\Delta_i^h + \Big(\left\lfloor \frac{t}{T_h} \right\rfloor - 1 \Big).\delta_i^h + \min \Big(\delta_i^h, t \bmod T_h \Big) \Big) \tag{4.1}$$

4.5.3 Channel Contention Delay under Graph Routing

In this section, we analyze the channel contention delay caused by one higher priority flow F_h to a lower priority one F_i under reliable graph routing. First we analyze the delay without considering channel hopping. Later, we will analyze the effect of channel hopping.

Let E_h and S_h denote the total number of dedicated links and total number of shared links of flow F_h, respectively. Since every dedicated link is scheduled on 2 dedicated slots, there are $2E_h + S_h$ assignments of channels for flow F_h.

Note that a packet is scheduled on multiple paths in its routing graph for fault tolerance. While a natural approach to analyzing channel contention delay of a flow under this scenario is to consider it as a parallel task, we observe that the scheduling on routing graphs experiences only a little parallelism making it more closer to sequential task scheduling due to the following two problems:

- *ACK-lost problem:* Assuming no packet duplication, we could schedule the link $w \to a$ for delivery through paths $s_h \to y \to z \to w \to a$ on slot 6, ignoring the fact that link $v \to a$ is already scheduled on slot 6 because the packet will be delivered through one path only (Figure 4.1(a)). But, in presence of ACK-lost problem, to avoid conflict among the duplicate packets (of the same packet), we cannot schedule link $w \to a$ on slot 6. Thus the two links $v \to a$ and $w \to a$ are scheduled sequentially, on slot 6 and slot 7, respectively.

- *Impact of transmission conflict on channel contention delay:* The second reason is that channel contention delay and transmission conflict delay are often correlated. Specifically, channel contention delay can increase when a flow experiences transmission conflict delay. Let us consider links $z \to w$ and $u \to x$ (in G_h) that can be scheduled on slot 5 when there are no other higher priority flow (Figure 4.1). In the presence of higher priority flows, if any of transmissions $z \to w$ and $u \to x$ in F_h is delayed, for example by 1 slot, due to transmission conflict with a higher priority flow, while the other can happen at slot 5, then these two transmissions have to be scheduled sequentially (instead of scheduling in parallel). Therefore, even though scheduling of F_h has some parallelism, in the worst case in presence of transmission conflict, it can cause channel contention delay on its lower priority flows like a flow that happens like a sequential task with execution requirements of $2E_h + S_h$ slots.

Based on the above observations, the analysis for upper bounding the channel contention delay reduces to that for a set of flows where each flow F_i has the worst-case time requirement of e_i slots through a single path route, where

$$e_i = 2E_i + S_i$$

Hence, we leverage our result in [155] whose analysis was given for flows with single-path routes to find the channel contention delay caused by F_h on F_i. Using that result, in any time interval of x slots, there are at most $m-1$ higher priority flows each flow F_h among which can cause at most $I_i^h(x)$ delay on F_i as expressed below

$$I_i^h(x, e_i) = \min\Bigg(x - e_i + 1,\ \left\lfloor \frac{x - e_h}{T_h} \right\rfloor e_h + e_h + \min\Big(e_h - 1, \max\Big((x - e_h) \bmod T_h - (T_h - R_h), 0\Big)\Big)\Bigg)$$

where R_h is the worst-case end-to-end delay of F_h. Each other higher priority flow F_h can cause at most $J_i^h(x, e_i)$ delay on F_i, where

$$J_i^h(x, e_i) = \min\Big(x - e_i + 1,\ \left\lfloor \frac{x}{T_h} \right\rfloor e_h + \min\big(x \bmod T_h,\ e_h\big)\Big)$$

Thus, considering a total of m channels, an upper bound $\Omega_i(x)$ of the channel contention delay caused by all higher priority flows on F_i in any time interval of x slots is derived as follows.

$$\Omega_i(x, e_i) = \left\lfloor \frac{1}{m}\Big(Z_i(x, e_i) + \sum_{h<i} J_i^h(x, e_i)\Big) \right\rfloor \tag{4.2}$$

with $Z_i(x, e_i)$ being the sum of the $\min(i-1, m-1)$ largest values of the differences $I_i^h(x, e_i) - J_i^h(x, e_i)$ among the higher priority flows $F_h,\ h < i$.

Effect of Channel Hopping. To every transmission, the scheduler assigns a *channel offset* between 0 and $m-1$ instead of an actual channel, where m is the total number of channels. All devices in the network maintain an identical list of available channels. At any time slot t, a channel offset c (i.e., $1, 2, \cdots, m-1$) maps to a channel that is different from the channel used in slot $t-1$ as follows.

$$\text{channel} = (c + t) \bmod m \tag{4.3}$$

Both the sender and the receiver of the corresponding transmission link switches to the new channel. As can be seen from Equation 4.3, at every time slot any 2 different channel offsets

always map to 2 different channels. The scheduler assigns at most one channel offset to a link at any time which maps to different physical channels in different time slots, keeping the total number of available channels at m always, and scheduling each link on at most one channel at any time. Hence, channel hopping does not have effect on channel contention delay.

4.5.4 End-to-End Delay Bound

Now both types of delays are incorporated together to develop an upper bound of the end-to-end delay of every flow. This is done for every flow in decreasing order of priority starting with the highest priority flow. Theorem 10 provides an upper bound R_i of end-to-end delay for every flow F_i.

Considering no delay from higher priority flows, let the *worst-case time requirement* of F_h in the **sen**sing phase be denoted by L_h^{sen}. For example, in Figure 4.1(a), $L_h^{\text{sen}} = 9$ slots (as described in Section 4.4). A similar scheduling is followed in the control phase also. Similarly, considering no delay from higher priority flows, let the *worst-case time requirement* of F_h in the **con**trol phase be denoted by L_h^{con}. Thus, considering no delay from higher priority flows, the *time requirement through a critical path* denoted by L_i, of flow F_i is

$$L_i = L_i^{\text{sen}} + L_i^{\text{con}} \tag{4.4}$$

Theorem 10. *Let x_i^* be the minimum value of $x \geq L_i$ that solves Equation 4.5 using a fixed-point algorithm.*

$$x = \Omega_i(x, e_i) + L_i \tag{4.5}$$

Then the end-to-end delay bound R_i of flow F_i is the minimum value of $t \geq x_i^*$ that solves Equation 4.6 using a fixed-point algorithm.

$$t = x_i^* + \sum_{h<i} \left(\Delta_i^h + \left(\left\lfloor \frac{t}{T_h} \right\rfloor - 1 \right) . \delta_i^h + \min\left(\delta_i^h, t \bmod T_h \right) \right) \tag{4.6}$$

Proof. According to Equation 4.2, x_i^* is calculated considering R_h (i.e., the end-to-end delay bound of F_h considering both channel contention delay and conflict delay) of each higher

priority flow F_h. According to Equation 4.2, $\Omega_i(x, L_i)$ is the channel contention delay caused by all higher priority flows on F_i in any time interval of x slots. Hence x_i^* is the bound of the end-to-end delay of F_i when it suffers only from channel contention delay caused by higher priority flows (and no conflict delay). Equation 4.1 provides the bound of transmission conflict delay of F_i. Hence, adding this value to x_i^* must be an upper bound of F_i's end-to-end delay under both channel contention and transmission conflict. □

Thus we can determine R_i for every flow F_i in decreasing order of priority starting with the highest priority flow using Theorem 10. In solving Equations 4.5 and 4.6, if x or t exceeds D_i, then F_i is decided to be "unschedulable". Thus, the worst-case time required to determine R_i can be $O(D_i)$ which implies a pseudo polynomial time complexity.

4.6 A Probabilistic End-to-End Delay Analysis

Graph routing provides a very conservative approach to scheduling transmissions in a WirelessHART network. In the scheduling used in the previous sections, there is a synchronization at the access points in the sense that the scheduling in the downlink graph of a flow (the control phase) is started after all links in its uplink subgraph are scheduled. However, there is high probability that a packet will be delivered through the dedicated path only because each link on the path is dedicated and scheduled twice. Therefore, whenever the gateway receives a sensor packet through the dedicated link, the corresponding control message can be calculated and delivered through the downlink graph's dedicated link in the next available slot avoiding synchronization at the access points. The corresponding retry on the shared slot can be scheduled only after all links on the uplink subgraph of the flow are scheduled. The advantage of such a scheduling policy is that the actual end-to-end delay in most cases will be substantially shorter since a packet follows the dedicated links in most cases. Under this scheduling, we can determine a probabilistic delay bound that is tighter than the bound derived in the last section but represents a bound with high probability.

Considering the dedicated route has E_i links, and p_k as the probability of a successful transmission along link k, the probability of being successful upon 2 transmissions through link k is $1 - (1 - p_k)^2$. Therefore, the probability that a packet will be delivered through the

dedicated links is

$$\prod_{k=1}^{E_i}\Big(1-(1-p_k)^2\Big) \tag{4.7}$$

Let, in G_i, the path consisting of all dedicated links be called *dedicated path*. Let ${\Delta'}_i^h$ denote the total number of transmissions of (one instance of) F_h that share a node on the dedicated path of F_i. Similarly, let ${\delta'}_i^h$ denote the maximum conflict delay caused by one instance of F_h on the bottleneck link on F_i's dedicated path (i.e., a link on F_i's dedicated path can share a node with at most ${\delta'}_i^h$ transmissions of F_h). Corollary 1 now follows from Theorem 10.

Corollary 1. *Let x_i^* be the minimum value of $x \geq 2E_i$ that solves Equation 4.8 using a fixed-point algorithm.*

$$x = \Omega_i(x, 2E_i) + 2E_i \tag{4.8}$$

Then the minimum value of $t \geq x_i^*$ that solves Equation 4.9 using a fixed-point algorithm is the worst-case end-to-end delay bound of flow F_i with a probability of at least $\prod_{k=1}^{E_i}\Big(1-(1-p_k)^2\Big)$.

$$t = x_i^* + \sum_{h<i}\bigg({\Delta'}_i^h + \Big(\Big\lfloor\frac{t}{T_h}\Big\rfloor - 1\Big).{\delta'}_i^h + \min\Big({\delta'}_i^h, t \bmod T_h\Big)\bigg) \tag{4.9}$$

Proof. By Equation 4.2, $\Omega_i(x, 2E_i)$ represents the channel contention delay on the dedicated path of F_i. Following Theorem 10, the minimum value of $t \geq x_i^*$ that solves Equation 4.9 is the worst case delay bound for the dedicated route. The proof follows since a packet has $\prod_{k=1}^{E_i}\Big(1-(1-p_k)^2\Big)$ probability of being delivered through the dedicated route. □

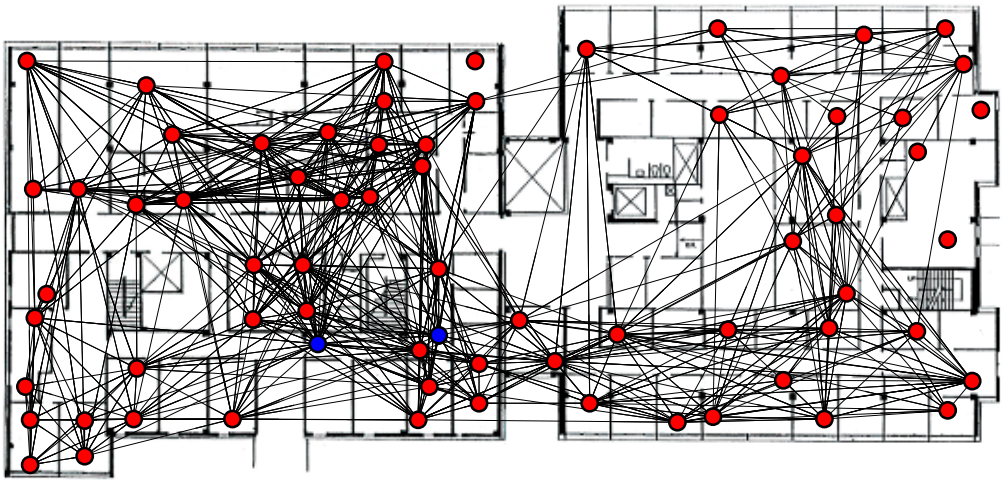
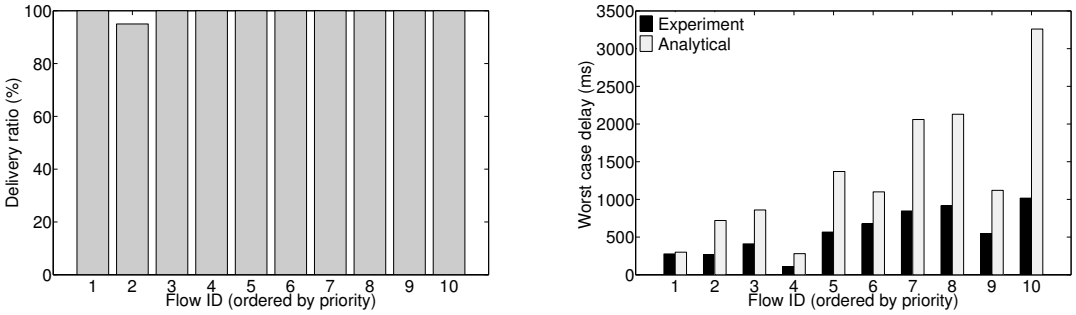


Figure 4.2: Testbed topology (access points are colored in blue)



(a) Delivery ratio

(b) Delay

Figure 4.3: Delay and reliability on testbed

4.7 Experiment

4.7.1 Testbed Experiment

Implementation

We evaluate our delay analysis on an indoor wireless testbed deployed in two buildings at Washington University [2]. The testbed consists of 69 TelosB motes, each equipped with Chipcon CC2420 radios compliant with the IEEE 802.15.4 standard. Note that the physical layer of WirelessHART is also based on IEEE 802.15.4. We implement a network protocol stack on the testbed which consists of a multi-channel TDMA MAC protocol with channel hopping based on Equation 4.3 and a routing protocol. The uplink and downlink graphs are generated using the graph routing algorithms presented in [93]. Time is divided into 10 ms slots and clocks are synchronized across the entire network using the Flooding Time Synchronization Protocol (FTSP) [131].

Experimental Setup

To avoid channels occupied by the campus Wi-Fi, we use IEEE 802.15.4 channels 11 to 14 in our experiments. For each link in the testbed, we measured its packet reception ratio (PRR) by counting the number of received packets among 250 packets transmitted on the link. Following the practice of industrial deployment, we only consider links with PRR higher than 90% on every channel to determine the testbed topology. Figure 4.2 is a topology of the testbed showing the node positions on the two buildings' floor plan. We use two nodes (colored in blue in the figure) as access points, which are physically connected to a root server (Gateway). The Network Manager runs on this root server. The rest of motes work as field devices.

We experiment by generating 10 flows on our testbed. The period of each flow is picked up from the range of $[10 * 2^8, 10 * 2^{11}]$ms. The relative deadline of each flow equals to its period. All flows are schedulable based on our delay analyses. Priorities of the flows are assigned based on Deadline Monotonic (DM) policy, a widely used scheduling policy in CPU

scheduling and in control area networks. DM assigns priorities to flows according to their relative deadlines; the flow with the shortest deadline being assigned the highest priority.

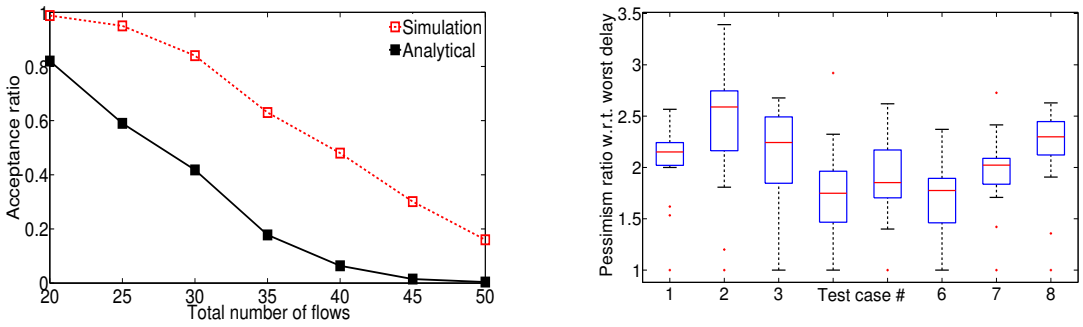
Results

We run our experiments long enough so that each superframe is run for at least 20 cycles. Based on our experimental results, we evaluate our proposed approaches in terms of reliability and delay. We use delivery ratio to measure reliability. The delivery ratio of a flow is defined as percentage of packets that are successfully delivered to destination. Then, we compare the worst case end-to-end delay observed in experiments with our analytical delay bounds.

Figure 4.3 shows our results. Figure 4.3(a) shows the delivery ratios of all 10 flows. As the figure shows, one flow has a delivery ratio of 95% while all other flows have 100% delivery ratio. This is reasonable as graph routing is designed for such a high degree of reliability through route, channel, and time diversity. This high delivery ratio demonstrates the effectiveness of graph routing. Figure 4.3(b) plots the maximum end-to-end delay observed in our experiments and the end-to-end delay bounds derived through our delay analysis. As the figure shows, our analytical delay bounds are no less than the experimental worst case delays, demonstrating that our delay analysis provides safe upper bounds of the actual delays. For this particular experiments, the bounds are at most 2.68 times that observed in experiments. Note that our analytical delays are the worst case delays while the longest delays observed in the experiments are not the worst-case delays as our testbed is not deployed in industrial environments. Hence, this ratio of 2.68 is expected to be smaller should the experiments be performed in an industrial environment.

4.7.2 Simulation

For more extensive evaluation, we now use the same testbed topology and evaluate the results in simulations. We generate flows by randomly selecting sources and destinations, and simulate their schedules in these topologies. Two nodes in the topology are selected as access points. The uplink and downlink graphs are generated using the same graph routing algorithms as the one we used in testbed experiment. The periods of the flows are considered



(a) Acceptance ratio

(b) Pessimism ratios

Figure 4.4: Worst case delay analysis performance in simulation

harmonic and are randomly generated in the range $[10 * 2^5, 10 * 2^{13}]$ms. In default settings, the deadlines are considered equal to periods. Later we decrease the deadlines. Priorities of the flows are assigned based on DM policy. In all cases, we use 12 channels for scheduling. We evaluate our analysis in terms of the following metrics. (a) *Acceptance ratio* defined as the proportion of the number of test cases deemed to be schedulable to the total number of test cases. (b) *Pessimism ratio* defined, for a flow, as the proportion of the analyzed theoretical upper bound to its maximum end-to-end delay observed in simulations.

Performance Analysis of the Worst Case Delay Analysis

Since there exists no prior work on delay analysis under reliable graph routing of WirelessHART, we analyze the effectiveness of our analysis by simulating the complete schedule of transmissions of all flows released within the hyper-period. In the figure, "Simulation" indicates the fraction of test cases that have no deadline misses in the simulations, and "Analytical" indicates the acceptance ratio of our delay analysis.

Figure 4.4(a) shows the acceptance ratios for 1000 test cases under varying number of flows. As the figure shows, for 20 flows, 986 test cases among 1000 are schedulable through simulations while our analysis has determined 818 cases as schedulable as its acceptance ratio is 0.818. For 30 flows 862 cases are schedulable through simulations among which 419 cases are deemed schedulable by our analysis, which is almost 50% of the total schedulable cases.

The acceptance ratios decrease sharply with the increase in the number of flows as the network becomes congested. However, in most cases, at least 50% of all schedulable cases are accepted.

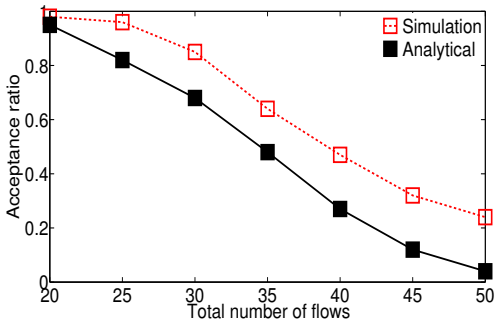
Figure 4.4(b) plots the pessimism ratios of the flows under our analysis for randomly selected 8 test cases each consisting of 30 flows. The figure plots pessimism ratios w.r.t. the worst case possible delay (i.e. the ratio of analytical delay to worst case possible delay in simulation). It indicates that the 75th percentile of the pessimism ratios is less than 2.5 in all but one test case where it is below 2.7 and the median is below 2.6. These results indicate that our delay bounds are not overly pessimistic for the particular cases we have tested.

In every setup, we have observed that the acceptance ratios of our analysis are close to those of simulation. In addition, all test cases accepted by our analysis meet their deadlines in the simulations which demonstrates that the bounds are safe. Our analysis hence can be used as an effective schedulability test for real-time flows under reliable graph routing.

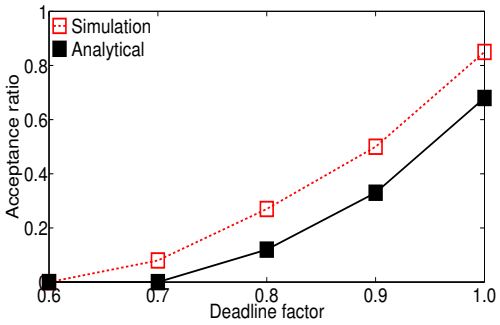
Performance Analysis of the Probabilistic Delay Analysis

Now we analyze the performance of our probabilistic analysis (Section 4.6). We show the acceptance ratios under the probabilistic delay bound in Figure 4.5. Figure 4.5(a) shows that the acceptance ratio is always close to that of simulation under various number of flows. For 30 flows, Figure 4.5(b) shows acceptance ratios under varying deadlines, where a flow's deadline is varied as *period* $*$*(deadline factor)*. Restoring the base deadlines (equal to period), Figure 4.5(c) shows acceptance rates under varying sampling rates, where a flow's rate is varied as $(old\ rate) * (rate\ factor)$. The results show that probabilistic delay bounds allow to accept more test cases, since these bounds are tighter than the worst case bounds.

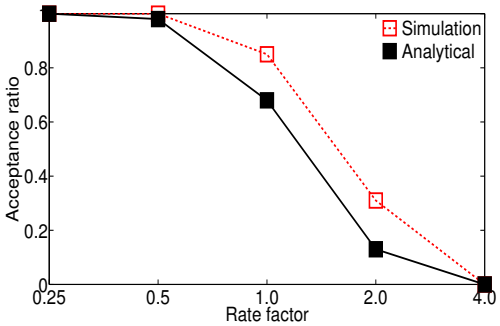
Figure 4.6 plots the distribution of pessimism ratios for 30-flows in 8 test cases by restoring the original rates of the flows. Compared to the ratios under the worst-case delay analysis (of Figure 4.4), these ratios are higher because here pessimism ratios are the ratio of analytical delay bound to the actual observed delay in simulation. Since most of the times a packet is delivered through dedicated routes, the actual delay is lot shorter than the worst case possible delay.



(a) Under varying number of flows (n)



(b) Under varying deadlines when $n = 30$



(c) Under varying rates when $n = 30$

Figure 4.5: Acceptance rate under probabilistic delay bound

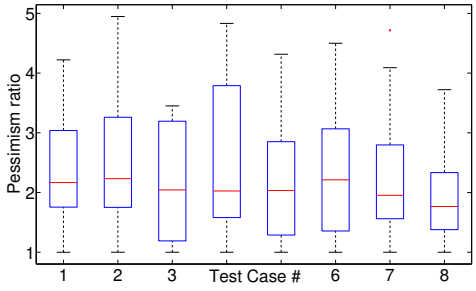


Figure 4.6: Pessimism ratio for 30 flows under probabilistic bound

Based on PRR, for each flow, the probability of delivering through the dedicated route was at least 0.90 (based on Equation 4.7). Therefore, the results suggest that the probabilistic delay bounds represent safe upper bounds with probability ≥ 0.90. Since graph routing in a WSAN is a conservative way of scheduling transmissions to ensure reliability at the cost of a high degree of redundancy, the worst case delay bound can be naturally pessimistic in many set ups. Hence, the probabilistic delay bound can be used as an alternative with the bounds being upper bound with high probability for soft real-time flows for which probabilistic bounds are sufficient.

4.8 Summary

Industrial wireless sensor-actuator networks must support reliable and real-time communication in hash environments. Industrial wireless standards such as WirelessHART adopt a reliable graph routing approach to handle transmission failures through retransmissions and route diversity. These mechanisms introduce substantial challenges in analyzing the schedulability of real-time flows. We have presented the first worst-case delay analysis under reliable graph routing. We have also proposed a probabilistic delay analysis that provides delay bounds with high probability. Experiments based on a wireless testbed of 69 nodes and simulations show that our analytical delay bounds are safe, and can be used as an effective schedulabity test for real-time flows under reliable graph routing.

Chapter 5

Real-Time Wireless: Priority Assignment for Fixed Priority Scheduling

WirelessHART is a new wireless sensor-actuator network standard specifically developed for process industries. A key challenge faced by WirelessHART networks is to meet the stringent real-time communication requirements imposed by process monitoring and control applications. Fixed-priority scheduling, a popular scheduling policy for real-time networks, has recently been shown to be an effective real-time transmission scheduling policy in WirelessHART networks. Priority assignment has a major impact on the schedulability of real-time flows in these networks. This paper investigates the open problem of priority assignment for periodic real-time flows in a WirelessHART network. We first propose an optimal priority assignment algorithm based on local search for any given worst case delay analysis. We then propose an efficient heuristic search algorithm for priority assignment. We also identify special cases where the heuristic search is optimal. Simulations based on random networks and the real topology of a physical sensor network testbed showed that the heuristic search algorithm achieved near optimal performance in terms of schedulability, while significantly outperforming traditional priority assignment policies for real-time systems.

5.1 Introduction

Wireless Sensor-Actuator Networks (WSANs) represent a new generation of communication technology for industrial process control. A feedback control system in process industries (e.g., oil refineries) is implemented in a WSAN for process monitoring and control applications. Networked control loops impose stringent reliability and real-time requirements for communication of sensors and actuators [166]. To meet these requirements in harsh industrial environments, WirelessHART has been developed as an open WSAN standard with unique features such as centralized network management and scheduling, multi-channel TDMA, redundant routes, and channel hopping [22, 56]. With the adoption of WirelessHART, recent years have seen successful real-world deployment of WSANs for process monitoring and control [56]. As they continue to evolve in process industries, real-time transmission scheduling issues are becoming increasingly important for WirelessHART networks.

In this paper, we consider a WirelessHART network that supports feedback control loops through periodic real-time data flows from sensors to controllers and then to actuators. We focus on priority assignment for real-time flows whose transmissions are scheduled based on a fixed priority policy. Due to its simplicity and efficiency, fixed priority scheduling is a commonly adopted real-time scheduling policy in CPU scheduling and traditional real-time networks (e.g., Control-Area Networks). Recent study has shown that fixed priority scheduling is an effective policy for real-time flows in WirelessHART networks and developed worst case delay analysis to be used for efficient schedulability test [155]. Priority assignment has a significant impact on the schedulability of real-time flows. However, optimal priority assignment for WirelessHART networks is a challenging and open problem that has not been addressed in the literature. An ideal priority assignment should not only enable real-time flows to meet their deadlines, but also work synergistically with real-time schedulability tests to support effective network capacity planning and efficient online admission control and adaptation.

For a given schedulability test, a priority assignment algorithm is *optimal* if it can find a priority ordering under which a set of flows is deemed schedulable by the test whenever there exists any such priority ordering. Since an optimal priority assignment is NP-hard for all but a few special cases, simple heuristics such as Deadline Monotonic and Rate Monotonic policies are commonly adopted in practice in real-time networks [122]. However, as shown

in our simulation results presented in this paper, the effectiveness of these heuristics in WirelesHART networks is far from the optimal.

This paper is the first to address the optimal priority assignment problem for real-time flows in WirelessHART networks. Specifically, our key contributions are four-fold: (1) We design a local search algorithm for priority assignment that is optimal for any given schedulability test based on worst case delay analysis; (2) We propose an efficient heuristic search algorithm for priority assignment; (3) We identify special cases where the heuristic search is optimal; (4) We present simulation results based on both random networks and the real topology of a physical sensor network testbed. Our results showed that the heuristic search algorithm achieved near-optimal performance in terms of schedulability, while significantly outperforming traditional real-time priority assignment policies.

In the rest of this paper, Section 5.2 describes the WirelessHART network model. Section 5.3 defines the priority assignment problem. Section 5.4 reviews existing schedulability tests and identifies the key insights underlying the priority assignment approach. Sections 5.5 and 5.6 present the optimal local search and the heuristic search algorithms, respectively. Section 5.7 presents the simulation results. Section 5.8 reviews the related works. Section 5.9 concludes the paper.

5.2 WirelessHART Network Model

We consider a WirelessHART network consisting of field devices, a gateway, and a centralized network manager. A *field device* is a sensor, an actuator or both, and is usually connected to process or plant equipment. The gateway provides the host system with access to the network devices. The network manager is located at the gateway and has the complete information of the network. It creates schedules, and distributes among the devices. The unique features that make WirelessHART particularly suitable for industrial process control are as follows.

Limiting Network Size. Experiences in process industries have shown the daunting challenges in deploying large-scale WSANs. The limit on the network size for a WSAN makes

the centralized management practical and desirable, and enhances the reliability and real-time performance. Large-scale networks can be organized by using multiple gateways or as hierarchical networks that connect small WSANs through traditional resource-rich networks such as Ethernet and 802.11 networks.

Time Division Multiple Access (TDMA). In contrast with CSMA/CA protocols, TDMA protocols provide predictable communication latencies, thereby making themselves an attractive approach for real-time communication. In WirelessHART networks, time is synchronized and slotted. The length of a time slot allows exactly one transmission and its associated acknowledgement between a device pair.

Route and Spectrum Diversity. Spatial diversity of routes allows messages to be routed through multiple paths to mitigate physical obstacles, broken links, and interference. Spectrum diversity gives the network access to all 16 channels defined in IEEE 802.15.4 physical layer and allows per time slot *channel hopping* to avoid jamming and mitigate interference from coexisting wireless systems. Besides, any channel that suffers from persistent external interference is *blacklisted* and not used. The combination of spectrum and route diversity allows a packet to be transmitted multiple times, over different channels over different paths, thereby handling the challenges of network dynamics in harsh and variable environments at the cost of redundant transmissions and scheduling complexity.

Handling Internal Interference. Due to difficulty in detecting interference between nodes and the variability of interference patterns, WirelessHART allows only one transmission in each channel in a time slot across the entire network, thereby avoiding spatial reuse of channels [56]. Thus, there are at most m concurrent transmissions across the network at any slot, with m being the number of channels. This design decision effectively avoids transmission failure due to interference between concurrent transmissions, and improves the reliability at the potential cost of reduced throughput. The potential loss in throughput is also mitigated due to small size of network.

With the above features, WirelessHART forms a mesh network modeled as a graph $G = (V, E)$, where the node-set V consists of the gateway and field devices, and the edge-set E is the set of communication links between the nodes. A node can send, receive, and route packets but cannot both send and receive in the same time slot. In addition, two

transmissions that have the same intended receiver interfere each other. A transmission involves exactly one pair of devices connected by an edge. Therefore, two transmissions that happen along edges uv and cd, respectively, are *conflicting* if $(u = c) \vee (u = d) \vee (v = c) \vee (v = d)$. Since conflicting transmissions cannot be scheduled in the same slot, transmission conflicts significantly contribute to communications delays.

5.3 Problem Definition

Real-time flows. We consider a WirelessHART network with a set of end-to-end flows. Associated with every flow are a sensor node called the *source*, an actuator called the *destination* of the flow, and one or more routes connecting its source to destination through the gateway (where controllers are located). Each flow periodically generates a packet at its source which has to be delivered to its destination within a deadline. A flow may need to deliver its packet through multiple routes. If the delivery through a route fails or some link on a route is broken, the packet can still be delivered through another route. In a schedule, time slots must be reserved for transmissions through each route associated with a flow for redundancy. For schedulability test and priority assignment purposes, through each of its associated routes a flow is treated as an individual flow with the same deadline and period. Therefore, from now onward the term 'flow' will refer to flow through a single route. That is, an original flow with ϕ routes is considered ϕ flows each with a single route. Thus, we consider there are n flows denoted by the set F. Each flow $b \in F$ is characterized by a period T_b, a deadline $D_b \leq T_b$, a source, a destination, and a route from its source to destination through the gateway. For each flow $b \in F$, the number of transmissions required to deliver a packet through its route is denoted by C_b. Thus, C_b is the number of time slots required by flow $b \in F$.

End-to-end delay. For a flow, if a packet generated at slot r is delivered to its destination at slot d then its *end-to-end delay* for this packet is defined as $d - r + 1$. The worst case end-to-end delay of flow $b \in F$ is denoted by L_b.

Fixed priority scheduling. In a *fixed priority scheduling policy*, each flow has a fixed priority. At any time slot, among all ready transmissions that do not conflict with the

transmissions already scheduled in the same slot, the transmission of the highest priority flow is scheduled on an available channel.

Schedulability. Transmissions are scheduled using m channels. The set of periodic flows F is called *schedulable* under a scheduling algorithm $\mathbb{A}$, if $\mathbb{A}$ is able to schedule all transmissions in m channels such that no deadline is missed, i.e., $L_b \leq D_b, \ \forall b \in F$.

Schedulability test. For $\mathbb{A}$, a schedulability test S is *sufficient* if any set of flows deemed schedulable by S is indeed schedulable by $\mathbb{A}$. S is *necessary* if any set of flows deemed unschedulable by S is indeed unschedulable by $\mathbb{A}$. S is *exact* if it is both sufficient and necessary. For a set of flows, an *end-to-end delay analysis* provides a sufficient schedulability test by showing that, for every flow, an upper bound of its worst case end-to-end delay is no greater than its deadline.

Priority assignment. In priority assignment, our objective is to assign a distinct priority to every flow. Given set F of n flows, we have priority levels 1 to n denoted by set $P = \{1, 2, \cdots, n\}$. Any *priority assignment* or *ordering* is, thus, a one-to-one function $f : P \rightarrow F$, where $f(i) = b$ if and only if the priority of flow $b \in F$ is $i \in P$. A priority level i is *higher* than another priority level j if and only if $i < j$. Given a priority assignment, $hp(b)$ denotes the set of flows whose priorities are higher than that of flow b.

Optimal priority assignment algorithm. For a schedulability test S, a priority assignment is called *acceptable* if under that assignment all flows are guaranteed to meet their deadlines according to S. For flow $b \in F$, let R_b denote its worst case end-to-end delay according to S. A priority assignment is *acceptable* denoted by $f^{opt} : P \rightarrow F$ if it satisfies S, i.e., $R_b \leq D_b, \ \forall b \in F$. For a schedulability test S, a priority assignment algorithm is called *optimal* if it can find an acceptable priority assignment whenever there exists any acceptable assignment. That is, if there exists any priority assignment under which S will determine the flows as schedulable, then an *optimal algorithm* is able to find that priority assignment.

5.4 End-to-End Delay Analysis

In this section, we analyze some properties of the existing schedulability tests for real-time flows in WirelessHART networks. These properties provide key insights for an optimal priority assignment algorithm, and intuition for efficient heuristics. We focus on worst case delay analysis (known as *response time analysis* in CPU scheduling), a common approach for schedulability tests in CPU scheduling [45,88] and WirekessHART networks [155]. These tests are based on efficient but pessimistic analysis of end-to-end delays, that provide a sufficient but not necessary condition for schedulability.

To find an optimal priority assignment policy for a schedulability test, the idea is to start from the lowest priority level, and to upper bound and lower bound the end-to-end delays of the flows according to the test, thereby avoiding unnecessary options for priority assignment at higher levels. To estimate these bounds, we identify a class of schedulability tests, named *Class-1*, in which the worst case end-to-end delay of a flow depends on the worst case end-to-end delays of higher priority flows. The other class of tests in which the worst case end-to-end delay of a flow is independent of the worst case end-to-end delays of higher priority flows is named as *Class-2*. Our proposed algorithms work with both classes of schedulability tests. While Class-1 tests are usually more precise than Class-2, Class-2 tests provide the advantages of simplifying the search for priority assignment. In the following, we analyze this using one representative schedulability test of each class.

5.4.1 Class-1 Schedulability Test

An example of a Class-1 schedulability test for real-time flows in WirelessHART networks was proposed in [155]. Given the fixed priorities of the flows, a lower priority flow can be delayed by the higher priority ones due to (a) *channel contention* (when all channels are assigned to transmissions of higher priority flows in a slot), and (b) *transmission conflicts* (when a transmission of the flow and a transmission of a higher priority flow conflict). $\Delta(b, a)$ denotes an upper bound of the delay that flow b can experience from a flow $a \in hp(b)$ due to transmission conflicts. $\Omega_b(x)$ denotes the total delay (in an interval of x slots) caused by all higher priority flows on b due to channel contention. According to this test, the worst case

end-to-end delay R_b of flow b is the minimum value of $y \geq x^*$ that solves Equation 5.1, where x^* is the minimum value of $x \geq C_b$ that solves Equation 5.2 using a fixed-point algorithm. If x or y exceeds D_b, then b is decided to be "unschedulable".

$$y = x^* + \sum_{a \in hp(b)} \left\lceil \frac{y}{T_a} \right\rceil . \Delta(b, a) \tag{5.1}$$

$$x = \left\lfloor \frac{\Omega_b(x)}{m} \right\rfloor + C_b \tag{5.2}$$

$\Delta(b, a)$ is calculated by finding the possible conflicting transmissions of a and b by comparing their routes. For b, $\sum_{a \in hp(b)} \lceil \frac{y}{T_a} \rceil \Delta(b, a)$ is an upper bound of its total delay (in an interval of y slots) due to transmission conflicts with $hp(b)$. $\Omega_b(x)$ is calculated based on a mapping of the transmission scheduling in a WirelessHART network to the multiprocessor scheduling. Specifically, $\Omega_b(x)$ is the delay of b when the flows are executed on multiprocessor, and is analyzed using the response time analysis considering each R_a as the worst case response time of $a \in hp(b)$. The authors used the state-of-the-art response time analysis for multiprocessor scheduling [88] as a representative method to calculate $\Omega_b(x)$. Since schedulability test is not the focus of our work, we skip its details and refer to [155]. We point out our observations on this test.

In this test, when set $hp(b)$ is known for b, the term $\sum_{a \in hp(b)} \lceil \frac{y}{T_a} \rceil \Delta(b, a)$ can be calculated. But $\Omega_b(x)$ depends on R_a of every $a \in hp(b)$ and, hence, is different for different priority ordering among $hp(b)$. Therefore, $\Omega_b(x)$ cannot be calculated if we know only set $hp(b)$. Thus, the bounds of R_b depend on the bounds of $\Omega_b(x)$. $\Omega_b(x)$ non-decreases with the increase of R_a, and non-increases with the decrease of R_a of $a \in hp(b)$. Hence, a lower bound and an upper bound of $\Omega_b(x)$ can be derived when it is calculated with a lower bound and upper bound, respectively, of R_a for every $a \in hp(b)$. Note that $\sum_{a \in hp(b)} \lceil \frac{y}{T_a} \rceil \Delta(b, a)$ is a dominating term in R_b due to high degree of conflicts in the network specially near the gateway (since all flows pass through the gateway). Therefore, our calculated bounds for R_b, for any $b \in F$, are tight even if we set $\Omega_b(x)$ to 0 to find a lower bound, or to maximum channel contention delay to find an upper bound of R_b.

5.4.2 Class-2 Schedulability Test

Now we present a schedulability test of Class-2. An optimal priority assignment policy for this test is comparatively easier since the worst case end-to-end delay of a flow can be derived whenever its higher priority flows are known. Such an observation has been made previously in [68,70] for priority assignment in multiprocessor scheduling by exploiting the analogy with that in uniprocessor scheduling [35].

When R_b is calculated using the fixed-point algorithm in Equation 5.2 for flow b, x^* represents the worst case response time of b when the flows are executed on multiprocessor. Now, to determine R_b using Equation 5.2, x^* for flow $b \in F$ is calculated based on the polynomial-time response time analysis for multiprocessor proposed in [45] as follows.

$$x^* = C_b + \left\lfloor \frac{1}{m} \sum_{a \in hp(b)} \min(W_b(a), D_b - C_b + 1) \right\rfloor \tag{5.3}$$

where $W_b(a) = \lambda_b(a).C_a + \min(C_a, D_b + D_a - C_a - \lambda_b(a).T_a)$; and $\lambda_b(a) = \lfloor \frac{D_b + D_a - C_a}{T_a} \rfloor$.

Here x^* for flow b is a function of C_b, D_b, and of C_a, T_a, and D_a of every $a \in hp(b)$, which remain unchnaged over every priority ordering among $hp(b)$. Hence, R_b can be found using Equations 5.1 and 5.3 when the set $hp(b)$ is known. This test, hence, represents Class-2 schedulability tests.

5.5 Priority Assignment Using Local Search

In this section, we exploit the observations made in Section 5.4 on the classification of schedulability tests and develop an optimal priority assignment algorithm based on local search (LS). Given a schedulability test S and set F of n flows, if there exists any acceptable priority assignment, then the optimal algorithm is able to find that assignment. If no acceptable assignment exists, then it returns a priority assignment that is likely to be good for schedulability of the flows. The proposed algorithm is compatible to any Class-1 and Class-2

schedulability test (Section 5.4). We also analyze some special cases where the algorithm runs in pseudo polynomial time.

The idea underlying local search is to lower bound and upper bound the worst case end-to-end delay according to S of every flow in an acceptable priority assignment. Starting from the lowest priority level, the algorithm explores different options, in the form of a search tree, for assigning priorities at higher levels. For a possible acceptable assignment in a subtree, if, for every flow, an upper bound of its worst case end-to-end delay according to S happens to be no greater than its deadline, then the subtree is a *sufficient branch* and all other branches are discarded. Upper bounding the delays, thus, provides a *sufficient condition* that guarantees that an acceptable assignment can be found in a branch. For a possible acceptable assignment in a subtree, if, for every flow, a lower bound of its worst case end-to-end delay according to S is no greater than its deadline, then the subtree is designated as a *necessary branch*. Thus, lower bounding the delays provides a *necessary condition*. Any branch that dissatisfies this condition is guaranteed not to lead to an acceptable assignment, and is discarded as an *unnecessary branch*.

Having the above idea, the search starts from any initial priority assignment $f : P \to F$. If an acceptable priority assignment exists, then it can be found by reordering f. Every node in the tree performs a reordering of the priorities, thereby representing a new priority assignment. Specifically, the search starts reordering from the lowest priority level n. When it reaches priority level $l \geq 1$, a node has $l - 1$ options to assign priority $l - 1$. For every option, it generates a child node. The branches in the subtree rooted at each child node represent different reordering from level 1 to $l - 1$. Considering f as the priority assignment at a node, we introduce the following notations to establish the bounds in its subtree:

- $R^{\text{opt}}_{f(i)}$: denotes the worst case end-to-end delay of $f(i)$ according to S in an acceptable priority assignment f^{opt}.
- $R^{\text{ub}}_{f(i)}$: denotes an upper bound of $R^{\text{opt}}_{f(i)}$.
- $R^{\text{lb}}_{f(i)}$: denotes a lower bound of $R^{\text{opt}}_{f(i)}$.
- $f_{l,k}$: denotes the priority assignment from level l to k in f, where $1 \leq l \leq k \leq n$ (Figure 5.1). In f, a partial priority assignment $f_{l,k}$ is called *acceptable* if $f_{l,k} = f^{\text{opt}}_{l,k}$,

i.e., the assignment from priority level l to k in f is the same as that in an acceptable priority assignment.

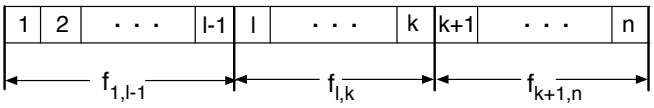


Figure 5.1: Priority assignment f at a node

- $R^{\infty}_{f(i)}$: denotes an upper bound of the end-to-end delay of $f(i)$ under infinite number of channels, i.e., when $f(i)$ experiences no channel contention and is delayed only due to transmission conflicts with the higher priority flows. Therefore, for flow $f(i)$, $R^{\infty}_{f(i)}$ is calculated using Equation 5.1 with x^* being replaced by $C_{f(i)}$.

5.5.1 Upper Bound of Worst Case End-to-End Delay

The upper bounds are calculated based on the observations made in Section 5.4. Specifically, an upper bound of the worst case end-to-end delay of a flow is determined by considering the upper bounds of its higher priority flows.

In a priority assignment f, let the assignment $f_{k+1,n}$ from level $k+1$ ($\leq n$) to n has been decided to be in an acceptable assignment. To decide whether the assignment $f_{l,k}$ from level l ($\leq k$) to k is also in that acceptable assignment, the upper bound $R^{\text{ub}}_{f(i)}$ for every $f(i),\ l \leq i \leq k$, is calculated as follows.

- The set $\{f(j)|1 \leq j < l\}$ is considered as the set of higher priority flows of $f(l)$. When $l = 2$, $R^{\text{ub}}_{f(1)}$ for flow $f(1)$ is set to $C_{f(1)}$. When $l > 2$, $R^{\text{ub}}_{f(j)}$ of every flow $f(j),\ 1 \leq j < l$, is set to its deadline $D_{f(j)}$.
- If $l \leq m$, flow $f(i),\ l \leq i \leq m$, does not experience any channel contention, and hence $R^{\text{ub}}_{f(i)} = R^{\infty}_{f(i)}$. For every other flow $f(i),\ m < i \leq k$, $R^{\text{ub}}_{f(i)}$ is calculated according to schedulability test S using $R^{\text{ub}}_{f(j)}$ as the worst case end-to-end delay of $f(j)$ for every $j < i$.

- If $l > m$, then for every flow $f(i),\ l \leq i \leq k$, $R^{\text{ub}}_{f(i)}$ is calculated according to S using $R^{\text{ub}}_{f(j)}$ as the worst case end-to-end delay of $f(j)$ for every $j < i$.

The upper bound calculation is shown as Procedure $\text{S}^{ub}(f,\ l,\ k)$. In this procedure, $\text{S}(f(i))$ returns $f(i)$'s worst case end-to-end delay according to S using $R^{\text{ub}}_{f(j)}$ as the worst case end-to-end delay of $f(j)$ for every $j < i$. $\text{S}^{ub}(f,\ l,\ k)$ returns *false* if the upper bound is greater than deadline for any flow $f(i),\ l \leq i \leq k$. Otherwise, it returns *true*. Thus, if $\text{S}^{ub}(f,\ l,\ k)$ returns *true*, then it is a guarantee that there exists a priority assignment among the flows $\{f(j)|1 \leq j < l\}$ such that using that assignment (from priority level 1 to $l-1$), and the current assignment $f_{l,k}$ (from level l to k), the resulting assignment from level 1 to k is the same as that in an acceptable assignment. $\text{S}^{ub}(f,\ l,\ k)$ thus provides a *sufficient condition* to determine if reordering $f_{1,l-1}$ can guarantee an acceptable assignment. Any partial assignment $f_{l,k}$ is said to *satisfy* $\text{S}^{ub}(f,\ l,\ k)$, if $\text{S}^{ub}(f,\ l,\ k)$ returns *true*.

Procedure $\text{S}^{ub}(f, l, k)$: Upper Bound Calculation

```
bool S^ub(f, l, k)
begin
    if l = 2 then R^ub_f(1) ← C_f(1);
    ;
    else
        for j = 1 to l − 1 do R^ub_f(j) ← D_f(j); ;
    end
    if l ≤ m then
        l' = min(m, k);
        for i = l to l' do
            R^ub_f(i) ← R^∞_f(i);                          /* Exact value */
            if R^ub_f(i) > D_f(i) then return false; ;
        end
        l ← l' + 1;
    end
    for i = l to k do
        R^ub_f(i) ← S(f(i));                               /* Using test S */
        if R^ub_f(i) > D_f(i) then return false; ;
    end
    return true;
end
```

From our discussions in Section 5.4, in the above upper bound calculation, $R^{\text{ub}}_{f(i)}$ for every $f(i),\ l \leq i \leq k$, includes its exact delay due to transmission conflicts (according to S). This delay is a dominating term in the worst case end-to-end delay of a flow due to high degree of conflicts in the network specially near the gateway (since all the flows pass through the gateway). As a result, our upper bound estimation becomes precise, thereby making the sufficient condition strong.

Theorem 11. *Let $f : P \rightarrow F$ be any priority assignment. If there exists an acceptable priority assignment $f^{opt}_{1,l-1},\ 2 \leq l \leq n+1$, among flows $\{f(i)|1 \leq i < l\}$, then, for $f_{l,k},\ l \leq k \leq n$, $R^{ub}_{f(i)}$ calculated in Procedure $S^{ub}(f, l, k)$ is an upper bound of $R^{opt}_{f(i)}$ for every $f(i),\ l \leq i \leq k$.*

Proof. First, let S be a Class-2 schedulability test (Section 5.4). For each flow $f(i), l \leq i \leq k$, we know its higher priority flows, and $R^{\text{ub}}_{f(i)}$ does not depend on any $R^{\text{ub}}_{f(j)}$ where $j < i$. Hence, $R^{\text{ub}}_{f(i)} = R^{\text{opt}}_{f(i)}$.

Now, let S be a Class-1 schedulability test (Section 5.4). For any $i,\ l \leq i \leq k$, such that $i \leq m$, flow $f(i)$ does not experience any channel contention. Hence, $R^{\text{ub}}_{f(i)} = R^{\infty}_{f(i)} = R^{\text{opt}}_{f(i)}$ holds. For any $i,\ l \leq i \leq k$, such that $i > m$, Procedure $\text{S}^{ub}(f, l, k)$ computes $R^{\text{ub}}_{f(i)}$ for flow $f(i)$ according to S by considering the upper bounds $R^{\text{ub}}_{f(j)}$ for every higher priority flow $f(j)$ where $j < i$. Hence, based on our observations in Section 5.4, $R^{\text{ub}}_{f(i)}$ is an upper bound of $R^{\text{opt}}_{f(i)}$. □

Theorem 12. *Let there exists an acceptable priority assignment $f^{opt} : P \rightarrow F$. Let $f : P \rightarrow F$ be any priority assignment such that $f_{k+1,n}$ satisfies $S^{ub}(f, k+1, n)$. Then $f_{k+1,n} = f^{opt}_{k+1,n}$, i.e., priority assignment from level $k+1$ to level n in f is the same as that in an acceptable assignment. In other words, there is an ordering of priorities from level 1 to k in f that will give an acceptable priority assignment.*

Proof. Assume to the contrary that there exists no priority ordering among the flows $\{f(j)|1 \leq j \leq k\}$ for which $f_{k+1,n}$ is a part of an acceptable priority assignment. Therefore, there must be at least one flow $f(j),\ 1 \leq j \leq k$, that cannot be assigned any priority from level 1 to k. This implies that we must be able to assign some priority j', where $k < j' \leq n$, to this particular flow $f(j)$ since there exists an acceptable priority assignment. But its worst case end-to-end delay at a lower priority level j' must be no less than that at the higher priority

level j. That is, if $f(j)$ is schedulable at the lower priority level j', it must be schedulable at the higher priority level j which contradicts our assumption. □

Lemma 13. *Let there exists an acceptable priority assignment $f^{opt} : P \to F$. Let $f : P \to F$ be any priority assignment such that $f_{k+1,n} = f^{opt}_{k+1,n},\ 0 \leq k < n$. Now if $f_{l,k},\ 1 \leq l \leq k$ satisfies $S^{ub}(f, l, k)$, then $f_{l,n} = f^{opt}_{l,n}$.*

Proof. By Theorem 11, $R^{\text{ub}}_{f(i)} \leq D_{f(i)},\ l \leq i \leq k$. Having assignment $f_{k+1,n}$, $R^{\text{ub}}_{f(i)}$ will not change since each $f(i')$ with $i' > k$ is a lower priority flow of $f(i),\ l \leq i \leq k$. By Theorem 12, $f_{l,k}$ is in an acceptable assignment. Hence, $f_{l,n} = f^{\text{opt}}_{l,n}$. □

5.5.2 Lower Bound of Worst Case End-to-End Delay

Similar to upper bound calculation, a lower bound of the worst case end-to-end delay of a flow is determined by considering the lower bounds of its higher priority flows.

In a priority assignment f, let the assignment $f_{k+1,n}$ from level $k+1 (\leq n)$ to n has been decided to be in an acceptable assignment. To decide whether the assignment $f_{l,k}$ from level $l (\leq k)$ to k is also in that acceptable assignment, the lower bound $R^{\text{lb}}_{f(i)}$ for every $f(i),\ l \leq i \leq k$, is calculated as follows.

- The set $\{f(j) | 1 \leq j < l\}$ is the set of higher priority flows of $f(l)$. $R^{\text{lb}}_{f(j)}$ of every flow $f(j),\ 1 \leq j < l$, is set to its number of transmissions $C_{f(j)}$.
- If $l \leq m$, then flow $f(i),\ l \leq i \leq m$, does not experience any channel contention and, hence, $R^{\text{lb}}_{f(i)} = R^{\infty}_{f(i)}$. For every other flow $f(i),\ m < i \leq k$, $R^{\text{lb}}_{f(i)}$ is calculated according to schedulability test S using the $R^{\text{lb}}_{f(j)}$ as the worst case end-to-end delay of $f(j)$ for every $j < i$.
- If $l > m$, then for every flow $f(i),\ l \leq i \leq k$, $R^{\text{lb}}_{f(i)}$ is calculated according to S using the $R^{\text{lb}}_{f(j)}$ as the worst case end-to-end delay of $f(j)$ for every $j < i$.

The procedure for calculating the lower bounds is shown as Procedure $\text{S}^{lb}(f,\ l,\ k)$. In this procedure, $\text{S}(f(i))$ returns $f(i)$'s worst case end-to-end delay according to S using $R^{\text{lb}}_{f(j)}$ as

the worst case end-to-end delay of $f(j)$ for every $j < i$. $\mathrm{S}^{lb}(f,\ l,\ k)$ returns *false* if the lower bound is greater than deadline for any flow $f(i),\ l \leq i \leq k$. Otherwise, it returns *true*. Thus, if $\mathrm{S}^{lb}(f,\ l,\ k)$ returns *false*, it is a guarantee that no ordering of flows $\{f(j)|1 \leq j < l\}$ can be in an acceptable assignment. $\mathrm{S}^{lb}(f,\ l,\ k)$ thus provides a *necessary condition* to determine if reordering $f_{1,l-1}$ can guarantee an acceptable assignment. Any partial assignment $f_{l,k}$ is said to *satisfy* $\mathrm{S}^{lb}(f,\ l,\ k)$, if $\mathrm{S}^{lb}(f,\ l,\ k)$ returns *true*.

Procedure $\mathrm{S}^{lb}(f, l, k)$: Lower Bound Calculation

```
bool S^lb(f, l, k)
begin
    for i = 1 to l − 1 do R^lb_{f(i)} ← C_{f(i)}; ;
    if l ≤ m then
        l' = min(m, k);
        for i = l to l' do
            R^lb_{f(i)} ← R^∞_{f(i)};                          /* Exact value */
            if R^lb_{f(i)} > D_{f(i)} then return false; ;
        end
        l ← l' + 1;
    end
    for i = l to k do
        R^lb_{f(i)} ← S(f(i));                                 /* Using test S */
        if R^lb_{f(i)} > D_{f(i)} then return false; ;
    end
    return true;
end
```

From our discussions in Section 5.4, $R^{\mathrm{lb}}_{f(i)}$ for every $f(i),\ l \leq i \leq k$, includes its exact delay due to transmission conflicts (according to S). This delay is a dominating term in the worst case end-to-end delay of a flow due to high degree of conflicts in the network specially near the gateway (since all the flows pass through the gateway). As a result, like the upper bounds, our lower bound estimation also becomes precise, thereby making the necessary condition strong.

Theorem 14. *Let $f : P \rightarrow F$ be any priority assignment. Let f^{opt} be an acceptable priority assignment such that there exists an ordering among flows $\{f(j)|1 \leq j < l\}$ which is also in f^{opt}. Then, for $f_{l,k},\ l \leq k \leq n$, $R^{lb}_{f(i)}$ calculated in $S^{lb}(f, l, k)$ is a lower bound of $R^{opt}_{f(i)}$ for every $f(i),\ l \leq i \leq k$.*

Proof. Similar to Theorem 11, $R^{\text{lb}}_{f(i)} = R^{\text{opt}}_{f(i)}$ for every $f(i),\ l \leq i \leq k$, when S is a Class-2 schedulability test. When S is Class-1 schedulability test, the proof is similar to Theorem 11. □

Corollary 2 follows from Theorem 14.

Corollary 2. *For any given priority ordering $f : P \to F$, if $f_{l,k},\ l \leq k \leq n$, does not satisfy $S^{\text{lb}}(f, l, k)$, then no acceptable priority assignment can be found from f by reordering the flows from level 1 to $l-1$.*

5.5.3 Local Search Framework

Now we structure the search for an acceptable priority assignment into a local search (LS) framework. Starting from an initial assignment, the algorithm performs a reordering of the priorities at every node of its search tree, thereby creating a new assignment. Specifically, the search starts from the lowest priority level and investigates if any flow that has higher priority in current assignment can be assigned this lower priority and generates a child node representing this new assignment. The branches are discarded or explored based on the lower bounds and upper bounds calculated for a branch.

The search tree has as its root node a Deadline Monotonic (DM) priority ordering. It has a maximum of $n+1$ levels with the root being at level $n+1$. If the DM priority assignment is acceptable, then the algorithm terminates. Otherwise, the search branches down by creating new nodes. Every node represents a complete priority assignment a part of which is guaranteed to be in an acceptable assignment. Therefore, besides the priority assignment f, every node has two attributes l and k, where $1 \leq l \leq n,\ \ l \leq k \leq n+1$ (Figure 5.1). In priority assignment f at a node, its part $f_{k+1,n}$ is *guaranteed* to be in an acceptable assignment; $f_{l,k}$ is *not guaranteed but may be* in an acceptable assignment; and $f_{1,l-1}$ is *yet undecided*. Thus, k is the level on the path from root to this node such that every node on this path from level n to $k+1$ has satisfied the sufficient condition. The steps of the search are:

1. The root starts with $l = n+1$ and $k = n$ since no part of its assignment is yet final.

2. For the undecided part $f_{1,l-1}$ of priority assignment f at a node at tree-level l, the node creates a child node at tree level $l-1$ for every i, $1 \leq i \leq l-1$, by exchanging the priorities between $f(i)$ and $f(l-1)$.

3. If a child node created in Step 2 satisfies the sufficient condition, then its k becomes 1 less than its l meaning that $f_{k+1,n}$ is now guaranteed to be in an acceptable assignment (according to Theorem 12 and Lemma 13). Hence, all other branches are discarded. This child is expanded further by going to Step 2.

4. If a child node created in Step 2 cannot satisfy the necessary condition, it is closed (according to Corollary 2). Otherwise, it is expanded further by going to Step 2.

5. The search continues creating new nodes until it reaches a node at tree level 1 where k becomes 0 which indicates that an acceptable assignment has been found or until there exists no unexpanded node for which neither the necessary nor the sufficient condition is satisfied. In the latter case, no acceptable assignment is found and the priority assignment of the current node is returned.

The pseudo code is shown as LS Priority Assignment Algorithm. If DM priority assignment f is acceptable, then Procedure S$(f, 1, n)$ returns *true*. Otherwise, the root with f, $l = n+1$, and $k = n$ expands by calling procedure LS(*root*). The attributes l, k, and the priority assignment f at any node nd in the search tree is denoted by $nd.l$, $nd.k$, and $nd.f$, respectively. In LS(Node nd), if $nd.k = 0$ for current node nd, then the search terminates by returning $nd.f$ as an acceptable assignment f^*. Otherwise, for every flow $nd.f(i)$ starting from $i = l-1$ to 1, it generates a child node ch with $ch.k = nd.k$ at level $ch.l = nd.l - 1$, and SwapPriority$(ch.f(i), ch.f(ch.l))$ exchanges the priorities between $ch.f(i)$ and $ch.f(ch.l)$. If $\mathrm{S}^{ub}(ch.f, ch.l, ch.k)$ returns *true*, then $ch.k$ is updated to $ch.l - 1$, and all other branches are discarded, and only this child is expanded as a sufficient branch by calling LS(ch). If $\mathrm{S}^{lb}(ch.f, ch.l, ch.k)$ returns *false*, then ch is closed. Otherwise, it is expanded as a necessary branch. If the tree cannot expand any more and $k > 0$ at every node, then no acceptable assignment exists, and f of current node is returned as f^*.

Theorem 15. *For a given set F of n flows and a schedulability test S, there exists an acceptable priority assignment of F if and only if the priority assignment f^* returned by the LS algorithm is acceptable.*

Procedure LS(Node nd): Local Search

```
bool LS(Node nd)
begin
    if nd.k = 0 then
        f* ← nd.f; return true;                                  /* Acceptable */
    end
    for i = nd.l − 1 down to 1 do
        Create a Child Node ch;
        ch.f ← nd.f; ch.l ← nd.l − 1; ch.k ← nd.k;
        SwapPriority(ch.f(i), ch.f(ch.l));
        if S^ub(ch.f, ch.l, ch.k) = true then
            ch.k ← ch.l − 1;                                     /* Sufficient */
            if LS(ch) = true then                                    /* branch */
                return true;                                          /* found */
            break;                                          /* Cut other branches */
        end
        if S^lb(ch.f, ch.l, ch.k) = false then
            continue;                                       /* Close this child */
        else if LS(ch) = true then
            return true;                                    /* Necessary branch */
        end
    end
    f* ← nd.f;                                                  /* No acceptable */
    return false;                                          /* assignment exists */
end
```

Algorithm 4: Algorithm: LS Priority Assignment

input : Set F of n flows, and schedulability test S
output: $f^* : P \to F$, where $P = \{1, 2, \cdots, n\}$
$f \leftarrow$Deadline Monotonic priority assignment;
if $S(f, 1, n) = true$ **then** /* DM satisfies S */
  $f^* \leftarrow f$; *return* "acceptable assignment found";
end
Create a Node *root* with attributes $f,\ l,\ k$;
$root.f \leftarrow f;\ root.l \leftarrow n + 1;\ root.k \leftarrow n;$
if $LS(root) = true$ **then**
  return "acceptable assignment found";
else
  return "no acceptable assignment exists";

Proof. Let there exists an acceptable priority assignment of F. By Theorem 12 and Lemma 13, if the search stops with $k = 0$ at a node, then the priority assignment of that node must be acceptable. Suppose to the contrary the search has stopped at a node nd with $k > 0$ and priority assignment f. Since an acceptable priority assignment exists, by Theorem 12, there must exist a priority ordering $f'_{1,k}$ among flows $\{f(i)|1 \leq i \leq k\}$ in f such that $f'_{1,k}$ and the current assignment in f from level $k+1$ to n will give an acceptable assignment. Hence, at least one necessary branch must reach level 1 from nd, and at least one such branch that reaches a node nd' at level 1 must correspond to $f'_{1,k}$. Node nd' must satisfy the sufficient condition and its k' is updated to $1 - 1 = 0$ which contradicts our assumption. To prove in the other direction, let f^* returned by the algorithm is considered acceptable. This implies that the search has stopped with $k = 0$. By Theorem 12 and Lemma 13, f^* must satisfy S. □

5.5.4 Analysis

Now we analyze the LS Algorithm for some special cases where it runs in pseudo polynomial time.

Case 1. According to Section 5.4, when S is a Class-2 schedulability test, both the lower bound and the upper bound calculated for a flow are its exact worst case end-to-end delay according to S. That is, both the necessary condition and the sufficient condition become exact. As a result, the search tree consists of just one path. If there exists an acceptable assignment, then the path reaches level 1. Otherwise, it stops at some level where no flow can be assigned that priority. In either case, the search always has $l = k$ and, at every level l ($\leq n + 1$), it tests at most $l - 1$ flows. Thus the algorithm runs in $O(n^2t)$ time, where t is the time to calculate the worst case end-to-end delay of a flow using S and is pseudo polynomial.

Case 2. Based on our observations in Section 5.4, when $m \geq n$, there is no channel contention and, hence, $R^{\infty}_{f(i)}$ is the worst case end-to-end delay for every flow $f(i)$. Hence, similar to Case 1, both the sufficient condition and the necessary condition are exact, and the algorithm runs in $O(n^2t)$ time. When $n > m$, the same thing happens when the value of k becomes no greater than m during the search.

Case 3. If the DM priority assignment is acceptable, then the search stops immediately and returns that ordering. Assigning DM priorities takes $O(n \log n)$ time, and to verify if it is acceptable by S, we need $O(nt)$ time. Hence, the algorithm runs in $O(n \log n + nt)$ time.

5.6 Priority Assignment Using Heuristic Search

While the proposed LS algorithm is optimal and runs efficiently in most cases (as shown in simulation in Section 5.7), a faster execution time cannot be guaranteed theoretically for all the time. Therefore, in this section, we propose an efficient near-optimal heuristic search. It is also based on the similar strategy but is forced to discard many branches by expanding only the branches deemed good. Hence, it runs much faster at the cost of loosing the optimal behavior in some cases.

The LS algorithm can take longer time mostly when it is hard to find a sufficient branch. The key idea behind the heuristic search (HS), therefore, is to loose this condition for faster execution. To determine whether the subtree rooted at a node ch is sufficient, the LS algorithm calls Procedure $S^{ub}(ch.f, ch.l, ch.k)$. The procedure determines the branch as sufficient only if the upper bound of the worst case end-to-end delay of every flow $ch.f(i)$, $l \leq i \leq k$, is no greater than its deadline. Since this is an overestimate, the HS algorithm instead checks only for current level $ch.l$. That is, only if the upper bound of the worst case end-to-end delay of flow $ch.f(ch.l)$ is no greater than its deadline, it discards all other branches and expands only this branch. Note that such a branch is still good (but not guaranteed to be the best as the new condition is not sufficient) since every node from level n to $ch.l$ on this branch has either satisfied this new condition or the necessary condition which we have previously argued to be a strong condition because of precise lower bound estimation.

The HS algorithm considers every level from $ch.l$ to $ch.k$ in Procedure $\mathrm{S}^{lb}(ch.f, ch.l, ch.k)$ as the necessary condition. Since the *new condition* $S^{ub}(ch.f, ch.l, ch.l) = true$ does not mean that $f_{l,n}$ is acceptable, the search updates k as long as the new condition is satisfied on a root to leaf path, and stops updating it after the first time the new condition is violated on that path. That is, $ch.k$ is updated only if $ch.k \geq ch.l - 1$. The HS algorithm is, thus, pseudo coded by making two changes in Procedure LS(Node nd) of the LS algorithm:

1. Replace the condition $S^{ub}(ch.f, ch.l, ch.k) = true$ with $S^{ub}(ch.f, ch.l, ch.l) = true$.

2. Before the statement $ch.k \leftarrow ch.l - 1$, add the check **if**$(ch.k \geq ch.l - 1)$.

By Theorem 12, the partial priority assignment $f_{k+1,n}$ of f at a node of the search tree in the HS algorithm is still guaranteed to be a part of an acceptable assignment (if there exists one at all). However, when the algorithm terminates at a node nd, some node on a level from $nd.k$ to $nd.l$ (when $nd.k \neq nd.l$) on the path from the root to nd may have violated the sufficient condition which the HS algorithm is not aware of. In that case, the algorithm is not optimal. However, our simulation studies have shown that such cases hardly happen in practice.

Analysis. In Case 1 and Case 2 (Subsection 5.5.4), the optimal LS algorithm always maintains $l = k$. Hence, the new condition used in the HS algorithm becomes a sufficient condition. Case 1 and Case 2, thus, hold for the HS algorithm. That is, if $m \geq n$ or S is a Class-2 schedulability test, then it is optimal and runs in $O(n^2 t)$ time, where t is the time to calculate the worst case end-to-end delay of a flow using S. Besides, Case 3 always holds for it. It trivially dominates DM in that whenever the DM priority assignment is acceptable the HS algorithm also determines that assignment as acceptable and runs in $O(n \log n + nt)$ time. In other cases for Class-1 tests, although the execution time of the HS algorithm is theoretically exponential, it can be guaranteed to run faster in practice. A long execution time can happen if the new condition is hardly satisfied or the necessary condition cannot discard enough branches. Note that the new condition is hardly satisfied when the flows have very tight deadlines. However, in this case, the necessary condition will discard many branches. Again, the necessary condition may not discard enough branches if the deadlines are not tight. In this case, the new condition is easily satisfied to discard all other branches, thereby making the search faster.

5.7 Performance Evaluation

We evaluate our priority assignment algorithms through simulations based on both random topologies and the real topology of a physical testbed. We compare the heuristic search (HS) with the optimal local search (LS) algorithm and the following priority assignment

policies: (a) *Deadline Monotonic (DM)* assigns priorities to flows according to their relative deadlines; (b) *Proportional Deadline monotonic (PD)* assigns priorities to flows based on *relative subdeadline* defined for a flow as its relative deadline divided by the total number of transmissions along its route.

Metrics. We evaluate the algorithms in terms of the following metrics. (a) *Acceptance ratio:* fraction of the test cases deemed schedulable according to the schedulability test used. (b) *Execution time:* average execution time (with the 95% confidence interval) needed to generate a priority assignment.

Simulation Setup. A fraction (θ) of nodes is considered as sources and destinations. A node with the highest degree is selected as the gateway. The *reliability* of a link is represented by the *packet reception ratio (PRR)* along it. The most reliable route connecting a source to a destination is selected as the first route. For additional routes (for redundancy) between the same pair, we exclude the links used by existing routes between the pair and select the next most reliable route. Period T_b of every flow b is generated randomly in a range denoted by $T_\sim = 2^{i \sim j}$ slots, $i \leq j$. The relative deadline D_b of every flow b is randomly generated in a range between C_b and $\alpha * T_b$ slots, for $0 < \alpha \leq 1$. The algorithms have been implemented in C and tested on a Macbook Pro laptop. Table 5.1 summarizes the notations used in this section.

N :	Number of nodes in the network
m :	Number of channels
ρ :	Edge-density of the network
θ :	Fraction of total nodes that are source or destination
γ :	Number of routes between every source and destination
$T_\sim$:	Period range
α :	Deadline parameter (e.g., $C_b \leq D_b \leq \alpha * T_b$, for flow b)

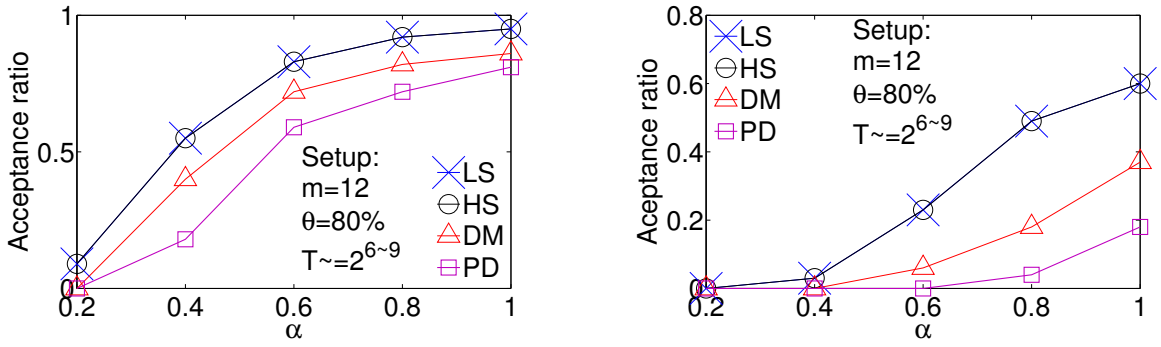
Table 5.1: Notations used in evaluation

5.7.1 Simulations with Testbed Topologies

We evaluate our algorithms on the topology of a physical indoor testbed in Bryan Hall of Washington University [2]. The testbed consists of 48 TelosB motes each equipped with a Chipcon CC2420 radio compliant with IEEE 802.15.4. At transmission power of 0 dBm,

every node broadcasts 50 packets in a round-robin fashion. The neighbors record the sequence numbers of the packets they receive. This cycle is repeated for 5 rounds. Then every link with a higher than 80% PRR is considered *reliable* and used as a link of the topology. Using 12 channels, we compare HS, LS, DM, and PD considering the Class-1 schedulability test presented in Section 5.4 on this topology.

Varying deadlines. We generate flows in the network by randomly selecting the sources and destinations considering $\theta = 80\%$ (i.e., 40% of the total nodes are sources while another 40% are destinations). The periods of the flows are randomly generated in range $2^{6\sim9}$ time slots. We generate 100 test cases and plot the acceptance ratios in Figure 5.2 by varying the deadlines of the flows (by changing α). For $\gamma = 1$ (Figure 5.2(a)), when $\alpha = 0.4$, there are acceptable assignments in 55% test cases but PD is able to find an acceptable assignment only in 18% cases. Thus, the difference between the acceptance ratios of LS and PD is 0.37 when $\alpha = 0.4$. For any $\alpha \geq 0.4$, the difference remains at least 0.14. For DM, this difference is 0.10 to 0.15. When $\gamma = 2$ (Figure 5.2(b)), the differences are 0.23 to 0.45 for PD, and 0.17 to 0.31 for DM. HS performs almost like an optimal algorithm in this setup since it selects good branches and uses a strong necessary condition to discard unnecessary branches (as explained in Section 5.6).



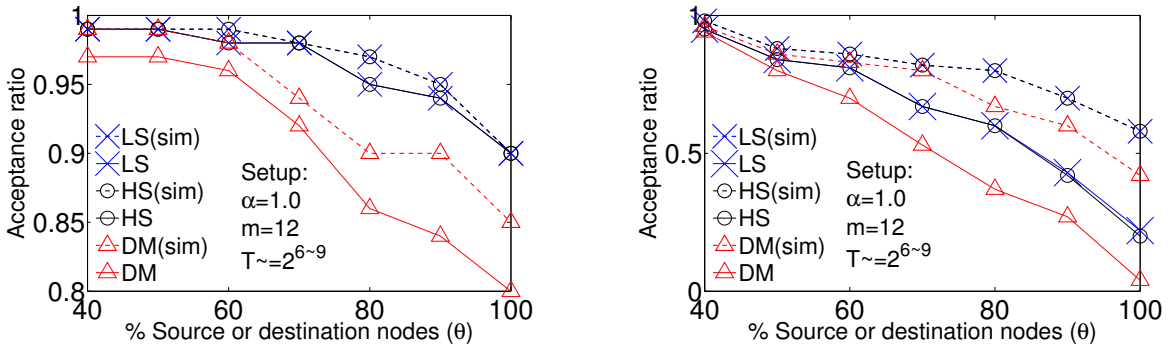
(a) Acceptance ratio when $\gamma = 1$

(b) Acceptance ratio when $\gamma = 2$

Figure 5.2: Performance under varying deadlines

Varying sources and destinations. Now we vary θ resulting in different numbers of flows in the network. Since PD performs worst, we omit any further comparison with PD. For every θ, we generate 100 test cases and show the performances in Figure 5.3. Since

we use a sufficient schedulability test, there are cases when a priority assignment is not acceptable by the test but the flows may meet their deadlines if they are scheduled using that priority assignment. Therefore, every priority assignment generated by an algorithm is tested in simulation by scheduling all flows within their hyper-period. In the figure, each curve "sim" shows the fractions of test cases that have no deadline misses in simulations. When $\gamma = 1$ (Figure 5.3(a)), the difference between the acceptance ratios of LS and DM is always 0.02 to 0.11. Their difference in simulation is 0.02 to 0.05 when $\theta \geq 60\%$. When $\gamma = 2$ (Figure 5.3(b)), their difference is 0.11 to 0.23 in acceptance ratio, and 0.02 to 0.16 in simulation. Here, HS performs like LS when $\gamma = 1$. When $\gamma = 2$, its acceptance ratio is 0.01 to 0.02 less than that of LS, if $\theta \geq 80\%$.



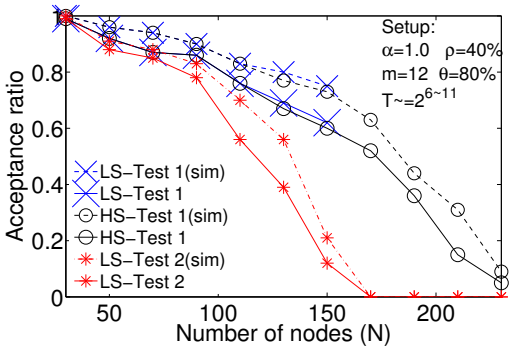
(a) Acceptance ratio when $\gamma = 1$

(b) Acceptance ratio when $\gamma = 2$

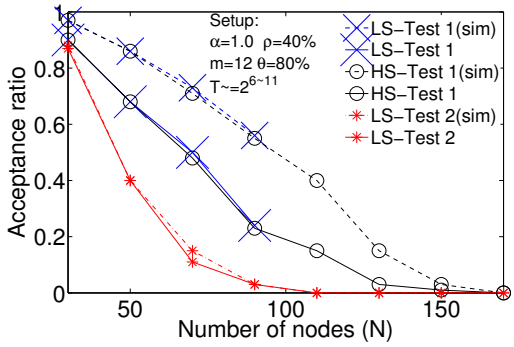
Figure 5.3: Performance under varying number of sources and destinations

5.7.2 Simulations with Random Topologies

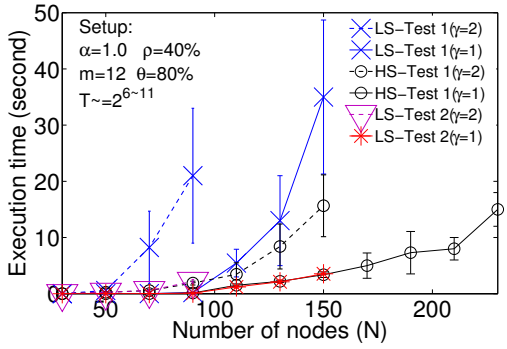
We test the scalability of our algorithms on random topologies of different number of nodes (N). For every N, we generate 100 random networks, each with an edge-density (ρ) of 40%, i.e., with $N(N-1)40/200$ edges. PRR of each edge is randomly assigned between 0.80 and 1.0. In this setup, periods are in range $2^{6\sim11}$ slots to accommodate large networks. Here, we consider both the Class-1 schedulability test (Test 1) and the Class-2 schedulability test (Test 2) presented in Section 5.4. Starting with $N = 30$, we increase N as long as HS can find acceptable assignments and plot the performances in Figure 5.4.



(a) Acceptance ratio when $\gamma = 1$



(b) Acceptance ratio when $\gamma = 2$



(c) Execution time

Figure 5.4: Performance under varying network sizes

For $\gamma = 1$ (Figure 5.4(a)), HS is able to find an acceptable assignment in every case when $N \leq 110$. When $N > 110$, there is a difference from 0.01 to 0.02 between LS and HS in both acceptance ratio and simulation. For $\gamma = 2$ (Figure 5.4(b)), the difference in both acceptance ratio and simulation is 0.01 to .02 when $N > 50$. The figures also indicate that the acceptance ratio with Test 2 is much lower than that with Test 1. For Test 2 in this set up, there exists an acceptable assignment when $N \leq 150$ and $\gamma = 1$ (Figure 5.4(a)), and when $N \leq 90$ and $\gamma = 2$ (Figure 5.4(b)). This is because Test 2 is a less effective schedulability test compared to Test 1. However, both LS and HS are optimal for Test 2 (and we plot it only for LS).

We abort LS if it cannot complete any test case in 10 minutes. Using Test 1, we have been able to record its performance when $N \leq 150$ for $\gamma = 1$, and $N \leq 90$ for $\gamma = 2$. With $\gamma = 1$, its average execution time remains within 5s when $N \leq 100$, and increases sharply to 36s

for $N = 150$ (Figure 5.4(c)). In contrast, HS takes 14s on average when $N = 230$ and $\gamma = 1$. When $\gamma = 2$, its average execution time is 15s when $N = 150$. If $N > 150$, HS cannot find an acceptable priority assignment for any test case. Using Test 2, the execution time of LS remains less than 5s (as long as there exists an assignment acceptable by Test 2). These results indicate that HS is an effective priority assignment algorithm as it is near-optimal and scales better than LS.

5.8 Related Works

For transmission scheduling in wireless sensor networks, schedulability analysis has been addressed in previous works [25,60] considering the fixed priorities as given. For WirelessHART networks, convergecast scheduling for linear and tree [165,188,189] topologies, and real-time flow scheduling for arbitrary topologies [157] have been addressed in recent works. The schedulability analysis proposed in [155] assumes that the priorities of flows are given. None of these works addresses the priority assignment for real-time flows. Complementary to these works on scheduling for WirelessHART networks, we propose an optimal and a near-optimal priority assignment algorithm for real-time flows with fixed priorities.

Alur et al. [32] have proposed a mathematical framework to model and analyze schedules using automata for WirelessHART networks. But their formal method approach can be computationally expensive making it suitable only for *offline* design. In contrast, our heuristic search is suitable for *online* admission control and adaptation, which is needed to handle both dynamic workloads and topology changes common in wireless networks. Moreover, our approach is tailored for fixed priority scheduling that is commonly adopted in real-time networks due to its simplicity and efficiency.

5.9 Summary

WirelessHART is an important standard for wireless sensor-actuator networks that have increasing adoption in process industries. This paper is the first to address the priority assignment for real-time flows in WirelessHART networks. We have proposed an optimal

algorithm based on local search and an efficient heuristic for priority assignment. Simulations on random networks and a sensor network testbed topology showed that the heuristic achieved near-optimal performance in terms of schedulability, while significantly outperforming traditional priority assignment policies for real-time systems.

Acknowledgement

This research was supported by NSF under grants CNS-0448554 (CAREER), CNS-1017701 (NeTS), and CNS-0708460 (CRI).

Chapter 6

Near Optimal Rate Selection for Wireless Control Systems

With the advent of industrial standards such as WirelessHART, process industries are now gravitating towards wireless control systems. Due to limited bandwidth in a wireless network shared by multiple control loops, it is critical to optimize the overall control performance. In this paper, we address the scheduling-control co-design problem of determining the optimal sampling rates of feedback control loops sharing a WirelessHART network. The objective is to minimize the overall control cost while ensuring that all data flows meet their end-to-end deadlines. The resulting constrained optimization based on existing delay bounds for WirelessHART networks is challenging since it is non-differentiable, non-linear, and not in closed-form. We propose four methods to solve this problem. First, we present a subgradient method for rate selection. Second, we propose a greedy heuristic that usually achieves low control cost while significantly reducing the execution time. Third, we propose a global constrained optimization algorithm using a simulated annealing (SA) based penalty method. We study SA method under both constant factor penalty and adaptive penalty. Finally, we formulate rate selection as a differentiable convex optimization problem that provides a quick solution through a convex optimization technique. This is based on a new delay bound that is convex and differentiable, and hence simplifies the optimization problem. We study both the gradient descent method and the interior point method to solve it. We evaluate all methods through simulations based on topologies of a 74-node wireless sensor network testbed. The subgradient method is disposed to incur the longest execution time as well as the highest control cost among all methods. Among the SA based constant penalty method, the greedy heuristic, and the gradient descent method, the first two represent the opposite ends of the

tradeoff between control cost and execution time, while the third one hits the balance between the two. We further observe that the SA based adaptive penalty method is superior to the constant penalty method, and that the interior point method is superior to the gradient method. Thus the interior point method and the SA based adaptive penalty method are the two most effective approaches for rate selection. While both methods are competitive against each other in terms of control cost, the interior point method is significantly faster than the penalty method. As a result, the interior point method upon convex relaxation is more suitable for online rate adaptation than the SA based adaptive penalty method due to their significant difference in run-time efficiency.

6.1 Introduction

With the advent of industrial wireless standards such as WirelessHART [22], recent years have seen successful real-world deployments of process control systems over wireless sensor-actuator networks (WSANs). In a wireless control system, the control performance not only depends on the design of control algorithms, but also relies on real-time communication over the shared wireless network. The choice of sampling rates of the feedback control loops must balance between control performance and real-time communication. A low sampling rate usually degrades the control performance while a high sampling rate may cause excessive communication delays causing degraded performance. The coupling between real-time communication and control requires a scheduling-control co-design approach to optimize the control performance subject to stringent bandwidth constraints of the wireless network.

In this paper, we address the sampling rate optimization problem for multiple feedback control loops sharing a WirelessHART network. A *feedback control loop* periodically delivers data from sensors to the controller, and then delivers the control messages to the actuators through the network. We consider a wireless control system wherein transmissions over a multi-hop WSAN are scheduled based on fixed priorities. The objective is to determine the optimal sampling rates of the feedback control loops to minimize their total control cost, subject to the constraints that their end-to-end network delays are within their respective sampling periods. To our knowledge, this is the first work on scheduling-control co-design for WirelessHART networks.

We formulate the sampling rate optimization problem based on existing end-to-end delay bounds [155, 158] for data flows in multi-hop WirelessHART mesh networks. The resulting constrained non-linear optimization problem is challenging because the existing delay bounds are *non-differentiable* and *not in closed-form*. To address this difficult scheduling-control co-design problem in wireless control systems based on WirelessHART networks, we study and propose four methods:

- First, to handle non-differentiability and non-convexity of the delay bounds, we develop a subgradient based method to find sampling rates through Lagrangian relaxation.
- Second, we propose a time-efficient greedy heuristic that usually achieves low control cost, and is suitable for large-scale WSANs and online rate selection.
- Third, we propose a global constrained optimization algorithm that adopts a penalty approach based on simulated annealing (SA). We study SA method under both constant factor penalty and adaptive penalty.
- Finally, we derive a convex and differentiable delay bound by relaxing an existing delay bound. Then, we formulate the co-design as a differentiable convex optimization problem that can be quickly solved using any traditional convex optimization technique. We study both the gradient descent method and the interior point method to solve it.

We evaluate the proposed algorithms through simulations based on the real network topologies of a wireless sensor network testbed of 74 TelosB motes. Our evaluations have been performed in three different ways:

- In the first case, we use the constant factor penalty method for SA. To solve the reduced convex optimization problem, we use a gradient descent method as a widely used convex optimization technique. The results demonstrate that, among all methods, SA achieves the least control cost while requiring the longest execution time. In contrast, the greedy heuristic runs faster but leads to higher control cost. The gradient descent method based on the new delay bound hits the balance between control cost and execution time. Interestingly, due to high nonlinearity and existence of a large number of local extrema, the subgradient method is both ineffective and inefficient.

- In the second case, we study SA method under two different scenarios: under the constant factor penalty method and under the adaptive penalty method. Our studies indicate that the adaptive penalty method on average outperforms the constant factor penalty method in terms of execution time.
- Finally, we compare the performance of the gradient descent method and the interior point method to solve the reduced convex optimization problem. We observe that the interior point method significantly outperforms the gradient descent method. Our results indicate that the SA based adaptive penalty method and the interior point method are the two most effective approaches for solving our co-design problem, and hence we compare the performances of these two methods. Our results indicate that both methods are competitive against each other in terms of control cost, the interior point method is significantly faster than the penalty method. As a result, the interior point method upon convex relaxation is more suitable for online rate adaptation than the SA based adaptive penalty method due to their significant difference in run-time efficiency.

In the rest of the paper, Section 6.2 reviews related works. Section 6.3 presents the network model. Section 6.4 describes the control loop model. Section 6.5 formulates the rate selection problem. Sections 6.6, 6.7, and 6.8 present the subgradient method, the greedy heuristic method, and the SA based penalty method, respectively, for rate selection. Section 6.9 derives a convex delay bound and presents the gradient descent method and the interior point method for rate selection. Section 6.10 presents evaluation results. Section 6.11 concludes the paper.

6.2 Related Works

There have been extensive studies on real-time CPU scheduling and control co-design in single-processor systems (see survey [182]). Some notable works [49, 160, 161] among them address rate selection under schedulability constraints. However, these works do not apply for networked control systems since network induced delays have significant effects on control

performance, and the schedulability analysis through the network is usually more complicated than CPU scheduling. Following the seminal work on integrated communication and control [90], a number of works [50,66,82,117,118,132,190] have treated the co-design in networked control systems. However, these works have not been designed for wireless networks where end-to-end delay analysis introduces challenging non-linear optimization problems.

For wireless control system, a conceptual study of a wireless real-time system dedicated for remote sensor/actuator control in production automation has been presented in [109]. Wireless control co-design has been studied in [125,126,183]. But these works do not consider multi-hop wireless networks. The rate selection under schedulability constraints for multi-hop wireless sensor network (WSN) has been studied in [127,163]. But these works consider a simplified network model where a WSN is cellular with a base station functioning as a router at the center of each cell. An inner cell is surrounded by 6 cells. The base station in a cell uses 7 orthogonal channels for communication with 6 surrounding cells, periodically enabling transmission in each direction. The utilization based analysis used for this model does not apply for common WSANs based on industrial standards such as WirelessHART. To our knowledge, there exists no utilization based schedulability analysis for multi-hop wireless networks. This lack of simple analytical model to efficiently analyze real-time performance excludes the use of scheduling-control co-design approaches developed for CPU scheduling or wired networks.

As WirelessHART networks [22,56] are becoming the mainstream for wireless control systems in process industries, recent works have focused on control and scheduling issues in WirelessHART networks [32, 93, 140, 155–157, 165, 188]. However, these works have addressed either scheduling [140, 156, 157, 165, 188], routing [93], delay analysis [155], or framework to model schedules [32], and have not considered the scheduling-control co-design problems such as rate selection. In contrast, we have developed the co-design approach to determine near optimal sampling rates of the feedback control loops which minimize their overall control cost and ensure their real-time schedulability. To our knowledge, this paper is the first to address scheduling-control co-design for WirelessHART networks.

6.3 Control Network Model

We consider a wireless control system where feedback control loops are closed over a WirelessHART network. The WirelessHART standard [22, 56] has been specifically designed to meet the critical needs for industrial process monitoring and control. We consider a WirelessHART network consisting of a set of field devices (sensors and actuators) and one gateway. A WirelessHART network is characterized by small size and a *centralized network manager* installed in the gateway. The network manager determines the routes, and schedule of transmissions. The controllers for feedback control loops are installed in the gateway. The sensor devices deliver their sensor data to the controllers, and the control messages are then delivered to the actuators through the network.

Time is synchronized, and transmissions happen based on TDMA. A time slot is 10ms long, and allows exactly one transmission and its acknowledgement between a device pair. In a *dedicated slot*, there is only one sender for each receiver. In a *shared slot*, more than one sender can attempt to transmit to the same receiver. The network uses 16 channels defined in IEEE 802.15.4 and allows per time slot channel hopping. Each transmission in a time slot happens on a different channel. A device cannot both transmit and receive at the same time; nor can it receive from more than one sender at the same time. Two transmissions *conflict* when they involve a common node.

A directed list of paths that connect a source and destination pair is defined as a *routing graph*. For communication between a pair, transmissions are scheduled on the routing graph by allocating one link for each en-route device starting from the source, followed by allocating a second dedicated slot on the same path to handle a retransmission, and then by allocating a third shared slot on a separate path to handle another retry. This conservative practice leaves a huge number of allocated time slots unused since only one route is chosen based on network conditions, thereby degrading the schedulability. To address this, existing end-to-end delay analysis [155, 158] considers only collision-free schedule based on dedicated slots. Since delay analysis is not the focus of this paper, we use existing end-to-end delay bounds. If, in the future, any delay bound is derived by considering shared time slots, that bound can be applied to define the constraints in our co-design problem.

6.4 Control Loop Model

The wireless control system consists of n feedback control loops, each denoted by F_i, $1 \leq i \leq n$. Associated with each control loop are a sensor node and an actuator. In each loop, the dynamics of the plant is described as a Linear Time Invariant (LTI) system and can be written as

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t)$$

where $x(t)$ is the plant state, $u(t)$ is the controller output, $y(t)$ is the system output. A, B, C are constant matrix describing the system dynamics. Although we assume LTI in this work, the framework proposed can be extended to time-varying and/or non-linear systems. For each loop, we consider the state feedback controller:

$$u(t) = Lx(t)$$

where L is control gain designed by the control theory. The *quality of control* (QoC) of each loop is measured by the following performance index function [160]:

$$J(u) = \lim_{H \to \infty} \int_0^H \left(x^T(t) Q x(t) + u^T(t) W u(t) \right) dt$$

Where Q and W are quadratic weight matrix representing the importance of deviation of control objective $x(t)$ and control effort $u(t)$[2]. H is the time horizon during which the cost function is calculated. A great value of $J(u)$ thus indicates either a great deviation of the desired state or a great control effort to bring the state to its reference value. An optimal control theory, such as Least Quadratic Regulator (LQR), solves the optimization problem:

$$J^*(u) = \text{minimize } J(u)$$

to derive an optimal controller. Although $J(u)$ is often related to an optimal control problem, it can be used as a general control performance index not limited to some specific controller.

[2]We assume the system converges to the origin.

Considering the digital implementation of a control loop F_i, the optimal control performance may deviate from its continuous counterpart J_i^* respecting the sampling frequency f_i (Hz). Usually, there is complicated interaction between the deviation and the sampling frequency. However, similar to [160], the deviation with respect to the sampling frequency can be approximated as follows

$$J_i = J_{D,i}^* - J_i^* = \alpha_i e^{-\beta_i f_i} \tag{6.1}$$

where $J_{D,i}^*$ is the optimal control performance of the digital implementation, α_i is the magnitude coefficient, and β_i is the decay rate.

Each control loop F_i maintains a minimum required frequency of $f_i^{\min}$ Hz and a maximum allowable frequency of $f_i^{\max}$ Hz. To maintain an acceptable control performance, the end-to-end communication (sensor-controller-actuator) delay for every sensor data and its associated control message must remain within the sampling period T_i. For any control loop F_i, we express its sampling period T_i in terms of time slots. Since 1 slot=10ms, its sampling *rate* or *frequency* is

$$f_i = \frac{100}{T_i} \text{ Hz}$$

Transmissions are scheduled on m channels, and using *rate monotonic* policy where a loop with higher rate has higher priority, breaking ties arbitrarily. The set of control loops $F = \{F_1, F_2, \cdots, F_n\}$ will always be assumed to be ordered by priorities. F_h has higher priority than F_i if and only if $h < i$. That is, each F_h, $1 \leq h \leq i-1$, is a higher priority loop of F_i. In a *fixed priority scheduling policy*, among all transmissions that can be scheduled in a time slot, the one belonging to the highest priority control loop is scheduled on an available channel first. The complete schedule is divided into superframes. A *superframe* represents transmissions in a series of time slots that repeat infinitely and represent the communication pattern of a group of devices. In rate monotonic scheduling, flows having the same period are assigned in the superframe of length equal to their period. We will use C_i to denote the number of transmissions (i.e., time slots) required by F_i for end-to-end communication. The end-to-end delay for F_i is denoted by R_i (time slots). The set of control loops F is *schedulable*, if $R_i \leq T_i$, $\forall 1 \leq i \leq n$.

6.5 Formulation of the Rate Selection Problem

In this section, we formulate the rate selection problem as a constrained non-linear optimization problem. The objective is to minimize the overall control cost of the feedback control loops subject to their real-time schedulability constraints. Based on the selected rates, the control loops are scheduled using rate monotonic policy.

In order to capture the online interaction between control algorithms and the scheduler, a number of issues must be considered. It must be possible to dynamically adjust the control loop parameters, e.g., their rates, in order to compensate for changes in the workload. It can also be advantageous to view this parameter adjustment strategy in the scheduler as a controller. Control design methods must also take the schedulability constraints into account to guarantee real-time communication through the network. Besides, it should be possible to compensate for wireless deficiencies (e.g., lossy links). Briefly, there are three main factors that affect coupling between the control system and wireless network: (1) the rates of the control loops, (2) the end-to-end delays, and (3) the packet loss. As explained in Section 6.3, a packet delivery in WirelessHART networks achieves high degree of reliability through route and spectrum diversity. As a consequence, the probability of packet loss is very low [56]. Therefore, our co-design approach focuses on rates and end-to-end delays.

We use the end-to-end delay bounds derived in [158] which are an improved and extended analysis proposed in [155]. In fact, the analysis in [158] has two ways to derive a delay bound: in pseudo-polynomial time and in polynomial time. Note that a pseudo-polynomial time bound makes the schedulability constraints extremely expensive to check at every step of optimization in the co-design, thereby making a non-linear optimization approach almost impractical. Therefore, in this section, we formulate the problem using the polynomial-time delay bounds that are somewhat less precise than pseudo-polynomial ones. In the polynomial time analysis, the worst case end-to-end delay R_i of F_i is determined as follows

$$R_i = \left\lfloor \frac{1}{m} \sum_{h=1}^{i-1} \Omega_i^h \right\rfloor + \sum_{h=1}^{i-1} \Theta_i^h + C_i \tag{6.2}$$

Where m denotes the total number of channels; Ω_i^h is the delay that a higher priority loop F_h causes on F_i due to *channel contention*, and is determined as follows

$$\Omega_i^h = \min\Big(T_i - C_i + 1, \left\lfloor \frac{T_i + T_h - C_h}{T_h} \right\rfloor C_h + \min\big(C_h, T_i + T_h - C_h - \left\lfloor \frac{T_i + T_h - C_h}{T_h} \right\rfloor T_h\big)\Big)$$

And Θ_i^h is the delay that a higher priority loop F_h can cause on F_i due to *transmission conflict*, and is determined as follows

$$\Theta_i^h = \Delta_i^h + \Big(\left\lfloor \frac{T_i}{T_h} \right\rfloor - 1\Big)\delta_i^h + \min\Big(\delta_i^h, T_i - \left\lfloor \frac{T_i}{T_h} \right\rfloor T_h\Big)$$

where δ_i^h denotes the maximum delay that a single transmission of F_i can suffer from F_h, and Δ_i^h denotes the total maximum delay that all transmissions of F_i can suffer from F_h, $1 \leq h < i$, due to transmission conflict. These values are calculated based on how the routes of F_i and F_h intersect each other. For any given routes of F_i and F_h, δ_i^h and Δ_i^h are constant, and their derivation can be found in [155, 158].

We now define the performance index of the control system that can describe how the control performance depends on the rates and delays of the control loops. Note that, when the controller is implemented, the system performance will deviate from the ideal value of the performance measure attained using continuous-time control, and the deviation will depend on the sampling rate. As mentioned in Equation 6.1 like [160], we quantify this deviation by defining the control cost for every control loop F_i by a monotonic and convex function

$$J_i = \alpha_i e^{-\beta_i f_i} \tag{6.3}$$

where α_i is the magnitude co-efficient, β_i is the decay rate, f_i (in Hz) is the rate of F_i. Considering w_i as the weight of F_i, for a set of chosen rates $f = \{f_1, f_2, \cdots, f_n\}$, where f_i is the rate of F_i, the total control cost of the system stands

$$J(f) = \sum_{i=1}^{n} w_i \alpha_i e^{-\beta_i f_i} \tag{6.4}$$

Function $J(f)$ describes how the control performance depends on the rates (i.e., frequencies) and delays of the control loops. Namely, the higher the rates, the better the performance. However, a too high rate of some loop may cause congestion in the network, resulting in a very low rate for some other loop, thereby degrading the performance. Therefore, we choose the total control cost $J(f)$ as the *performance index*. As can be seen in Equation 6.4, the weight w_i of any control loop F_i indicates its relative importance to the user in terms of the control cost, and hence impacts the overall control cost. A higher weight indicates a more important control loop to the user. The weights are application specific and can be assigned based on the importance of different control loops to the user. Our work is not concerned with determining the weights, and simply considers that the weights are given as input parameters for control loops.

We can now formulate the scheduling-control co-design as a non-linear constrained optimization problem, where our objective is to determine the optimal sampling rates that minimize the total control cost. The co-design must guarantee that the end-to-end delay R_i of every loop F_i is within its deadline T_i. Besides, every control loop F_i must maintain its minimum required rate of $f_i^{\min}$ Hz and the maximum allowable rate of $f_i^{\max}$ Hz for an acceptable control performance. In the scheduling-control co-design, our objective thus boils down to finding rates $f = \{f_1, f_2, \cdots, f_n\}$ so as to

$$\begin{aligned} &\text{minimize } J(f) \\ &\text{subject to } R_i \leq T_i, \ \forall 1 \leq i \leq n \\ &\qquad\qquad f_i \geq f_i^{\min}, \ \forall 1 \leq i \leq n \\ &\qquad\qquad f_i \leq f_i^{\max}, \ \forall 1 \leq i \leq n \end{aligned} \tag{6.5}$$

where $f_i = 100/T_i$ Hz, and R_i is as defined in Equation 6.2, $\forall 1 \leq i \leq n$.

6.6 Subgradient Method for Rate Selection

Subgradient based methods are an established and standard approach for nonlinear optimization. In this section, we develop a subgradient based approach to determine the sampling

rates for control cost optimization in the scheduling-control co-design formulated in the previous section.

In the optimization problem defined in (6.5) for co-design, the objective function $J(f) : \mathbb{R}^n \rightarrow \mathbb{R}$ is convex while the non-linear constraints $R_i \leq T_i$, $\forall 1 \leq i \leq n$, are not convex. This optimization problem is challenging since the constraints $R_i \leq T_i$ are not differentiable, making any traditional gradient-based optimization unsuitable. To generate approximate solutions to the primal problem defined in 6.5, we consider approximate solutions to its dual problem. Here, the dual problem is the one arising from Lagrangian relaxation of the inequality constraints $R_i \leq T_i$, and is given by

$$\begin{aligned} &\text{maximize } L(f, \lambda) \\ &\text{subject to } \lambda \geq 0 \end{aligned} \tag{6.6}$$

where $L(f, \lambda)$ is the *Lagrangian dual function* defined by

$$\begin{aligned} L(f, \lambda) = &\inf\{J(f) + \sum_{i=1}^{n} \lambda_i (R_i - T_i)\} \\ \text{such that } &f_i^{\min} \leq f_i \leq f_i^{\max}, \quad \forall 1 \leq i \leq n \end{aligned}$$

Here $\lambda \in \mathbb{R}^n$ is the vector of *Lagrange multipliers*.

Figure 6.1 shows the surface of the dual problem in (6.6) for changing the rates of 2 control loops (and keeping all other loops' rates unchanged) considering data flows of 12 control loops simulated on our WSN testbed topology (shown in Figure 6.4). The figure shows that $L(f, \lambda)$ is highly nonlinear in rates. This implies the difficulty of the problem. Besides, the function $L(f, \lambda)$ is not differentiable everywhere. Therefore, traditional optimization approaches based on gradient calculation cannot be applied directly to solve it. Hence, we first adopt a subgradient optimization method to determine the rates. Note that f belongs to a finite range. Steps of the subgradient method are presented as Algorithm 5.

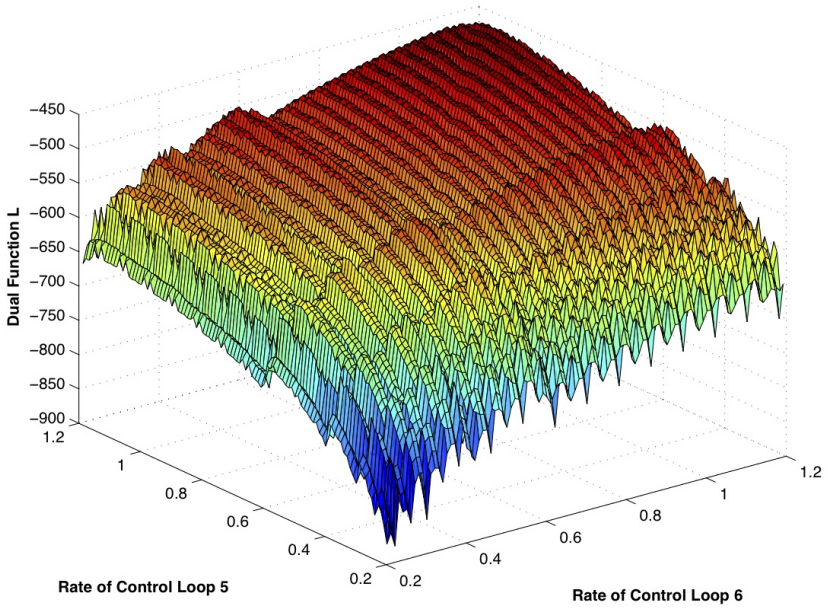


Figure 6.1: Surface of the dual function in 6.6

Thus, the scheduling-control co-design defined in (6.5) can be solved using any existing subgradient solver (e.g., SSMS [18]). Both the speed of convergence and the quality of solution largely depend on the step size selection. As a traditional subgradient method, Algorithm 5 is guaranteed to converge under any diminishing step size or dynamically adjusted step size such as Polyak step size [142].

6.7 Greedy Heuristic for Rate Selection

While a subgradient method is a standard approach for non-linear optimization, it can run very slowly for many practical problems that have too many local extrema and are highly non-linear. Due to a large number of local extrema and complicated subgradient direction in our optimization problem, the subgradient based method proposed in the previous section for rate selection may turn out to be not quite efficient. Therefore, in this section, we propose a simple intuitive greedy heuristic that can run very fast and scale very well.

Algorithm 5: Subgradient Method for Rate Selection

Input: $[f_i^{\min}, f_i^{\max}], w_i, \alpha_i, \beta_i \ \forall F_i, \ 1 \leq i \leq n$;
Output: $f_i, \forall F_i$, and total control cost J;
$f_i \leftarrow f_i^{\min}, \ \forall F_i, 1 \leq i \leq n$; `/* validity check */`
Assign priorities using rate monotonic policy;
if *$\exists F_i$ such that $R_i > T_i$* **then return** *unschedulable;* ;

Step 0: Set time $t = 0$. Choose initial Lagrange multipliers $\lambda^t = 0$. Let f^t be the primal variables corresponding to Lagrange multipliers λ^t.

while *stop condition not true* **do**

Step 1: Determine the rate monotonic priorities of the loops under current f. Solve the Lagrangian subproblem $L(f, \lambda)$. That is, given the dual variables λ^t, determine the primal variables f^t as follows

$$f^t \in \arg\min\{J(f) + \sum_{i=1}^{n} \lambda_i (R_i - T_i)\}$$
$$\text{such that } f^{\min} \leq f^t \leq f^{\max}$$

This gives a subgradient $s^t = R^t - T^t$ at λ^t. If $s^t = 0$, then stop. The algorithm has converged. Current λ^t gives the optimal value of the dual, and current f^t gives an approximated value of the primal.

Step 2: Compute the Lagrange multipliers for next time as follows

$$\lambda^{t+1} = \max\{0, \lambda^t + \gamma^t s^t\}$$

where γ^t is the step size.

Step 3: Update $t = t + 1$ and go to Step 1.

end

The greedy heuristic starts by selecting a rate of $f_i^{\min}$ for each control loop F_i. Note that, for valid rate ranges $[f_i^{\min}, f_i^{\max}]$, the control loops should be schedulable when each loop F_i selects a rate of $f_i^{\min}$. Otherwise, the test case is simply rejected since no rate selection exists that can satisfy the schedulability constraints. For valid rate ranges, the algorithm has the highest control cost in the beginning. Therefore, it will keep decreasing the cost as long the loops are schedulable. This is done by increasing the sampling rates of the loops. The algorithm selects one control loop to increase the rate in each step, and uses a step size of μ by which the rate is increased. For loop F_i, the decrease in control cost due to an increase in current rate f_i by μ is determined as

$$w_i \alpha_i e^{-\beta_i f_i} - w_i \alpha_i e^{-\beta_i (f_i + \mu)}$$

In every step, the greedy heuristic increases the rate of the control loop that decreases the control cost most while satisfying the schedulability constraints of the loops. It keeps

Algorithm 6: Greedy Heuristic

Input: $[f_i^{\min}, f_i^{\max}], w_i, \alpha_i, \beta_i \; \forall F_i, \; 1 \leq i \leq n$, and a step size μ;
Output: $f_i, \forall F_i$, and total control cost J;
$f_i \leftarrow f_i^{\min}, \; \forall F_i, 1 \leq i \leq n$; /* initialize rates */
Assign priorities using rate monotonic policy;
if $\exists F_i$ *such that* $R_i > T_i$ **then return** *unschedulable;* ;
while *true* **do**
  $max \leftarrow 0$; /* maximum control cost decrease */
  $k \leftarrow null$; /* index of the best control loop */
  for *each* $F_i, i = 1, 2, \cdots, n$ *such that* f_i *can further increase* **do**
    $J_i^{\text{old}} \leftarrow w_i \alpha_i e^{-\beta_i f_i}$; /* current cost of F_i */
    $f_i \leftarrow f_i + \mu$; /* increase rate by μ */
    Reassign priorities using rate monotonic policy;
    if $R_j \leq T_j, \; \forall 1 \leq j \leq n$ **then** /* if schedulable */
      $J_i^{\text{new}} \leftarrow w_i \alpha_i e^{-\beta_i f_i}$; /* new cost of F_i */
      if $J_i^{\text{old}} - J_i^{\text{new}} \geq max$ **then**
        $max \leftarrow J_i^{\text{old}} - J_i^{\text{new}}$;
        $k \leftarrow i$; /* F_k is the best candidate */
      end
    end
    $f_i \leftarrow f_i - \mu$; /* put back F_i's rate */
  end
  if *max=0* **then** /* no f_i can further increase */
    return current $f_i, \forall F_i$, and total control cost J
  end
  $f_k \leftarrow f_k + \mu$; /* increase rate of loop F_k */
end

increasing the rates in this way as long as some loop's rate can be increased while keeping all loops schedulable. When no loop's rate can be increased anymore, the algorithm terminates, and returns the current control cost J, and the selected rates. The pseudo code of the greedy heuristic method is presented as Algorithm 6.

6.8 Rate Selection Using a penalty approach with simulated annealing

The greedy heuristic proposed in the previous section can execute very fast and, in some cases, may significantly minimize the control cost. But due to complicated nonlinear constraints, in many cases, it can get stuck in local extrema and, hence, its performance (in

terms of control cost) may not be guaranteed. Therefore, in this section, we explore a global optimization framework based on simulated annealing that can handle non-differentiability and escape local extrema. In particular, we propose a method that extends the standard simulated annealing through a penalty approach to address the constraints for rate selection.

Simulated annealing (SA) is a global optimization framework that is suitable for problems where gradient information is not available. It uses a global parameter called *temperature* to control the probability of accepting a new solution that is worse than the current one. The temperature decreases gradually as the algorithm gradually converges. SA is proven to be able to achieve global optimality under certain theoretical conditions. SA is particularly suitable for our problem since it does not require differentiability of functions, and it employs stochastic global exploration to escape from local minima.

However, while the original SA is designed for unconstrained optimization, our co-design problem is a constrained optimization problem. To find a feasible solution using SA for our co-design problem, we use a ℓ_1- penalty method [58]. In this method, we introduce a new objective function

$$g = J(x) + pV(x),$$

where J is the control cost, $V = \max\{0, R_i - T_i | i = 1 \cdots n\}$ is the *violation of schedulability constraints*, and $p > 0$ is the *penalty factor*. The penalty method starts with a low penalty 0.25 and an initial temperature set to 1000*n, where n is the number of control loops.

At each iteration, we use SA to minimize g under a fixed p. If it cannot find a feasible solution with that setting, we increase the penalty p and temperature and start over the SA algorithm. We call this method the *constant factor penalty method*. Theoretically, such a penalty method can find the constrained global optimal solution when the unconstrained optimization is optimal and p is large enough. The new penalty at the i^{th} iteration is calculated by multiplying p at the $(i-1)^{th}$ iteration by four, and the new temperature is calculated by multiplying the original temperature by the iteration number i. This process is continued until we find a feasible solution or the maximum number of iteration is reached. The maximum number of iteration is currently set to 100. In all SA experiments, we set the final temperature and total number of steps to be 0.01 and 200,000, respectively.

SA based adaptive penalty method. While the constant factor penalty method can find the constrained global optimal solution, it may require a long time to run. In contrast, the execution time can be reduced significantly using an adaptive penalty method. Intuitively, the adaptive penalty method increases the penalty when the current solution is far away from the optimal one, and decreases the penalty otherwise. For our rate selection problem, we measure the constraint violation at the current solution point, and use it as a penalty parameter. In the constant factor penalty method, the previous step's penalty is multiplied by a constant c_1 to get a new penalty. In contrast, in the adaptive penalty method, the constraint violation,after multiplying by a constant c_2, is added to the previous step's penalty to get a new one.

6.9 Rate Selection Through Convex Optimization

Since the co-design problem in (6.5) is non-differentiable and non-convex, we have adopted subgradient method and simulated annealing to solve it. In this section, we derive a differentiable and convex delay bound by relaxing the pseudo-polynomial time delay bound proposed in [155, 158]. Then, we formulate the rate selection problem as a convex optimization problem. Since the new convex delay bounds can be more pessimistic than the original delay bounds, there can be cases where, for any set of selected rates, the constraints based on the original delay bounds are satisfied, but those based on the new delay bounds are not satisfied. To satisfy the constraints in the new (convex) formulation, we may need to decrease some rates, thereby increasing the control cost. Therefore, the control cost obtained upon solving the new problem can be worse than that obtained based on original delay bounds. However, the main advantage of the new formulation is that it can be solved quickly using any existing convex optimization technique (e.g., gradient descent method or interior point method). A quick solution is specially preferred, when the network condition changes frequently, and the network manager needs to recalculate the rates quickly in response to network dynamics.

For each loop F_i, we derive a differentiable and convex delay bound R_i^{cvx} as follows. Based on the pseudo-polynomial time analysis in [155, 158], if loop F_i has an end-to-end delay of x time slots, the channel contention delay Ω_i^h that a higher priority loop F_h can cause on F_i is

bounded as follows

$$\Omega_i^h \leq \left\lfloor \frac{x}{T_h} \right\rfloor C_h + C_h + (C_h - 1) \leq \frac{x}{T_h} C_h + 2C_h - 1$$

Similarly, the transmission conflict delay Θ_i^h that a higher priority loop F_h can cause on F_i is bounded as follows

$$\begin{aligned} \Theta_i^h &= \Delta_i^h + \Big(\left\lfloor \frac{x}{T_h} \right\rfloor - 1 \Big) \delta_i^h + \min \Big(\delta_i^h, x - \left\lfloor \frac{x}{T_h} \right\rfloor T_h \Big) \\ &\leq \Delta_i^h + \Big(\frac{x}{T_h} - 1 \Big) \delta_i^h + \delta_i^h = \Delta_i^h + \frac{x}{T_h} \delta_i^h \end{aligned}$$

Note that the above upper bounds of Θ_i^h and Ω_i^h are both differentiable and continuous. If a control loop F_i has an end-to-end delay of x time slots, then using the above upper bounds of Θ_i^h and Ω_i^h, the end-to-end delay bound x can be written similar to Equation 6.2 as follows

$$\begin{aligned} x &= \frac{1}{m} \sum_{h=1}^{i-1} \Big(\frac{x}{T_h} C_h + 2C_h - 1 \Big) + \sum_{h=1}^{i-1} \Big(\Delta_i^h + \frac{x}{T_h} \delta_i^h \Big) + C_i \\ &= \frac{x}{m} \sum_{h=1}^{i-1} \frac{C_h}{T_h} + \frac{1}{m} \sum_{h=1}^{i-1} \Big(2C_h - 1 \Big) + \sum_{h=1}^{i-1} \Delta_i^h + x \sum_{h=1}^{i-1} \frac{\delta_i^h}{T_h} + C_i \\ \Leftrightarrow x \left(1 - \frac{1}{m} \sum_{h=1}^{i-1} \frac{C_h}{T_h} - \sum_{h=1}^{i-1} \frac{\delta_i^h}{T_h} \right) &= \frac{1}{m} \sum_{h=1}^{i-1} \Big(2C_h - 1 \Big) + \sum_{h=1}^{i-1} \Delta_i^h + C_i \end{aligned}$$

Thus,

$$x = \frac{\frac{1}{m} \sum_{h=1}^{i-1} \Big(2C_h - 1 \Big) + \sum_{h=1}^{i-1} \Delta_i^h + C_i}{1 - \frac{1}{m} \sum_{h=1}^{i-1} \frac{C_h}{T_h} - \sum_{h=1}^{i-1} \frac{\delta_i^h}{T_h}} = R_i^{\text{cvx}} \tag{6.7}$$

Lemma 16. *For any control loop F_i, the end-to-end delay bound R_i^{cvx} derived in Equation 6.7 is convex in f.*

Proof. Note that R_i^{cvx} is twice-differentiable. Hence R_i^{cvx} is convex iff its Hessian matrix is positive semidefinite. Let the constant (the numerator in R_i^{cvx}): $\frac{1}{m}\sum_{h=1}^{i-1}\Big(2C_h - 1\Big) + \sum_{h=1}^{i-1}\Delta_i^h + C_i = Q_i$. Using $T_i = 100/f_i$ the denominator of R_i^{cvx}: $1 - \frac{1}{100m}\sum_{h=1}^{i-1} f_h C_h - \frac{1}{100}\sum_{h=1}^{i-1} f_h \delta_i^h = Z_i$. Letting $\frac{C_h}{100m} + \frac{\delta_i^h}{100} = q_h$, for $h = 1, 2, \cdots, i-1$, the gradient is given by

$$\nabla R_i^{\text{cvx}}(f_1, f_2, \cdots, f_{i-1}) = \begin{pmatrix} \frac{Q_i}{Z_i^2} q_1 \\ \frac{Q_i}{Z_i^2} q_2 \\ \vdots \\ \frac{Q_i}{Z_i^2} q_{i-1} \end{pmatrix}$$

The Hessian matrix H is given by: $H =$

$$\begin{bmatrix} 2\frac{Q_i}{Z_i^3}q_1^2 & 2\frac{Q_i}{Z_i^3}q_1q_2 & 2\frac{Q_i}{Z_i^3}q_1q_3 & \cdots & 2\frac{Q_i}{Z_i^3}q_1q_{i-1} \\ 2\frac{Q_i}{Z_i^3}q_2q_1 & 2\frac{Q_i}{Z_i^3}q_2^2 & 2\frac{Q_i}{Z_i^3}q_2q_3 & \cdots & 2\frac{Q_i}{Z_i^3}q_2q_{i-1} \\ 2\frac{Q_i}{Z_i^3}q_3q_1 & 2\frac{Q_i}{Z_i^3}q_3q_2 & 2\frac{Q_i}{Z_i^3}q_3^2 & \cdots & 2\frac{Q_i}{Z_i^3}q_3q_{i-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 2\frac{Q_i}{Z_i^3}q_{i-1}q_1 & 2\frac{Q_i}{Z_i^3}q_{i-1}q_2 & 2\frac{Q_i}{Z_i^3}q_{i-1}q_3 & \cdots & 2\frac{Q_i}{Z_i^3}q_{i-1}^2 \end{bmatrix}$$

Note that $\frac{Q_i}{Z_i} > 0$, $q_h > 0, \forall h$. Now the leading principal minors of H:

$$\Big|\ 2\tfrac{Q_i}{Z_i^3}q_1^2\ \Big| > 0, \qquad \begin{vmatrix} 2\frac{Q_i}{Z_i^3}q_1^2 & 2\frac{Q_i}{Z_i^3}q_1q_2 \\ 2\frac{Q_i}{Z_i^3}q_2q_1 & 2\frac{Q_i}{Z_i^3}q_2^2 \end{vmatrix} = 0,$$

$$\begin{vmatrix} 2\frac{Q_i}{Z_i^3}q_1^2 & 2\frac{Q_i}{Z_i^3}q_1q_2 & 2\frac{Q_i}{Z_i^3}q_1q_3 \\ 2\frac{Q_i}{Z_i^3}q_2q_1 & 2\frac{Q_i}{Z_i^3}q_2^2 & 2\frac{Q_i}{Z_i^3}q_2q_3 \\ 2\frac{Q_i}{Z_i^3}q_3q_1 & 2\frac{Q_i}{Z_i^3}q_3q_2 & 2\frac{Q_i}{Z_i^3}q_3^2 \end{vmatrix} = 0, \quad \cdots, \quad \Big|\ H\ \Big| = 0.$$

Thus all leading principle minors become non-negative. Therefore, Hessian matrix H is positive semidefinite. Hence, R_i^{cvx} is convex in f. □

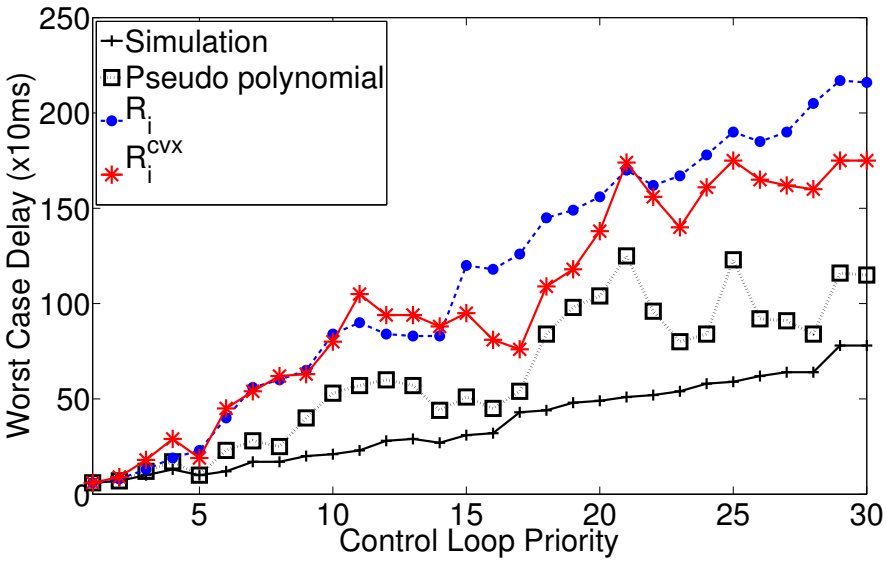


Figure 6.2: End-to-end delay bounds on testbed topology

Figure 6.2 shows how (pessimistic) the derived convex bound R_i^{cvx} is for a test case on our WSN testbed topology (Figure 6.4). The simulation generates data flows for 30 control loops in the network and randomly assigns, for each loop, a harmonic period that is also a multiple of 10ms (i.e., 1 time slot) in a range $[320ms, 5120ms]$. The loops are assigned rate monotonic priority, and are sorted along the X-axis from the highest to the lowest priority. Using 12 channels, the delay bounds R_i (Equation 6.2), R_i^{cvx}, and the delay bound based on the pseudo-polynomial time analysis in [158] are shown in the figure for each loop F_i. The loops are scheduled up to their hyper-period, and for each loop, its maximum end-to-end delay observed in simulations (marked by 'simulation') is also shown. The figure indicates that R_i^{cvx} overestimates the delay at most 2 times that estimated by the pseudo-polynomial analysis. R_i^{cvx} is also highly competitive against the polynomial time delay bound R_i. Since, neither R_i^{cvx} nor R_i dominates the other, we study the results under both bounds. The advantage with R_i^{cvx} is that the co-design problem can be formulated as a convex optimization problem that can be solved quickly using any existing convex optimization technique (e.g., gradient descent method or interior point method).

Now we reformulate the optimization problem in (6.5) using above expression of R_i^{cvx} as follows. Here, we have to select rates $f = \{f_1, f_2, \cdots, f_n\}$ so as to

$$\begin{aligned} &\text{minimize } J(f) \\ &\text{subject to } R_i^{\text{cvx}} \leq T_i, \ \forall 1 \leq i \leq n \\ &\qquad\qquad f_i \geq f_i^{\min}, \ \forall 1 \leq i \leq n \\ &\qquad\qquad f_i \leq f_i^{\max}, \ \forall 1 \leq i \leq n \end{aligned} \tag{6.8}$$

where $f_i = 100/T_i$ Hz, and R_i^{cvx} is as defined in Equation 6.7, $\forall 1 \leq i \leq n$.

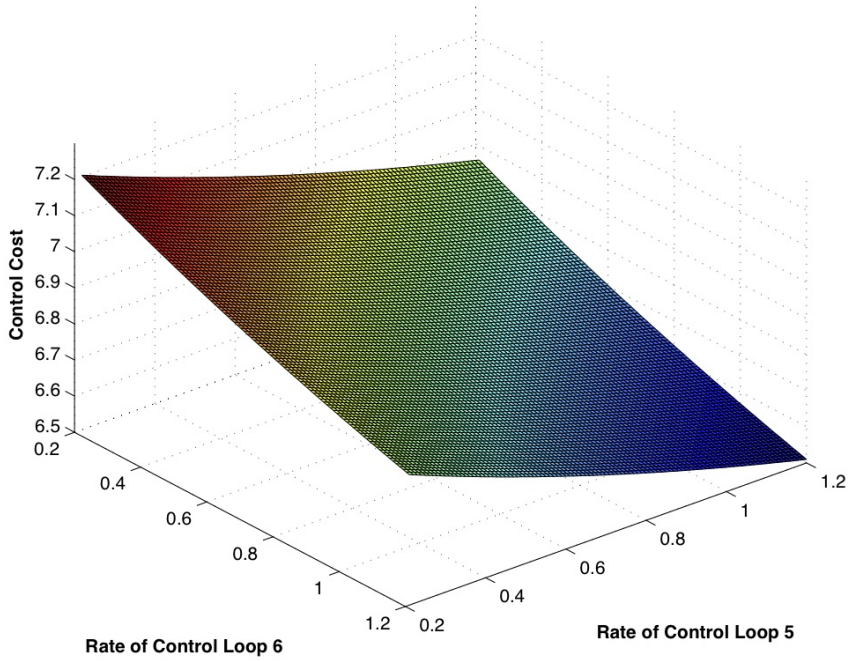


Figure 6.3: Surface of the primal function of Problem in 6.8

The above formulated problem is a convex optimization problem. Figure 6.3 indicates the smoothness of the function in Problem 6.8 for changing the rates of 2 control loops (and keeping all other loops' rates unchanged) considering data flows of 12 control loops simulated on a testbed topology (shown in Figure 6.4).

6.9.1 Gradient Descent Method

First we adopt a gradient descent method for solving the above convex optimization problem. To find a solution to the primal problem, we consider solutions to its dual problem. Here also, the dual problem is formed through Lagrangian relaxation of inequality constraints $R_i^{\text{cvx}} \leq T_i$, and is given by

$$\begin{aligned} &\text{maximize } L(f, \lambda) \\ &\text{subject to } \lambda \geq 0 \end{aligned}$$

Where $L(f, \lambda)$ is the Lagrangian dual function defined by

$$\begin{aligned} L(f, \lambda) = &\inf\{J(f) + \sum_{i=1}^{n} \lambda_i (R_i^{\text{cvx}} - T_i)\} \\ \text{such that } &f_i^{\min} \leq f_i \leq f_i^{\max}, \quad \forall 1 \leq i \leq n \end{aligned}$$

Here $\lambda \in \mathbb{R}^n$ is the vector of Lagrange multipliers. In the dual, $L(f, \lambda)$ is differentiable and, hence, the classical approach of maximizing the function would be the steepest descent method that computes a sequence of iterations to update the multipliers as follows

$$\lambda^{t+1} = \lambda^t + \gamma^t \nabla L(f, \lambda)$$

Note that at every step, the priorities of the control loops are updated according to rate monotonic policy based on new updated rates to calculate R_i^{cvx}. In solving the dual function, we follow the gradient at the current position, with a specified step size γ, to reach points with a higher function value. Unlike Algorithm 5, now we have unique subgradient (which is the gradient) at the current position. In our case, this evaluates to

$$\lambda^{t+1} = \lambda^t + \gamma^t (R^{\text{cvx}} - T)$$

Any traditional step size rule (either vanishing or dynamic) can be applied to reach the solution in a gradient descent way. Also, the solution can be found simply by using any standard convex optimization tool such as CVX [86].

6.9.2 Interior Point Method

The interior point method is a deterministic algorithm for solving nonlinear optimization problems, and often becomes a faster approach for many convex optimization problems. Hence, in addition to the gradient descent method, we adopt the interior point method for solving the above convex optimization problem. This method uses a self-concordant barrier function to encode the convex set. It reaches an optimal solution by traversing the interior of the feasible region. Since our new formulation is a convex optimization problem, it can be transformed into minimizing the objective function over a convex set. The idea is to encode the feasible set using a barrier function.

For the interior point method, the logarithmic barrier function associated with our convex optimization problem in (6.8) is given by

$$B(f,\mu) = J(f) - \mu(\sum_{i=1}^{n} \ln(T_i - R_i^{\text{cvx}}) + \sum_{i=1}^{n} \ln(f_i - f_i^{\min}) + \sum_{i=1}^{n} \ln(f_i^{\max} - f_i)) \tag{6.9}$$

where μ is a small positive scalar called the *barrier parameter*. As μ converges to zero, the minimum of $B(f,\mu)$ should converge to a solution of (6.8).

The barrier function gradient is given by

$$g_b = g - \mu \sum_{j=1}^{k} \frac{1}{c_j(f)} \bigtriangledown c_j(f), \tag{6.10}$$

where g is the gradient of the original function $J(f)$ and $\bigtriangledown c_j(f)$ is the gradient of $c_j(f)$. Here, $c_j(f)$ stands for the j-th constraint in the problem formulation (6.8) and k equals to $3n$. Considering $\lambda \in \mathbb{R}^k$ as Lagrange multipliers, the *perturbed complementarity condition* is

$$\lambda_j c_j(f) = \mu, \forall i = 1, \ldots, m \tag{6.11}$$

We try to find those (x_μ, λ_μ) which turn the gradient of the barrier function to zero. Applying (6.11) to (6.10), we get the equation for gradient as follows.

$$g - A^T\lambda = 0 \tag{6.12}$$

where A is Jacobian matrix of the constraint $c(f)$.

The intuition behind (6.12) is that the gradient of $J(f)$ should lie in the subspace spanned by the constraints' gradients. The perturbed complementarity with small μ can be understood as the condition that the solution should either lie near the boundary $c_j(f) = 0$ or that the projection of the gradient g on the constraint component $c_j(f)$ normal should be almost zero. Applying Newton's method to (6.11) and (6.12), we get an equation for (p_f, p_λ), the search direction in the (f, λ) space at each iteration:

$$\begin{pmatrix} W & -A^T \\ \Lambda A & C \end{pmatrix} \begin{pmatrix} p_f \\ p_\lambda \end{pmatrix} = \begin{pmatrix} -g + A^T\lambda \\ \mu 1 - C\lambda \end{pmatrix},$$

where W is the Hessian matrix of $J(f)$ and Λ is a diagonal matrix of λ. The condition $\lambda \geq 0$ should be enforced at each step. This can be done by choosing an appropriate step size α:

$$(f, \lambda) \rightarrow (f + \alpha p_f, \lambda + \alpha p_\lambda).$$

6.10 Evaluation

In this section, we evaluate the proposed algorithms for near optimal rate selection for feedback control loops in wireless control systems. We evaluate the algorithms through simulations based on the real topologies of a WSN testbed. Our WSN testbed is deployed in two buildings (Bryan Hall and Jolley Hall) of Washington University in St Louis [2]. The testbed consists of 74 TelosB motes each equipped with Chipcon CC2420 radios compliant with the IEEE 802.15.4 standard (WirelessHART is also based on IEEE 802.15.4).

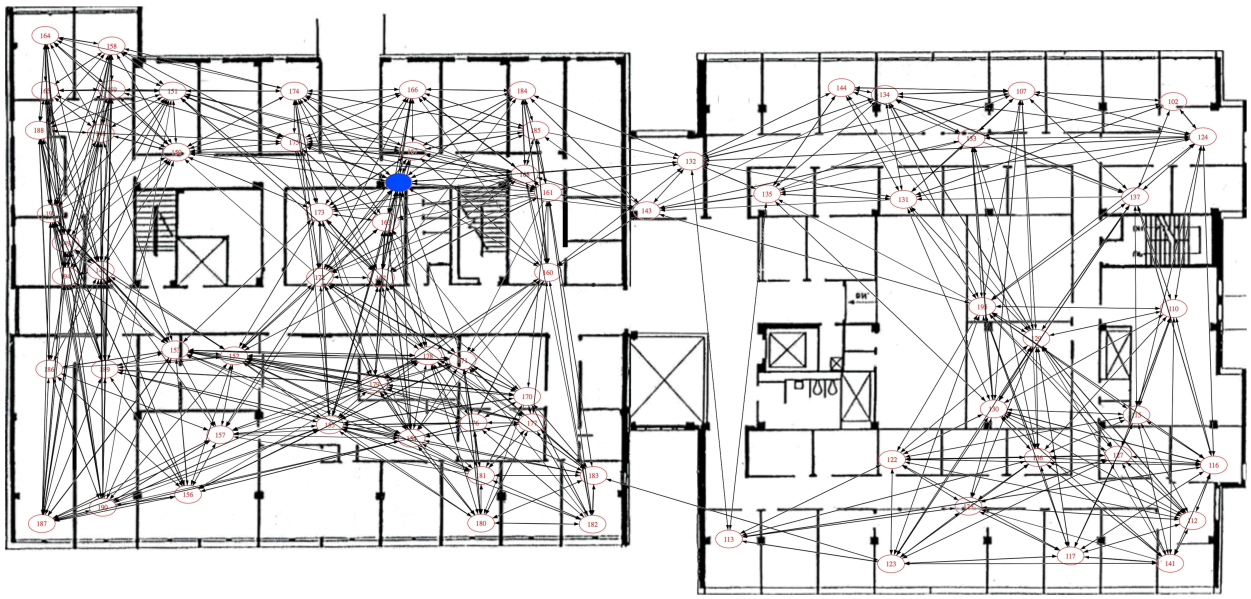
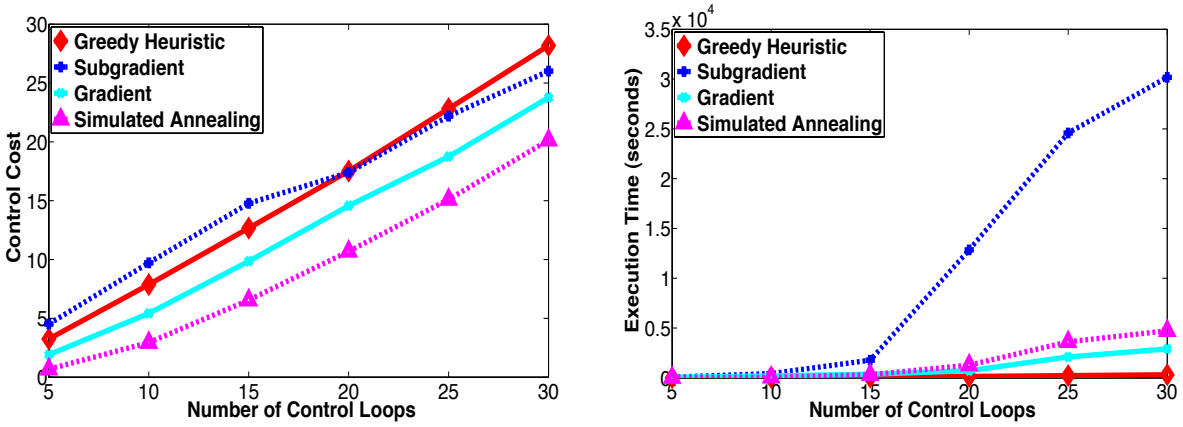


Figure 6.4: Testbed topology at transmission power of -5 dBm (the gateway is colored in blue)

6.10.1 Simulation Setup

We simulate the networked control loops by generating data flows in our testbed topologies. The topologies are determined in the following way. Setting the same transmission power at every node, a node broadcasts 50 packets while its neighbors record the sequence numbers of the packets they receive. After a node completes sending its 50 packets, the next sending node is selected in a round-robin fashion. This cycle is repeated giving each node 5 rounds to transmit 50 packets in each round. Every link with a higher than 80% *packet reception ratio (PRR)* is considered a reliable link to derive the topology of the testbed. Figure 6.4 shows the network topology (embedded on the floor plans of two buildings) when each node's transmission power is set to -5 dBm. We have tested our algorithms using the topologies at 4 different transmission power levels: 0 dBm, -1 dBm, -3 dBm, -5 dBm.

In each topology, the node with the highest number of neighbors is designated as the gateway. A set of nodes is considered as sources (sensors), while another set as destinations (actuators). We select the same source and destination pairs in each topology. The most reliable routes (based on PRR) are used for data flow between source and destination pairs. Each data flow is associated with a control loop. The weight of each control loop is set to 1. The



(a) Control cost (b) Execution time

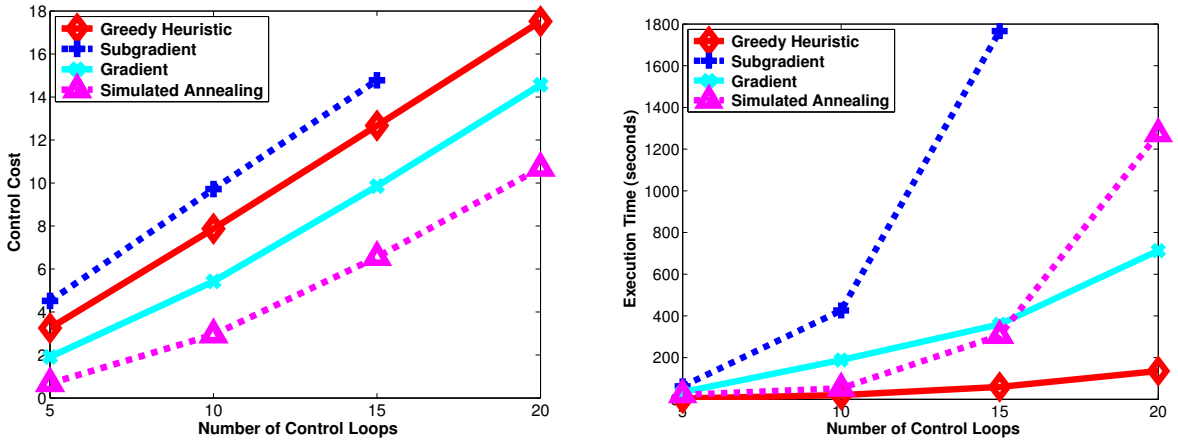
Figure 6.5: Performance comparison on topology at transmission power -5 dBm

decay rate (β) and magnitude coefficient (α) of the loops have been assigned according to those used for bubble control systems in [160]. The penalty based simulated annealing has been implemented based on Python Simulated Annealing Module [144]. For interior point method, we have used IPOPT [96] solver. All other algorithms have been implemented in MATLAB. The tests have been performed on a Mac OS X machine with 2.4 GHz Intel Core 2 Duo processor.

6.10.2 Performance Study of Four Methods

We first evaluate the subgradient method, the greedy heuristic, the SA based constant factor penalty method, and the gradient descent method in terms of achieved *control cost* and *execution time*. For SA, in the first case, we use the constant factor penalty method. The maximum number of iterations is set to $2,00000$. For the reduced convex optimization problem, we use the gradient descent method. We perform the simulations using 12 channels.

Figures 6.5 shows the results for 30 control loops simulated on the testbed topology when every node's transmission power is set to -5 dBm. Figure 6.5(a) indicates that the control cost in the simulated annealing (SA) based penalty method is consistently a lot less than all other methods. The control cost in the gradient method is larger but very close to that of SA, and a lot less than the greedy heuristic and the subgradient method. The greedy



(a) Control cost

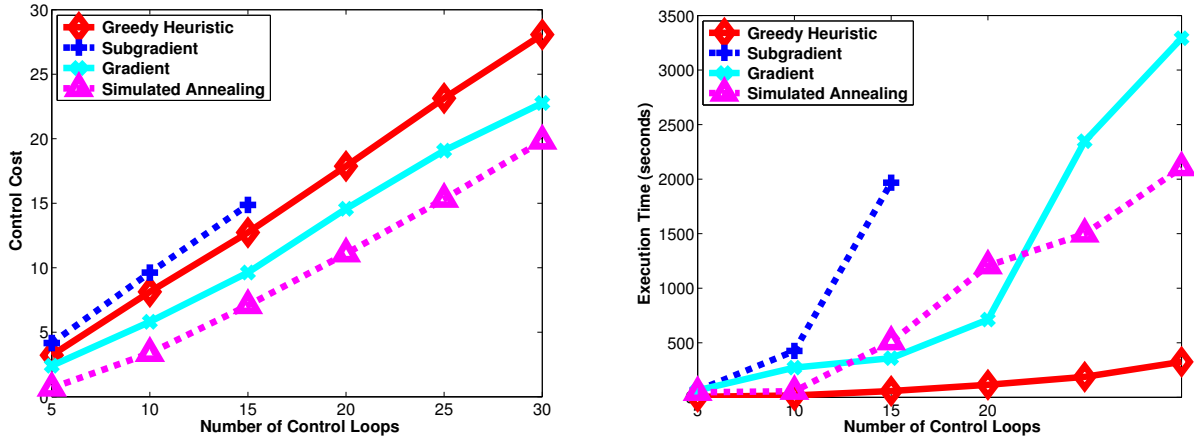
(b) Execution time

Figure 6.6: Performance comparison on topology at transmission power -3 dBm

heuristic is always achieving control cost higher than the gradient method, but less than the subgradient method when number of loops is more than 25. Similar results are observed in Figures 6.6 for the testbed topology with transmission power -3 dBm.

Figure 6.7 shows the results for 30 control loops on the testbed topology with transmission power -1 dBm. Figure 6.7(a) indicates that the control cost in SA is consistently a lot less than all other methods. The control cost in the gradient method is at most 1.2 times that of SA, and is a lot less than the greedy heuristic and the subgradient method. The greedy heuristic is always achieving control cost higher than the gradient method, but less than the subgradient method. The subgradient method takes a long execution time. According to Figure 6.7(b), its time increases exponentially with the number of loops. The gradient method runs faster than SA when the number of loops is increased beyond 10 but does not become larger than 20.

Figure 6.8 shows the results for 30 control loops simulated on the testbed topology when every node's transmission power is set to 0 dBm. Figure 6.8(a) indicates that the control cost in the simulated annealing (SA) based penalty method is consistently a lot less than all other methods. The control cost in the gradient method is very close to that of SA for each number of loops. The control cost in the gradient method is at most 1.12 times that of SA, and is a lot less than the greedy heuristic and the subgradient method. The greedy heuristic is always achieving control cost higher than the gradient method, but less than the



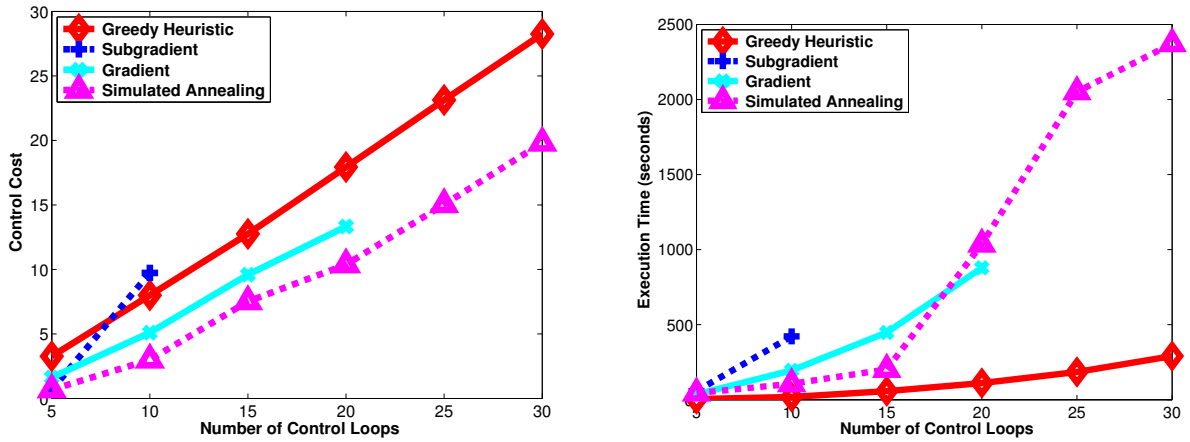
(a) Control cost

(b) Execution time

Figure 6.7: Performance comparison on topology at transmission power -1 dBm

subgradient method when number of loops is more than 5. The subgradient method takes a long execution time, and we were not able to get its results for more than 10 loops. For more than 20 loops, we have also observed that the gradient method takes a longer execution time (Figure 6.8(b)). The gradient method turns out to be a better option for a moderate number loops. According to Figure 6.8(b), the execution time of SA increases exponentially with the number of loops, but always remains less than the subgradient method. The simple greedy heuristic is a lot faster than other methods.

The results demonstrate that, among all methods, SA achieves the least control cost while requiring the longest execution time. The subgradient method turns out to be worse than all other algorithms both in terms of execution time and in terms of control cost. This is quite reasonable as our optimization problem is highly nonlinear and there exist a large number of local extrema. The subgradient direction becomes highly complicated and therefore both its execution time and control cost get worse. The greedy heuristic incurs control cost at most 2.67 times that of SA, while keeping the execution time very low. The gradient based descent method incurs control cost at most 1.35 times that of SA, while keeping the execution time less than SA in most cases. Therefore, to get near optimal results at the cost of longer execution time, SA turns out to be a prominent method. To get results very quickly and for scalability with a moderate control cost, the greedy heuristic turns out to be the best option. To achieve moderate control cost (not as high as greedy and not as low as SA) within



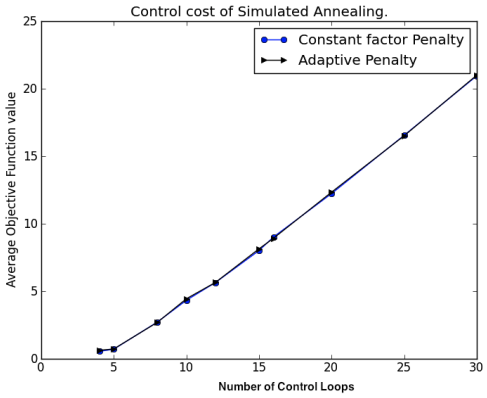
(a) Control cost

(b) Execution time

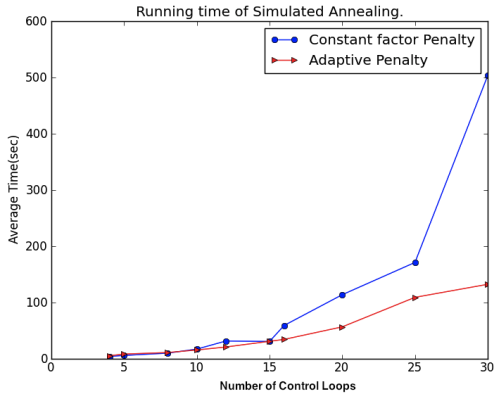
Figure 6.8: Performance comparison on topology at transmission power 0 dBm

a reasonable time (not as fast as greedy, not as slow as SA), the gradient descent method appears to be a promising approach.

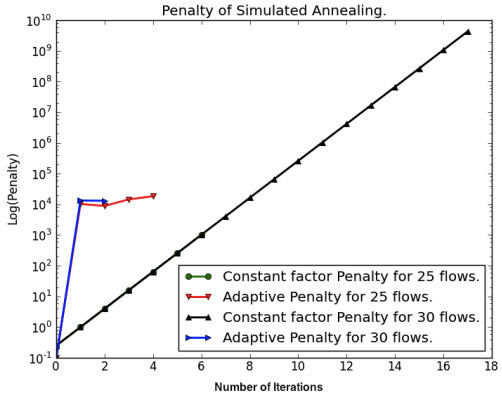
In Figures 6.5, 6.6, 6.7, and 6.8, we observe a small variation in control cost and execution time for the same method in different topologies. In particular, there is a small increase in control cost when the transmission power is reduced. In these tests only the transmission power is changed, while all other setups are the same. The topology of the network changes as the transmission power is changed. Usually, at the highest transmission power (0 dBm), the network has very high connectivity, while the connectivity usually decreases with the decrease in the transmission power. Therefore, at a low transmission power, the routes between a source (sensor) and a (destination) pair can be longer than those between the same pair at a higher transmission power. This may lead to the selection of lower sampling rates for some control loops, thereby increasing the overall control cost. The change in network topology also causes some differences in execution times for the same method under different transmission powers.



(a) Control cost



(b) Execution time



(c) Penalty

Figure 6.9: Performance comparison of Adaptive Vs Constant Penalty SA

6.10.3 SA based Constant Factor Penalty Method Versus Adaptive Penalty Method

We now study the SA based penalty method further. This study has been performed using 12 channels and the topology collected at transmission power -5 dBm. Here, we study SA by setting the maximum number of iterations to a smaller value. In particular, the maximum number of iterations is set to $2,0000$. Note that this setting reduces the execution time significantly. This is what we observe in Figure 6.9 also. In addition, we can see that the control cost under our settings is not significantly higher than that in the previous test with a larger maximum number of iterations.

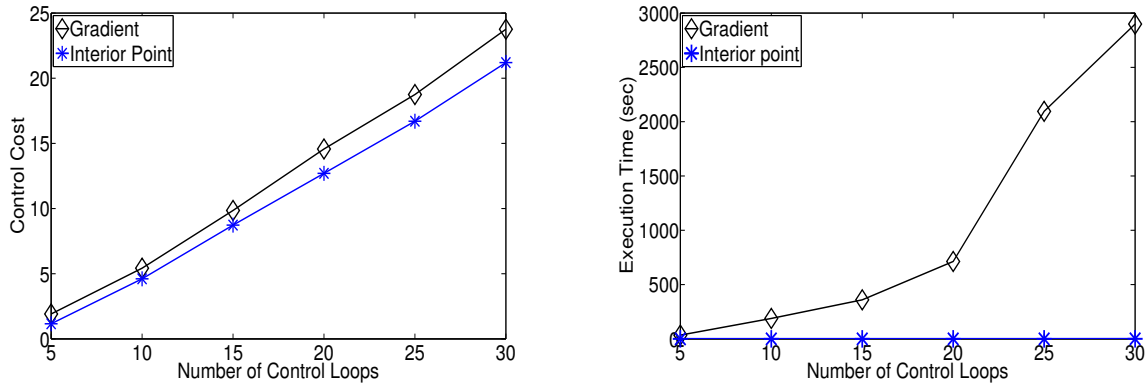
Here we compare the SA based adaptive penalty method, and the constant factor penalty method. Figure 6.9 plots the average execution time and the average objective function value (control cost) of five different runs as it seems that one method was better than other in some cases. The adaptive method outperforms constant factor method on average. In constant factor penalty method, the previous step's penalty is multiplied by four to get a new penalty. In the adaptive method, the constraint violation,after multiplying by a constant 10, is added to the previous step's penalty to get a new one. The running time of both methods were almost the same for small numbers of flows, but the running time of adaptive method was around 70% less than that of constant factor method for larger numbers of flows (Figure 6.9(b)) while the control cost remains close between the two methods (Figure 6.9(a)). Figure 6.9(c) shows the penalties at different iterations in these two methods. It indicates that the adaptive penalty sharply increases the penalty in initial iterations and then reduces the penalty as the solution approaches the final solution. These results indicate that SA based adaptive penalty method is superior to SA based constant factor penalty method.

6.10.4 Evaluating the Interior Point Method

We now solve the reduced convex optimization problem using the interior point method. We have used the IPOPT solver [96] for interior point method, while for gradient descent method we have implemented the optimizer in MATLAB. In this section, we perform the evaluation based on the topology at transmission power of -5 dBm.

Interior Point Method versus Gradient Descent Method

We have compared the performances of the interior point method and the gradient descent method using 12 channels. The results in Figure 6.10 indicates that the interior point method is superior to the gradient descent method. Its execution time is significantly less than the gradient method. In particular, the execution time for interior point method is no more than 1 second for each case. These results indicate that the interior method is much more efficient than the gradient method in solving the relaxed convex optimization problem while also outperforming the gradient method in terms of control cost.



(a) Control cost (b) Execution time

Figure 6.10: Interior Point Method versus Gradient Method for Convex Optimization

Interior Point Method versus the SA based Adaptive Penalty Method

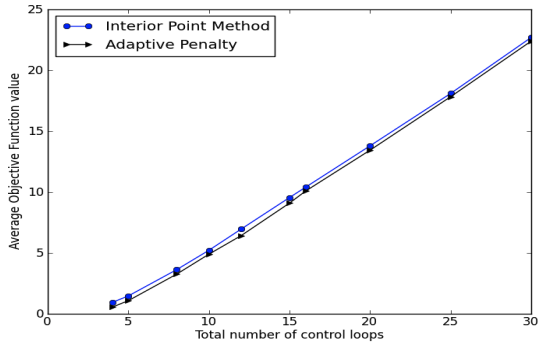
Using the topology at transmission power of -5 dBm, we now compare the performances of the interior point method and the SA based adaptive penalty method as these two seem to be the most competitive and effective approaches for our co-design problem. We test the results under different numbers of channels. Figure 6.11 shows the results under these two methods when the number of channels are 4, 8, 12, and 16.

Surprisingly, in the figure, we notice that the control cost does not decrease significantly as we increase the number of channels. This happens because the end-to-end delays of the control loops through the network are dominated by transmission conflicts instead of the number of channels. Namely, for our small network topology, many channels remain unused as many transmissions cannot be scheduled simultaneously due to transmission conflict. Therefore, adding channels does not necessarily increase the schedulability of the control loops as two conflicting transmissions cannot be scheduled in the same time slot, no matter how many channels are available. (Note that this is a significant difference between multi-processor scheduling and transmission scheduling in wireless networks. In the former case, the schedulability increases as the number of processor increases, while in the latter case the schedulability is determined by both the number of channels and the degrees of conflict in the network.)

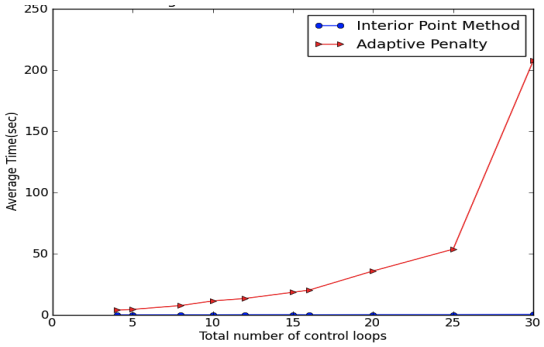
Figure 6.11 also indicates that the interior point method runs significantly faster than SA based adaptive penalty method, but the control cost incurred in the latter is slightly less than that in the former. As a result, these two methods represent the opposite ends of the tradeoff between control cost and execution time, while the interior method is likely the most effective approach in practice due to its run time efficiency. Specifically, as seen in the results in Figures 6.11(a), 6.11(c), 6.11(e), and 6.11(g), SA based adaptive penalty method and the interior point method incur almost the same control cost which results from similar configurations of sampling rates under both methods. Therefore both methods are competitive against each other in terms of control performance. The *execution time* is also an important metric in wireless control systems because the wireless networks in industrial environments are subject to network dynamics and frequent topology changes. The network manager may need to recalculate the sampling rates frequently in response to dynamics. As can be seen the execution times of the two methods in Figures 6.11(b), 6.11(d), 6.11(f), and 6.11(h), the interior point method is significantly faster than SA based adaptive penalty method. As a result, the interior point method is more suitable for online rate adaptation than the adaptive penalty method due to their significant difference in run-time efficiency.

6.11 Summary

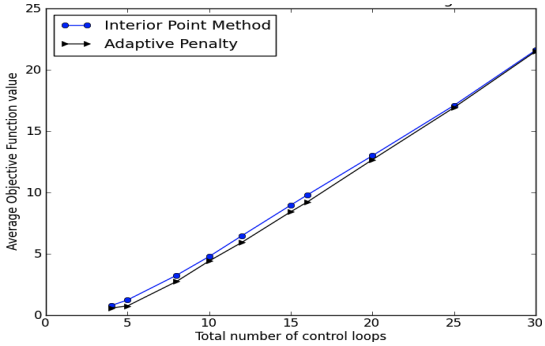
Recent industrial standards such as WirelessHART have enabled real-world deployment of wireless control systems. Due to limited bandwidth in wireless sensor-actuator networks, it is important to optimize the control performance through a wireless-control co-design approach. This paper addresses the problem of determining the optimal sampling rates of feedback control loops sharing a WirelessHART network. The objective is to minimize the overall control cost while ensuring that all data flows meet their end-to-end deadlines. The resulting constrained optimization problem based on existing delay bounds for data flows in WirelessHART networks is difficult since it is non-differentiable, non-linear, and not in closed-form. We propose four approaches to solve this challenging problem: (1) a subgradient method, (2) a simulated annealing (SA) based penalty method, (3) a time-efficient greedy heuristic method, and (4) a convex optimization method based on a new delay bound that is convex and differentiable. We study SA method under both constant factor penalty and



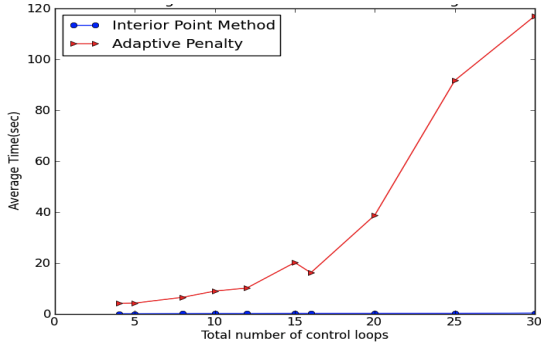
(a) Control cost (using 4 channels)



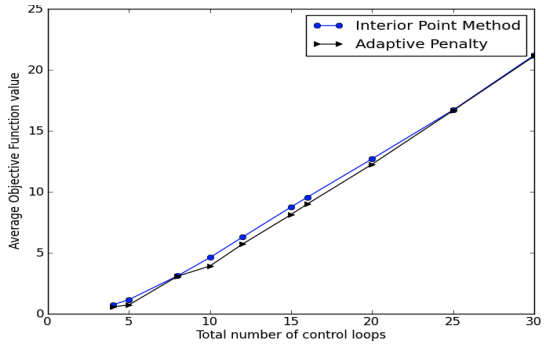
(b) Execution time (using 4 channels)



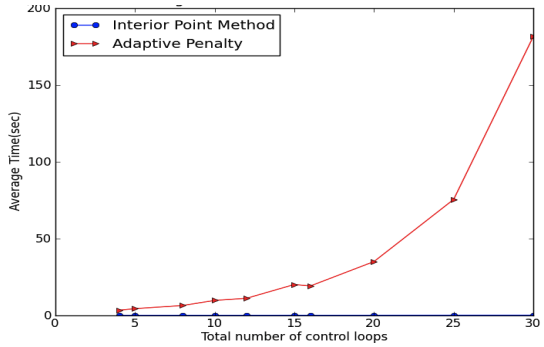
(c) Control cost (using 8 channels)



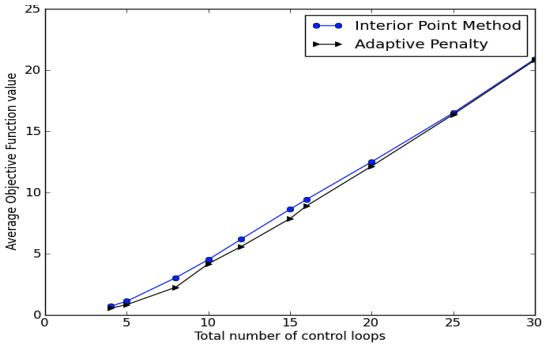
(d) Execution time (using 8 channels)



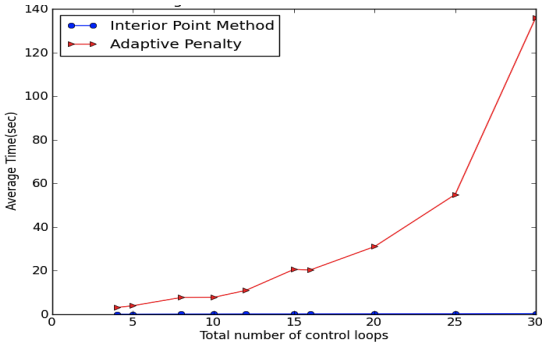
(e) Control cost (using12 channels)



(f) Execution time (using 12 channels)



(g) Control cost (using 16 channels)



(h) Execution time (using 16 channels)

Figure 6.11: Interior Point Method versus Adaptive Penalty Method

adaptive penalty. To solve the reduced convex optimization problem, we study both the gradient descent method and the interior point method.

We then perform a simulation study of the different approaches based on real testbed topologies and simulated control systems. Interestingly, while subgradient methods are commonly adopted to solve non-linear constrained optimization problems, it leads to the highest control cost *and* significant computation times in solving our optimization problem. We found that it is due to a large number of local minima and high nonlinearity of our problem. Among the SA based constant penalty method, the greedy heuristic, and the gradient descent method, the first one consistently achieves the minimum control cost while incurring the longest execution time. Conversely, the second one results in higher control cost using the shortest execution time. The third one that solves our reduced convex optimization problem based on our new delay bound hits the balance between control cost and execution time.

We further observe that the SA based adaptive penalty method is superior to the constant penalty method, and that the interior point method is superior to the gradient method. Thus the interior point method and the SA based adaptive penalty method are the two most effective approaches for rate selection. While these two methods are competitive against each other in terms of control cost, the interior point method is significantly faster than the penalty method. The execution time is an important metric in wireless control systems because the wireless networks in industrial environments are subject to network dynamics, and the network manager may need to recalculate the sampling rates frequently. As a result, the interior point method upon convex relaxation is more suitable than the SA based adaptive penalty method due to their significant difference in run-time efficiency, especially for online rate adaptation in response to changes in the workload and wireless conditions. Our results represent a promising step towards wireless-control co-design involving complex interactions between control performance and real-time communication.

Acknowledgements

This research was supported by NSF under grants CNS-1144552 (NeTS), CNS-1035773 (CPS), CNS-1017701 (NeTS), CNS-0708460 (CRI), a Microsoft Research New Faculty Fellowship, and a Sloan-Kettering Center grant.

Chapter 7

Distributed Channel Allocation Protocols for Wireless Sensor Networks

Interference between concurrent transmissions can cause severe performance degradation in wireless sensor networks (WSNs). While multiple channels available in WSN technology such as IEEE 802.15.4 can be exploited to mitigate interference, channel allocation can have a significant impact on the performance of multi-channel communication. This paper proposes a set of distributed protocols for channel allocation in WSNs with theoretical bounds. We first consider the problem of minimizing the number of channels needed to remove interference in a WSN, and propose both receiver-based and link-based distributed channel allocation protocols. Then, for WSNs with an insufficient number of channels, we formulate a fair channel allocation problem whose objective is to minimize the maximum interference (MinMax) experienced by any transmission link in the network. We prove that MinMax channel allocation is NP-hard, and propose a distributed link-based MinMax channel allocation protocol. Finally, we propose a distributed protocol for link scheduling based on MinMax channel allocation that creates a conflict-free schedule for transmissions. The proposed decentralized protocols are efficient, scalable, and adaptive to channel condition and network dynamics. Simulations based on the topologies and data traces collected from a WSN testbed of 74 TelosB motes have shown that our channel allocation protocols significantly outperform a state-of-the-art channel allocation protocol.

7.1 Introduction

Interference between concurrent transmissions can cause severe performance degradation in wireless sensor networks (WSNs). Multi-channel communication is an attractive approach to reducing interference and enhancing spatial reuse. Since channels are a scarce resource in a WSN, channel allocation significantly influences the performance of multi-channel WSNs. It is particularly important in Time Division Multiple Access (TDMA) based WSNs where interfering transmissions scheduled at the same time slot must be assigned different channels. It is, therefore, important to allocate the channels to reduce interference and increase the number of concurrent transmissions.

Channel allocation has been widely studied for wireless ad-hoc networks. These protocols are not applicable to WSNs since the applications, routing, node resources, and network structure in WSNs are quite different from traditional ad-hoc networks. While channel allocation has also been studied for WSNs, most of them focus on simple heuristics [106, 113, 114] without any performance guarantee for channel hopping or just focus on centralized solutions [84, 175].

In this paper, we formulate optimal channel allocation as constrained optimization problems, and propose a set of distributed channel allocation protocols with theoretical bounds for WSNs. We first consider the problems of minimizing the number of channels needed to remove interference in a WSN for both receiver-based and link-based channel allocation. A receiver-based channel allocation is suitable for both CSMA/CA and TDMA protocols. A link-based channel allocation allows better spatial reuse due to the flexibility in assigning different channels to different senders, but it is more suitable for TDMA protocols under which the receiver can switch to channels according to the expected sender scheduled in each time slot. We present distributed protocols for both receiver-based and link-based channel allocation.

WSNs usually have a moderate number of channels (e.g., 16 channels specified IEEE 802.15.4), and noisy environments may further reduce the number of available channels due to black-listing [22]. Therefore, there may not exist enough channels to remove all interference. Existing works on channel allocation with an insufficient number of channels usually consider receiver-based allocation and propose centralized heuristics [84, 175, 184]. A recently

proposed distributed protocol for channel allocation in WSNs has addressed receiver-based allocation to minimize total interference suffered by all receivers [187]. In contrast, we formulate a link-based fair channel allocation problem whose objective is to minimize the maximum interference (MinMax) experienced by any transmission link in a WSN. The key advantage of the MinMax objective is that it can mitigate bottlenecks in a WSN where a node or link experiences excessive interference. We prove that MinMax channel allocation is NP-hard, and propose a distributed MinMax channel allocation protocol. Furthermore, since channel allocation cannot always resolve all transmission conflicts due to an insufficient number of channels, it is complemented by a time slot assignment algorithm to create a conflict-free schedule. We propose a distributed protocol for link scheduling based on MinMax channel allocation. Our contributions are:

- We present distributed protocols for both receiver-based and link-based interference-free minimum channel allocation.
- We formulate a link-based fair channel allocation problem, called MinMax channel allocation, whose objective is to minimize the maximum interference experienced by any transmission link in a WSN, and prove it to be NP-hard.
- We propose a distributed protocol for MinMax channel allocation in WSN.
- We propose a distributed protocol for link scheduling based on MinMax channel allocation.

The proposed algorithms are efficient, scalable, and adaptive to channel condition and network dynamics. We provide the time complexity and performance bound of each algorithm. Simulations using the real topologies and data traces collected from a WSN testbed have shown that our protocols significantly outperform a state-of-the-art protocol [187].

In the rest of this paper, Section 7.2 reviews related work. Section 7.3 describes the network model. Section 7.4 formulates the problems. Sections 7.5 and 7.6 present the proof of NP-hardness and the distributed protocols for interference-free and MinMax channel allocation, respectively. Section 7.7 presents the link scheduling protocol. Section 7.8 presents evaluation results. Section 7.9 concludes the paper.

7.2 Related Work

Multi-channel MAC protocols have been extensively studied for wireless ad-hoc network [26, 30, 31, 37, 46, 47, 59, 67, 72, 83, 89, 107, 111, 121, 133–135, 137, 146, 147, 149–151, 162, 164, 171–173, 180, 185, 186]. However, there are some key differences between these existing protocols for traditional wireless ad-hoc network and the channel allocation protocols proposed in this paper for WSN as detailed below.

First, the protocols in [26,31,59,67,72,83,107,111,121,135,137,146,147,151,162,173,180,185, 186] assume that the hardware is able to listen to multiple channels simultaneously. But each sensor device is usually equipped with a single radio transceiver (e.g., TelosB mote [19] with Chipcon CC2420 radio) that cannot transmit and receive at the same time, and cannot operate on different channels simultaneously. Second, the protocols in [30,31,89,107,133,146,151] involve heavy centralized computation such as linear programming [31, 107], mixed integer linear programming [30], and subgradient method [89, 133]. But a WSN has limited bandwidth (e.g. 250kbps in 802.15.4 network), and each sensor device has limited memory (e.g. 10KB in TelosB motes [19]) and limited processing power (8MHz MSP430 microcontroller in TelosB motes), making a WSN unsuitable for such heavy-weight computations. Third, the protocols in [37,149,164,171] use RTS/CTS for channel negotiation. But, due to limited bandwidth in WSNs, the MAC layer packet size in WSNs is much smaller (typically 30∼50 bytes) than that of general ad hoc networks (typically 512+ bytes). Hence, RTS/CTS control packets result in significant overhead for WSN, thereby making these protocols unsuitable for WSN.

Graph theory based multi-channel protocols for wireless ad-hoc networks studied in [46, 47, 72, 83, 134, 135, 150, 172, 186] are most related to our work. The protocols in [72, 83, 134, 135, 150, 186] are distributed but assume that each node can listen to multiple channels simultaneously (as already discussed), while those in [46, 47] are based on single radio in each node but consider centralized solutions. The work in [172] that uses a distributed approach considering single radio in each node, and the work in [83] that uses a game theoretic approach are particularly related to our work. But, both works focus on maximizing link data rate instead of interference-free minimum channel allocation or minimizing the maximum interference, which are our focus.

In summary, none of the above protocols is applicable for WSNs since the applications, routing, and network structure in WSNs are quite different from traditional ad-hoc networks. For example, in contrast to traditional ad hoc networks designed to support general communication patterns and routes, WSNs are typically involved in monitoring applications requiring data collection with unique communication patterns and routing structures. Sensor nodes are prone to failures, and the network topology changes more frequently. Besides, sensor nodes mainly use broadcast communication paradigms whereas most traditional ad-hoc networks are based on point-to-point communications. In a WSN, nodes are usually densely deployed, and the number of nodes can be several orders of magnitude higher than that in a traditional wireless ad-hoc network.

Channel allocation has also been studied for WSN in recent years. MMSN [193] is an early multi-channel protocol proposed for WSN. MMSN ignores routing information for channel allocation. In contrast, we propose routing-aware channel allocation protocols that do not assign channels to the links not involved in traffic. Tree-based Multi-Channel Protocol (TMCP) proposed in [181] uses the distance-based interference model which does not hold in practice as shown by recent empirical studies [184]. TMCP has been extended in [184] to employ inter-channel RSS models for interference assessment in channel allocation [184]. All these protocols are *centralized*, and lack any performance bound. The protocols proposed in [106, 113, 114] use simple heuristics for channel hopping. These protocols do not address interference-free minimum channel allocation or minimizing the maximum interference, which are the focus of our work in this paper.

Interference-aware channel allocation based on graph-theory has been studied in [63, 84, 175] for WSN. But the work in [63] is designed for unit-disk graph. The work in [175] assigns a channel to each flow. The work in [84] shows that minimizing schedule length for multi-channel arbitrary network is NP-hard, and presents a constant factor approximation algorithm for unit-disk graph [84]. These algorithms are *centralized*. Due to frequent topology changes, distributed protocols are more suitable for WSNs. A distributed game theory based protocol has been proposed in [187] for channel allocation in WSN. It addresses only receiver-based allocation, and minimizes total interference suffered by all receivers.

In contrast to existing channel allocation protocols for WSNs, we present distributed protocols for both receiver-based and link-based interference-free minimum channel allocation.

The key novelty of our work lies in formulating a link-based fair channel allocation problem, called MinMax channel allocation, whose objective is to minimize the maximum interference experienced by any transmission link in a WSN. In addition, we prove that an optimal MinMax channel allocation is NP-hard. Furthermore, we propose a distributed protocol for MinMax channel allocation based on heuristic. We also propose a distributed protocol for link scheduling based on MinMax channel allocation. The key advantage of the MinMax objective is that it can mitigate bottlenecks in a WSN where a node or link experiences excessive interference.

7.3 Network Model

A WSN consists of a set of sensor nodes. A node, called the *base station*, serves as the sink of the network. A *communication link* $e = (u, v)$ indicates that the packets transmitted by node u may be received by v. We assume that every communications link is symmetric. This assumption holds for WSNs relying on acknowledgement for reliable communication (e.g., WirelessHART networks [22] based on IEEE 802.15.4). An *interference link* $e = (u, v)$ indicates that u's transmission interferes with any transmission intended for v even though u's transmission may not be successfully received by v. Thus, any two concurrent transmissions that happen on the same channel are *conflicting* if there is an interference link from one's sender to the other's receiver. Several practical protocols [124, 129] exist that model interference in WSNs using Signal-to-Noise plus Interference Ratio (SNIR). A set of transmissions on the same channel is conflict-free if the SNIR of all receivers exceeds a threshold. For example, RID [192] is a distributed protocol for determining interference links in a WSN based on Received Signal Strength (RSS) measurements.

We model a WSN as an *Interference-Communication* (IC) graph, a notion introduced in [61]. In the IC graph $G = (V, E)$, V is the set of sensor nodes (including the sink s); E is the set of communication or interference links between the nodes. A subset of the communication links forms the routing tree that is used for data collection at the sink. Let $E_T \subseteq E$ denote the set of links in the routing tree. Any link $e = (u, v)$ in E_T indicates that v is the parent of u. For any node u, we use p_u to denote its parent in the routing tree. Since the transmissions along non-tree links do not aim at the receiver, every non-tree link (that is not a part of the

routing tree) is an *interference link*. $E_I = E - E_T$ is the set of all interfering links in the IC graph. Any link $e = (u, v)$ in E_I indicates an interference link from u to v. A node cannot both send and receive at the same time, nor can it receive from more than one sender at the same time. The set of channels available in the WSN is denoted by M. We use m to denote $|M|$ i.e. the total number of channels. The channels are numbered through 1 to m. In this work, we particularly focus on TDMA based WSN.

7.4 Problem Formulation

In *receiver-based channel allocation*, each sensor node is assigned a fixed channel to receive message; the neighbors which have messages to deliver to it should use this channel to send. In this allocation, the leaves (i.e., nodes without children) in the routing tree do not receive any message, and hence are not assigned any channel. Let the nodes that receive message (i.e., the nodes other than leaves) be denoted by $R \subset V$. Therefore, the *receiver-based channel allocation* is a function $f : R \mapsto M$, where M is the set of channels.

In *link-based channel allocation*, every link $e \in E_T$ is assigned a channel so that every transmission along that link happens on that channel. In contrast to receiver-based channel allocation, here for the same receiver, different senders can use different channels, thereby providing more flexibility in avoiding interference. Any *link-based assignment* is a function $f : E_T \mapsto M$. Since every node has unique sending link, a link-based channel assignment function can also be defined as $f : V - \{s\} \mapsto M$, where s is the root (i.e., the sink) of the routing tree and it does not send to anyone. Thus every sender in the network is assigned a channel. For reception, the receiver uses the same channel that the sender uses to transmit.

Interference caused by siblings (in the routing tree) to each other cannot be resolved by channel assignment because the shared parent cannot receive from more than one of them at the same time. This can be resolved through a time-slot assignment. Therefore, for channel allocation purpose, we are concerned only about interference through non-tree links E_I (that are not parts of the routing tree), and simply use the term 'conflict' to denote the interference through these links. In the worst-case, the maximum number of transmissions that can be conflicting through interference links with a transmission along link (u, v) is equal to the total number of incoming interference links of v and outgoing interference links of u. Thus,

we define *conflict of transmission link* (u, v) or *conflict of node* u as the maximum number of transmissions that can be conflicting through interference links with a transmission of node u. For a node u, in a channel assignment f, we use $C(u, f)$ to denote its conflict, and define as follows (where p_u is the parent of u):

$$C(u, f) = \big|\{z|\big((z, p_u) \in E_I \vee (u, p_z) \in E_I\big) \wedge f(z) = f(u)\}\big|$$

That is, $C(u, f)$ counts the total number of nodes that use the same channel as u's and that has either an outgoing interfering link to the parent of u or an incoming interfering link to its parent from u. The interference a node receives is not only decided by the number of interference sources, but also by the strengths of interfering signals. However, to develop an efficient and distributed approach, we consider the metric $C(u, f)$ as the signals can be determined based on a threshold on signal strength. $C(u, f)$ is an effective metric and can be used for effective channel allocation because the total number of interfering signals has strong correlations with transmission failures and retries. The higher the value of $C(u, f)$, the more transmissions that u's transmission may conflict with. Namely, the more interfering links a receiver hears, the more retries a message needs to be successfully received by that receiver, thereby incurring longer delay. For example, in [187], the total number of interfering links a receiver hears was shown to be approximately linear with the total number of retries a message needs to be successfully received by that receiver.

Problem 1: Receiver-based interference-free channel allocation. The number of channels is usually fixed and limited in practice. Our first objective is to minimize the total number of channels to remove all interferences in the IC graph $G = (V, E)$. Let $f(R)$ denote the range of function $f : R \mapsto M$, i.e., the set of channels used in f. In *receiver-based interference-free channel allocation*, our objective is to determine a channel assignment $f : R \mapsto M$ so as to

$$\begin{array}{ll} \text{Minimize} & |f(R)| \\ \text{subject to} & C(u, f) = 0,\ \forall (u, v) \in E_T \end{array}$$

Problem 2: Link-based Interference-free channel allocation. While receiver-based channel allocation is simple in the sense that a receiver can avoid switching to different

channels for different senders, it can end up with extra interference for some transmission link, thereby limiting the communication possibilities for some nodes. Such a limitation of receiver-based channel allocation can be significantly overcome by adopting link-based allocation. In *link-based interference-free channel allocation*, our objective is to determine a channel assignment $f : V - \{s\} \mapsto M$ to

$$\begin{aligned} &\text{Minimize} \quad |f(V - \{s\})| \\ &\text{subject to} \quad C(u, f) = 0, \ \forall u \in V - \{s\} \end{aligned}$$

Problem 3: Minimizing Maximum interference (MinMax) channel allocation. The number of channels required to remove all interference may be greater than the total available channels. Therefore, when the available channels are not sufficient to remove all interference, a fair channel allocation is the one that minimizes the maximum interference experienced by any transmission link in G. Since link-based channel allocation allows better spatial reuse of channels, we use link-based allocation for MinMax objective. In MinMax channel allocation, our objective is to determine a link-based channel assignment $f : V - \{s\} \mapsto M$ so as to

$$\begin{aligned} &\text{Minimize} \quad \max\{C(u, f) | u \in V - \{s\}\} \\ &\text{subject to} \quad f(u) \in M, \ \forall u \in V - \{s\} \end{aligned}$$

Problem 4: Link scheduling. After MinMax channel allocation, a conflict-free schedule is required to avoid transmission conflicts through both tree (transmission) links and the residual interference links. This needs to be resolved through time slot assignment. That is, after channel allocation in phase 1, we consider the link scheduling in phase 2. While it may be possible to combine two phases into one, such an approach complicates the optimization problem as the solution space becomes larger. Instead, decoupling it into two phases simplifies the optimization problem for conflict resolution. Hence, in our solution approach, channel allocation is done in the first phase, which is followed by a time slot assignment in the second phase. In TDMA, a transmission needs one time slot, and a sequence of time slots forms a *frame*. The frame is repeated continuously. Every link is assigned a relative time slot within a frame and it is activated at that slot of the frame. Therefore, here

our objective is to schedule all links to minimize the frame length. Thus, for link scheduling, after MinMax channel allocation, our objective is to determine a time slot assignment $g : E_T \mapsto \{1, 2, 3, \cdots\}$ so as to

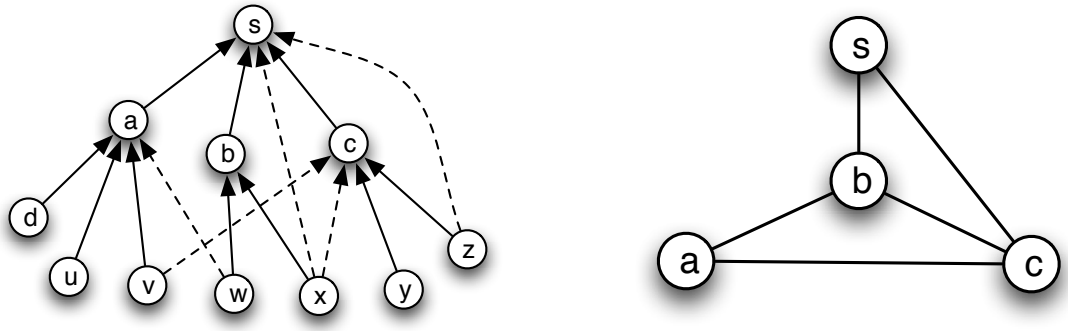
$$\text{Minimize} \quad |g(E_T)|$$

7.5 Interference-free Channel Allocation

7.5.1 Receiver-based Channel Allocation

We first consider receiver-based channel allocation to minimize the number of channels to eliminate all interference. This problem has been proven to be NP-hard in [84]. In the following, we provide a distributed algorithm based on *vertex-coloring* for this problem.

Two receivers are called *interfering* if the transmission of some child of one receiver is interfered by the transmission of some child of the other receiver. In order to eliminate all interference, every receiver must be assigned a channel that is different from all of its interfering receivers' channels. Therefore, for the given IC graph $G = (V, E)$, we can assume a *receiver-based conflict-graph*, denoted by $G_R = (R, E_R)$, that consists of all receivers R as nodes, and an edge (in E_R) between every interfering receiver pair. For example, Figure 7.1(b) shows the receiver-based conflict-graph of the IC graph of Figure 7.1(a). In an IC graph, we use dotted lines and solid lines to indicate interference links and transmission links, respectively. Considering every *channel* as a *color*, vertex-coloring of G_R provides the solution for receiver-based interference-free channel allocation in G to minimize the number of channels (colors). To construct the conflict graph, each node needs to know the channel conditions between itself and other nodes. This requirement can be met in practice, even in scenarios with fast channel fading or network dynamics, since the fast-fading and fluctuating channels can be blacklisted at deployment time and infrequent maintenance cycles in wireless sensor networks, e.g., as specified in the WirelessHART standard for industrial wireless sensor networks [22]. Our approach can leverage existing algorithms specifically designed to detect conflict graphs efficiently in wireless sensor networks [124, 192]. These methods use RSS measurements and determine and store conflict graphs in a distributed fashion: a node



(a) IC graph G (b) Conflict graph G_R

Figure 7.1: IC graph and receiver-based conflict graph

only knows its incoming/outgoing communication and interference edges based on SNIR. Hence, conflict graph construction method is distributed and efficient in practice.

For vertex-coloring in distributed manner, the best known deterministic algorithm [40] employs $D^{1+O(1)}$ colors, where D is the maximum degree of the given graph. The distributed methods for vertex-coloring available in the literature of theoretical computer science [40] involve multiple phases. A phase starts only after its previous phase converges. Since the WSN devices are characterized by low power and resources, these algorithms are too heavy-weight for WSNs. Here we present a simple and deterministic distributed protocol suitable for WSNs, which can employ at most $\Delta_R + 1$ channels, with Δ_R being the degree of the receiver-based conflict graph.

Let $N_R(u)$ denote the neighbors of node u in G_R. In our distributed method, every node $u \in R$ has to communicate with its neighbors $N_R(u)$ in G_R. Note that two neighbors u and v in G_R may not be one-hop away from one another in IC graph G. In such cases, u and v communicate with one another by increasing their transmission power like what is done in [61, 192]. If this is not possible, communication between u and v is done through the end-to-end route between u and v. Channel allocation is done iteratively and every round consists of communication between the neighbors in G_R. In every communication round, all nodes use the same channel. Once the algorithm converges, every node uses the channel determined by the algorithm for subsequent communication. The distributed

receiver-based interference-free channel allocation protocol consists of the following steps comprising a procedure that is invoked iteratively:

1. In the beginning, every node $u \in R$ is assigned channel 1 (the smallest numbered channel). In every round, each node $u \in R$ broadcasts a message containing its ID and chosen channel to its neighbors $N_R(u)$.

2. Considering the current channel allocation among neighbors $N_R(u)$, every node u repeatedly switches to the smallest channel not used by any of its neighbors. Two neighbors cannot switch channels simultaneously. If two neighbors in G_R want to switch at the same time, the node with the smallest ID wins (as a local agreement among neighbors) and switches channel.

3. After choosing the channel, each node u broadcasts its chosen channel in a message (that also contains its ID) to its neighbors $N_R(u)$ in G_R.

4. The procedure is repeated until every node has chosen a channel different from its neighbors in G_R and cannot choose a channel that is smaller than its current channel.

As stated before, two neighbors in G_R do not execute the procedure simultaneously.

time \ node	a	b	c	s
Round 1	1	1	1	1
Round 2	2	1	1	2
Round 3	2	3	1	2

Table 7.1: Channels selected in different rounds by the receiver nodes in Receiver-based channel assignment

The above algorithm converges when every node in G_R has chosen a channel different from those of its neighbors $N_R(u)$, and cannot switch to a smaller channel. In every round, the total number of messages that are sent or received by a node u is $O(|N_R(u)|)$. Theorem 17 proves the convergence of the algorithm. Theorem 18 shows that the algorithm requires at most $\Delta + 1$ channels, where Δ is the maximum degree of G. For the network shown in Figure 7.1(a), the channels selected by the nodes in different rounds (up to the convergence) of Receiver-based channel assignment are shown in Table 7.1 for any $m > 3$.

Theorem 17. *Receiver-based interference-free channel allocation algorithm converges in $|E_I|$ rounds, where $|E_I|$ is the total number of interfering links in G.*

Proof. Until the algorithm converges, in every round at least one node switches its channel that is different from its neighbors in the receiver-based conflict graph G_R. If a node u switches to a channel that is different from its neighbors' channels, the interference links between u and its neighbors $N_R(u)$ are removed. Since no two neighbors in G_R switch channels in the same round, at least one interfering link in G is removed in every round. Since the total interfering links in G is $|E_I|$, the algorithm converges in at most $|E_I|$ rounds. □

Theorem 18. *Receiver-based interference-free channel allocation algorithm requires at most $\Delta + 1$ channels, where Δ is the maximum degree in G.*

Proof. Let Δ_R be the maximum degree of G_R. The channels are numbered 1,2, $\cdots$ in increasing order. Every node initially has channel 1. Every time a node switches channel, it switches to the smallest channel not used by the neighbors. Hence, the largest possible channel to which a node can switch is $\Delta_R + 1$, which happens if all first Δ_R channels are chosen by its neighbors in G_R. Hence, the algorithm employs at most $\Delta_R + 1$ channels. Since $\Delta \geq \Delta_R$, the theorem follows. □

7.5.2 Link-based Channel Allocation

Receiver-based allocation can end up with extra interference for some transmission link, thereby limiting the communication possibilities for some nodes. As a result, when all transmission conflicts are completely resolved through a time slot assignment phase, the schedule length becomes longer if a receiver-based allocation is adopted. This limitation can be significantly overcome by adopting a link-based allocation since it allows better spatial reuse. This is illustrated in Figure 7.2 through a simple example considering m= 2. The number in the rectangle beside every receiver shows its assigned channel. Under this receiver-based allocation, every time node w transmits, none of a's children should transmit. This problem can be avoided using a link-based channel allocation instead (as shown beside the links) by assigning channel 1 to node w, and channel 2 to node x.

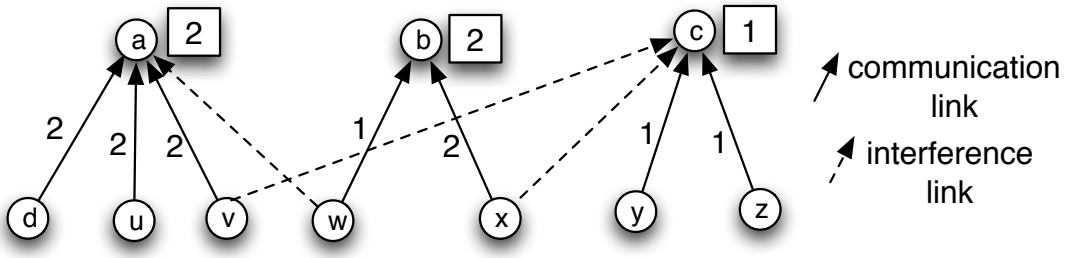


Figure 7.2: Link-based channel allocation

A reduction similar to the one used in [84] (that proves that receiver-based interference-free channel assignment is NP-hard) can also be used to prove that link-based interference-free channel allocation is NP-hard as shown in Theorem 19.

Theorem 19. *Given a routing tree T on an IC graph $G = (V, E)$, and a total of m channels, it is NP-complete to decide whether there exists some channel allocation f to the links in T such that G becomes interference-free.*

Proof. The problem is in NP since, given an instance of the problem, we can verify in $O(|E|)$ time whether the network is interference-free. Following reduction from vertex-coloring implies NP-hardness. Given any instance $\langle \mathbb{G}, k \rangle$ of the vertex-coloring problem in graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, we create a sink node s as the parent of every $u \in \mathbb{V}$, and create a child for every u. Now for every edge $(u, v) \in \mathbb{E}$, we create an interfering link between u's child and v (or between v's child and u). This constructs an IC graph $G = (V, E)$. A channel allocation f uses the color of $u \in \mathbb{V}$ as the channel of u's child, and uses any channel c, $1 \leq c \leq k$, for u in G. Thus, $\mathbb{G}$ can be vertex-colored with k colors if and only if f can remove all interference links from G using k channels. □

Now we present a distributed algorithm for link-based channel allocation to minimize the number of channels in order to eliminate all interfering links. This approach is also similar to the distributed vertex-coloring adopted for receiver-based allocation in the previous subsection.

Two senders in G are called *interfering* if one's transmission is interfered by the other. In order to eliminate all interference, every sender's transmission link must be assigned a

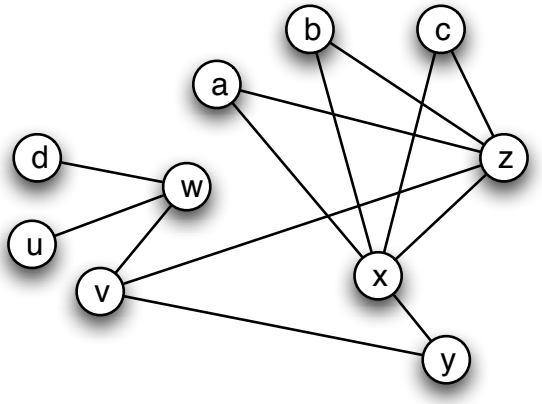


Figure 7.3: Link-based conflict graph G_L of G

channel that is different from those of its interfering senders. Therefore, for the IC graph $G = (V, E)$, we can assume a *link-based conflict-graph*, denoted by $G_L = (V - \{s\}, E_L)$, that consists of all senders $V - \{s\}$ as nodes, and an edge (in E_L) between every interfering sender pair. For example, Figure 7.3 shows the link-based conflict-graph of the IC graph of Figure 7.1(a). Considering every *channel* as a *color*, vertex-coloring of G_L provides the solution for link-based interference-free channel allocation in G to minimize the number of channels (colors).

Using the same distributed algorithm as the one used for receiver-based channel allocation in the preceding subsection, we now vertex color graph G_L. As stated before, two neighbors in G_L do not execute the algorithm simultaneously. If two neighbors in G_L want to execute simultaneously, the node with the smallest ID wins (as a local agreement among neighbors) and executes to switch its channel. When the entire distributed algorithm converges, every sender (i.e., every sender's transmission link) is assigned a channel that is different from any interfering sender's channel. This algorithm converges within $|E_I|$ rounds, and employs at most $\Delta_L + 1$ channels, where Δ_L is the maximum degree in G_L (proofs are similar to those of Theorems 17 and 18). For the network shown in Figure 7.1(a), the channels selected by the nodes in different rounds (up to the convergence) of Link-based channel assignment are shown in Table 7.2 for any $m > 3$.

time \ node	a	b	c	d	u	v	w	x	y	z
Round 1	1	1	1	1	1	1	1	1	1	1
Round 2	2	2	2	2	2	2	1	1	1	1
Round 3	2	2	2	2	2	2	1	3	1	1

Table 7.2: Channels selected in different rounds by the sender nodes in Link-based channel assignment

7.6 MinMax Channel Allocation

Note that WSNs usually have a moderate number of channels (e.g., 16 channels for WSNs based on IEEE 802.15.4), and noisy environments may further reduce the number of available channels due to blacklisting [22]. Therefore, there may not exist enough channels to remove all interference using the algorithms presented in the previous section. In such a situation, we adopt MinMax channel allocation whose objective is to minimize the maximum interference experienced by any transmission link across the network. Since receiver-based allocation may not minimize the maximum interference experienced by a transmission link (Subsection 7.5.2), we follow a link-based approach for MinMax channel allocation.

We first prove that MinMax allocation is NP-hard by showing that its decision version is NP-complete.

Theorem 20. *Given a routing tree T on an IC graph $G = (V, E)$, m channels, and an integer k, it is NP-complete to decide if there exists a channel allocation f to the links in T such that the maximum conflict in G is at most k.*

Proof. When $k = 0$, the decision problem of the theorem represents a decision version of the link-based interference-free channel allocation (Problem 2) that has been proved to be NP-complete in Theorem 19. Thus, this decision problem is a generalization of that of Problem 2 and, hence, is NP-complete. □

Now we present a distributed algorithm for MinMax channel allocation. In the protocol, every node needs to communicate with its neighbors in link-based conflict graph G_L (see Subsection 7.5.2 and Figure 7.3 for G_L) to compute its conflict. For any node u, the set

of its neighbors in G_L is denoted by $N_L(u)$. Communication in the neighborhood in G_L is done based on the same approach presented in the previous section. Distributed MinMax algorithm consists of the following procedure that is invoked iteratively:

1. Before the invocation of the procedure, every node $u \in V - \{s\}$ is assigned a random channel in the range between 1 and m. Every node $u \in V - \{s\}$ broadcasts a message containing its ID and channel to its neighbors $N_L(u)$ in G_L.

2. Considering the current channel allocation among the neighbors in G_L, every node calculates its conflict $C(u, f)$ and broadcasts again to the neighbors $N_L(u)$.

3. For each node u, once it receives the message containing $C(v, f)$ from each neighbor v in G_L, node u calculates its conflict $C(u, f)$ on every channel. Any channel used by a neighbor v with $C(v, f) > C(u, f)$ is considered unavailable at u. That is, node u excludes all channels used by the neighbors with higher conflicts in the current round. This is done because switching to such a channel increases the neighbor's conflict which may increase the maximum conflict in the network. Among the available channels, node u switches to the channel that results in the smallest $C(u, f)$, breaking ties arbitrarily. Two neighbors cannot switch channels simultaneously. If two neighbors want to switch at the same time, the node with the smallest ID wins.

4. After choosing the channel, every node broadcasts its chosen channel to its neighbors in G_L.

5. The procedure repeats as long as some node u can decrease $C(u, f)$ using its available channels.

As stated before, two neighbors in G_L do not execute the procedure simultaneously. If two neighbors in G_L want to execute simultaneously, the node with the smallest ID wins (as a local agreement among neighbors) and executes to switch its channel.

In each communication round, all nodes use the same channel for communication. Once the algorithm converges, every node uses the channel determined by the algorithm for subsequent communication. Each node u needs to send or receive $O(|N_L(u)|)$ messages in a round. The algorithm converges when no node can decrease its conflict using its available channels.

time \ node	a	b	c	d	u	v	w	x	y	z
Round 1(random)	2	1	2	1	1	1	2	1	1	1
Round 2	2	2	2	1	1	2	2	1	1	1

Table 7.3: Channels selected in different rounds by the sender nodes in MinMax channel assignment when $m = 2$

Theorem 21 proves its convergence. For the network shown in Figure 7.1(a), the channels selected by the nodes in different rounds (up to the convergence) of Link-based channel assignment are shown in Table 7.3 considering $m = 2$.

Theorem 21. *MinMax Channel Allocation converges in $|E_I|$ rounds, where $|E_I|$ is the total number of interfering links in G.*

Proof. Since MinMax algorithm is repeated as long as some node u can decrease its $C(u, f)$ using its available channels, in every round at least one node switches its channel. Assuming the neighbors of u in G_L keep their channels unchanged, changing the channel of u that decreases $C(u, f)$ implies that the total number of interference links between u and its neighbors decreases. Since no two neighbors in G_L switch channel simultaneously, at least one interfering link in G is removed in every round. Hence, similar to Theorem 17, the algorithm converges in at most $|E_I|$ rounds. □

Theorem 22. *Upon MinMax Channel Allocation, the maximum conflict in G is at most $\lfloor \frac{C_{\max}}{m} \rfloor$, where $C_{\max}$ is the maximum conflict in G under single channel.*

Proof. Let $d(u)$ denote the degree of node u in link-based conflict graph G_L of G. The value $d(u)$ is equal to the conflict of u under single channel. We first prove that, when MinMax Algorithm converges, at most $\lfloor \frac{d(u)}{m} \rfloor$ neighbors in G_L can have the same channel as the one assigned to u, for any node u. Suppose to the contrary, after the algorithm converges, there exists some node v such that $\lfloor \frac{d(v)}{m} \rfloor + 1$ of its neighbors in G_L have the same channel as the one assigned to v. Let c be the channel assigned to v, and $Z \subseteq N_L(v)$ be the neighbors of v in G_L that have been assigned channel c. Now according to *the pigeon-hole principle*, there must be at least one channel $c' \neq c$ such that at most $\lfloor \frac{d(v)}{m} \rfloor$ neighbors of v have been assigned channel c'. If $\exists z \in Z$ such that $C(v, f) \leq C(z, f)$, then z will switch to channel c'

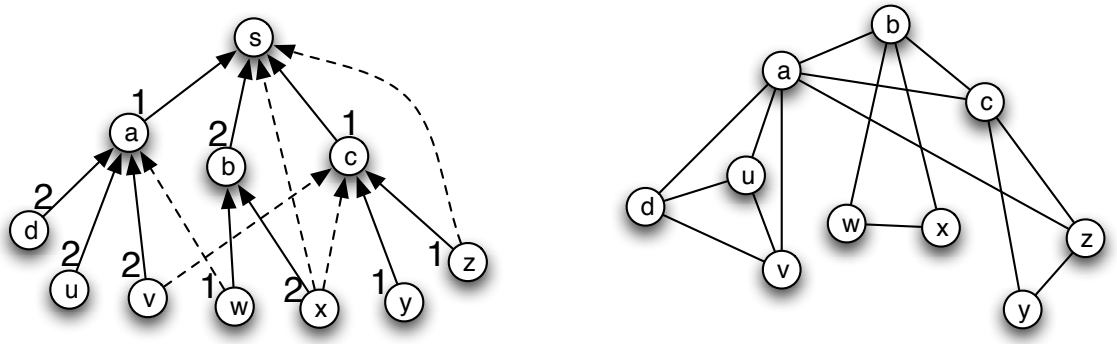
since it can decrease its $C(z, f)$. If $C(z, f) \leq C(v, f)$, then v will switch to channel c' since it can decrease its $C(v, f)$. Both cases contradict with the hypothesis that the algorithm has converged. Therefore, when MinMax Algorithm converges, at most $\lfloor \frac{d(u)}{m} \rfloor$ neighbors in G_L can have the same channel as the one assigned to u, for any node u. Since $C_{\max}$ is equal to the maximum degree in G_L, the theorem follows. □

The key advantage of the MinMax objective is that it can mitigate bottlenecks in a WSN where a node or link experiences excessive interference. The simulation results (presented in Section 7.8) indicate that the MinMax objective is more effective than minimizing the total interference in the network in terms of critical network metrics such as latency.

7.7 Distributed Link Scheduling

Note that channel allocation cannot resolve all transmission conflicts in a WSN due to two reasons. First, the number of available channels is limited and may not suffice to remove all interference. Second, each WSN device is equipped with a half-duplex radio that prevents a node from both transmitting and receiving at the same time, and also prevents reception from two senders simultaneously. Therefore, a channel allocation is complemented by a time slot assignment. Namely, any two conflicting transmissions are assigned different time slots. While this can be achieved through a joint channel allocation and time slot assignment, performing channel allocation and time slot assignment in two different phases simplifies this optimization problem. In this section, we present a distributed algorithm for time slot assignment after MinMax channel allocation. Namely, we first perform MinMax channel allocation. Then, we perform a time slot assignment that avoids transmission conflicts through both tree links and the residual interference links to create a conflict-free schedule.

In the time slot assignment algorithm, every link is assigned a relative time slot in a frame, and the link is activated at that slot of the frame. The frame is repeated continuously. Note that, after MinMax channel allocation, the network can still be considered as a new IC graph with reduced interference. Therefore, a proof similar to Theorem 19 implies that scheduling all links to minimize the frame length is NP-hard. We provide a distributed method for time slot assignment that minimizes the frame length.



(a) G (b) Schedule conflict graph G_S

Figure 7.4: IC graph and schedule conflict graph

To resolve the conflict through both tree links and residual interference links after MinMax channel allocation, we determine a schedule conflict graph G_S of IC graph G as follows:

- Ignore all interfering links that are removed by MinMax channel allocation.
- Add links between siblings. The links between parent and children remain unchanged.
- For every interfering link (u, v) from u to v that still exists after channel allocation f, add a link from u to every child z of v with $f(z) = f(u)$.

For the IC graph G shown in Figure 7.1(a), let Figure 7.4(a) shows the channel allocation, where the number beside a sender shows its assigned channel. Then Figure 7.4(b) shows its schedule conflict graph G_S. In a TDMA schedule, any two nodes that are neighbors in G_S must be scheduled on different time slots. We use the same distributed algorithm as the one used for interference-free channel allocation. We run the algorithm considering schedule conflict graph G_S. Now, instead of channel, we allocate a time slot to every node in G_S. Every node starts with slot 1. In each round, the nodes switch to the smallest slot not assigned to any neighbor in G_S. The maximum time slot assigned to a node indicates the length of the frame, since the frame will repeat after this slot.

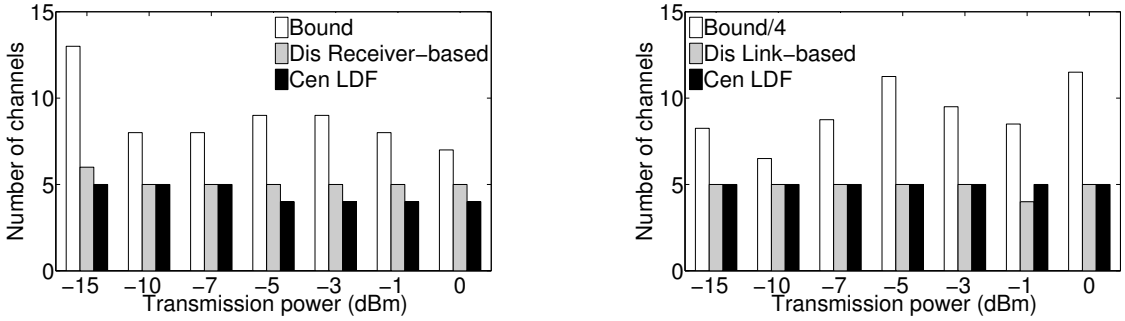
Theorem 23. *The frame length determined by the distributed link scheduling algorithm is at most* $\lfloor \frac{C_{\max}}{m} \rfloor + \Delta_T + 1$*, where* $C_{\max}$ *is the maximum conflict in* G *under single channel,* Δ_T *is the maximum degree of the routing tree.*

Proof. According to Theorem 18, the total time slots used in the frame is at most $\Delta_S + 1$, where Δ_S is the maximum degree in G_S. After MinMax channel allocation, $\Delta_S \leq \lfloor \frac{C_{\max}}{m} \rfloor + \Delta_T$. Hence, the bound follows. □

7.8 Evaluation

We evaluate our channel allocation and link scheduling protocols on the topologies of an indoor WSN testbed [2] spread over two buildings (Bryan Hall and Jolley Hall) of Washington University in St louis. The testbed consists of 74 TelosB motes each equipped with a Chipcon CC2420 radio compliant with IEEE 802.15.4. We have developed a discrete-event simulator that operates based on interference data traces collected from the testbed. The traces were obtained by having each node in the testbed take turns broadcasting a sequence of 50 packets. All nodes operated on channel 26 of IEEE 802.15.4. While the application transmits packets as soon as possible, the MAC layer applied for each transmission a randomized back-off uniformly distributed in the interval [10ms, 170ms]. The batch of 50 packets takes 4.5s on average to transmit. The remainder of the nodes recorded the Received Signal Strength (RSS) of the packets they receive. The short delay between the transmissions of packet pertaining to the same batch allows us to capture the short-term variability of RSS. We have collected 7 sets of data traces at 7 transmission (Tx) power levels: -15, -10, -7, -5, -3, -1, 0 dBm. Collecting the data traces over three consecutive days captured the long-term variability. RSS traces collected from the 74-node testbed are used to configure the simulations.

The network topologies used in the simulations are based on RSS traces collected from the testbed. We determine the communication and interference links between nodes as follows. A node A may communicate with a node B if node B's RSS average during A's transmissions exceeds a threshold of -85 dBm. Prior empirical studies have shown that links with RSS above this threshold typically have high packet reception rate (PRR) [168]. Interference links are determined based on RID protocol [192]. RID models interference as a graph that is constructed as follows. To determine whether the transmissions of other nodes can interfere with a communication link (A, B), RID calculates the Signal to Noise Plus Interference Ratio (SNIR) at node B for each set of k senders ($k = 3$ in our setup) assuming they transmit



(a) Receiver-based allocation

(b) Link-based allocation

Figure 7.5: Channel allocation on testbed topologies to remove all interferences

simultaneously as A transmits to B. For each set of senders $S(B)$, RID computes the SNIR at B when A and the set of senders $S(B)$ transmit simultaneously. The RSS of a link is computed as the average of the four 50 packet batches collected from the testbed. The RSS of missing packets is overestimated to equal the receiver sensibility of CC2420 (-90 dBm). If the computed SNIR is below a threshold a link from each node in $S(B)$ to B is added as an interference link. The SNIR threshold was set to 5 dB consistent with empirical studies that showed that meeting this threshold is usually sufficient for correctly decoding packets in the presence of interference [184, 192]. The routing tree on a topology is constructed based on high quality links.

We also evaluate scalability of our protocols using random topologies. A random network is generated with an edge-density of 50%, i.e. with $n(n-1)50/200$ edges for a network with n nodes. Packet reception rate (PRR) along a link is assigned randomly in a range $[0.60, 1.0]$. A node with the highest degree is selected as the sink. A subset of links forms the routing tree. All other links are interference links.

7.8.1 Interference-free Channel Allocation

Figure 7.5 shows the number of channels required in our interference-free channel allocation (1 run) on testbed topologies at different Tx power. For receiver-based channel allocation (Figure 7.5(a)), our protocol requires no more than 6 channels (marked as 'Dis Receiver-based' in the figure) in every topology, and these values are less than the theoretical upper

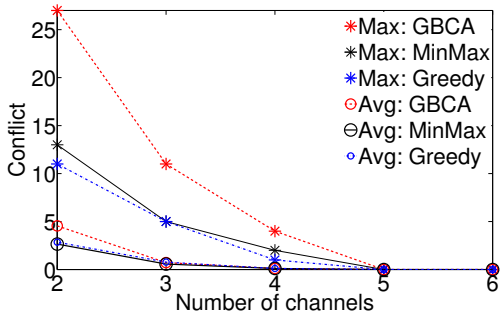


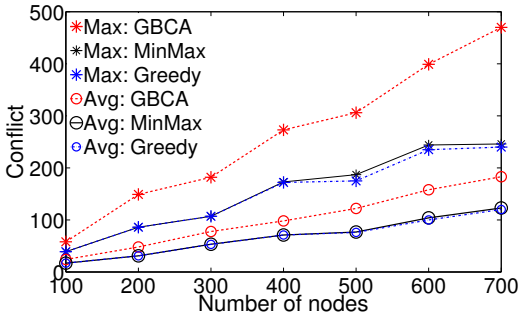
Figure 7.6: MinMax channel allocation on testbed topology with -5 dBm Tx power

bound. We compare the results against a well-known centralized heuristic, called *Largest Degree First* (LDF) [84] (where a node is assigned the first available frequency in non-increasing order of degrees). While LDF is inherently more effective at the cost of centralized behavior, the figure indicates that the numbers of channels required by the centralized LDF and that by our distributed protocol vary at most by 1. For the link-based allocation (Figure 7.5(b)), the number of channels required by our protocol is much less than its theoretical bound.

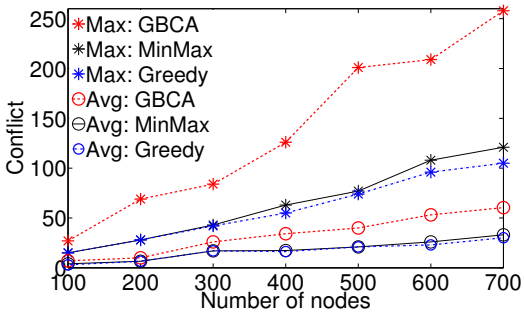
7.8.2 MinMax Channel Allocation

Now we evaluate the MinMax algorithm. We plot the maximum conflict among all transmission links and the average conflict per transmission link after channel allocation. Each data point is the average of 5 runs. We compare the results with that of GBCA [187], the only known distributed protocol that minimizes the total interferences in the network in a receiver-based allocation. We also compare the performance with a *centralized greedy approach* that works as follows. Every time it determines the link that experiences the maximum conflict. If there exists a link such that switching its channel to a different one decreases the maximum conflict, then it switches to that channel. Any sender that does not affect the maximum conflict switches to the channel that results in maximum decrease in its own conflict.

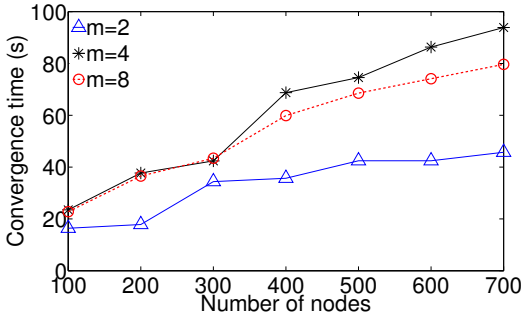
Figure 7.6 shows the performance of MinMax protocol on the testbed topology with -5 dBm Tx power under varying number of channels. It shows that the maximum conflict in GBCA



(a) Conflict using 2 channels



(b) Conflict using 4 channels

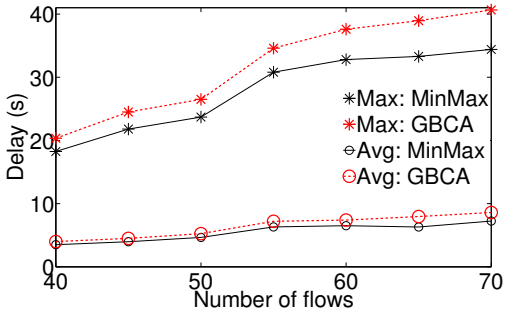


(c) Convergence time

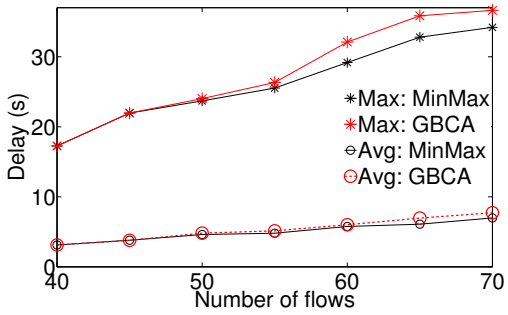
Figure 7.7: MinMax channel allocation on random topologies

using 2 channels is 27 while that in MinMax is only 13. The average conflict per link is 4.55 in GBCA, and 2.87 in MinMax. The centralized greedy heuristic results in a maximum conflict of 11, and an average of 2.85 per link. Both maximum and average conflict in GBCA are higher than those in MinMax allocation since GBCA does not aim to minimize the maximum conflict.

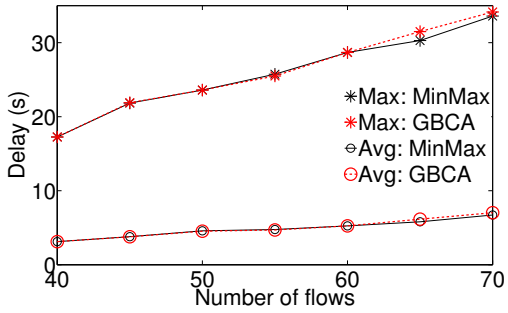
Figure 7.7 shows the performance of MinMax protocol on random topologies with different number of nodes. Figure 7.7(a) shows that the performance gap between GBCA and MinMax increases with the increase of network size. In a 700-node network with 2 channels, the maximum conflicts in GBCA, MinMax, and centralized greedy heuristic are 470, 246, and 240, respectively; the average conflicts per link in GBCA, MinMax, and centralized greedy heuristic are 183, 123, and 120, respectively. Figures 7.7(b) shows the similar results using 4 channels. The results show that MinMax protocol is highly effective in minimizing the maximum interference. It also results in less (compared to GBCA) average conflict which



(a) Delay using 2 channels



(b) Delay using 3 channels



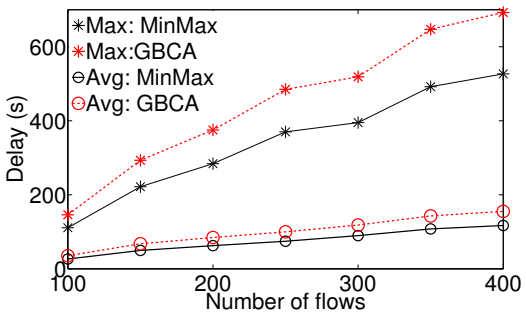
(c) Delay using 4 channels

Figure 7.8: Network performance on testbed topology at -5 dBm

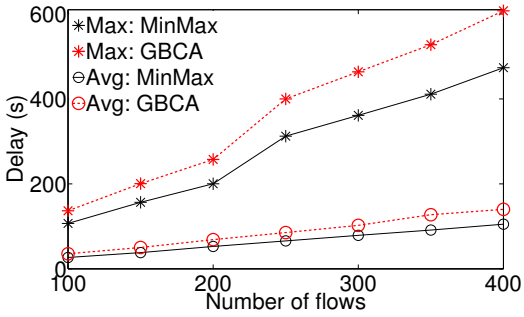
is very close to that of the centralized greedy algorithm. The MinMax protocol converges in 39s when the number of nodes is no greater than 300 (Figures 7.7(c)). For a 700-node network with 4 channels, it converges in 87s.

7.8.3 Latency under MinMax Channel Allocation

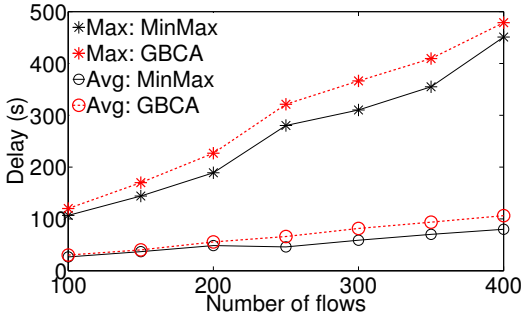
Here we implement our distributed link scheduling protocol after both MinMax and GBCA channel allocation. We consider TDMA with each time slot of 10ms (similar to WirelessHART [22] based on 802.15.4). For scheduling, each node periodically generates a packet resulting in a flow to the sink. All node have the same period. We record the maximum packet delay and the average packet delay in both protocols. The delay of a packet is counted as the difference between the time when it is delivered to the sink and the time when it was



(a) Delay using 2 channels



(b) Delay using 4 channels



(c) Delay using 8 channels

Figure 7.9: Network performance on random topology of 400 sensor nodes

released at its source. In every run, a set of source nodes is selected randomly. Each data point is the average of 5 runs.

Figure 7.8 shows the delays under different number of flows on the testbed topology at -5 dBm Tx power. Figure 7.8(a) shows that the maximum delay among 70 flows under GBCA using 2 channels is 40.65s while that under MinMax allocation is only 34.40s. The average delay per packet is 8.60s under GBCA, and 7.24s under MinMax. In every setup, the 95% confidence interval remains within $\pm 1.7s$ for maximum delay, and within $\pm 0.43s$ for average delay for each protocol. The performance difference between GBCA and MinMax increases in larger networks as shown for random topologies of 400 nodes in Figure 7.9. For 400 flows and 2 channels (Figure 7.9(a)), the maximum delay is 692.61s under GBCA, and 526.68s under MinMax; the average delay per packet is 155.18s under GBCA, and 117.04s under MinMax. In every setup, the 95% confidence interval remains within $\pm 16.7s$ for maximum

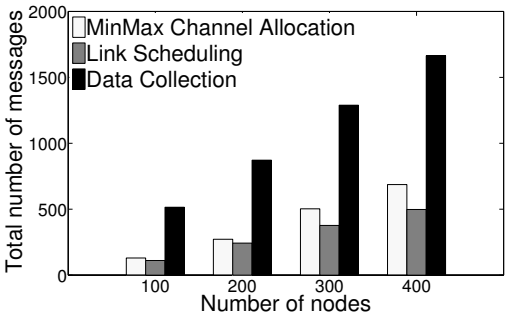


Figure 7.10: Comparison of message cost for channel allocation and one round of data collection

delay, and within $\pm 4.65s$ for average delay for each protocol. The results indicate that MinMax allocation is more effective in terms of packet latency.

7.8.4 Channel Allocation Message Overhead

Figure 7.10 shows the total number of messages used in MinMax channel allocation and link scheduling along with the total number of data transmissions. We used a setup similar to the preceding experiment. We consider networks with 101, 201, 301, and 401 nodes where in each case 1 node serves as the sink while all the other nodes are the sources of data. Every source node periodically generates a packet, all nodes having the same period. We compare the message overhead with the number of data messages in *one* cycle of data collection. Note that realistically channel allocation will be needed after *multiple* rounds of data collection. This result shows that even when compared to *one* cycle of data collection, channel allocation and link scheduling have lower message overhead. For example, the proportion of the total data transmissions to the total messages needed for channel allocation and link scheduling is 0.7 for 1 cycle of data collection in a network of 400 nodes. For c cycles of data collection, this fraction becomes $\frac{0.7}{c}$. Usually, upon channel allocation once, a multi-channel application (such as data collection) can run continuously based on that allocation until some network condition changes. For example, the message overhead will be below 3% of the data load if data allocation and scheduling are performed once every 25 rounds of data collection. The message overhead is therefore acceptable in many deployment scenarios.

7.9 Summary

We have proposed a set of distributed protocols for channel allocation in WSNs. For WSNs with an insufficient number of channels, we have proposed a fair channel allocation protocol that minimizes the maximum interference experienced by any transmission link. In the future, we plan to design traffic-aware protocol, and implement the results on testbeds.

Chapter 8

CapNet: A Real-Time Wireless Management Network for Data Center Power Capping

Data center management (DCM) is increasingly becoming a significant challenge for enterprises hosting large scale online and cloud services. Machines need to be monitored, and the scale of operations mandates an automated management with high reliability and real-time performance. Existing wired networking solutions for DCM come with high cost. In this paper, we propose a wireless sensor network as a cost-effective networking solution for DCM while satisfying the reliability and latency performance requirements of DCM. We have developed CapNet, a real-time wireless sensor network for power capping, a time-critical DCM function for power management in a cluster of servers. CapNet employs an efficient event-driven protocol that triggers data collection only upon the detection of a potential power capping event. We deploy and evaluate CapNet in a data center. Using server power traces, our experimental results on a cluster of 480 servers inside the data center show that CapNet can meet the real-time requirements of power capping. CapNet demonstrates the feasibility and efficacy of wireless sensor networks for time-critical DCM applications.

8.1 Introduction

The continuous, low-cost, and efficient operation of a datacenter heavily depends on its management network and system. A typical data center management (DCM) system handles

physical layer functionality such as powering on/off a server, motherboard sensor telemetry, cooling management, and power management. Higher level management capabilities such as system re-imaging, network configuration, (virtual) machine assignments, and server health monitoring [29, 97] depend on DCM to work correctly. DCM is expected to function even when the servers do not have a working OS or the data network is not configured correctly [3].

Today's DCM is typically designed in parallel to the production data network (in other words, *out of band*), with a combination of Ethernet and *serial connections* for increased redundancy. There is a cluster controller for a rack or a group of racks, which are connected through Ethernet to a central management server. Within the clusters, each server has a motherboard microcontroller (BMC - Baseboard Management Controller) that is connected to the cluster controller via point-to-point serial connections. For redundancy reasons, every server is typically connected to two independent controllers on two different fault domains, so there is at least one way to reach the server under any single point of failure. Unfortunately, this architecture does not scale. The overall cost of management network increases super-linearly with the number of servers in a data center. At the same time, massive cabling across racks increases the chance for human errors and prolongs the server deployment latency.

This paper presents a different approach to data center management network at the rack granularity, by replacing serial cable connections with low cost wireless links. Low power wireless sensor network technology such as IEEE 802.15.4 has intrinsic advantages in this application.

- *Cost:* Low-power radios (i.e., IEEE 802.15.4) are cheaper individually than wired alternatives and the cost scales linearly with the number of servers.
- *Embedded*: These radios can be physically small and be integrated onto motherboard to save precious rack space.
- *Reconfigurability:* Wireless sensor networks can be self-configuring and self-repairing with the broadcast media to prevent human cabling error.
- *Low power:* With a small on-board battery, the DCM based on wireless can continue to function on batteries providing monitoring capabilities even when the rack experiences a power supply failure.

However, whether a wireless DCM can meet the high reliability requirement for data center operation is not obvious for several reasons. The amount of sheet metals, electronics, and cables may completely shield RF signal propagation within racks. Furthermore, although typical traffic on a DCM is low, emergency situations might need to be handled in real time, which could require the design of new protocols.

Power capping is an example of emergency event that imposes real-time requirements. Today, data center operators commonly oversubscribe the power infrastructure by installing more servers to an electric circuit than it is rated. The rationale is that servers seldom reach their peak at the same time. By over-subscription, the same data center infrastructure can host more servers than otherwise. In the rare event when the aggregate power consumption of all servers exceeds the circuit's power capacity, some servers must be slowed down (i.e. power capped), through dynamic frequency and voltage scaling (DVFS) or CPU throttling, to prevent the circuit breaker from tripping. Every magnitude of oversubscription is associated with a trip time which is a *deadline* by which power capping must be performed to avoid circuit breaker tripping.

This paper studies the feasibility and advantages of using low-power wireless for DCM. In two data centers, we empirically evaluate IEEE 802.15.4 link qualities in server racks to show that the overall packet reception rate is high. We further dive into the power capping scenario and design **CapNet**, a wireless **Net**work for power **Cap**ping, that employs an event-driven real-time control protocol for power capping over wireless DCM. The protocol uses distributed event detection to reduce the overhead of regularly polling all nodes in the network. Hence, the network throughput can be used by other management tasks when there is no emergency. When a potential power surge is detected, the controller uses a sliding window and collision avoidance approach to gather power measurements from all servers, and then issues power capping commands to a subset of them. We deployed and evaluated CapNet in a data center. Using server power traces, our experimental results on a cluster of 480 servers in the data center show that CapNet can meet the real-time requirements of power capping. It demonstrates the feasibility and efficacy in power capping like wired DCM with a fraction of the cost..

8.2 The Case for Wireless DCM (CapNet)

Typical wired DCM solutions in data centers scale poorly with increase in number of servers. The serial-line based point-to-point topology incurs additional costs as we connect more of them together. Here, we compare the costs of the wired DCM to our proposed wireless based solution (CapNet) by considering the cost of the management network, and by measuring the quality of in-rack wireless links.

8.2.1 Cost Comparison with Wired DCM

To compare the hardware cost, we consider the cost of the DiGi switches ($3917/48port [4]), controller cost (approx. $500/rack [5]), cable cost ($2/cable [6]) and additional management network switches ($3000/48port on average [7]). We do not include the labor or management costs for cabling for simplicity of costing model, but note that these costs are also significant with wired DCMs. We assume that there are 48 servers per rack, and there can be up to 100,000 servers that need to be managed, which are typical for large data centers. For the wireless DCM based CapNet solution, we assume IEEE 802.15.4 (ZigBee) technologies for its low cost benefits. The cost of network switches at the top level layer stays, but the cost of DiGi can be significantly reduced. We assume $10 per wireless controller, which is essentially an Ethernet to ZigBee relay. For wireless receivers on the motherboard, we assume $5 per server for the RF chip and antenna as the motherboard controller is already in place [8].

# of servers	Wired-N	Wired-2N	CapNet-N	CapNet-2N
10	7450	14900	3060	6070
100	16560	33120	3530	6560
1000	98820	197640	8210	11420
10000	980780	1961560	79090	108180
100000	9772280	19544560	781840	1063680

Table 8.1: System cost (in US Dollar) comparison and scalability

We develop a simple cost model based on these individual costs and compute the total devices needed for implementing management over number of servers ranging from 10 to 100,000 (in order to capture how cost scales with the number of servers). We consider solutions across two dimensions 1) Wired vs Wireless, and 2) N-redundant vs 2N-redundant

(A 2N redundant system consists of two independent switches, DiGis and paths through the management system). Table 8.1 shows the cost comparison across these solutions. We see that a wired N-redundant DCM solution (Wired-N) for 100,000 servers is 12.5× the cost of a wireless N-redundant DCM solution (CapNet-N). If we increase the redundancy of the management network to 2N, the cost of a wired solution (between Wired-2N and Wired-N) doubles. In contrast, the cost of a wireless solution increases only by 36% (due to 2N controllers and 2N switches at the top level). The resulting cost of Wired-2N is 18.4× that of CapNet-2N. Given the significant cost difference between wired DCM and CapNet, we next explore whether wireless is feasible for communication within racks.

8.2.2 Choice of Wireless - IEEE 802.15.4

We are particularly interested in low bandwidth wireless like IEEE 802.15.4 instead of IEEE 802.11 for a number of reasons. First, the payload size for data center management is small and hence a ZigBee (IEEE 802.15.4) network bandwidth is sufficient for control plane traffic. Second, in WiFi (IEEE 802.11) there is a limit on how many nodes an access point can support in the infrastructure mode since it has to maintain an IP stack for every connection, and this impacts scalability in a dense deployment. Third, to support management features, the data center management system should still work when the rack is unpowered. A small backup battery can power ZigBee longer at much higher energy efficiency. Finally, ZigBee communication stack is simpler than WiFi so the motherboard (BMC controller) microcontroller can remain simple. Although we do not rule out other wireless technologies, we chose to prototype with ZigBee in this paper.

8.2.3 Radio Environment inside Racks

We did not find any previous study that evaluated the signal strength within the racks through servers and sheet metal. The sheet metals inside the enclosure are known to weaken radio signal, giving a harsh environment for radio propagation inside racks. RACNet [119] studied wireless characteristics in data centers, but only across racks when all radios are mounted at the top of the rack. Therefore, we first perform an in-depth 802.15.4 link layer

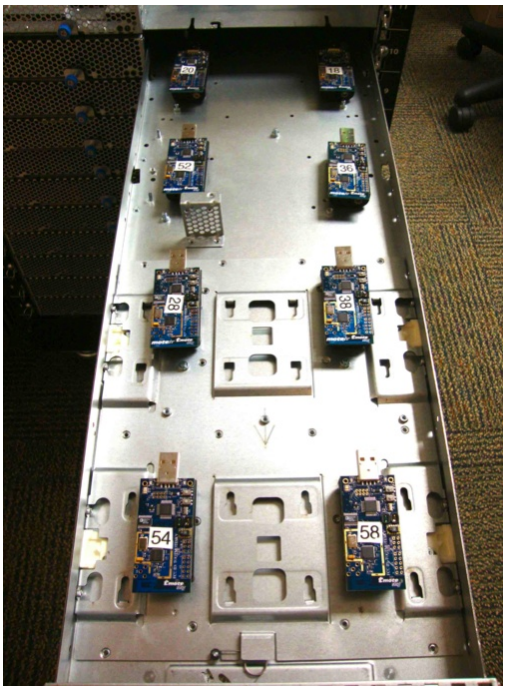
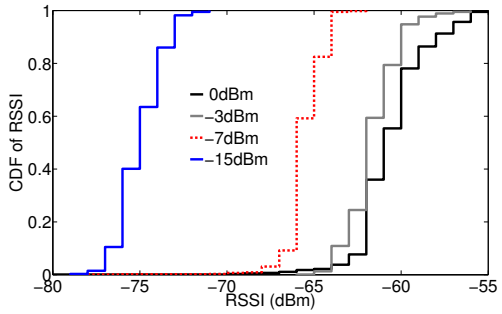


Figure 8.1: Mote placed in bottom sled

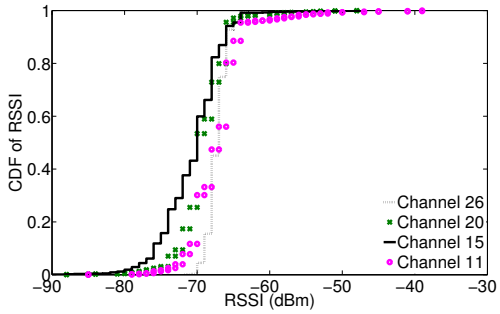
measurement study based on in-rack radio propagation inside a data center of Microsoft Corporation.

Setup. The data center used for measurement study has racks that consist of multiple chassis in which servers are housed. A chassis is organized into two columns of sleds. In all experiments, one TelosB mote is placed on top of the rack (ToR), inside the rack enclosure. The other motes are placed in different places in a chassis in different experiments. Figure 8.1 shows the placement of 8 motes inside a bottom sled (which is open in the figure but was closed during the experiment). While measuring the downward link quality, the node on ToR is the sender and the nodes in the chassis receive. Then we reverse the sender and the receiver to measure the upward link quality. In each setup, the sender transmits packets at 4Hz. The payload size of each packet is 29 bytes. Through a week-long test capturing the long-term variability of links, we collected signal strengths and packet reception rate (PRR).

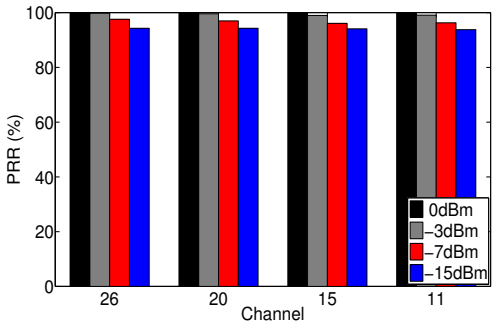
Results. Figure 8.2(a) shows the cumulative distribution function (CDF) of Received Signal Strength Indicator (RSSI) values at a receiver inside the bottom sled for 1000 transmissions



(a) RSSI when Tx power varies (channel 26)



(b) RSSI on various channels (Tx power -3dBm)



(c) PRR at a receiver

Figure 8.2: Downward signal strength and PRR in bottom sled

from the node on ToR for different transmission (Tx) power using IEEE 802.15.4 channel 26. For -7dBm or higher Tx power, RSSI is greater than -70dBm in 100% cases. RSSI values in ZigBee receivers are in the range $[-100, 0]$. Previous study [169] on ZigBee shows that when the RSSI is above -87dBm (approx.), PRR is at least 85%. As a result, we see that signal strength at the receiver in bottom sled is quite strong. Figure 8.2(b) shows the CDF of RSSI values at the same receiver for 1000 transmissions from the node on ToR on different channels at Tx power of -3dBm. Both figures indicate a strong signal strength, and in each experiment the PRR was at least 94% (Figure 8.2(c)). We observed similar results in all other setups of the measurement study, and omit those results.

The measurement study reveals that low-power wireless, such as IEEE 802.15.4, is viable for communication within data center racks and can be reliable for telemetry purpose. We now focus on the power capping scenario and CapNet design for real-time power capping over wireless DCM.

8.3 CapNet Design Overview

8.3.1 The Power Capping Problem

Power infrastructure bears huge capital investment for a data center, up to 40% of the total cost of a large data center that can cost hundreds of millions of US Dollars [91]. Hence, it is desirable to use the provisioned infrastructure to its maximum rated capacity. The capacity of a branch circuit is provisioned during design time, based on upstream transformer capacity during normal operation or UPS/Generator capacity when running on backup power. To improve data center utilization, a common practice in enterprise data centers is to do *oversubscription* [75, 81, 120, 138]. This method allocates servers in a circuit exceeding the rated capacity (i.e. *cap*), since not all servers reach their maximum power consumption at the same time. Hence, there is a circuit breaker (CB) that trips to protect expensive equipment. The peak power consumption above the cap has a specified time limit, called a *trip time*, depending on the magnitude of over-subscription (as shown in Figure 8.3 for Rockwell Allen-Bradley 1489-A circuit breaker). If the over-subscription continues for longer than the

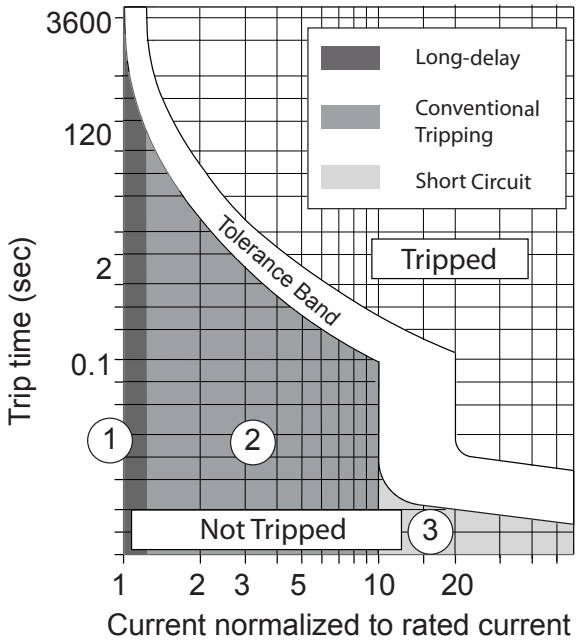


Figure 8.3: The trip curve of Rockwell Allen-Bradley 1489-A circuit breaker at 40°C [23]. X-axis is oversubscription magnitude. Y-axis is trip time.

trip time, the CB will trip and cause undesired server shutdowns and power outages disrupting data center operation. *Power capping* is the mechanism to bring the aggregate power consumption back to the cap. An overload condition under practical current draw trips the CB on a time scale from several hundred milliseconds to hours, depending on the magnitude of the overload [23]. These trip times are the *deadlines* for the corresponding oversubscription magnitudes within which power capping must be done to prevent CB tripping to avoid power loss or damage to expensive equipment.

To enable power capping for a rack or cluster, a *power capping manager* (also called *controller*) collects all servers' power consumption and determines the cluster-level aggregate power consumption. If the aggregate consumption is over the cap, the manager generates control messages asking a subset of the servers to reduce their power consumptions through CPU frequency modulation (and voltage if using DVFS) or utilization throttling. The application level quality of service may require different servers to be capped at different levels. So the central controller needs all individual server readings. In some graceful throttling policies, the control messages are delivered by the BMC Controller to the host OS or VMs, which introduce additional latency due to OS stack [48, 120]. To avoid abrupt changes to

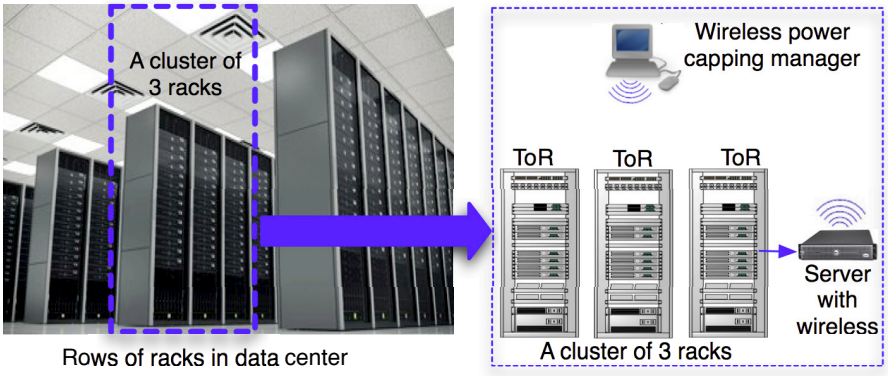


Figure 8.4: Wireless DCM architecture

application performance, the controller may change the power consumption incrementally and require multiple iterations of the feedback control loop before the cluster settles down to below the power cap [120, 177]. These control policies have been studied extensively by previous work and are out of the scope of this paper.

8.3.2 Power Capping over Wireless DCM

Servers in a data center are stacked and organized into racks. One or more racks can comprise a power management unit, called a *cluster*. Figure 8.4 shows the wireless DCM architecture inside a data center. All servers in a cluster incorporate a wireless transceiver that connects to the BMC microcontroller. Each server is capable of measuring its own power consumption. A cluster power capping manager can either directly measure the total power consumption using a power meter, or, to achieve fine-grained power control, aggregates the power consumption from individual servers. We focus on the second case due to its flexibility. When the aggregate power consumption approaches the circuit capacity, the manager issues capping commands over wireless links to individual servers. The main difference compared to a wired DCM is the broadcast wireless media and challenge of scheduling communication to meet the real-time demands.

To reduce extra coordination and to enable spatial spectrum reuse, we assume a single IEEE 802.15.4 channel for communication inside a cluster. Using multiple channels, multiple

clusters can run in parallel. Channel allocation can be done using existing protocols that minimize inter-cluster interference, and is not the focus of our paper. For protocol design, we focus on a single cluster of n servers.

8.3.3 A Naive Periodic Protocol

A naive approach for a fine-grained power capping policy is to always monitor the servers by periodically collecting the power consumption readings from individual servers. The manager periodically computes the aggregate power. Whenever the aggregate power exceeds the cap, it generates a control message. Upon finishing the aggregation and control in η iterations, it resumes the periodic aggregation again.

8.3.4 Event-Driven CapNet

Oversubscribing data centers may provision for the 95-th (or more) percentile of the peak power, and require capping for 5% (or less) of the time, which may be an acceptable hit on performance in relation to cost savings [48]. Thus power capping is a rare event, and the naive periodic protocol is an overkill as it saturates the wireless media by always preparing for the worst case. Other delay-tolerant telemetry messages cannot get enough network resources. An ideal wireless protocol should generate significant traffic only when a significant power surge occurs. Therefore, CapNet employs an event-driven policy that is designed to trigger power capping control operation only when a potential power capping event is predicted. Due to the rareness and emergency nature of power surge, the network can suspend other activities to handle power capping. It provides real-time performance and a sustainable degree of reliability without consuming much network resource. The details of the protocol is explained in the next section.

8.4 Power Capping Protocol

We design a distributed event detection policy, where we assign local caps to each individual server from their global (cluster-level) cap. When a server observes a local power surge based on its own power reading, it can trigger the collection of the power consumption of all the servers to detect a potential surge in the aggregate power consumption of cluster. If a cluster-level power surge is detected, the system initiates a power capping action. As many servers can simultaneously exceed their local caps, a standard CSMA/CA protocol can suffer from significant packet loss due to excessive contention and collisions. Similarly, a slot stealing TDMA (Time Division Multiple Access) protocol such as Z-MAC [152] would suffer from the same problem as those servers will try to steal slot simultaneously. Furthermore, pure TDMA based protocols do not fit well for our problem since they need to have a predefined communication schedule for all nodes. Finally, as power aggregate consumption can be quite dynamic, it may be infeasible to predict an upcoming power peak based on historical readings. This observation leads us to avoid a predictive protocol that proactively schedule data collection based on historical power readings.

While a global detection is possible by just monitoring at the branch circuit level, say using a power meter, it cannot support fine-grained and flexible power capping policies such as those based on individual server-priority or reducing powers of individual servers based on their power consumptions. Also, a centralized measurement introduces a single point of failure. That is, if the power meter fails, power oversubscription will fail also. In contrast, our distributed approach is more resilient to failure. If individual measurement fails, the system can always assume a maximum power consumption at that server and keep the whole cluster going.

The event-driven protocol runs in 3 phases as illustrated in Figure 8.5: **detection**, **aggregation**, and **control**. The event detection phase generates alarms based on local power surges. Upon detecting a potential event, CapNet runs the second phase which invokes a power aggregation protocol. False detection may happen when some servers generate alarms exceeding the local caps, but the aggregate value is still under the cap. This is corrected in the aggregation phase, where the controller determines the aggregate power consumption. The impact of a false positive case is that the system runs into the aggregation phase which incurs additional wireless traffic. The control phase is executed only if the alarms are true.

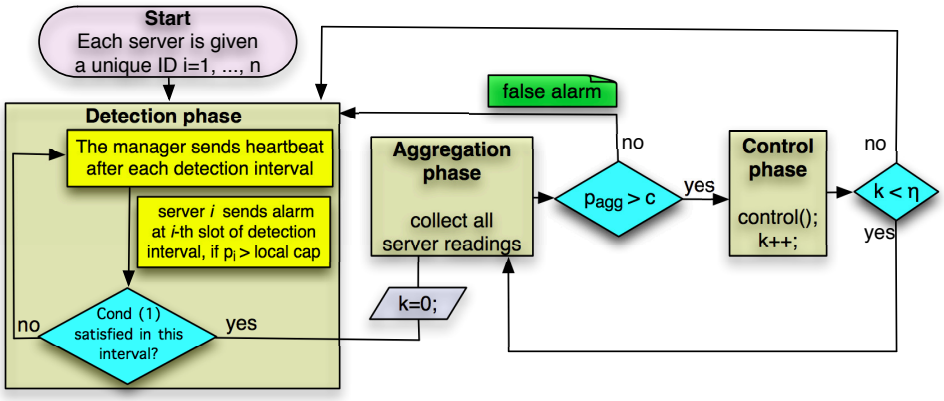


Figure 8.5: CapNet's event-driven protocol flow diagram

We normalize each server's power consumption value between 0 and 1 by dividing its instantaneous power consumption by the maximum power consumption of an individual server. This normalized power consumption value of server i is denoted by p_i, where $0 \leq p_i \leq 1$, and is used in this paper as a server's power consumption. The cap of a cluster of n servers is denoted by c, and the total power consumption of n servers is considered as the aggregate power consumption and is denoted by p_{agg}.

Assigning local cap. If $p_{agg} > c$, a necessary condition is that some servers' (at least one) individual power consumption values locally exceed the value $\frac{c}{n}$. Therefore, a possible way is to assign $\frac{c}{n}$ as each server's local cap. However, there can be situations where only one server exceeds $\frac{c}{n}$ while all other servers are under $\frac{c}{n}$, thereby triggering an aggregation phase upon a single server's alarm. As a result, this policy will generate many false alarms. Therefore, to suppress false alarms, we assign a slightly smaller local cap, and consider alarms from multiple servers before aggregation phase. Thus we use a value $0 < \alpha \leq 1$ close to 1 and assign $\frac{\alpha c}{n}$ as the local cap for each server. A server i reports alarm if $p_i > \frac{\alpha c}{n}$.

Each server is assigned a unique ID i, where $i = 1, 2, \cdots, n$. The manager broadcasts a *heartbeat* packet at every h time units called *detection interval*. The detection interval of length h is slotted among n slots, with each slot length being $\left\lfloor \frac{h}{n} \right\rfloor$. The value of h is selected in a way so that a slot is long enough to accommodate one transmission and its acknowledgement. After receiving the heartbeat message, the server clocks are synchronized.

8.4.1 Detection Phase

Each node i, $1 \leq i \leq n$, takes its sample (i.e., power consumption value p_i) at the i-th slot in the detection phase. If its reading is over the cap i.e. $p_i > \frac{\alpha c}{n}$, it generates an alarm and sends the reading (p_i) to the manager as an acknowledgement of the heartbeat message. Otherwise, it ignores the heartbeat message, and does nothing. If an alarm is received at the s-th slot, the manager determines, based on whether the network is reliable or not, whether an aggregation phase has to be started. Let the servers who have sent alarms in the current detection window so far be denoted by A.

Reliable Network. Let an alarm be generated in the s-th slot of a detection interval. Considering a reliable network we can consider that no server message was lost. Therefore, each of the other $s - |A|$ servers among the first s servers has a power consumption reading of at most $\frac{\alpha c}{n}$ as it has not generated an alarm. Each of the remaining $n - s$ servers can have a power consumption value of at most 1. Thus based on the alarm at s-th slot, the manager can estimate an aggregate power of $\sum_{j \in A} p_j + (s - |A|)\frac{\alpha c}{n} + (n - s)$. Hence, if an alarm is generated at the s-th slot, the manager will start aggregation phase if

$$\sum_{j \in A} p_j + (s - |A|)\frac{\alpha c}{n} + (n - s) > c \tag{8.1}$$

Unreliable Network. Now we consider a scenario where some server alarms were lost. As a result, if an alarm is generated in the s-th slot of a detection window, each of the other $s - |A|$ servers among the first s servers may have a power consumption reading of at most 1 as its alarm is assumed to be lost. Therefore, each of the $n - |A|$ servers can have power consumption of at most 1, making an estimated aggregate power of $\sum_{j \in A} p_j + (n - |A|)$. Thus, if an alarm is generated in the s-th slot, the manager will start aggregation phase if

$$\sum_{j \in A} p_j + (n - |A|) > c \tag{8.2}$$

If there are no alarms in the detection phase or all alarm messages were lost due to transmission failure, the controller resumes the next detection phase (to detect the surges again using the same mechanism) when the current phase is over.

8.4.2 Aggregation Phase

To minimize aggregation latency, CapNet adopts a sliding window based protocol to determine aggregate power consumption denoted by p_{agg}. The controller uses a window of size ω. At anytime, it selects ω servers (or, if there are fewer than ω servers whose readings are not yet collected, then selects all of them) in a round-robin fashion who will send their readings consecutively in the next window. These ω server IDs are ordered in a message. In the beginning of the window, the controller broadcasts this message, and starts a timer of length $\tau_d + \omega\tau_u$ after the broadcast, where τ_d denotes the maximum downward communication time (i.e., the maximum time required for a controller's packet to be delivered to a server) and τ_u denotes the maximum upward communication time (server to controller). Upon receiving the broadcast message, any server whose ID is in order i, $1 \leq i \leq \omega$, in the message transmits its reading after $(i-1)\tau_u$ time. Other servers ignore the message. If the timer fires or packets from all ω nodes are received, the controller creates the next window of ω servers that are yet to be scheduled or whose packets were missed (in the previous window). A server is scheduled in at most γ consecutive windows to handle transmission failures, where γ is the worst-case ETX (expected number of transmissions for a successful delivery) in the network. The procedure continues until all server readings are collected or there is no server that was retried γ times.

8.4.3 Control Phase

Upon finishing the aggregation phase, if $p_{agg} > c$, where c is the cap, it starts the control phase. The control phase generates a capping control command using a control algorithm, and then the controller broadcasts the message requesting a subset of the servers to be capped. To handle broadcast failures, it repeats the broadcast γ times (since the broadcast is not acknowledged). The servers react to the capping messages by DVFS or CPU throttling that incurs an operating system (OS) level latency as well as a hardware-induced delay [48]. If the control algorithm requires η-iteration, then after the capping control command is executed in the first round, the controller will again run the aggregation phase to reconfirm that capping was done correctly. The procedure iterates up to $(\eta - 1)$ more iterations.

Upon finishing the control, or after the aggregation phase upon a false alarm, it resumes the detection phase.

Given the time criticality for power capping, it is important for CapNet to achieve bounded latency. In the following subsection, we provide an analytical upper bound for the total latency required for power capping by CapNet. The analysis can be used by system administrators to configure the cluster to ensure power capping meets the timing constraints.

8.4.4 Latency Analysis

Given the time criticality for power capping, it is important for CapNet to achieve bounded latency. Here, we provide an analytical latency upper bound for CapNet's power capping latency that consists of detection phase latency, aggregation latency, OS level latency, and hardware latency. In practice, the actual latency is usually lower than the bound. The analysis can be used by system administrators to configure the cluster to ensure power capping meets the timing constraints.

Aggregation latency. For n servers in the cluster, the total aggregation delay L_{agg} under no transmission failure can be upper bounded as follows. Note that each window of ω transmissions can take at most $(\tau_u\omega + \tau_d)$ time units. There can be at most $\lfloor \frac{n}{\omega} \rfloor$ windows where in each window ω servers transmit. Then, the last window will take only $(n \bmod \omega + \tau_d)$ time to accommodate the remaining $(n \bmod \omega)$ servers. Hence,

$$L_{agg} \leq (\tau_u\omega + \tau_d) \left\lfloor \frac{n}{\omega} \right\rfloor + (n \bmod \omega + \tau_d)$$

Considering γ as the worst-case ETX in the network,

$$L_{agg} \leq \left((\tau_u\omega + \tau_d) \left\lfloor \frac{n}{\omega} \right\rfloor + (n \bmod \omega + \tau_d) \right) \gamma \tag{8.3}$$

The above value is only an analytical upper bound, and in practice the latency can be a lot shorter.

Latency in detection phase. The time spent in the detection phase is denoted by L_{det}. In a detection window the protocol never will need the readings from the last $\lfloor c \rfloor - 1$ servers

as an aggregation phase must start before this should a power capping needed (assuming that not all alarms were lost). Therefore the alarms generated within the first $(n - \lfloor c \rfloor + 1)$ slots must trigger aggregation phase. Hence,

$$L_{det} \leq \left\lfloor \frac{h}{n} \right\rfloor (n - \lfloor c \rfloor + 1) \tag{8.4}$$

Total power capping latency. To handle a power capping event, a detection phase and an aggregation phase are followed by a control message that is broadcasted γ times and takes $\tau_d \gamma$ time. In addition, once the control message reaches a server, there is an operating system level latency, and after processor frequency changes, there is a hardware-induced delay. Let the OS level latency and the hardware level latency in the worst case be denoted by L_{os} and L_{hw}, respectively. Thus, the total power capping latency in one iteration, denoted by L_{cap}, is bounded as

$$L_{cap} \leq L_{det} + L_{agg} + \tau_d \gamma + L_{os} + L_{hw}$$

A η-iteration control means that once power capping command is executed, the controller will again need to collect all readings from servers, and reconfirm that capping was done correctly in $(\eta - 1)$ more iterations. Therefore, for η-iteration control, the above bound is given by

$$L_{cap} \leq L_{det} + (L_{agg} + \tau_c \gamma + L_{os} + L_{hw})\eta \tag{8.5}$$

8.5 Experiments

In this section, we present the experimental results of CapNet. The objective is to evaluate the effectiveness and robustness of CapNet in meeting the real-time requirements of power capping under data center realistic settings.

8.5.1 Implementation

The wireless communication side of CapNet is implemented in NesC on TinyOS [20] platform. To comply with realistic data center practices, we have implemented the control management

at the power capping manager side. In our current implementation, wireless devices are plugged to the servers directly through their serial interface.

8.5.2 Workload Traces

We use workload demand traces from multiple geo-distributed data centers run by a global corporation over a period of six consecutive months. Each cluster consists of several hundreds of servers that span multiple chassis and racks. These clusters run a variety of workloads including Web-Search, Email, Map-Reduce jobs, and cloud applications, catering to millions of users around the world. Each cluster uses homogeneous hardware, though there could be differences across clusters. We use workload traces of 2 representative server clusters: C1 and C2. In both clusters each individual server has CPU utilization data of 6 consecutive months in every 2 minutes interval. While we recognize that full system power is composed of storage, memory and other components, in addition to CPUs, several previous works show that a server's utilization is roughly linear to its power consumption [57, 62, 76, 148]. Hence, we use server's CPU utilization as a proxy for power consumption in all experiments.

8.5.3 Experimental Setup

Experimental Methodology

We experiment with CapNet using TelosB motes for wireless communication. First we deployed 81 motes (1 for manager, 80 for servers) in Microsoft's data center in Redmond, WA. When we experiment with more than 80 servers to test scalability, one mote emulates multiple servers and communicates for them. For example, when we experiment for 480 servers, mote 1 works for first 6 servers, then mote 2 works for next 6 servers, and so on. We place all 80 motes in racks. The manager node is placed on ToR and connected through its serial interface to a PC that works as the manager. No mote in the rack has direct line of sight with the manager. Using the workload demand traces, CapNet is run in a trace-driven fashion. For every server the reading at a time stamp sent from its corresponding wireless mote is taken from these traces at the same time stamp. While the data traces are

of 6-month long, our experiment does not run for actual 6-month. When we take a subset of those traces, say for 4 weeks, the protocols skip the long time intervals where there is no peak. For example, when we know (looking ahead into the traces) there is no peak between time t_1 and t_2, the protocols skip the times between t_1 and t_2. Thus our experiments finish in several days instead of 4 weeks.

Oversubscription and Trip Time

We use the trip times from Figure 8.3 as the basis, in order to determine the different caps required in various experiments. X-axis shows the ratio of current draw to the rated current and is the *magnitude of oversubscription*. Y-axis shows the corresponding trip time. The trip curve is shown as a tolerance band. The upper curve of the band indicates *upper bound (UB) trip times* above which is the tripped area, meaning that the circuit breaker will trip if the duration of the current is longer than the UB trip time. The lower curve of the band indicates *lower bound (LB) trip times* under which is the not-tripped area. This band between 2 curves is the area where it is non-deterministic if the circuit breaker will trip. LB trip time is a very conservative bound. In our experiments we use both LB and UB of conventional trip times to verify the robustness of CapNet.

CapNet Parameters

For all experiments, we use channel 26 and Tx power of -3dBm. The payload size of each packet sent from the server nodes is 8 bytes, which is enough for sending power consumption reading. The maximum payload size of each packet sent from the manager is 29 bytes, the maximum default size in IEEE 802.15.4 radio stack for TelosB motes. This payload size is set large to contain the schedules as well as control information. For aggregation protocol, window size ω is set to 8. A larger window size can reduce aggregation latency, but requires the payload size of the manager's message to be larger (since the packet contains ω node IDs indicating the schedule for next window). In the aggregation protocol both τ_d and τ_u were set to 25ms. The manager sets its timeout using these values. These values are relatively larger compared to the maximum transmission time between two wireless devices. The time required for communication between two wireless devices is in the range of several

milliseconds. But in our design the manager node is connected through its serial interface to a PC. The TelosB's serial interface does not always incur a fixed latency for communication between PC and the mote through serial. Upon experimenting and observing a wide variation of this time, we have set τ_d and τ_u to 25ms.

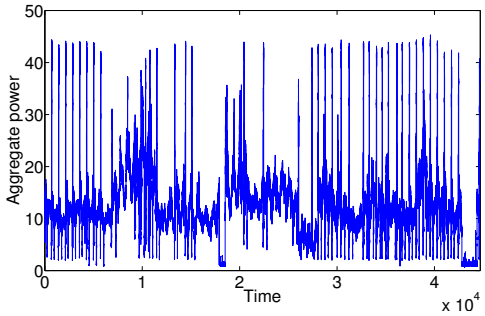
Control Emulation

In our experiments, we emulate the final control action since we use workload traces. We assume that one packet is enough to contain the entire control message. To handle control broadcast failure, we repeat control broadcast $\gamma = 2$ times. Our extensive measurement study through data center racks indicated that this is also the maximum ETX for any link between two wireless motes. Upon receiving the control broadcast message, the nodes generate an OS level latency and hardware level latency. We use the maximum and minimum OS level and hardware level time required for power capping experimented on three servers with different processors: Intel Xeon L5520 (frequency 2.27GHz, 4 cores), Intel Xeon L5640 (frequency 2.27GHz, dual socket, 12 cores with hyper-threading), and an AMD Opteron 2373EE (frequency 2.10GHz, 8 cores with hyper-threading), each running Windows Server 2008 R2 [48]. The ranges of OS level and hardware level latencies are in the range of 10-50ms and 100-300ms, respectively [48]. We generate OS and hardware level latencies using a uniform distribution in this range.

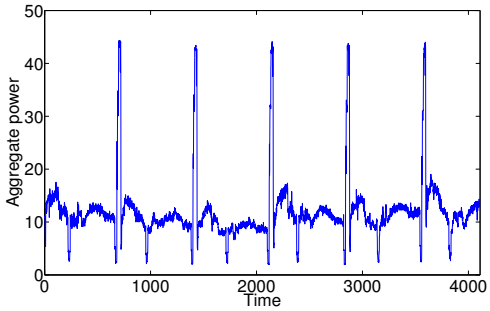
8.5.4 Power Peak Analysis of Data Centers

We first analyze whether CapNet protocol is consistent with the data center power behavior leveraging our data traces. For brevity, we present the trace analysis results of 3 racks: Racks R1 and R2 from Cluster C1, and Rack R3 from Cluster C2.

To give an idea on how power consumption varies over time in a data center, Figure 8.6(a) shows the aggregate power of 60 servers on RACK R1 in cluster C1 for 2 consecutive months which is zoomed in for 6 consecutive days in Figure 8.6(b). For each rack, we use the 95-th percentile of aggregate power over 2 consecutive months as the power cap.

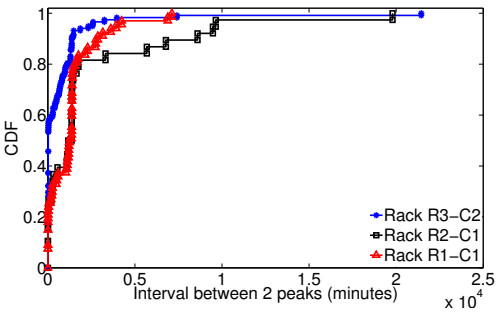


(a) Aggregate power (2 Months)

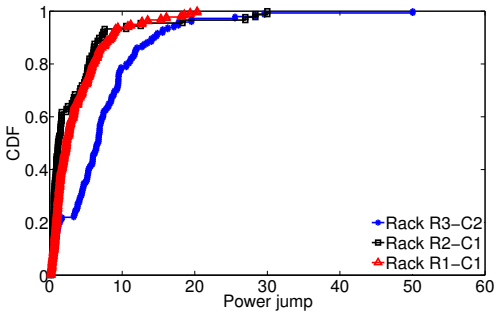


(b) Aggregate power (1st 6 days zoomed-in)

Figure 8.6: 60 Servers on Rack R1 in Cluster C1

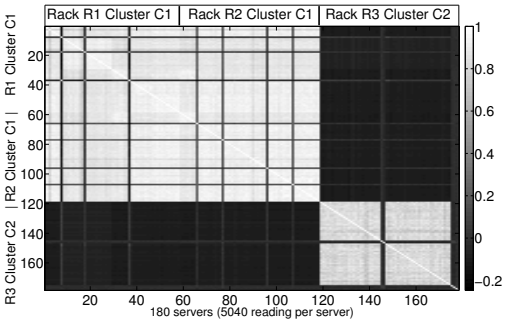


(a) Time interval between 2 peaks

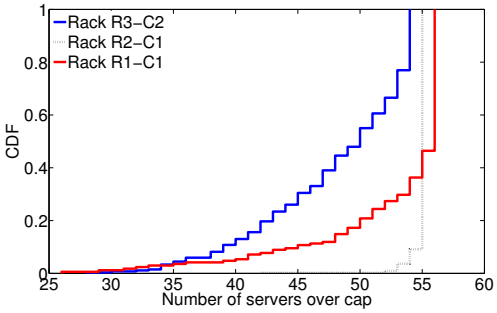


(b) Power jump

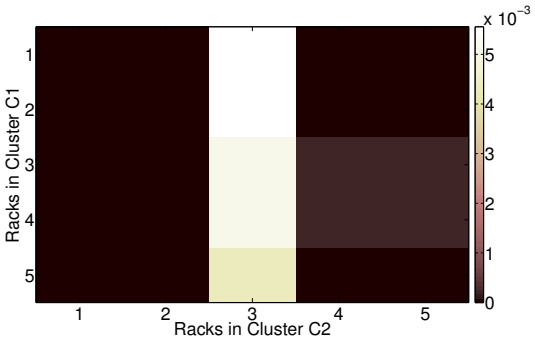
Figure 8.7: Power characteristics (2 month data)



(a) Correlations among 180*180 server pairs in 3 racks in 2 clusters



(b) Total number (out of 60) of servers that exceed local cap $\frac{c}{60}$



(c) Probability of simultaneous peak between two different clusters

Figure 8.8: Correlations among servers, racks, and clusters

We first explore the power dynamics of the servers and the unpredictability of power capping events. Using 2-month long data, Figure 8.7 shows that the time intervals between two consecutive peaks can range between few minutes to several hundred hours. We define *power jump* as the difference between the power that exceeds the cap and the preceding measurement that is below the cap. As Figure 8.7(b) shows that power jumps can vary between 0 to 51 for 60 servers in each rack (while their aggregate power is in range $[0, 60]$). This result shows the motivation for an event-driven protocol.

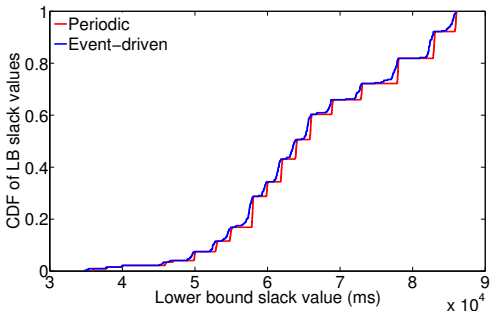
Figure 8.8 illustrates the correlations across 180 servers from different racks and clusters using their raw power consumption data over 1 week. The image is a visualization of a 180×180 matrix, indexed by the server number. That is, the entry indexed at $[i, j]$ in this matrix is the correlation coefficient of the values (5040 samples) between the i-th and the j-th server. We can clearly see that the servers in the same rack are strongly positively correlated, and those

in the same cluster are also positively correlated. But the servers between clusters are less or negatively correlated. This usually happens because the servers in the same cluster hosts similar workloads leading to synchronous power characteristics [57]. We further assume a local cap of $\frac{c}{60}$ (considering $\alpha = 1$) for each individual server, and show in Figure 8.8(b) the CDF of the number of servers that exceed local caps when the cluster level aggregate power exceeds cap c. The figure shows that in 80% cases when the rack level aggregate power exceeds cap c, the numbers of servers (among 60 servers per rack) that are over the local cap are 43, 55, and 50 for Rack R3, R1, and R2, respectively. The strong *intra-cluster synchrony* in power surge suggests the feasibility of detecting a cluster-level power surge based on local server-level measurements. Figure 8.8(c) shows probabilities of different racks in 2 clusters to be at peak simultaneously. The entry indexed at $[i, j]$ in this 2D matrix is the probability that the i-th rack in cluster 1 and the j-th rack in cluster 2 are at peak simultaneously. The probabilities were found in the range $[0, 0.0056]$. This strong *inter-cluster asynchrony* implies that using an event-driven protocol (that performs wireless communications only upon detecting an event) significantly minimizes inter cluster interference caused by transmissions generated by the event-driven CapNet in different clusters.

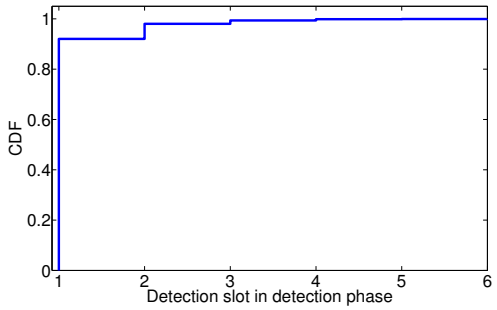
We observe strong synchrony in power behavior among the servers in the same cluster and strong asynchrony among between different clusters. The major implication of the trace analysis is that CapNet protocol is consistent with real data center power behavior. As the intra-cluster synchrony suggests the potential efficacy of a local event detection policy, our protocol is particularly effective in the presence of strong intra-cluster synchrony that exists in enterprise data centers as observed in our trace analysis. However, in absence of intra-cluster synchrony in power peaks, CapNet will not cause unnecessary power capping control or more wireless traffic than a periodic protocol. The synchrony only enhances CapNet's performance.

8.5.5 Power Capping Results

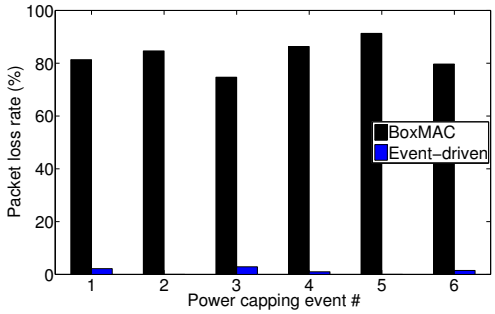
Now we present our experimental results with CapNet's event-driven protocol. First we compare its performance with the periodic protocol and a representative CSMA/CA protocol. We then analyze its scalability in terms of number of servers. First we experiment only for the simple case, where a single iteration of control loop can settle to a sustained power level,



(a) CDF of lower bound slack



(b) CDF of detection slots in detection phase



(c) Packet Loss Rate

Figure 8.9: Performance of Event-Driven protocol on 60 servers (4 weeks)

and then we also analyze scalability in terms of number of control iterations, where multiple iterations are needed to settle to a sustained power level. We have also experimented it under different caps and in presence of interfering clusters. In all experiments, detection phase length, h, was set to $100 * n$ ms, where n is the number of servers. We set this value because this makes each slot in the detection phase equal to 100ms, which is enough for receiving one alarm as well as for sending a message from the manager to the servers. Setting a larger value reduces the number of cycles of detection phase, but reduces the granularity of monitoring. For assigning a local cap of $\frac{\alpha c}{n}$ to the servers, we first experiment with $\alpha = 1$. Later, we experiment under different values of α. Condition 8.1 is used for detection and starting an aggregation phase. In the results, ***slack*** is defined as the difference between the trip time (i.e. deadline) and the total latency required for power capping. That is, a negative value of slack implies a deadline miss. We use *LB slack* and *UB slack* to define the slack calculated considering LB trip time and UB trip time, respectively. In our results, in cases timing requirement can be loose, while there are cases where these are very tight, and the results are shown for all cases. We particularly care for tight deadlines, and want to avoid any deadline misses.

Performance Comparison with Base Lines

Figure 8.9 presents the results using 60 servers on one rack for single-iteration control loop. We used 4 weeks long data traces for this rack. We set the 95-th percentile of all aggregate powers values of all data points in every 2-minute interval as its cap c. For assigning local cap we use $\alpha = 1$. In running the protocols using these traces, the protocols observe all peaks. The upper bound of aggregation latency (L_{agg}) given in (8.3) in Section 8.4.4 was set as the period of the periodic protocol. Figure 8.9(a) shows the LB slacks for both the event-driven protocol and the periodic one. The figure only plots the CDF for the cases where the magnitude of oversubscription was above 1.5 for better resolution as the slack was too big for a smaller magnitudes (which are not of interest). Since UB trip times are easily met, we also omit those results. The non-negative LB slack values for each protocol indicate that it easily meets the trip times. Hence there is no benefit in using non-stop communications (i.e., the naive periodic protocol).

While the slacks in event-driven protocol are shorter than those in the periodic protocol because the former spends some time in the detection phase, in 80% cases event-driven protocol can provide a slack of more than 57.15s while the periodic protocol provides 57.88s. The difference is not significant because as shown in Figure 8.9(b) in 90% cases among all power capping events the detection happened in the first slot of the detection cycle. Only in 10% cases, it was after the first slot of the detection phase, and all detection happened within the 6-th slot, although the phase had a total of 60 slots (for 60 servers, one slot per server). These results indicate that CapNet's local detection policy can quickly determine the events. This is also an implication that experimental values of power capping latencies are quite different (or shorter) from the pessimistic analytical values derived in (8.5) in Section 8.4.4. Also, in this experiment, 94.16% of the total detection phases did not have any transmission from the servers. Therefore, if we compare with the periodic protocol that needs to continue communication always in the network, the event-driven protocol suppresses transmissions at least by 94.16% while the real-time performance of two protocols are similar.

We also evaluate the performance when BoxMAC (the default CSMA/CA based protocol in TinyOS [20]) is used for power capping communication for up to first 6 capping events in the data traces. Figure 8.9(c) shows that it experiences packet loss rate over 74% while performing communication for a power capping event. This happens because all 60 nodes try to send at the same time, and the back-off period in 802.15.4 CSMA/CA under default setting is too short, which leads to frequent repeated collisions. Since we lose most of the packets, we do not consider latency under CSMA/CA. Increasing the back-off period reduces collisions but results in long communication delays. In subsequent experiments, we exclude CSMA/CA as it does not fit for power capping.

Scalability in Terms of Number of Servers

In our data traces each rack has at most 60 active servers. To test with more servers, we combine multiple racks in the same cluster since they have similar pattern of power consumption (as we have already discussed in Subsection 8.5.4). For sake of experimentation time, in all subsequent experiments we set cap at 98-th percentile (that would result in a smaller number of capping events). The lower bound slack distribution are shown in Figure 8.10 for 120, 240, and 480 servers by merging 2, 4, and 8 racks, respectively (for

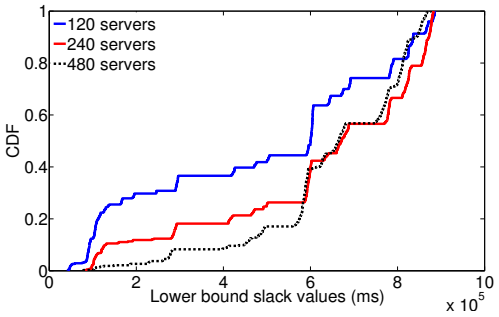


Figure 8.10: CDF of LB slack under various numbers of servers (4 weeks)

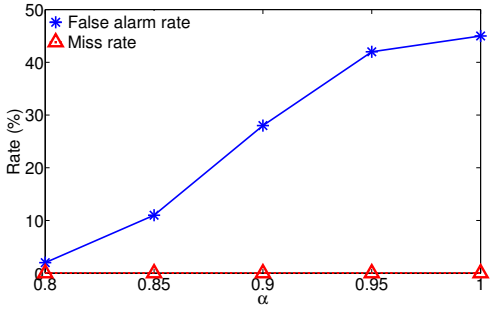


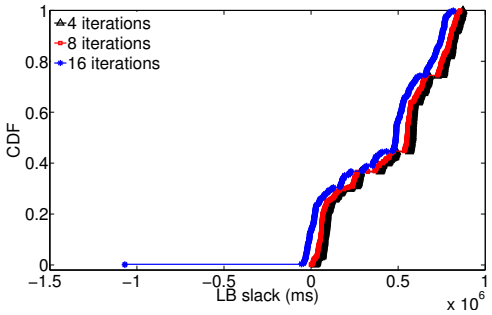
Figure 8.11: Deadline (trip time) miss rate and false alarm rate under varying α

single iteration capping). Hence, for single iteration, the deadlines are easily met for even 480 servers (since in each setup, 100% of all slack values are positive).

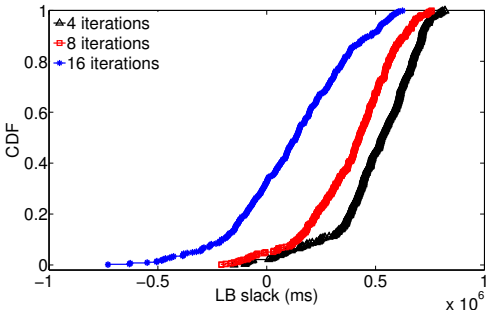
Experiments under Varying α

Now we experiment with different values of α for assigning a local cap of $\frac{\alpha c}{n}$ to the servers using 480 servers. The results in Figure 8.11 show the

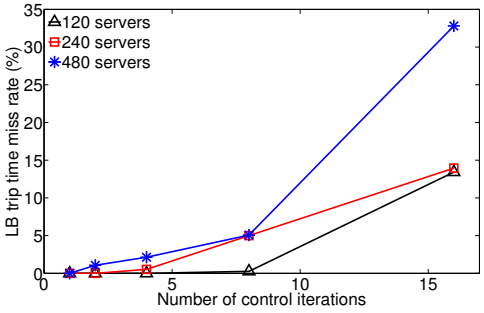
tradeoff between false alarm rate and power capping latency under varying α. As we decrease the value of α from 1 to 0.80, the false alarm rate decreases from 45% to 2%. This happens because with decreased value of α, CapNet considers multiple alarms before detecting a potential event. Note that this alarm rate is very small compared to the whole time window since power capping happens in at most 5% cases. Therefore alarms are also generated rarely. Since waiting for multiple alarms increases the latency in detection, the total power capping latency increases as the value of α decreases. However, as this latency increase happens only



(a) LB slack for 120 servers



(b) LB slack for 480 servers



(c) Miss rate (LB trip time)

Figure 8.12: Multi-iteration capping under event-driven protocol (4 weeks)

in the detection phase which is negligible compared to the total capping latency, there is hardly any impact on deadline miss rates. The figure shows a deadline miss rate of 0 under varying α.

Scalability in Number of Control Iterations

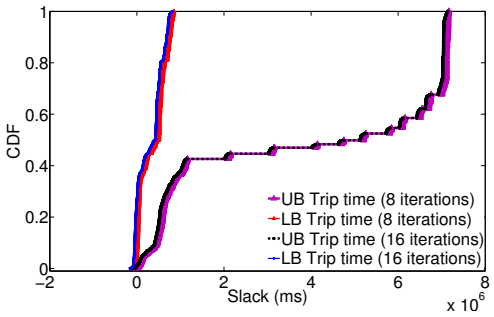
Now we consider a conservative case where multiple iterations of control loop are required to settle to a sustained power level [48, 120, 177]. The number of iterations required for the rack-level loop as experimented in [177] can be up to 16 in the worst case (which happens very rarely). Hence, we now conduct experiments considering multiple numbers of control iterations (up to 16 assuming a pessimistic scenario). We plot the results in Figure 8.12 for various numbers of servers under various number of iterations. As shown in Figure 8.12(a), for 120 servers under 16-iteration case, we have 13% cases with negative slack meaning that the LB trip times were missed. However, the UB trip times were met in 100% cases. Note that we have considered a quite pessimistic set up here because using 16-iteration as well

as trying to meet the lower bound of trip times are both very conservative considerations. For 120 servers under 8 iterations, in 0.13% cases slacks were negative. However, in 80% cases the slacks were 92.492s, 66.694s, and 22.238s for 4, 8, and 16 iterations, respectively indicating that the trip times were easily met, and the system could oversubscribe safely. For 4-iteration, the minimum slack was 23.2s. To preserve figure resolution, we do not show the UB slacks since they were all positive. For 480 servers (Figures 8.12(b), 8.12(c)), 98.95%, 97.86%, 94.93%, and 67.2% LB trip times were met for 2, 4, 8, and 16 iterations, respectively. For 240 nodes, we miss deadlines in 5% cases under 8-iteration and 13.94% cases under 16-iteration.

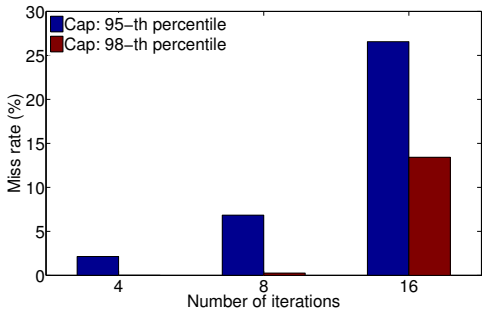
For all cases we met UB trip times in 100% cases. Note that assuming 16-iteration and considering the LB trip times are very conservative assumption as it can rarely happen. Hence, the above results show that, even for 480 servers, the latencies incurred in CapNet for power capping remain within even the conservative latency requirements in most cases.

Experiments under Varying Caps

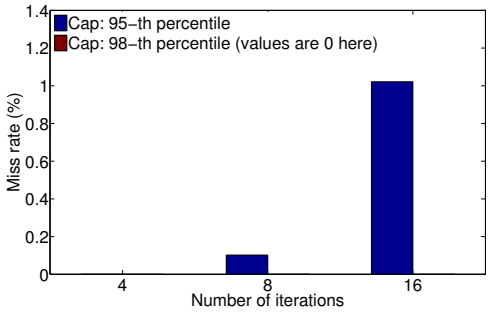
In all experiments we have performed so far, CapNet was able to meet UB trip times. Now we make some setup changes to encounter some scenario where UB trip times can be smaller, by making oversubscription magnitude higher. For this purpose, we now decrease the cap to decrease the trip times so as to make scenarios to miss upper bound trip times to see the robustness of the protocol. Now again we set the 95-th percentile of aggregate power as the cap. This would give the previous capping events shorter deadlines since a smaller cap implies a larger magnitude of oversubscription. For the sake of experiment time, we only tested with 120 servers and their 4 week data traces. Figure 8.13 shows that we now miss more LB trip times and miss some UB trip times as well since the deadlines now become shorter. However, UB trip times are missed only in 0.11% and 1.02% cases under 8 and 16 iterations, respectively, while LB deadlines were missed in 2.14%, 6.84%, and 26.56% cases under 4, 8, and 16 iterations, respectively. All deadlines were met for up to 3 iterations (and not shown in the figures). We have shown the results only for higher number of iterations that rarely happen. These results demonstrate the robustness for larger magnitude of oversubscription in that even when we use 16-iteration only 1.02% UB trip times are missed.



(a) CDF of slack values (Cap: 95-th percentile)



(b) LB trip time miss rate



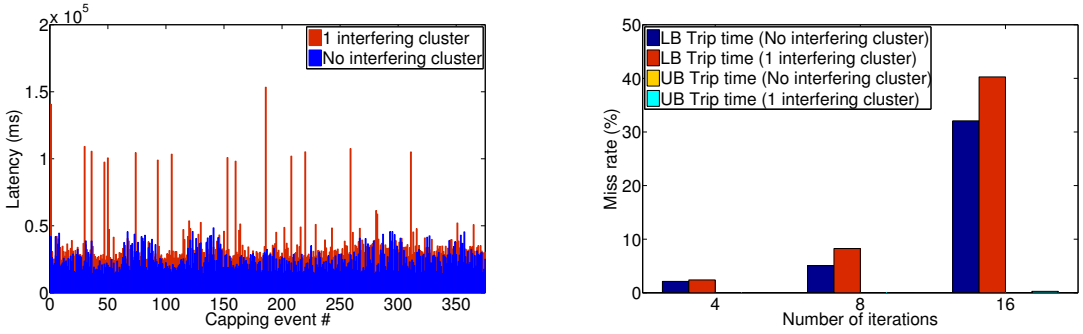
(c) UB trip time miss rate

Figure 8.13: Capping under different caps on 120 servers (4 weeks)

Experiments in Presence of Multiple Clusters

We have shown through data center trace analysis in Figure 8.8(c) in Section 8.5.4 that the probability that two clusters are over the cap simultaneously is no greater than 0.0056. Yet, in this section we perform some experiment from a pessimistic point of view. In particular, we perform an experiment and see the performance of CapNet under an interfering cluster.

We mimic an interfering cluster of 480 servers in the following way. We select a nearby cluster and place a pair of motes in the rack: one at the ToR and the other inside the rack. We set their Tx power at maximum (0dBm). The mote at the ToR represents its manager and carries on a pattern of communication like a real manager to control 480 servers. The mote inside the rack responds as if it were connected to each of 480 servers. Specifically, the manager executes a detection phase of $100 * 480$ms, and the node in the rack randomly selects a slot between 1 and 480. On that slot, it generates an alarm with probability 5% since capping happens in no more than 5% cases. Whenever the manager receives the alarm,



(a) Capping latency

(b) Miss rate

Figure 8.14: Capping for 480 servers under interfering cluster

it generates a burst of communication in the pattern like what it would have done for 480 servers. After finishing this pattern of communication it resumes the detection phase.

We run the main cluster (system used for experiment) using 4 weeks data traces, and plot the results in Figure 8.14. Figure 8.14(a) shows the latencies for different capping events in 4 weeks data both under interference and without interference (when there was no other cluster). Under interfering cluster, the delays mostly increase. This happens because the event-driven protocol experiences packet loss and uses retransmission for those, thereby increasing network delays. While the maximum increase was 124.63s, in 80% cases the increase was less than 15.089s. We noticed that such big increase happened due to the loss of alarms in a detection phase that resulted in a detection in the next phase (i.e., while the phase length is 48s). Still power capping was successful in all cases but those when the control broadcast was lost. Among 375 events, 4 broadcasts were lost at some server even after 2 repeatations, resulting in control failure in 1.06% cases. This value became 0 in multi-iteration cases. For multi-iteration cases, at least one control broadcasts was successful that resulted in no capping failure for control message loss. However, as the delay due to transmission failure and recovery increased in detection phase, we experienced capping failure. For 16-iteration, we missed the upper bound of trip time in 40.27% cases and lower bound of trip times in 32.08% cases. However, we use a conservative assumption here. For 4 iteration miss rate was 5.06% and 8.26% only. And for 2-iteration they are only 2.13% and 2.4% which are very marginal. The result indicates that even under interference, CapNet demonstrates robustness in meeting the real-time requirements of power capping.

8.6 Discussions and Future Work

While our paper addresses feasibility, protocol design and implementation, several engineering challenges such as security, EMI and fault tolerance needs to be addressed.

Fault Tolerance. One important challenge is handling the failure of power capping manager in a cluster. To address this, power capping managers can be connected among themselves either through a different band or through a wired backbone. As a result, when some manager fails, a nearby one can take over its servers. This paper focuses on communication within a single cluster. DCM fault detection, isolation, and node migration need to be studied in future work.

Security. Another challenge is the security of the management system itself. Since the system relies on wireless control, someone might be able to maliciously tap into the wireless network and take control of the data center. There are two typical approaches to handle this security issue: First, the signal itself should be attenuated by the time it reaches outside the building. We can identify secure locations inside the data center from which the controller can communicate, and identify a signature for the controllers which would be known to the server machines. Second, it is possible to encrypt wireless messages, for example, using MoteAODV (+AES) [36]. We can also use shielding within the data center to keep the RF signals contained within the enclosed region.

EMI & Compliance. While less emphasized in research studies, a practical concern of introducing wireless communications in data centers is that they do not adversely impact other devices. There are FCC certified IEEE 802.15.4 circuit design available(e.g. [9]). Previous work has also used WiFi and ZigBee in live data centers for monitoring purposes [119].

8.7 Related Work

In order to reduce the capital spending on data centers, enterprise data centers use an over-subscription approach as studied in [75, 81, 120, 138], which is similar to over-booking in airline reservations. Server vendors and data center solutions providers have started to

offer power capping solutions [15,16]. Power capping using feedback control algorithms [178] has been studied for individual servers. In contrast, the study of this paper concentrates to coordinated power capping which is more desirable in data centers as it allows servers to exploit power left unused by other servers. While such power capping has been studied before [77,108,120,145,177,191], all existing solutions rely on wired network for controller-server communication. In contrast, we focus on wireless networking for power capping. We have outlined the advantages of wireless management in Section 8.2.

Previous work on using wireless network in data centers exists on applications to high bandwidth (e.g. with 60GHz radio) production data network [194]. In contrast, CapNet is targeted at data management functions that have much lower bandwidth requirement while demanding real-time communication through racks. RACNet [119] is a passive monitoring solution in the data center that monitors temperature or humidity across racks where all radios are mounted at the top of the rack. Our solution enables active control and requires communication through racks and server enclosures, and hence encounters fundamentally different challenges. Also, RACNet also does not have real-time features, while CapNet is designed to meet the real-time requirements in power capping.

8.8 Summary

Power capping is a time-critical management operation for data centers that commonly over-subscribe power infrastructure for cost savings. In this paper, we have designed CapNet, a low-cost, real-time wireless management network for data centers and validated its feasibility for power capping. We deployed and evaluated CapNet in an enterprise data center. Using server power traces, our experimental results on a cluster of 480 servers inside the data center show that CapNet can meet the real-time requirements of power capping. CapNet represents a promising step towards applying low power wireless networks to time-critical, close-loop control in DCM.

Chapter 9

Multi-core Real-Time Scheduling for Generalized Parallel Task Models

Multi-core processors offer a significant performance increase over single-core processors. They have the potential to enable computation-intensive real-time applications with stringent timing constraints that cannot be met on traditional single-core processors. However, most results in traditional multiprocessor real-time scheduling are limited to sequential programming models and ignore intra-task parallelism. In this paper, we address the problem of scheduling periodic parallel tasks with implicit deadlines on multi-core processors. We first consider a synchronous task model where each task consists of segments, each segment having an arbitrary number of parallel threads that synchronize at the end of the segment. We propose a new task decomposition method that decomposes each parallel task into a set of sequential tasks. We prove that our task decomposition achieves a resource augmentation bound of 4 and 5 when the decomposed tasks are scheduled using global EDF and partitioned deadline monotonic scheduling, respectively. Finally, we extend our analysis to a directed acyclic graph (DAG) task model where each node in the DAG has a unit execution requirement. We show how these tasks can be converted into synchronous tasks such that the same decomposition can be applied and the same augmentation bounds hold. Simulations based on synthetic workload demonstrate that the derived resource augmentation bounds are safe and sufficient.

9.1 Introduction

In recent years, multi-core processor technology has improved dramatically as chip manufacturers try to boost performance while minimizing power consumption. This development has shifted the scaling trends from increasing processor clock frequencies to increasing the number of cores per processor. For example, Intel has recently put 80 cores in a Teraflops Research Chip [10] with a view to making it generally available, and ClearSpeed has developed a 96-core processor [11]. While hardware technology is moving at a rapid pace, software and programming models have failed to keep pace. For example, Intel [10] has set a time frame of 5 years to make their 80-core processor generally available due to the inability of current operating systems and software to exploit the benefits of multi-core processors.

As multi-core processors continue to scale, they provide an opportunity for performing more complex and computation-intensive tasks in real-time. However, to take full advantage of multi-core processing, these systems must exploit intra-task parallelism, where parallelizable real-time tasks can utilize multiple cores at the same time. By exploiting intra-task parallelism, multi-core processors can achieve significant real-time performance improvement over traditional single-core processors for many computation-intensive real-time applications such as video surveillance, radar tracking, and hybrid real-time structural testing [95] where the performance limitations of traditional single-core processors have been a major hurdle.

The growing importance of parallel task models for real-time applications poses new challenges to real-time scheduling theory that had previously mostly focused on sequential task models. The state-of-the-art work [112] on parallel scheduling for real-time tasks with intra-task parallelism analyzes the *resource augmentation bound* using partitioned Deadline Monotonic (DM) scheduling. A *resource augmentation* under a scheduling policy quantifies processor-speed up factor (how much we have to increase the processor speed) with respect to an optimal algorithm to guarantee the schedulability of a task set under that policy. The state-of-the-art work [112] considers a synchronous task model, where each parallel task consists of a series of sequential or parallel segments. We call this model *synchronous*, since all the threads of a parallel segment must finish before the next segment starts, creating a synchronization point. However, that task model is *restrictive* in that, for every task, all the segments have an *equal* number of parallel threads, and the execution requirements of

all threads in a segment are equal. Most importantly, in that task model, the number of threads in every segment is *no greater* than the total number of processor cores.

While the work presented by [112] represents a promising step towards parallel real-time scheduling on multi-core processors, the restrictions on the task model make the solutions unsuitable for many real-time applications that often employ different numbers of threads in different segments of computation. In addition, it analyzes the resource augmentation bound under partitioned DM scheduling only, and does not consider other scheduling policies such as global EDF. In this work, we consider real-time scheduling on multi-core processors for a more general synchronous task model. Our tasks still contain segments where the threads of each segment synchronize at its end. However, in contrast to the restrictive task model addressed in [112], for any task in our model, each segment can contain an *arbitrary* number of parallel threads. That is, different segments of the same parallel task can contain different numbers of threads, and segments can contain more threads than the number of processor cores. Furthermore, the execution requirements of the threads in any segment can vary. This model is more portable, since the same task can be executed on machines with small as well as large numbers of cores. Specifically, our work makes the following new contributions to real-time scheduling for periodic parallel tasks.

- For the general synchronous task model, we propose a task decomposition algorithm that converts each implicit deadline parallel task into a set of constrained deadline sequential tasks.
- We derive a resource augmentation bound of 4 when these decomposed tasks are scheduled using global EDF scheduling. To our knowledge, this is the first resource augmentation bound for global EDF scheduling of parallel tasks.
- Using the proposed task decomposition, we also derive a resource augmentation bound of 5 for our more general task model under partitioned DM scheduling.
- Finally, we extend our analyses for a Directed Acyclic Graph (DAG) task model where each node in a DAG has a unit execution requirement. This is an even more general model for parallel tasks. Namely, we show that we can transform unit-node DAG tasks into synchronous tasks, and then use our proposed decomposition to get the same resource augmentation bounds for the former.

We evaluate the performance of the proposed decomposition through simulations based on synthetic workloads. The results indicate that the derived bounds are safe and sufficient. In particular, the resource augmentations required to schedule the decomposed tasks in our simulations are at most 2.4 and 3.4 for global EDF and partitioned DM scheduling, respectively, which are significantly smaller than the corresponding theoretical bounds.

In the rest of the paper, Section 9.2 describes the parallel synchronous task model. Section 9.3 presents the proposed task decomposition. Section 9.4 presents the analysis for global EDF scheduling. Section 9.5 presents the analysis for partitioned DM scheduling. Section 9.6 extends our results and analyses for unit-node DAG task models. Section 9.7 presents the simulation results. Section 9.8 reviews related work. Finally, we conclude in Section 9.9.

9.2 Parallel Synchronous Task Model

We primarily consider a synchronous parallel task model, where each task consists of a sequence of computation segments, each segment having an arbitrary number of parallel threads with arbitrary execution requirements that synchronize at the end of the segment. Such tasks are generated by parallel *for* loops, a construct common to many parallel languages such as OpenMP [17] and CilkPlus [12].

We consider n periodic synchronous parallel tasks with implicit deadlines (i.e. deadlines are equal to periods). Each task τ_i, $1 \leq i \leq n$, is a sequence of s_i segments, where the j-th segment, $1 \leq j \leq s_i$, consists of $m_{i,j}$ parallel threads. First we consider the case when, for any segment of τ_i, all parallel threads in the segment have equal execution requirements. For such τ_i, the j-th segment, $1 \leq j \leq s_i$, is represented by $\langle e_{i,j}, m_{i,j} \rangle$, with $e_{i,j}$ being the worst case execution requirement of each of its threads. When $m_{i,j} > 1$, the threads in the j-th segment can be executed in parallel on different cores. The j-th segment starts only after all threads of the $(j-1)$-th segment have completed. Thus, a parallel task τ_i in which a segment consists of equal-length threads is shown in Figure 9.1, and is represented as $\tau_i : (\langle e_{i,1}, m_{i,1} \rangle, \langle e_{i,2}, m_{i,2} \rangle, \cdots, \langle e_{i,s_i}, m_{i,s_i} \rangle)$ where

- s_i is the total number of segments in task τ_i.

- In a segment $\langle e_{i,j}, m_{i,j} \rangle$, $1 \leq j \leq s_i$, $e_{i,j}$ is the worst case execution requirement of each thread, and $m_{i,j}$ is the number of threads. Therefore, any segment $\langle e_{i,j}, m_{i,j} \rangle$ with $m_{i,j} > 1$ is a *parallel segment* with a total of $m_{i,j}$ parallel threads, and any segment $\langle e_{i,j}, m_{i,j} \rangle$ with $m_{i,j} = 1$ is a *sequential segment* since it has only one thread. A task τ_i with $s_i = 1$ and $m_{i,s_i} = 1$ is a sequential task.

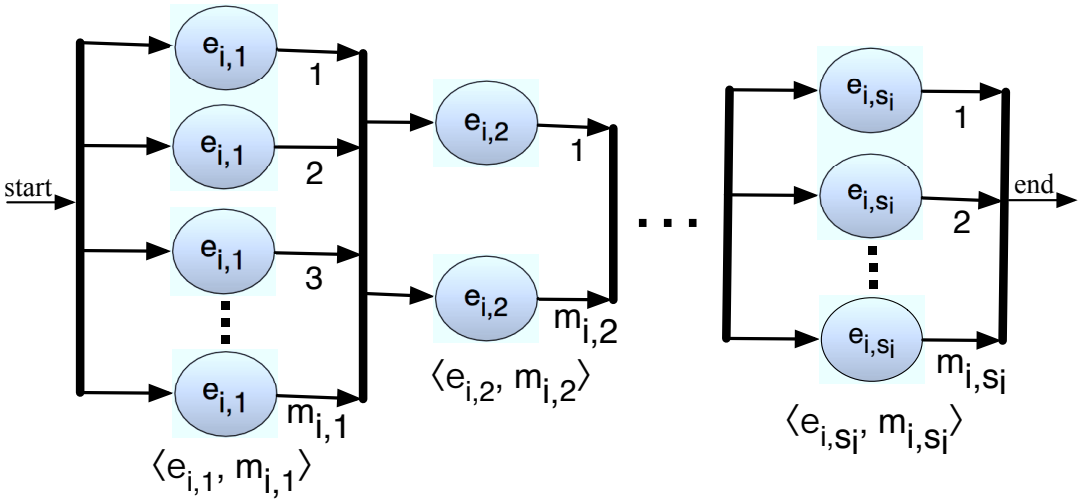
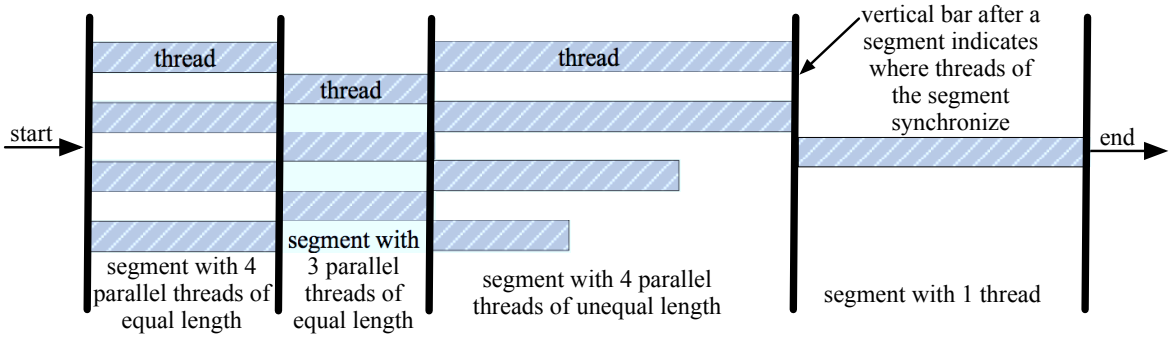


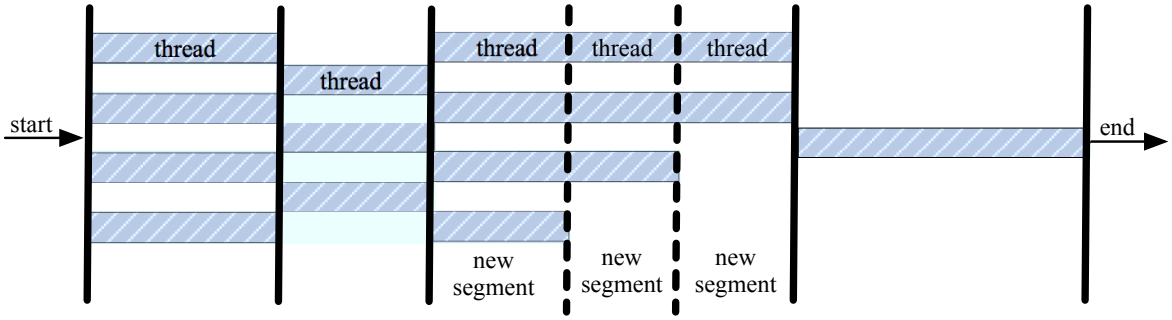
Figure 9.1: A parallel synchronous task τ_i

Now, we consider the case when the execution requirements of parallel threads in a segment of τ_i may differ from each other. An example of such a task is shown in Figure 9.2(a), where each horizontal bar indicates the length of the execution requirement of a thread. As the figure shows, the parallel threads in the third segment have unequal execution requirements. By adding a new synchronization point at the end of each thread in a segment, any segment consisting of threads of unequal length can be converted to several segments each consisting of threads of equal length as shown in Figure 9.2(b). Specifically, for the task with unequal-length threads in a segment shown in Figure 9.2(a), Figure 9.2(b) shows the corresponding task in which each segment consists of equal-length threads. Thus, in any synchronous parallel task, any segment consisting of threads of different execution requirements can be converted to several segments each consisting of threads of an equal execution requirement without changing any task parameter such as period, deadline, or execution requirement. It is worth noting that such a conversion is not entirely loss-less since it adds additional synchronization points. It is entirely possible that some task system that would be schedulable

with another transformation might not be schedulable with our proposed one. However, since the execution requirements do not change, it has no effect on the utilization of the system, and our bounds depend on the utilization of the system. Hence, we concentrate only to the task model where each segment in a task consists of equal-length threads (such as the one shown in Figure 9.1).



(a) A synchronous task with unequal-length threads in a segment



(b) The corresponding synchronous task with equal-length threads in each segment (each dotted vertical line indicates a newly added synchronization point at the end of a thread)

Figure 9.2: Conversion of a segment with unequal-length threads to segments with equal-length threads in a synchronous parallel task

Therefore, considering a multi-core platform consisting of m processor cores, we focus on scheduling n parallel tasks denoted by $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where each τ_i is represented as $\tau_i : (\langle e_{i,1}, m_{i,1}\rangle, \langle e_{i,2}, m_{i,2}\rangle, \cdots, \langle e_{i,s_i}, m_{i,s_i}\rangle)$ (as the one shown in Figure 9.1). The period of task τ_i is denoted by T_i. The deadline D_i of τ_i is equal to its period T_i. Each task τ_i generates (potentially) an infinite sequence of jobs, with arrival times of successive jobs separated by T_i time units. Jobs are fully independent and preemptive: any job can be

suspended (preempted) at any time instant, and is later resumed with no cost or penalty. The task set is said to be *schedulable* when all tasks meet their deadlines.

9.3 Task Decomposition

In this section, we present a decomposition of the parallel tasks into a set of sequential tasks. In particular, we propose a strategy that decomposes each implicit deadline parallel task (like the one shown in Figure 9.1) into a set of constrained deadline (i.e. deadlines are no greater than periods) sequential tasks by converting each thread of the parallel task into its own sequential task and assigning appropriate deadlines to these tasks. This strategy allows us to use existing schedulability analysis for multiprocessor scheduling (both global and partitioned) to prove the resource augmentation bounds for parallel tasks (to be discussed in Sections 9.4 and 9.5). Here, we first present some useful terminology. We then present our decomposition and a density analysis for it.

9.3.1 Terminology

Definition 1. *The* minimum execution time *(i.e. the* critical path length*)* P_i *of task* τ_i *on a multi-core platform where each processor core has unit speed is defined as*

$$P_i = \sum_{j=1}^{s_i} e_{i,j}$$

Observation 1. *On a unit-speed multi-core platform, any task* τ_i *requires at least* P_i *units of time even when the number of cores* m *is infinite.*

On a multi-core platform where each processor core has speed ν, the critical path length of task τ_i is denoted by $P_{i,\nu}$ and is expressed as follows.

$$P_{i,\nu} = \frac{1}{\nu} \sum_{j=1}^{s_i} e_{i,j} = \frac{P_i}{\nu}$$

Definition 2. *The* maximum execution time *(i.e. the* work*)* C_i *of task* τ_i *on a multi-core platform where each processor core has unit speed is defined as*

$$C_i = \sum_{j=1}^{s_i} m_{i,j}.e_{i,j}$$

That is, C_i is the execution time of τ_i on a unit-speed single core processor if it is never preempted. On a multi-core platform where each processor core has speed ν, the maximum execution time of task τ_i is denoted by $C_{i,\nu}$ and is expressed as follows.

$$C_{i,\nu} = \frac{1}{\nu}\sum_{j=1}^{s_i} m_{i,j}.e_{i,j} = \frac{C_i}{\nu} \tag{9.1}$$

Definition 3. *The* utilization u_i *of task* τ_i*, and the* total utilization $u_{sum}(\tau)$ *for the set of* n *tasks* τ *on a unit-speed multi-core platform are defined as*

$$u_i = \frac{C_i}{T_i}; \qquad\qquad u_{sum}(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Observation 2. *If the total utilization* u_{sum} *is greater than* m*, then no algorithm can schedule* τ *on* m *identical unit speed processor cores.*

Definition 4. *The* density δ_i *of task* τ_i*, the* maximum density $\delta_{\max}(\tau)$ *and the* total density $\delta_{sum}(\tau)$ *of the set of* n *tasks* τ *on a unit-speed multi-core platform are defined as follows:*

$$\delta_i = \frac{C_i}{D_i}; \quad \delta_{sum}(\tau) = \sum_{i=1}^{n} \delta_i; \quad \delta_{\max}(\tau) = \max\{\delta_i | 1 \leq i \leq n\}$$

For an implicit deadline task τ_i*,* $\delta_i = u_i$*.*

9.3.2 Decomposition

Following is the high-level idea of the decomposition of a parallel task τ_i.

1. In our decomposition, each thread of the task becomes its own sequential subtask. These individual subtasks are assigned release times and deadlines. Since each thread of a segment is identical (with respect to its execution time), we consider each segment one at a time, and assign the same release times and deadlines to all subtasks generated from threads of the same segment.

2. Since a segment $\langle e_{i,j}, m_{i,j} \rangle$ has to complete before segment $\langle e_{i,j+1}, m_{i,j+1} \rangle$ can start, the release time of the subtasks of segment $\langle e_{i,j+1}, m_{i,j+1} \rangle$ is equal to the absolute deadline of the subtasks of segment $\langle e_{i,j}, m_{i,j} \rangle$.

3. As stated before, we analyze the schedulability of the decomposed tasks based on their densities. The analysis is largely dependent on the total density (δ_{sum}) and the maximum density (δ_{max}) of the decomposed tasks. Therefore, we want to keep both δ_{sum} and δ_{max} bounded and as small as possible. In particular, we need δ_{max} to be at most 1, and we want δ_{sum} over all tasks to be at most u_{sum} after decomposition. To do so, usually we need to assign enough slack among the segments. It turns out to be difficult to assign enough slack to each segment if we consider unit-speed processors. However, we can increase the available slack by considering higher speed processors. In particular, we find that decomposing on speed 2 processors allows us enough slack to decompose effectively, keeping both δ_{sum} and δ_{max} at desired levels. Decomposing on processors of higher speed is also possible but leads to lower efficiency. On the other hand, decomposing on speed 1 processors would be clearly non-optimal (in terms of the availability of slack). In particular, if we do the decomposition with α-speed processor, where $\alpha \geq 1$, then the best value of α to minimize both δ_{sum} and δ_{max} becomes 2. Therefore, in the remainder of the paper we restrict ourselves to decomposition on 2-speed processors. Thus, the *slack* for task τ_i, denoted by L_i, that is distributed among the segments is its slack on speed 2 processors, given by

$$L_i = T_i - P_{i,2} = T_i - \frac{P_i}{2} \tag{9.2}$$

This slack is distributed among the segments according to a principle of "equitable density" meaning that we try to keep the density of each segment approximately rather than exactly equal by maintaining a uniform upper bound on the densities. To do this,

we take both the number of threads in each segment and the computation requirement of the threads in each segment into consideration while distributing the slack.

In order to take the computation requirement of the threads in each segment into consideration, we assign proportional slack fractions instead of absolute slack. We now formalize the notion of *slack fraction*, $f_{i,j}$, for the j-th segment (i.e. segment $\langle e_{i,j}, m_{i,j} \rangle$) of task τ_i. *Slack fraction* $f_{i,j}$ is the fraction of L_i (i.e. the total slack) to be allotted to segment $\langle e_{i,j}, m_{i,j} \rangle$ proportionally to its minimum computation requirement. Each thread in segment $\langle e_{i,j}, m_{i,j} \rangle$ has a minimum execution time of $\frac{e_{i,j}}{2}$ on 2-speed processor cores, and is assigned a slack value of $f_{i,j}\frac{e_{i,j}}{2}$. Each thread gets this "extra time" beyond its execution requirement on 2-speed processor cores. Thus, for each thread in segment $\langle e_{i,j}, m_{i,j} \rangle$, the relative deadline is assigned as

$$d_{i,j} = \frac{e_{i,j}}{2} + f_{i,j}.\frac{e_{i,j}}{2} = \frac{e_{i,j}}{2}(1 + f_{i,j}) \tag{9.3}$$

Equation 9.3 shows how a thread's deadline $d_{i,j}$ is calculated based on its assigned slack fraction $f_{i,j}$. For example, if a thread has $e_{i,j} = 4$ and it is assigned a slack fraction of $f_{i,j} = 1.5$, then its relative deadline is $2(1+1.5) = 5$. That is, the thread has been assigned a slack value of $d_{i,j} - \frac{e_{i,j}}{2} = 5 - \frac{4}{2} = 3$ (or, equivalently $f_{i,j}\frac{e_{i,j}}{2} = 3$) on 2-speed cores. Similarly, the value of 0 of $f_{i,j}$ implies that the thread has been assigned no slack on 2-speed processor cores. Note that since the slack fraction and hence the slack can not be negative, the density of a segment is at least 1. Therefore, in order to satisfy our maximum density requirement — that $\delta_{\max}$ is at most 1 after decomposition on speed-2 processors — we must ensure that the slack fraction is never negative.

Since a segment cannot start before all previous segments complete, the release offset of a segment $\langle e_{i,j}, m_{i,j} \rangle$ is assigned as

$$\phi_{i,j} = \sum_{k=1}^{j-1} d_{i,k} \tag{9.4}$$

Thus, the density of each thread in segment $\langle e_{i,j}, m_{i,j} \rangle$ on 2-speed cores is

$$\frac{\frac{e_{i,j}}{2}}{d_{i,j}} = \frac{\frac{e_{i,j}}{2}}{\frac{e_{i,j}}{2}(1 + f_{i,j})} = \frac{1}{1 + f_{i,j}}$$

Since a segment $\langle e_{i,j}, m_{i,j} \rangle$ consists of $m_{i,j}$ threads, the segment's density on 2-speed processor cores is

$$\frac{m_{i,j}}{1 + f_{i,j}} \tag{9.5}$$

Note that to meet the deadline of the parallel task on 2-speed processor cores, the segment slack should be assigned so that

$$f_{i,1}.\frac{e_{i,1}}{2} + f_{i,2}.\frac{e_{i,2}}{2} + f_{i,3}.\frac{e_{i,3}}{2} + \cdots + f_{i,s_i}.\frac{e_{i,s_i}}{2} \leq L_i.$$

In our decomposition, we always assign the maximum possible segment slack on 2-speed processor cores and, therefore, for our decomposition, the above inequality is in fact an equality.

Since after assigning slack, we want to keep the density of each segment about equal, we must take the number of threads of the segment into consideration while assigning slack fractions. As stated before, we need to keep the sum of densities bounded on speed-2 processors after decomposition. To determine whether slack assignment to a segment is critical or not, we calculate a threshold based on task parameters. The segments whose number of threads is greater than this threshold are computation intensive, and hence assigning slack to such segments is critical. The remaining segments are deemed to be less computation intensive, and hence assigning slack to such segments is less critical. Hence, to calculate segment slack according to equitable density, we classify segments into two categories based on their computation requirements and slack demand:

- *Heavy segments* are those which have $m_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$. That is, they have many parallel threads, and hence are computation intensive.
- *Light segments* are those which have $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$. That is, these segments are less computation intensive.

The threshold $\frac{C_{i,2}}{T_i - P_{i,2}}$ is chosen to ensure that no thread is assigned any negative slack on 2-speed processor cores. We later (in Subsection 9.3.2) prove that every thread is indeed assigned a non-negative slack which, in fact, guarantees that its density is at most 1 on 2-speed processor cores.

Using the above categorization, we also classify parallel tasks into two categories: tasks that have some or all heavy segments versus tasks that have only light segments, and analyze them separately as follows.

Tasks with some (or all) heavy segments

For the tasks which have some heavy segments, we treat heavy and light segments differently while assigning slack. In particular, we assign no slack to the light segments; that is, segments with $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$ of τ_i are assigned $f_{i,j} = 0$. The total available slack L_i is distributed among the heavy segments (segments with $m_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$) in such a way that each of these segments has the same density.

For simplicity of presentation, we first distinguish notations between the heavy and light segments. Let the heavy segments of τ_i be represented as $\{\langle e^h_{i,1}, m^h_{i,1}\rangle, \langle e^h_{i,2}, m^h_{i,2}\rangle, \cdots, \langle e^h_{i,s^h_i}, m^h_{i,s^h_i}\rangle\}$, where $s^h_i \leq s_i$ (superscript h standing for 'heavy'). Then, let

$$P^h_{i,2} = \frac{1}{2}\sum_{j=1}^{s^h_i} e^h_{i,j}; \qquad C^h_{i,2} = \frac{1}{2}\sum_{j=1}^{s^h_i} m^h_{i,j}.e^h_{i,j} \tag{9.6}$$

The light segments are denoted as $\{\langle e^\ell_{i,1}, m^\ell_{i,1}\rangle, \langle e^\ell_{i,2}, m^\ell_{i,2}\rangle, \cdots, \langle e^\ell_{i,s^\ell_i}, m^\ell_{i,s^\ell_i}\rangle\}$, where $s^\ell_i = s_i - s^h_i$ (superscript ℓ standing for 'light'). Then, let

$$P^\ell_{i,2} = \frac{1}{2}\sum_{j=1}^{s^\ell_i} e^\ell_{i,j}; \qquad C^\ell_{i,2} = \frac{1}{2}\sum_{j=1}^{s^\ell_i} m^\ell_{i,j}.e^\ell_{i,j} \tag{9.7}$$

Now, the following equalities must hold for task τ_i.

$$P_{i,2} = \frac{P_i}{2} = P^h_{i,2} + P^\ell_{i,2}; \qquad C_{i,2} = \frac{C_i}{2} = C^h_{i,2} + C^\ell_{i,2} \tag{9.8}$$

Now we calculate slack fraction $f^h_{i,j}$ for all heavy segments (i.e. segments $\langle e^h_{i,j}, m^h_{i,j}\rangle$, where $1 \leq j \leq s^h_i$ and $m^h_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$) so that they all have equal density on 2-speed processor cores.

That is,

$$\frac{m^h_{i,1}}{1+f^h_{i,1}} = \frac{m^h_{i,2}}{1+f^h_{i,2}} = \frac{m^h_{i,3}}{1+f^h_{i,3}} = \cdots = \frac{m^h_{i,s^h_i}}{1+f^h_{i,s^h_i}} \tag{9.9}$$

In addition, since all the slack is distributed among the heavy segments, the following equality must hold.

$$f^h_{i,1}.e^h_{i,1} + f^h_{i,2}.e^h_{i,2} + f^h_{i,3}.e^h_{i,3} + \cdots + f^h_{i,s^h_i}.e^h_{i,s^h_i} = 2.L_i \tag{9.10}$$

It follows that the value of each $f^h_{i,j}$, $1 \leq j \leq s^h_i$, can be determined by solving Equations 9.9 and 9.10 as shown below. From Equation 9.9, the value of $f^h_{i,j}$ for each j, $2 \leq j \leq s^h_i$, can be expressed in terms of $f^h_{i,1}$ as follows.

$$f^h_{i,j} = (1+f^h_{i,1})\frac{m^h_{i,j}}{m^h_{i,1}} - 1 \tag{9.11}$$

Putting the value of each $f^h_{i,j}$, $2 \leq j \leq s^h_i$, from Equation 9.11 into Equation 9.10:

$$\begin{aligned}
2L_i &= f^h_{i,1}e^h_{i,1} + \sum_{j=2}^{s^h_i}\left(\left((1+f^h_{i,1})\frac{m^h_{i,j}}{m^h_{i,1}} - 1\right)e^h_{i,j}\right) \\
&= f^h_{i,1}e^h_{i,1} + \sum_{j=2}^{s^h_i}\left(\frac{m^h_{i,j}}{m^h_{i,1}}e^h_{i,j} + f^h_{i,1}\frac{m^h_{i,j}}{m^h_{i,1}}e^h_{i,j} - e^h_{i,j}\right) \\
&= f^h_{i,1}e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum_{j=2}^{s^h_i} m^h_{i,j}e^h_{i,j} + \frac{f^h_{i,1}}{m^h_{i,1}}\sum_{j=2}^{s^h_i} m^h_{i,j}e^h_{i,j} - \sum_{j=2}^{s^h_i} e^h_{i,j}
\end{aligned}$$

From the above equation, we can determine the value of $f^h_{i,1}$ as follows.

$$
\begin{aligned}
f^h_{i,1} &= \frac{2L_i + \sum_{j=2}^{s^h_i} e^h_{i,j} - \frac{1}{m^h_{i,1}} \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j}}{e^h_{i,1} + \frac{1}{m^h_{i,1}} \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j}} \\
&= \frac{2L_i + (\sum_{j=2}^{s^h_i} e^h_{i,j} + e^h_{i,1}) - (e^h_{i,1} + \frac{1}{m^h_{i,1}} \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j})}{e^h_{i,1} + \frac{1}{m^h_{i,1}} \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j}} \\
&= \frac{2L_i + \sum_{j=1}^{s^h_i} e^h_{i,j}}{e^h_{i,1} + \frac{1}{m^h_{i,1}} \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j}} - 1
\end{aligned}
$$

In the above equation, replacing $\sum_{j=1}^{s^h_i} e^h_{i,j}$ with $2P^h_{i,2}$ from Equation 9.6, we get

$$
f^h_{i,1} = \frac{2L_i + 2P^h_{i,2}}{e^h_{i,1} + \frac{1}{m^h_{i,1}} \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j}} - 1 = \frac{m^h_{i,1}(2L_i + 2P^h_{i,2})}{m^h_{i,1} e^h_{i,1} + \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j}} - 1
$$

Similarly, in the above equation, replacing $(m^h_{i,1} e^h_{i,1} + \sum_{j=2}^{s^h_i} m^h_{i,j} e^h_{i,j})$ with $2C^h_{i,2}$ from Equation 9.6, the value of $f^h_{i,1}$ can be written as follows.

$$
\begin{aligned}
f^h_{i,1} &= \frac{m^h_{i,1}(2L_i + 2P^h_{i,2})}{2C^h_{i,2}} - 1 = \frac{m^h_{i,1}(L_i + P^h_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 && \text{(From 9.8)} \\
&= \frac{m^h_{i,1}((T_i - P_{i,2}) + P^h_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 && \text{(From 9.2)} \\
&= \frac{m^h_{i,1}(T_i - (P^h_{i,2} + P^\ell_{i,2}) + P^h_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 && \text{(From 9.8)} \\
&= \frac{m^h_{i,1}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1
\end{aligned}
$$

Now putting the above value of $f^h_{i,1}$ in Equation 9.11, for any heavy segment $\langle e^h_{i,j}, m^h_{i,j}\rangle$, we get

$$f^h_{i,j} = \frac{m^h_{i,j}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 \tag{9.12}$$

Intuitively, the slack never should be negative, since the deadline should be no less than the computation requirement of the thread. Since $m^h_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$, according to Equation 9.12, the quantity $\frac{m^h_{i,j}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} > 1$. This implies that $f^h_{i,j} > 0$. Now, using Equation 9.5, the density of every segment $\langle e^h_{i,j}, m^h_{i,j}\rangle$ is

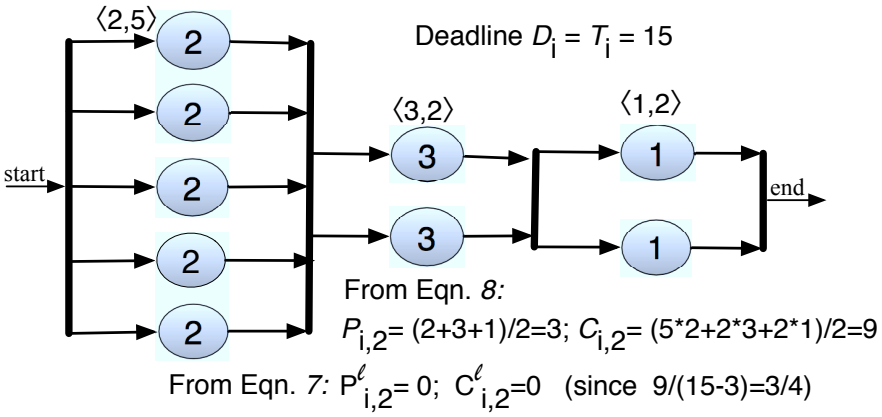
$$\frac{m^h_{i,j}}{1 + f^h_{i,j}} = \frac{m^h_{i,j}}{1 + \frac{m^h_{i,j}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1} = \frac{C_{i,2} - C^\ell_{i,2}}{T_i - P^\ell_{i,2}} \tag{9.13}$$

Figure 9.3 shows a simple example of decomposition for a task τ_i consisting of 3 segments.

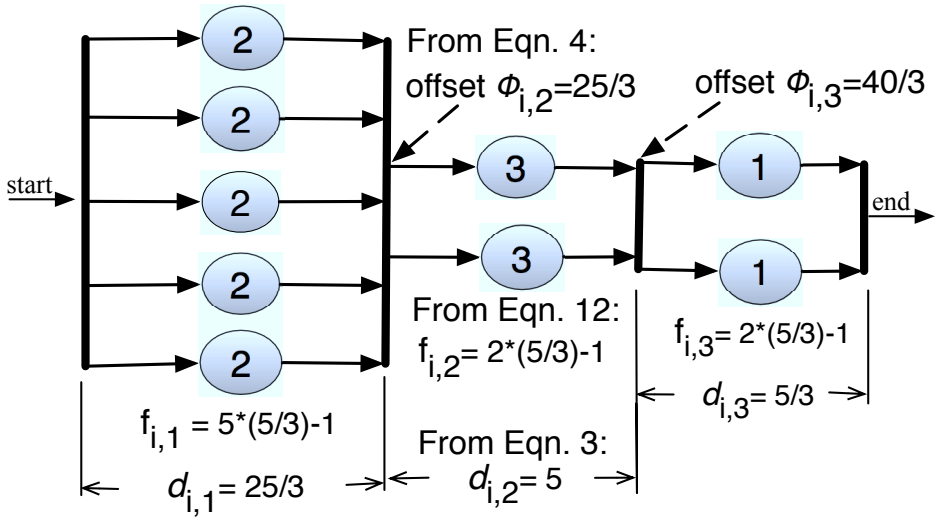
Tasks with no heavy segments

When the parallel task does not contain any heavy segments, we just assign the slack proportionally (according to the length of $e_{i,j}$) among all segments. That is,

$$f_{i,j} = \frac{L_i}{P_{i,2}} \tag{9.14}$$



(a) A parallel synchronous task τ_i



(b) Decomposed task τ_i^{decom}

Figure 9.3: An example of decomposition

By Equation 9.5, the density of each segment $\langle e_{i,j}, m_{i,j} \rangle$ is

$$\frac{m_{i,j}}{1+f_{i,j}} = \frac{m_{i,j}}{1+\frac{L_i}{P_{i,2}}} = m_{i,j}\frac{P_{i,2}}{L_i+P_{i,2}} = m_{i,j}\frac{P_{i,2}}{T_i} \tag{9.15}$$

9.3.3 Density Analysis

Once the above decomposition is done on task τ_i: $(\langle e_{i,1}, m_{i,1} \rangle, \cdots, \langle e_{i,s_i}, m_{i,s_i} \rangle)$, each thread of each segment $\langle e_{i,j}, m_{i,j} \rangle$, $1 \leq j \leq s_i$, is considered as a sequential multiprocessor subtask. We use τ_i^{decom} to denote task τ_i after decomposition. That is, τ_i^{decom} denotes the set of subtasks generated from τ_i through decomposition. Similarly, we use τ^{decom} to denote the entire task set τ after decomposition. That is, τ^{decom} is the set of all subtasks that our decomposition generates. Since $f_{i,j} \geq 0$, $\forall 1 \leq j \leq s_i$, $\forall 1 \leq i \leq n$, the maximum density $\delta_{\max,2}$ of any subtask (thread) among τ^{decom} on 2-speed processor core is

$$\delta_{\max,2} = \max\{\frac{1}{1+f_{i,j}}\} \leq 1 \tag{9.16}$$

We must now bound δ_{sum}. Lemma 24 shows that the density of every segment is at most $\frac{C_i/2}{T_i - P_i/2}$ for any task with or without heavy segments.

Lemma 24. *After the decomposition, the density of every segment $\langle e_{i,j}, m_{i,j} \rangle$, where $1 \leq j \leq s_i$, of every task τ_i on 2-speed processor cores is upper bounded by $\frac{C_i/2}{T_i - P_i/2}$.*

Proof. First, we analyze the case when the task contains some heavy segments. According to Equation 9.13, for every heavy segment $\langle e_{i,j}, m_{i,j} \rangle$, the density is

$$\begin{aligned}
\frac{C_{i,2} - C_{i,2}^{\ell}}{T_i - P_{i,2}^{\ell}} &\leq \frac{C_{i,2}}{T_i - P_{i,2}^{\ell}} && (\text{since } C_{i,2}^{\ell} \geq 0) \\
&\leq \frac{C_{i,2}}{T_i - P_{i,2}} && (\text{since } P_{i,2} \geq P_{i,2}^{\ell}) \\
&= \frac{C_i/2}{T_i - P_i/2}
\end{aligned}$$

For every light segment $\langle e_{i,j}, m_{i,j} \rangle$ (i.e., a segment with $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$), the slack fraction $f_{i,j} = 0$. That is, its deadline is equal to its computation requirement $\frac{e_{i,j}}{2}$ on 2-speed processor cores. Therefore, its density, by definition, is

$$\frac{m_{i,j}}{1 + f_{i,j}} = m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}} = \frac{C_i/2}{T_i - P_i/2}$$

For the case when there are no heavy segments in τ_i, for every segment $\langle e_{i,j}, m_{i,j} \rangle$ of τ_i, $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$. Since $T_i \geq P_{i,2}$ (Observation 1), the density of each segment $\langle e_{i,j}, m_{i,j} \rangle$ (Equation 9.15) of τ_i:

$$m_{i,j} \frac{P_{i,2}}{T_i} \leq m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}} = \frac{C_i/2}{T_i - P_i/2}$$

Hence, follows the lemma. □

Thus, our decomposition distributes the slack so that each segment has a density that is bounded above. Theorem 25 establishes an upper bound on the density of every task after decomposition.

Theorem 25. *The density $\delta_{i,2}$ of every τ_i^{decom}, $1 \leq i \leq n$, (i.e. the density of every task τ_i after decomposition) on 2-speed processor cores is upper bounded by $\frac{C_i/2}{T_i - P_i/2}$.*

Proof. After the decomposition, the densities of all segments of τ_i comprise the density of τ_i^{decom}. However, no two segments are simultaneous active, and each segment occurs exactly once during the activation time of task τ_i. Therefore, we can replace each segment with the segment that has the maximum density. Thus, task τ_i^{decom} can be considered as s_i occurences of the segment that has the maximum density, and therefore, the density of the entire task set τ_i^{decom} is equal to that of the segment having the maximum density which is at most $\frac{C_i/2}{T_i - P_i/2}$ (Lemma 24). Therefore, $\delta_{i,2} \leq \frac{C_i/2}{T_i - P_i/2}$. □

Lemma 26. *If τ^{decom} is schedulable, then τ is also schedulable.*

Proof. For each τ_i^{decom}, $1 \leq i \leq n$, its deadline and execution requirement are the same as those of original task τ_i. Besides, in each τ_i^{decom}, a subtask is released only after all its preceding segments are complete. Hence, the precedence relations in original task τ_i are

retained in τ_i^{decom}. Therefore, if τ^{decom} is schedulable, then a schedule must exist for τ where each task in τ can meet its deadline. □

9.4 Global EDF Scheduling

After our proposed decomposition, we consider the scheduling of synchronous parallel tasks. [112] show that there exist task sets with total utilization slightly greater than (and arbitrarily close to) 1 that are unschedulable with m processor cores. Since our model is a generalization of theirs, this lower bound still holds for our tasks, and conventional utilization bound approaches are not useful for schedulability analysis of parallel tasks. Hence, we (like [112]) use the *resource augmentation bound approach*, originally introduced in [141]. We first consider *global scheduling* where tasks are allowed to migrate among processor cores. We then analyze schedulability in terms of a resource augmentation bound. Since the synchronous parallel tasks are now split into individual sequential subtasks, we can use global Earliest Deadline First (EDF) scheduling for them. The global EDF policy for subtask scheduling is basically the same as the traditional global EDF where jobs with earlier deadlines are assigned higher priorities.

Under global EDF scheduling, we now present a schedulability analysis in terms of a resource augmentation bound for our decomposed tasks. For any task set, the *resource augmentation bound* ν of a scheduling policy $\mathbb{A}$ on a multi-core processor with m cores represents a processor speedup factor. That is, if there exists any (optimal) algorithm under which a task set is feasible on m identical unit-speed processor cores, then $\mathbb{A}$ is guaranteed to successfully schedule this task set on a m-core processor, where each processor core is ν times as fast as the original. In Theorem 28, we show that our decomposition needs a resource augmentation bound of 4 under global EDF scheduling.

Our analysis uses a result for constrained deadline sporadic sequential tasks on m processor cores proposed in [41] as re-stated here in Theorem 27. This result is a generalization of the result for implicit deadline sporadic tasks proposed in [85].

Theorem 27. *[41] Any constrained deadline sporadic sequential task set π with total density $\delta_{\text{sum}}(\pi)$ and maximum density $\delta_{\max}(\pi)$ is schedulable using global EDF strategy on m unit-speed processor cores if*

$$\delta_{\text{sum}}(\pi) \leq m - (m-1)\delta_{\max}(\pi)$$

Since we decompose our synchronous parallel tasks into sequential tasks with constrained deadlines, this result applies to our decomposed task set τ^{decom}. If we can schedule τ^{decom}, then we can schedule τ (Lemma 26).

Theorem 28. *If there exists any way to schedule a synchronous parallel task set τ on m unit-speed processor cores, then the decomposed task set τ^{decom} is schedulable using global EDF on m processor cores each of speed* 4.

Proof. Let there exist some algorithm $\mathbb{A}$ under which the original task set τ is feasible on m identical unit-speed processor cores. If τ is schedulable under $\mathbb{A}$, the following condition must hold (by Observation 2).

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \tag{9.17}$$

We decompose tasks considering that each processor core has speed 2. To be able to schedule the decomposed tasks τ^{decom}, suppose we need to increase the speed of each processor core ν times further. That is, we need each processor core to be of speed 2ν.

According to the definition of density, if a task has density x on a processor system S, then its density will be $\frac{x}{\nu}$ on a processor system that is ν times as fast as S. On an m-core platform where each processor core has speed 2ν, let the total density and the maximum density of task set τ^{decom} be denoted by $\delta_{\text{sum},2\nu}$ and $\delta_{\max,2\nu}$, respectively. From 9.16, we have

$$\delta_{\max,2\nu} = \frac{\delta_{\max,2}}{\nu} \leq \frac{1}{\nu} \tag{9.18}$$

The value $\delta_{\text{sum},2\nu}$ turns out to be the total density of all decomposed tasks. A task has a density of at most $\frac{C_i/2}{T_i - P_i/2}$ on 2-speed processor cores after decomposition. Therefore, its density on 2ν-speed cores is at most $\frac{\frac{C_i/2}{T_i - P_i/2}}{\nu} = \frac{C_i/2\nu}{T_i - P_i/2}$, where we cannot replace $P_i/2$ with $P_i/2\nu$. Although the critical path length on 2ν-speed cores is $P_i/2\nu$, our decomposition

was performed on 2-speed cores where critical path length was $P_i/2$. By Theorem 25 and Equation 9.1, the density of every task τ_i^{decom} on m identical processors each of speed 2ν is

$$\begin{aligned}\delta_{i,2\nu} &\leq \frac{\frac{C_i}{2\nu}}{T_i - \frac{P_i}{2}} \leq \frac{\frac{C_i}{2\nu}}{T_i - \frac{T_i}{2}} && \text{(since } P_i \leq T_i\text{)} \\ &= \frac{\frac{C_i}{2\nu}}{\frac{T_i}{2}} = \frac{1}{\nu}.\frac{C_i}{T_i}\end{aligned}$$

Thus, from 9.17,

$$\delta_{\text{sum},2\nu} = \sum_{i=1}^{n} \delta_{i,2\nu} \leq \frac{1}{\nu} \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{m}{\nu} \tag{9.19}$$

Note that, in the decomposed task set, every thread of the original task is a sequential task on a multiprocessor platform. Therefore, $\delta_{\text{sum},2\nu}$ is the total density of all threads (i.e. subtasks), and $\delta_{\text{max},2\nu}$ is the maximum density among all threads. Thus, by Theorem 27, the decomposed task set τ^{decom} is schedulable under global EDF on m processor cores each of speed 2ν if

$$\delta_{\text{sum},2\nu} \leq m - (m-1)\delta_{\text{max},2\nu} \tag{9.20}$$

Now using the values of $\delta_{\text{sum},2\nu}$ (Equation 9.19) and $\delta_{\text{max},2\nu}$ (Equation 9.18) into Condition (9.20), task set τ^{decom} is schedulable if

$$\begin{aligned}&\frac{m}{\nu} \leq m - (m-1)\frac{1}{\nu} \\ \Leftrightarrow\ &\frac{1}{\nu} + \frac{1}{\nu} - \frac{1}{m\nu} \leq 1 \quad \Leftrightarrow \quad \frac{2}{\nu} - \frac{1}{m\nu} \leq 1\end{aligned}$$

From the above condition, τ^{decom} must be schedulable if

$$\frac{2}{\nu} \leq 1 \quad \Leftrightarrow \quad \nu \geq 2 \quad \Leftrightarrow \quad 2\nu \geq 4$$

Hence follows the theorem. □

9.5 Partitioned Deadline Monotonic Scheduling

Using the same decomposition described in Section 9.3, we now derive a resource augmentation bound required to schedule task sets under partitioned Deadline Monotonic (DM) scheduling. Unlike global scheduling, in *partitioned scheduling*, each task is assigned to a processor core. Tasks are executed only on their assigned processor cores, and are not allowed to migrate among cores. We consider the FBB-FFD (Fisher Baruah Baker - First-Fit Decreasing) partitioned DM scheduling [80] which the previous work on parallel task scheduling [112] also uses as the scheduling strategy for parallel tasks in a more restricted model. In fact, the FBB-FFD Algorithm was developed for periodic tasks without release offsets while our decomposed subtasks have offsets. Therefore, first we present how the FBB-FFD Algorithm should be adapted to partition our subtasks with offsets, and then we analyze the resource augmentation bound.

9.5.1 FBB-FFD based Partitioned DM Algorithm for Decomposed Tasks

The original FBB-FFD Algorithm [80] is a variant of the first-fit decreasing bin-packing heuristic, and hinges on the notion of a request-bound function for constrained deadline periodic sequential tasks. For a sequential task π_i with execution requirement e_i, utilization u_i, and deadline d_i, its *request-bound function* $\mathrm{RBF}(\pi_i, t)$ for any time interval of length t is the largest cumulative execution requirement of all jobs that can be generated by π_i to have their arrival times within a contiguous interval of length t. In the FBB-FFD Algorithm, $\mathrm{RBF}(\pi_i, t)$ is approximated as

$$\mathrm{RBF}^*(\pi_i, t) = e_i + u_i t \tag{9.21}$$

Let the processor cores be indexed as $1, 2, \cdots, m$, and Π_q be the set of tasks already assigned to processor core q, where $1 \leq q \leq m$. Considering the tasks in decreasing DM-priority order and starting from the highest priority task, the FBB-FFD algorithm assigns a task π_i to the

first processor core q, $1 \leq q \leq m$, that satisfies the following condition

$$d_i - \sum_{\pi_j \in \Pi_q} \text{RBF}^*(\pi_j, d_i) \geq e_i \tag{9.22}$$

If no processor core satisfies the above condition for some task, then the task set is decided to be infeasible for partitioning.

To partition our decomposed subtasks based on the FBB-FFD algorithm, we need to adopt Condition (9.22) for decomposed subtasks. In τ_i^{decom}, let any subtask that belongs to the k-th segment in τ_i be denoted by $\tau_{i,k}$. Let the deadline and the worst case execution requirement of $\tau_{i,k}$ be denoted by $d_{i,k}$ and $e_{i,k}$, respectively. We need to update the expression of RBF* in Condition (9.22) by taking into account the release offsets. The RBF* of the subtasks of τ_j^{decom} that are assigned to processor core q for any interval of length t is denoted by $\text{RBF}_q^*(\tau_j^{\text{decom}}, t)$. Note that any two subtasks of τ_j^{decom} having different offsets belong to different segments in original task τ_j and hence are never active simultaneously. Therefore, the calculation of $\text{RBF}_q^*(\tau_j^{\text{decom}}, d_{i,k})$ in Condition (9.22) when $j \neq i$ is different from that when $j = i$.

To calculate $\text{RBF}_q^*(\tau_j^{\text{decom}}, d_{i,k})$ when $j = i$, we only need to calculate RBF* of all subtasks in the k-th segment that are assigned to processor core q, and call it RBF* of that segment (of τ_i) on core q. Since the subtasks of τ_i^{decom} that are in different segments will never be released and executed simultaneously, the sum of RBF* among all subtasks of the k-th segment assigned to core q is the $\text{RBF}_q^*(\tau_i^{\text{decom}}, d_{i,k})$. Let $m_{i,k,q}$ be the number of subtasks of τ_i^{decom} that belong to the k-th segment in τ_i and have already been assigned to processor core q. Thus,

$$\text{RBF}_q^*(\tau_i^{\text{decom}}, d_{i,k}) = \text{RBF}^*(\tau_{i,k}, d_{i,k}).m_{i,k,q} \tag{9.23}$$

To calculate $\text{RBF}_q^*(\tau_j^{\text{decom}}, d_{i,k})$ when $j \neq i$, we need to consider the worst case RBF* of a segment execution sequence starting from subtasks of any segment of task τ_j that are assigned to processor core q. Let $\text{RBF}_q^*(\tau_{j,l}^+, d_{i,k})$ be the RBF* of the segment execution sequence $\tau_{j,l}^+$ starting from subtasks of the l-th segment (of task τ_j) on core q. Since subtasks of τ_j^{decom} that are from different segments are never simultaneously active, we need to consider this in calculating $\text{RBF}_q^*(\tau_{j,l}^+, d_{i,k})$, where $j \neq i$. To do so, the approximation of $u_i d_{i,k}$ (considering $t = d_{i,k}$) in Equation (9.21) is modified to the sum of utilization of all the

segments that have been assigned to core q, while the approximation of e_i is limited to the real possible interference starting from the l-th segment to the following segments that could be released before time $(d_{i,k} \bmod T_j)$. Let $r_{j,p}$ be the release time of the p-th segment of task τ_j, determined by our decomposition. Note that the first segment will be executed after the last segment in the same task, for at least one period after its last release time. So the relative release time of the p-th segment, given that the l-th segment starts at relative time zero, will be $(r_{j,p} + T_j - r_{j,l}) \bmod T_j$. As long as the relative release time is no greater than deadline $d_{i,k}$, the segment will be executed before the execution of the k-th segment, which is released at relative time zero. Hence, $\mathrm{RBF}^*_q(\tau^+_{j,l}, d_{i,k})$ is expressed as follows.

$$\mathrm{RBF}^*_q(\tau^+_{j,l}, d_{i,k}) = \sum_{\tau_{j,p} \in \Pi_q, (r_{j,p}+T_j-r_{j,l}) \bmod T_j \le d_{i,k}} e_{j,p}.m_{j,p,q} + \sum_{\tau_{j,p} \in \Pi_q} u_{j,p}.m_{j,p,q}.d_{i,k} \tag{9.24}$$

Considering all possible segment execution sequences of task τ_j, the maximum RBF* on core q is an upper bound of $\mathrm{RBF}^*_q(\tau^{\mathrm{decom}}_j, d_{i,k})$. We simply use this upper bound for the value of $\mathrm{RBF}^*_q(\tau^{\mathrm{decom}}_j, d_{i,k})$, where $j \neq i$ in the FBB-FFD Algorithm. Thus,

$$\mathrm{RBF}^*_q(\tau^{\mathrm{decom}}_j, d_{i,k}) = \max\left\{\mathrm{RBF}^*_q(\tau^+_{j,l}, d_{i,k}) \middle| 1 \le l \le s_i\right\} \tag{9.25}$$

Now the decomposed subtasks are partitioned based on the FBB-FFD Algorithm in the following way. Recall that $d_{i,j}$ is the deadline and $e_{i,j}$ is the execution requirement of subtask $\tau_{i,j}$ (a subtask from the j-th segment of task τ_i). Let us consider Π_q to be the set of decomposed tasks τ^{decom}_i whose (one or more) subtasks have been assigned to processor core q. To assign a subtask to a processor core, the FBB-FFD based partitioned DM Algorithm sorts the unassigned subtasks in non-increasing DM-priority order. Let at any time $\tau_{i,j}$ be the highest priority subtask among the unassigned ones. Then the algorithm performs the following assignments. Subtask $\tau_{i,j}$ is assigned to the first processor core q, $1 \le q \le m$, that satisfies the following condition

$$d_{i,j} - \sum_{\tau^{\mathrm{decom}}_k \in \Pi_q} \mathrm{RBF}^*_q(\tau^{\mathrm{decom}}_k, d_{i,j}) \ge e_{i,j} \tag{9.26}$$

If no such q, $1 \leq q \leq m$, exists that satisfies the above condition for $\tau_{i,j}$, then the task set is determined to be infeasible for partitioning. Note that RBF* is calculated using Equation (9.23) if $k = i$, and using Equation (9.25), otherwise.

From our above discussions, the value of $\sum_{\tau_k^{\text{decom}} \in \Pi_q} \text{RBF}_q^*(\tau_k^{\text{decom}}, d_{i,j})$ used in Condition (9.26) is no less than the total RBF* of all subtasks assigned to processor core q. Since any task for which processor core q satisfies Condition (9.22) is DM schedulable on q (according to the FBB-FFD Algorithm), any subtask $\tau_{i,j}$ for which processor core q satisfies Condition (9.26) must be DM schedulable on q.

9.5.2 Analysis for the FBB-FFD based Partitioned DM Algorithm

We use an analysis similar to the one used in [112] to derive the resource augmentation bound as shown in Theorem 29. The analysis is based on the demand bound function of the tasks after decomposition.

Definition 5. *The* demand bound function *(DBF), originally introduced in [44], of a task τ_i is the largest cumulative execution requirement of all jobs generated by τ_i that have both their arrival times and their deadlines within a contiguous interval of t time units. For a task τ_i with a maximum computation requirement of C_i, a period of T_i, and a deadline of D_i, its DBF is given by*

$$DBF(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right)$$

Definition 6. *Based upon the DBF function, the* load *of task system τ, denoted by $\lambda(\tau)$, is defined as follows.*

$$\lambda(\tau) = \max_{t>0} \left(\frac{\sum_{i=1}^{n} DBF(\tau_i, t)}{t} \right)$$

From Definition 5, for a constrained deadline task τ_i:

$$\begin{aligned}\mathrm{DBF}(\tau_i, t) &\leq \max\left(0, \left(\left\lfloor \frac{t - D_i}{D_i} \right\rfloor + 1\right) C_i\right) \\ &\leq \left\lfloor \frac{t}{D_i} \right\rfloor C_i \ \leq \frac{t}{D_i}.C_i \ = \delta_i.t\end{aligned}$$

Based on the above analysis, we now derive an upper bound of DBF for every task after decomposition. Every segment of task τ_i consists of a set of constrained deadline subtasks after decomposition and, by Lemma 24, the total density of all subtasks in a segment is at most $\frac{C_i/2}{T_i - P_i/2}$. The constrained deadline subtasks are offset to ensure that those belonging to different segments of the same task are never simultaneously active. That is, for each task τ_i, each segment (and each of its subtasks) happens only once during the activation time of τ_i. Therefore, for decomposed task τ_i^{decom}, considering the segment having the maximum density in place of every segment gives an upper bound on the total density of all subtasks of τ_i^{decom}. Since, the density $\delta_{i,j}$ of any j-th segment of τ_i^{decom} is at most $\frac{C_i/2}{T_i - P_i/2}$, the DBF of τ_i^{decom} over any interval of length t is

$$\mathrm{DBF}(\tau_i^{\mathrm{decom}}, t) \leq \frac{C_i/2}{T_i - P_i/2}.t$$

The load of the decomposed task system τ^{decom} is

$$\lambda(\tau^{\mathrm{decom}}) = \max_{t>0} \left(\frac{\sum_{i=1}^{n} \mathrm{DBF}(\tau_i^{\mathrm{decom}}, t)}{t} \right) \leq \sum_{i=1}^{n} \frac{C_i/2}{T_i - P_i/2} \tag{9.27}$$

Theorem 29. *If there exists any (optimal) algorithm under which a synchronous parallel task set τ is schedulable on m unit-speed processor cores, then its decomposed task set τ^{decom} is schedulable using the FBB-FDD based partitioned DM Algorithm on m identical processor cores each of speed* 5.

Proof. [80] proves that any constrained deadline sporadic task set π with total utilization $u_{\mathrm{sum}}(\pi)$, maximum density $\delta_{\max}(\pi)$, and load $\lambda(\pi)$ is schedulable by the FBB-FFD Algorithm

on m unit-speed processor cores if

$$m \geq \frac{\lambda(\pi) + u_{\text{sum}}(\pi) - \delta_{\max}(\pi)}{1 - \delta_{\max}(\pi)}$$

Using the same method used in [80] for proving the above sufficient schedulability condition, it can be shown that our decomposed (sub)tasks τ^{decom} are schedulable by the FBB-FFD based partitioned DM scheduling (presented in Subsection 9.5.1) on m unit-speed processor cores if

$$m \geq \frac{\lambda(\tau^{\text{decom}}) + u_{\text{sum}}(\tau^{\text{decom}}) - \delta_{\max}(\tau^{\text{decom}})}{1 - \delta_{\max}(\tau^{\text{decom}})} \tag{9.28}$$

where $\delta_{\max}(\tau^{\text{decom}})$, $u_{\text{sum}}(\tau^{\text{decom}})$, and $\lambda(\tau^{\text{decom}})$ denote the maximum density, total utilization, and load, respectively, of τ^{decom} on unit-speed processor cores.

We decompose tasks considering that each processor core has speed 2. To be able to schedule the decomposed tasks τ^{decom}, suppose we need to increase the speed of each processor core ν times further. That is, we need each processor core to be of speed 2ν. Let the maximum density, total utilization, and load of task set τ^{decom} be denoted by $\delta_{\max,2\nu}$, $u_{\text{sum},2\nu}$, and $\lambda_{2\nu}$ respectively, when each processor core has speed 2ν. Using these notations in Condition (9.28), task set τ^{decom} is schedulable by the FBB-FFD based partitioned DM Algorithm on m identical processor cores each of speed 2ν if

$$m \geq \frac{\lambda_{2\nu} + u_{\text{sum},2\nu} - \delta_{\max,2\nu}}{1 - \delta_{\max,2\nu}} \tag{9.29}$$

From Equation 9.1:

$$u_{\text{sum},2\nu} = \sum_{i=1}^{n} \frac{\frac{C_i}{2\nu}}{T_i} = \frac{1}{2\nu} \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{u_{\text{sum}}}{2\nu} \tag{9.30}$$

From Equations 9.1 and 9.27:

$$\lambda_{2\nu} \leq \sum_{i=1}^{n} \frac{\frac{C_i}{2\nu}}{T_i - \frac{P_i}{2}} \leq \sum_{i=1}^{n} \frac{\frac{C_i}{2\nu}}{T_i - \frac{T_i}{2}} = \frac{1}{\nu} \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{u_{sum}}{\nu} \tag{9.31}$$

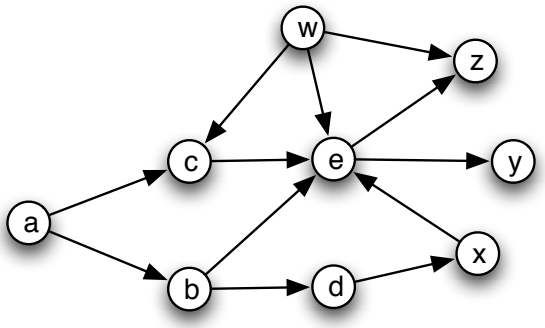
Using Equations 9.31, 9.30, 9.18 in Condition (9.29), task set τ^{decom} is schedulable if

$$m \geq \frac{\frac{u_{\text{sum}}}{\nu} + \frac{u_{\text{sum}}}{2\nu} - \frac{1}{\nu}}{1 - \frac{1}{\nu}}$$

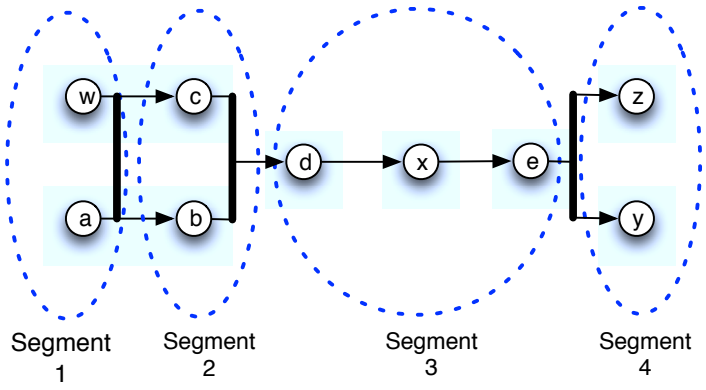
If the original parallel task set τ is schedulable by any algorithm on m unit-speed processor cores, then $u_{\text{sum}} \leq m$. Therefore, τ^{decom} is schedulable if

$$m \geq \frac{\frac{m}{\nu} + \frac{m}{2\nu} - \frac{1}{\nu}}{1 - \frac{1}{\nu}} \quad \Leftarrow \quad 2\nu - 2 \geq 3 \quad \Leftrightarrow \quad 2\nu \geq 5$$

Hence follows the theorem. □



(a) Unit-node DAG



(b) Parallel synchronous model

Figure 9.4: Unit-node DAG to parallel synchronous model

9.6 Generalizing to a Unit-node DAG Task Model

In the analysis presented so far, we have focused on synchronous parallel tasks. That is, there is a synchronization point at the end of each segment, and the next segment starts only after all the threads of the previous segment have completed. In this section, we show that even more general parallel tasks that can be represented as directed acyclic graphs (DAGs) with unit time nodes can be easily converted into synchronous tasks. Therefore, the above analysis holds for these tasks as well.

In the unit-node DAG model of tasks, each job is made up of nodes that represent work, and edges that represent dependencies between nodes. Therefore, a node can execute only after all of its predecessors have been executed. We consider the case where each node represents unit-time work. Therefore, a unit-node DAG can be converted into a synchronous task by simply adding new dependence edges as explained below.

If there is an edge from node u to node v, we say that u is the *parent* of v. Then we calculate the depth, denoted by $h(v)$, of each node v. If v has no parents, then it is assigned depth 1. Otherwise, we calculate the depth of v as

$$h(v) = \max_{u \text{ parent of } v} h(u) + 1$$

Each node with depth j is assigned to segment j. Then every node of the DAG is considered as a *thread* in the corresponding segment. The threads in the same segment can happen in parallel, and the segment is considered as a parallel segment of a synchronous task. If there are $k > 1$ consecutive segments each consisting of just one thread, then all these k segments are considered as one sequential segment of execution requirement k (by preserving the sequence). Figure 9.4 shows an example, where a DAG in Figure 9.4(a) is represented as a synchronous task in Figure 9.4(b). This transformation is valid since it preserves all dependencies in the DAG, and in fact only adds extra dependencies .

Upon representing a unit-node DAG task as a synchronous task, we perform the same decomposition proposed in Section 9.3. The decomposed task set can be scheduled using either global EDF or partitioned DM scheduling. Note that the transformation from a DAG task τ_i to a synchronous task preserves the work C_i of τ_i. Hence, the condition $\sum C_i/T_i \leq m$ used

in our analysis still holds. Besides, the transformation preserves the critical path length P_i of τ_i and, hence, the rest of the analysis also holds. Therefore, a set of unit-node DAG tasks can be scheduled with a resource augmentation bound of 4 under global EDF scheduling, and of 5 under partitioned DM scheduling.

9.7 Evaluation

In this section, we evaluate the proposed decomposition through simulations. We randomly generate synchronous parallel tasks, decompose them, and simulate their schedules under global EDF and partitioned DM policies considering multi-core processors with different number of cores. We validate the derived resource augmentation bounds by considering different speeds of the processor cores.

9.7.1 Task Generation

In our simulation studies, parallel synchronous task sets are generated in the following way. The number of segments of each task is randomly selected from the range $[10, 30]$. The number of threads in each segment is randomly selected from the range $[1, 90]$. The execution requirements of the threads in a segment are selected randomly from the range $[5, 35]$. Each task is assigned a valid harmonic period (i.e. period is no less than its critical path length) of the form 2^k, where k is chosen from the range $[6, 13]$. We generate task sets considering $m = 20, 40$, and 80 (i.e. for 20-core, 40-core, and 80-core processors). For each value of m, we generate 1000 task sets.

We want to evaluate our scheduler on task sets that an optimal scheduler could schedule on 1-speed processors. However, as we cannot compute this ideal scheduler, we assume that an ideal scheduler can schedule any task set that satisfies two conditions: (1) total utilization is no greater than m, and (2) each individual task is schedulable in isolation, that is, the deadline is no smaller than the critical path length. The second condition is implicitly satisfied by the way we assign period (which is the same as the deadline for our tasks) to tasks. Therefore, in our experiments, to generate a task set for each value of m (m being the

number of processor cores), we keep adding tasks to the set as long as their total utilization does not exceed m but is at least 98% of m, thereby (almost) fully loading a machine of speed 1 processors. Since a system with a larger value of m is able to schedule a task set with higher utilization, the task sets generated for different values of m are different. According to above parameters, the generated task sets for 20-core, 40-core, and 80-core processors have the average number of tasks per task set of 4.893, 6.061, and 8.791, respectively, and the average ratios for total utilization to the number of cores are 99.3%, 99.2%, and 99.1% respectively.

9.7.2 Simulation Setup

The tasks generated for a particular value of m are decomposed using our proposed decomposition technique. The decomposed tasks are then scheduled by varying the speed of the cores on a simulated multi-core platform. We evaluate the performance in terms of *failure ratio* defined as the proportion of the number of unschedulable task sets to the total number of task sets attempted. We consider partitioned DM and two versions of global EDF scheduling policies. Note that under partitioned DM (P-DM), to make the analysis work, segments cannot be released before the decomposed release offsets, while under global EDF a segment can either wait till its relative release time, or start as soon as its preceding segments complete. Our analysis holds for both versions of the global EDF policies, and here we evaluate both of them to understand if they perform differently in simulations. The policy where subtasks wait until their release offset is called standard global EDF (G-EDF) and the policy where tasks are released as soon their dependences are satisfied is called greedily synchronized global EDF (GSG-EDF).

For all three methods, two kinds of failure ratios are compared. For the first kind marked with "simu", we measure the actual failure ratio in that a task-set is considered unschedulable if any task in the task set misses its deadline. For the other kind marked with "test", a task set is considered unschedulable if any subtask (some thread in a decomposed segment) misses its deadline. Note that the overall task may still be scheduled successfully if a subtask misses deadlines. Therefore, "test" failure ratios are always no better (and generally worse than) than "simu" ones. Since subtask deadlines are assigned by the system and do not represent

real constraints, a subtask deadline miss is not important to a real system. However, we include this result in order to throughly investigate the safety of our analytical bounds.

Note that only the "simu" results reflect the job deadline misses, which can be compared with other methods without any decomposition. Besides all the simulation results, for P-DM, we also include the failure ratio of the analysis. The failure ratio marked by "P-DM-analysis" indicates the ratio of the number of task sets that the offset-aware FBB-FFD Algorithm failed to partition with guaranteed schedulability to the total number of task sets attempted. Note that since this partitioning algorithm is also pessimistic, there are some task sets that the algorithm can not guarantee, but in simulation all their deadlines are met anyway. Therefore, this failure ratio is generally worse than P-DM-test.

In Figures 9.5, 9.6, and 9.7, the failure ratios under G-EDF, GSG-EDF, and P-DM are compared on 20, 40 and 80 core machines, respectively. All cores always have the same speed, and we increase the speed gradually until all task sets are schedulable. In particular, we start by setting a speed of 1 (i.e. unit-speed) at every processor core. Then we keep increasing the speed of each core by 0.2 in every step, and schedule the same set of tasks using the increased speed.

9.7.3 Simulation Results

In the first set of simulations, we evaluate the schedulability of 1000 task sets generated for a 20-core processor. According to Figure 9.5, when each core has speed 1, the failure ratio under G-EDF-test is 0.988 and under G-EDF-simu is 0.27. That is, out of 1000 test cases, 988 cases of decomposed task sets are not schedulable (having at least one segment deadline miss in simulation), and 270 cases have at least one job deadline miss. As we gradually increase the speed of each core, the failure ratios for G-EDF-test decrease slowly and are much higher than those of G-EDF-simu. For example, when each core has speed of 1.2, 1.4, 1.6, and 1.8, the failure ratios for G-EDF-test are 0.969, 0.935, 0.9, and 0.849, respectively, while for G-EDF-simu the ratios are 0.169, 0.119, 0.097, and 0.084, respectively. When each core has speed 2.4 or more, all task sets are schedulable. Thus, the resource augmentation required for the tasks we have evaluated under G-EDF is only 2.4 for this simulation setting.

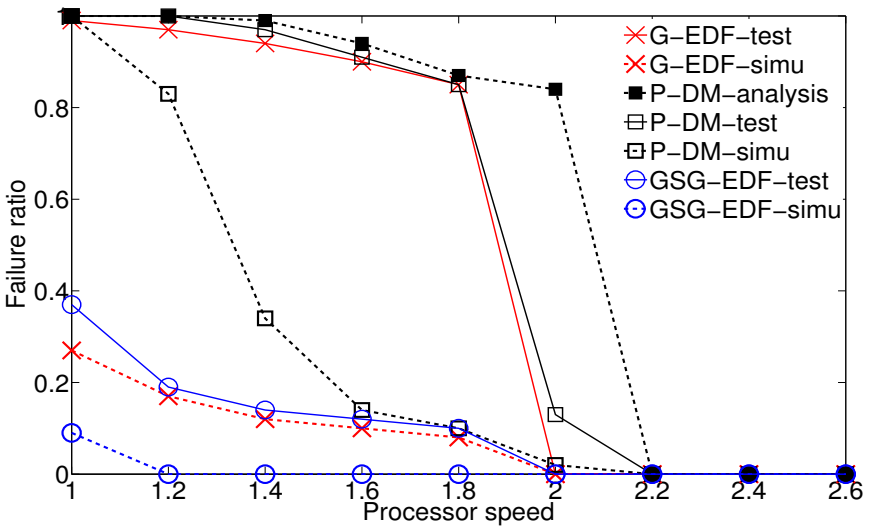


Figure 9.5: Schedulability on a 20-core processor.

Note one interesting fact about these figures. When speed is increased from 1.8 to 2, we observe a very sharp decrease in failure ratios of G-EDF-test. Specifically, at speeds 2 and 2.2, only one task set is unschedulable. This sharp decrease happens due to the following reason. In our method, the original decomposition occurs on speed-2 processors. Therefore, some (sub)tasks may have density greater than 1 on processors of speeds smaller than 2, meaning that these subtasks can never meet their deadlines at speed lower than 2. Since the decomposition guarantees that the maximum density among all (sub)tasks is at most 1 at speed 2, many task sets that were unschedulable at speed 1.8 become schedulable at speed 2.

Unlike G-EDF, GSG-EDF with greedy synchronization does not wait for any segment release time. It can utilize the slack from the decomposed task sets and hence has better performance than G-EDF. The failure ratios in GSG-EDF-test are 0.37, 0.193, 0.135, 0.117 and 0.101 at speed 1, 1.2, 1.4, 1.6, and 1.8, respectively. It is slightly worse than that of G-EDF-simu, and is much better than those of G-EDF-test. Out of 1000 cases in GSG-EDF-simu, there are only 86 and 4 job deadline misses at speed 1 and 1.2, respectively, and no deadline misses when speed is greater than 1.2. Note that "test" of GSG-EDF also has a decrease at speed 2 for the same reason as G-EDF. However, GSG-EDF-simu (unlike G-EDF-test, G-EDF-simu, or GSG-EDF-test) does not experience this sharp decrease caused by decomposition process

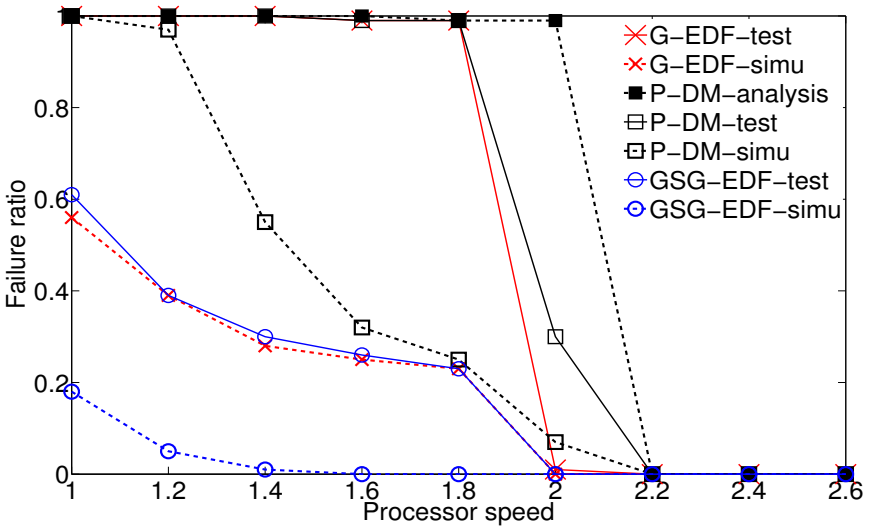


Figure 9.6: Schedulability on a 40-core processor

considering speed 2 processors. We speculate the reason is the following. The "simu" results basically measure if the last segment of the task misses a deadline. In the case of GSG-EDF, by the time the last segment comes around, the task has accumulated enough slack by starting the prior segments early. Therefore, even though the last segment's density may be larger than 1 (if it was released at its decomposed release time), it is released early enough that it can still generally meet its deadline.

Note that there are no theoretical guarantees about the comparison of job deadline misses between greedily synchronized global EDF for decomposed task sets and global EDF for original task sets. However, as we observe in practice, the resource augmentation required for the tasks we have evaluated under GSG-EDF is only 2.2, compared with 2.4 of G-EDF for the same 1000 task sets, suggesting that greedy synchronization provides some advantage over the standard version.

Besides G-EDF, Figure 9.5 also plots the failure ratios under P-DM (based on the offset-aware FBB-FFD Algorithm) on a 20-core processor. When each core has speed 1, all the 1000 test cases are unschedulable in both analysis and simulation under P-DM. With the increase in speed to 1.8, the failure ratios of P-DM-analysis and P-DM-test decrease slowly, while P-DM-simu decreases sharply. However, the result of P-DM-simu is not stable. A task set, which has no job deadline miss at certain speed, may result in job deadline misses

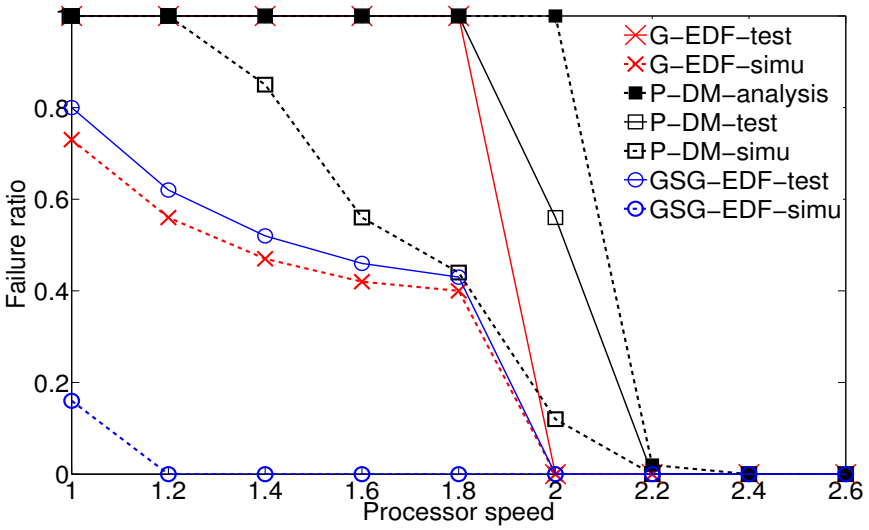


Figure 9.7: Schedulability on a 80-core processor

even when the speed increases. This is because the partitioning results for the same task set under different processor speed are different. Since for P-DM, the segment release time must be followed, the job deadlines are just part of the overall segment deadlines. With different partitioning results, different parts of segment deadlines may be missed. Therefore, the job deadline misses of the same task set under different processor speed are not stable. The failure ratio of P-DM-test decreases sharply when reaching the speed 1.8 and the P-DM-analysis decreases sharply when reaching 2. At speed 2.4, only one task set cannot be scheduled according to the analysis while all can be scheduled in simulation, demonstrating a resource augmentation of 2.4 for this specific simulation setting which is smaller than our theoretical bound of 5 for P-DM.

These results seem to indicate that the global EDF with greedy synchronization is slightly better than standard global EDF which is slightly better than P-DM. Note that we should only care about the "simu" results when comparing across policies.

Similar experiments for 40 and 80 cores are shown in Figures 9.6 and 9.7 respectively. Note that these task sets are different from those generated for the previous set of simulations, and have higher total utilization (as these are generated for a higher number of cores). The curves follow similar trends. Namely, here also the "test" results are generally worse than the

corresponding "simu" results, the P-DM-analysis results are slightly pessimistic and GSG-EDF is slightly better than G-EDF which is better than P-DM. On 40-core machines, the resource augmentation required by GSG-EDF, G-EDF and P-DM is 2.2, 2.6 and 2.6 respectively for these experiments. For 80 core machines, the respective resource augmentation requirements are 2.0, 2.4 and 3 respectively.

It is difficult to compare the results of the 20-core experiment with those of the 40-core and 80-core experiments since the task sets used in different experiments are different. There is no way of knowing if the differences we observe are due to the increase in the number of cores or because we happened to generate task sets of more or less inherent difficulty. However, the general trend (with the exception of GSG-EDF-simu when we go from 40 to 80 cores) seems to be that it is more difficult to schedule on a larger number of cores. This may be due to the fact that as the number of cores increases, we have a larger number of tasks and subtasks, increasing the number of deadlines, thereby increasing the chance of the event that a segment or job deadline miss occurs in a task set.

These results indicate that our theoretical bounds may be pessimistic for the particular cases we have experimented, since global EDF needs augmentation of at most 2.6 (in contrast to the analytical bound of 4) and P-DM needs augmentation of at most 3 (in contrast to the analytical bound of 5). Since the analytical bounds are guaranteed for any task set and for any number of processor cores, the results from our experiments indeed do not guarantee whether these bounds are very loose or tight in general, since the worst case task sets might have not appeared in our experiments. In addition, we have observed that the resource augmentation requirement slightly increases with the increase in the number of cores (and keeping the system almost fully loaded). Hence, it is conceivable that for thousands of processor cores and for very adversarial task sets, one might need an augmentation of 4 and 5 for global EDF and partitioned DM policies, respectively. This suggests some potential research direction towards exploring both better (smaller) augmentation bounds and tighter lower bounds.

9.8 Related Work

There has been extensive work on traditional multiprocessor real-time scheduling [69]. Most of this work focuses on scheduling sequential tasks on multiprocessor or multi-core systems. There has also been extensive work on scheduling of one or more parallel jobs on multiprocessors [27, 34, 39, 51, 53, 54, 71, 73, 74, 143]. However, the work in [27, 34, 39, 71, 73, 74, 143] does not consider task deadlines, and that in [51, 53, 54] considers soft real-time scheduling. In contrast to the goal (i.e. to meet all task deadlines) of a hard real-time system, in a *soft real-time system* the goal is to meet a certain subset of deadlines based on some application specific criteria.

There has been little work on hard real-time scheduling of parallel tasks. Anderson et al. [33] propose the concept of a megatask as a way to reduce miss rates in shared caches on multicore platforms, and consider Pfair scheduling by inflating the weights of a megatask's component tasks. Preemptive fixed-priority scheduling of parallel tasks is shown to be NP-hard by Han et al. [92]. Kwon et al. [110] explore preemptive EDF scheduling of parallel task systems with linear-speedup parallelism. Wang et al. [174] consider a heuristic for nonpreemptive scheduling. However, this work focuses on metrics like makespan [174] or total work that meets deadline [110], and considers simple task models where a task is executed on up to a given number of processors.

Most of the other work on real time scheduling of parallel tasks also address simplistic task models. Jansen [98], Lee et al. [115], and Collette et al. [64] study the scheduling of *malleable tasks*, where each task is assumed to execute on a given number of cores or processors and this number may change during execution. Manimaran et al. [130] study non-preemptive EDF scheduling for *moldable tasks*, where the actual number of used processors is determined before starting the system and remains unchanged. Kato et al. [103] address Gang EDF scheduling of moldable parallel task systems. They require the users to select at submission time the number of processors upon which a parallel task will run. They further assume that a parallel task generates the same number of threads as processors selected before the execution. In contrast, the parallel task model addressed in this paper allows tasks to have different numbers of threads in different stages, which makes our solution applicable to a much broader range of applications. For parallel tasks, Nelissen et al. [136] has studied real-time scheduling to minimize the required number of processors. In contrast, we study

schedulability of parallel tasks in terms of processor speed-up factor. Recently, processor speed up factor for a task represented as a DAG has been addressed by Baruah et al. [43]. But this work considers scheduling of a single task. In contrast, we consider scheduling a set of parallel tasks on multi-core platform.

Our work is most related to, and is inspired by, the recent work of Lakshmanan et al. [112] on real-time scheduling for a restrictive synchronous parallel task model. In their model, every task is an alternate sequence of parallel and sequential segments. All the parallel segments in a task have an *equal* number of threads, and that number *cannot exceed* the total number of processor cores. They also convert each parallel task into a set of sequential tasks, and then analyze the resource augmentation bound for partitioned DM scheduling. However, their strategy of decomposition is different from ours. They use a *stretch transformation* that makes a master thread of execution requirement equal to the task period, and assign one processor core exclusively to it. The remaining threads are scheduled using the FBB-FDD algorithm. Unlike ours, their results do not hold if, in a task, the number of threads in different segments vary, or exceed the number of cores. In addition, tasks that can be expressed as a DAG of unit time nodes cannot be converted to their task model in a work and critical path length conserving manner. Therefore, unlike ours, their analysis does not directly apply to these more general task models.

Our work in this paper has two major extensions beyond our previous work that appears as a conference version [153]. First, we have added a detailed description of how the FBB-FFD partitioned deadline monotonic scheduling policy should be adopted for decomposed tasks with offsets. The conference version of the paper provided a resource augmentation bound for the FBB-FFD algorithm, and did not provide the detailed procedure for partitioning of the tasks with offsets. Note that the FBB-FFD algorithm, by default, is not offset-aware. Hence, we have provided a modified version of this algorithm to make it offset-aware. Second, we have provided evaluation results based on simulations. The conference version of the paper did not provide any evaluation result. In addition, we have presented how our results will hold for task models where a synchronous task may have some segment containing threads of different execution requirements. The conference version of the paper considered the synchronous task model where each segment in a task consists of equal-length threads.

9.9 Summary

With the advent of the era of multi-core computing, real-time scheduling of parallel tasks is crucial for real-time applications to exploit the power of multi-core processors. While recent research on real-time scheduling of parallel tasks has shown promise, the efficacy of existing approaches is limited by their restrictive parallel task models. To overcome these limitations, in this paper we have presented new results on real-time scheduling for generalized parallel task models. First, we have considered a general synchronous parallel task model where each task consists of segments, each having an arbitrary number of parallel threads. Then we have proposed a novel task decomposition algorithm that decomposes each parallel task into a set of sequential tasks. We have derived a resource augmentation bound of 4 under global EDF scheduling, which to our knowledge is the first resource augmentation bound for global EDF scheduling of parallel tasks. We have also derived a resource augmentation bound of 5 for partitioned DM scheduling. Finally, we have shown how to convert a task represented as a Directed Acyclic Graph (DAG) with unit time nodes into a synchronous task, thereby holding our results for this more general task model.

Through simulations, we have observed that bounds in practice are significantly smaller than the theoretical bounds. These results suggest many directions of future work. First, we can try to provide better bounds and/or provide lower bound arguments that suggest that the bounds are in fact tight. In the future, we also plan to consider even more general DAG tasks where nodes have arbitrary execution requirements, and to provide analysis requiring no transformation to synchronous model. We also plan to address system issues such as cache effects, preemption penalties, and resource contention.

Acknowledgements

This research was supported by NSF under grants CNS-0448554 (CAREER) and CCF-1136073.

Chapter 10

Parallel Real-Time Scheduling of DAGs

Recently, multi-core processors have become mainstream in processor design. To take full advantage of multi-core processing, computation-intensive real-time systems must exploit intra-task parallelism. In this paper, we address the problem of real-time scheduling for a general model of deterministic parallel tasks, where each task is represented as a directed acyclic graph (DAG) with nodes having arbitrary execution requirements. We prove processor-speed augmentation bounds for both preemptive and non-preemptive real-time scheduling for general DAG tasks on multi-core processors. We first decompose each DAG into sequential tasks with their own release times and deadlines. Then we prove that these decomposed tasks can be scheduled using preemptive global EDF with a resource augmentation bound of 4. This bound is as good as the best known bound for more restrictive models, and is the first for a general DAG model. We also prove that the decomposition has a resource augmentation bound of 4 plus a constant non-preemption overhead for non-preemptive global EDF scheduling. To our knowledge, this is the first resource augmentation bound for non-preemptive scheduling of parallel tasks. Finally, we evaluate our analytical results through simulations that demonstrate that the derived resource augmentation bounds are safe in practice.

10.1 Introduction

As the rate of increase of clock frequencies is leveling off, most processor chip manufacturers have recently moved to increasing performance by increasing the number of cores on a chip. Intel's 80-core Polaris [10], Tilera's 100-core TILE-Gx, AMD's 12-core Opteron [13], and ClearSpeed's 96-core processor [11] are some notable examples of multi-core chips. With the rapid evolution of multi-core technology, however, real-time system software and programming models have failed to keep pace. Most classic results in real-time scheduling concentrate on sequential tasks running on multiple processors [69]. While these systems allow many tasks to execute on the same multi-core host, they do not allow an individual task to run any faster on it than on a single-core machine.

To scale the capabilities of individual tasks with the number of cores, it is essential to develop new approaches for tasks with intra-task parallelism, where each real-time task itself is a parallel task that can utilize multiple cores at the same time. Here, we take autonomous vehicle [104] as a motivating example. Such a system consists of a myriad of real-time tasks such as motion planning, sensor fusion, computer vision, and decision making algorithms that exhibit intra-task parallelism. For example, the decision making subsystem processes massive amounts of data from various types of sensors, where the data processing on different types of sensors can run in parallel. Such intra-task parallelism may enable timing guarantees for many complex real-time systems requiring heavy computation, whose stringent timing constraints are difficult to meet on traditional single-core processors.

There has been some recent work on real-time scheduling for parallel tasks, but it has been mostly restricted to the synchronous task model [78, 112, 153]. In the *synchronous model*, each task consists of a sequence of segments with synchronization points at the end of each segment. In addition, each segment of a task contains threads of execution that are of *equal* length. For synchronous tasks, the result in [78, 153] proves a resource augmentation bound of 4 under global earliest deadline first (EDF) scheduling. A *resource augmentation bound* ν of a scheduling policy $\mathbb{A}$ indicates that if there is any way to schedule a task set on m identical unit-speed processor cores, then $\mathbb{A}$ is guaranteed to successfully schedule it on m cores with each core being ν times as fast as the original.

While the synchronous task model represents the tasks generated by the *parallel for* loop construct common to many parallel languages such as OpenMP [17] and CilkPlus [12], most parallel languages also have other constructs for generating parallel programs, notably *fork-join* constructs. A program that uses fork-join constructs will generate a *non-synchronous* task, generally represented as a directed acyclic graph (DAG), where each thread (sequence of instructions) is a node, and the edges represent dependencies between the threads. A node's execution requirement can vary arbitrarily, and different nodes in the same DAG can have different execution requirements.

Another limitation of the state-of-the-art is that all prior work on parallel real-time tasks considers *preemptive scheduling*, where threads are allowed to preempt each other in the middle of execution. While this is a reasonable model, preemption can be a high-overhead operation since it often involves a system call and a context switch. An alternative scheduling model is to consider *node-level non-preemptive scheduling* (called non-preemptive scheduling in this paper), where once the execution of a particular node (thread) starts it cannot be preempted by any other thread. Most parallel languages and libraries have yield points at the end of threads (nodes of a DAG), allowing low-cost, user-space preemption at these yield points. For these languages and libraries, schedulers that switch context only when threads end can be implemented entirely in user-space, and therefore have low overheads. In addition, fewer switches imply lower caching overhead. In this model, since a node is never preempted, if it accesses the same memory location multiple times, those locations will be cached, and a node never has to restart on a cold cache.

This paper addresses the hard real-time scheduling of a set of generalized DAGs sharing a multi-core machine. We generalize the previous work in two important directions. First, we consider a general model of deterministic parallel tasks, where each task is represented by a general DAG in which nodes can have *arbitrary* execution requirements. Second, we address both *preemptive* and *non-preemptive* scheduling. In particular, we make the following new contributions.

- We propose a novel task decomposition to transform the nodes of a general DAG into sequential tasks. Since each node of the DAG becomes an individual sequential task, these tasks can be scheduled either preemptively or non-preemptively.

- We prove that any set of parallel tasks of a general DAG model, upon decomposition, can be scheduled using preemptive global EDF with a resource augmentation bound of 4. This bound is as good as the best known bound for more restrictive models [153] and, to our knowledge, is the *first* bound for a general DAG model.
- We prove that our decomposition requires a resource augmentation bound of $4 + 2\rho$ for non-preemptive global EDF scheduling, where ρ is the *non-preemption overhead* of the tasks. To our knowledge, this is the *first* bound for *non-preemptive* scheduling of parallel real-time tasks.
- We implement the proposed decomposition algorithm, and evaluate our analytical results for both preemptive and non-preemptive scheduling through simulations. The results indicate that the derived bounds are safe, and reasonably tight in practice, especially under preemptive EDF that requires a resource augmentation of 3.2 in simulation as opposed to our analytical bound of 4.

Section 10.2 reviews related work. Section 10.3 describes the task model. Section 10.4 presents the decomposition algorithm. Sections 10.5 and 10.6 present analyses for preemptive and non-preemptive global EDF scheduling, respectively. Section 10.7 presents the simulation results. Section 10.8 offers conclusions.

10.2 Related Work

There has been a substantial amount of work on traditional multiprocessor real-time scheduling focused on sequential tasks [69]. Scheduling of parallel tasks without deadlines has been addressed in [27, 28, 34, 71, 73, 143]. *Soft real-time scheduling*, (where the goal is to meet a subset of deadlines based on some application-specific criterion) has been studied for various parallel task models and optimization criteria such as cache misses [33, 52], makespan [174], and total work done within the deadlines [110].

An exact (i.e., both sufficient and necessary) schedulability analysis under *hard real-time system* (where the goal is to meet all task deadlines) is intractable for most cases of parallel tasks [92]. Early works on hard real-time parallel scheduling make simplifying assumptions

about task models. For example, the study in [103,130] considers EDF scheduling of parallel tasks where the actual number of processors used by a particular task is determined before starting the system, and remains unchanged.

Recently, *preemptive* real-time scheduling has been studied [78, 112, 153] for *synchronous* parallel tasks with implicit deadlines. In [112], every task is an alternate sequence of parallel and sequential *segments* with each parallel segment consisting of multiple threads of *equal* length that synchronize at the end of the segment. All parallel segments in a task have an *equal* number of threads which cannot *exceed* the number of processor cores. Each thread is transformed into a subtask, and a resource augmentation bound of 3.42 is claimed under partitioned Deadline Monotonic (DM) scheduling. This result was later generalized for synchronous model with arbitrary numbers of threads in segments, with bounds of 4 and 5 for global EDF and partitioned DM scheduling, respectively [153], and also to minimize the required number of processors [136].

Our earlier work [153] has proposed a simple extension to a synchronous task scheduling approach that handles *unit-node DAG* where each node has unit execution requirement. The work in [78] is a system implementation of our work in [153]. Our approach in [153] converts each task to a synchronous task allowing direct application of the approach designed for synchronous tasks. While it simplifies the resource augmentation analysis, the assumption that each node has unit execution requirement is highly restrictive and does not hold in general since this model does not represent a parallel task that most parallel languages generate. Most parallel languages that use fork-join constructs generate a *non-synchronous* task, generally represented as a DAG where each node's execution requirement can vary arbitrarily, and different nodes in the same DAG can have different execution requirements. While the proposed decomposition in this paper and that in [153] have similarity in that both decompose a parallel task into sequential subtasks by distributing the available slack among the subtasks, the decomposition in [153] for restrictive model is not applicable for general DAG. If it is extended to general DAG, it may split each node of a DAG into multiple subtasks, thereby disallowing node-level non-preemptive scheduling. Also, it will make preemptive scheduling inefficient and costly due to excessive numbers of contexts switches due to node splitting and artificially increased synchronization.

Scheduling and analysis of general DAGs introduces a challenging open problem. For this general model, an augmentation bound has been analyzed recently in [43], but it considers a *single* DAG on a multi-core machine with preemption. In this paper, we investigate the open problem of scheduling and analysis for a set of any number of general DAGs on a multi-core machine. We consider both preemptive and non-preemptive real-time scheduling of general DAG tasks on multi-core processors, and provide resource augmentation bound under both policies.

10.3 Parallel Task Model

We consider n periodic parallel tasks to be scheduled on a multi-core platform consisting of m identical cores. The task set is represented by $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task $\tau_i, 1 \leq i \leq n$, is represented as a Directed Acyclic Graph (DAG), where the *nodes* stand for different execution requirements, and the *edges* represent dependencies between the nodes.

A node in τ_i is denoted by $W_i^j, 1 \leq j \leq n_i$, with n_i being the total number of nodes in τ_i. The *execution requirement* of node W_i^j is denoted by E_i^j. A directed edge from node W_i^j to node W_i^k, denoted as $W_i^j \rightarrow W_i^k$, implies that the execution of W_i^k cannot start until W_i^j finishes. W_i^j, in this case, is called a *parent* of W_i^k, while W_i^k is its *child*. A node may have 0 or more parents or children, and can start execution only after all of its parents have finished execution. Figure 10.1 shows a task τ_i with $n_i = 10$ nodes.

The *execution requirement* (i.e., *work*) C_i of task τ_i is the sum of the execution requirements of all nodes in τ_i; that is, $C_i = \sum_{j=1}^{n_i} E_i^j$. Thus, C_i is the *maximum execution time* of τ_i if it was executing on a single processor of speed 1. For task τ_i, the *critical path length*, denoted by P_i, is the sum of execution requirements of the nodes on a critical path. A *critical path* is a directed path that has the maximum execution requirement among all other paths in DAG τ_i. Thus, P_i is the *minimum execution time* of τ_i meaning that it needs at least P_i time units on unit-speed processor cores even when the number of cores m is infinite. The *period* of task τ_i is denoted by T_i and the deadline D_i of each task τ_i is considered *implicit*, i.e., $D_i = T_i$. Since P_i is the minimum execution time of task τ_i even on a machine with an infinite number of cores, the condition $T_i \geq P_i$ must hold for τ_i to be schedulable (i.e. to

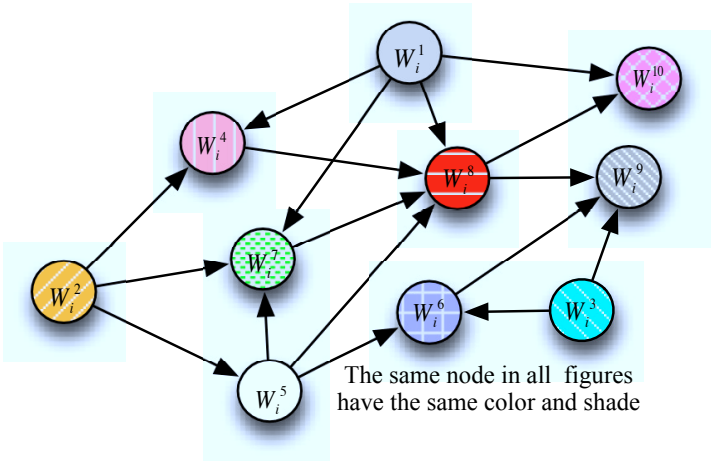
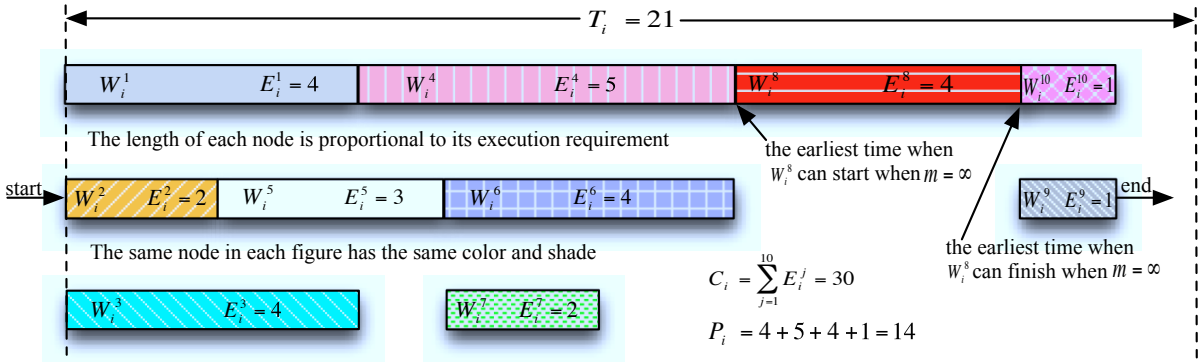


Figure 10.1: A parallel task τ_i represented as a DAG

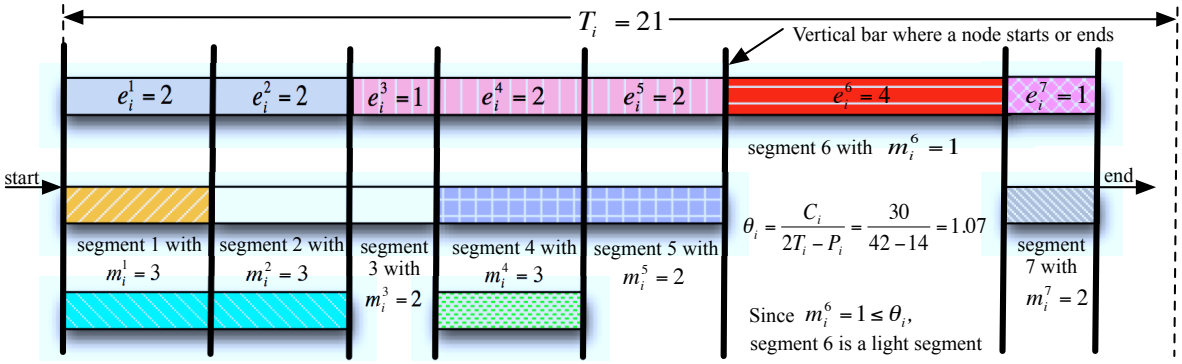
meet its deadline). A task set is said to be *schedulable* when all tasks in the set meet their deadlines.

10.4 Task Decomposition

We schedule parallel tasks by decomposing each parallel task into smaller sequential tasks. The main intuition for decomposing a parallel task into a set of sequential tasks is that the scheduling of parallel task reduces to the scheduling of sequential tasks, thereby allowing us to exploit the rich literature on the latter. In particular, this strategy allows us to leverage existing schedulability analysis for traditional multiprocessor scheduling, both preemptive and non-preemptive, of sequential tasks. In this section, we present a decomposition technique for a parallel task under a general DAG model. Upon decomposition, each node of a DAG becomes an individual sequential task, called a *subtask*, with its own deadline and with an execution requirement equal to the node's execution requirement. We will use the terms 'subtask' and 'node' interchangeably. All nodes of a DAG are assigned appropriate deadlines and release offsets such that when they execute as individual subtasks all dependencies among them in the original DAG are preserved. The deadlines of the subtasks of a DAG are assigned by splitting the DAG's deadline. The decomposition will ensure that if the subtasks of a DAG are schedulable, then the DAG must be schedulable. Thus, an



(a) τ_i^{∞}: a timing diagram for when τ_i executes on an infinite number of processor cores



(b) τ_i^{syn}

Figure 10.2: τ_i^{∞} and τ_i^{syn} of DAG τ_i (of Figure 10.1)

implicit deadline DAG is decomposed into a set of constrained deadline (i.e. deadline is no greater than period) sequential subtasks with each subtask corresponding to a node of the DAG.

Our schedulability analysis for parallel tasks entails deriving a resource augmentation bound [112, 153]. In particular, our result aims at procuring the following claim: If an optimal algorithm can schedule a task set on a machine of m unit-speed processor cores, then our algorithm can schedule this task set on m processor cores, each of speed ν, where ν is the *resource augmentation* factor. Since an optimal algorithm is unknown, we pessimistically assume that an optimal scheduler can schedule a task set if each task of the set has a critical-path length no greater than its deadline, and the total utilization of the task set is no greater than m. No algorithm can schedule a task set that does not meet these conditions. Our resource

augmentation analysis is based on the densities of the decomposed tasks, where the *density* of any task is the ratio of its execution requirement to its deadline. We first present terminology used in decomposition. Then, we present the proposed decomposition technique, followed by a density analysis of the decomposed tasks.

10.4.1 Terminology

Our proposed decomposition technique converts each implicit deadline DAG task into a set of constrained deadline sequential tasks, and is based on the following definitions that are applicable for any task, not limited to just parallel tasks.

The *utilization* u_i of a task τ_i, and the *total utilization* $u_{\text{sum}}(\tau)$ for any task set τ of n tasks are defined as

$$u_i = \frac{C_i}{T_i}; \qquad u_{\text{sum}}(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

If the total utilization u_{sum} is greater than m, then no algorithm can schedule τ on m identical unit-speed processor cores.

The *density* δ_i of any task τ_i, and the *total density* $\delta_{\text{sum}}(\tau)$ and the *maximum density* $\delta_{\text{max}}(\tau)$ for any set τ of n tasks are defined as follows.

$$\delta_i = \frac{C_i}{D_i}; \quad \delta_{\text{sum}}(\tau) = \sum_{i=1}^{n} \delta_i; \quad \delta_{\text{max}}(\tau) = \max\{\delta_i | 1 \leq i \leq n\} \tag{10.1}$$

The *demand bound function* (DBF) of task τ_i is the largest cumulative execution requirement of all jobs generated by τ_i that have both arrival times and deadlines within a contiguous interval of t time units. For any task τ_i, the DBF is given by

$$\text{DBF}(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right) \tag{10.2}$$

Based on the DBF, the *load*, denoted by $\lambda(\tau)$, of any task set τ consisting of n tasks is defined as follows.

$$\lambda(\tau) = \max_{t>0} \left(\frac{\sum_{i=1}^{n} \text{DBF}(\tau_i, t)}{t} \right) \tag{10.3}$$

10.4.2 Decomposition Algorithm

The decomposition algorithm converts each node of a DAG into an individual sequential subtask with its own execution requirement, release offset, and a constrained deadline. The release offsets are assigned so as to preserve the dependencies of the original DAG, namely, to ensure that a node (subtask) can start after the deadlines of all the parent nodes (subtasks). That is, a node starts after its *latest* parent finishes. The (relative) deadlines of the nodes are assigned by splitting the task deadline into intermediate subdeadlines. The intermediate subdeadline assigned to a node is called *node deadline*.

Note that once task τ_i is released, it has a total of T_i time units to finish its execution. The proposed decomposition algorithm splits this deadline T_i into node deadlines by preserving the dependencies in τ_i. For task τ_i, the deadline and the offset assigned to node W_i^j are denoted by D_i^j and Φ_i^j, respectively. Once appropriate values of D_i^j and Φ_i^j are determined for each node W_i^j (respecting the dependencies in the DAG), task τ_i is decomposed into nodes. Upon decomposition, the dependencies in the DAG need not be considered, and each node can execute as a traditional sequential multiprocessor task. Hence, the decomposition technique for τ_i boils down to determining D_i^j and Φ_i^j for each node W_i^j as presented below. The presentation is accompanied by an example using the DAG τ_i from Figure 10.1. For the example, we assign execution requirement of each node W_i^j as follows: $E_i^1 = 4$, $E_i^2 = 2$, $E_i^3 = 4$, $E_i^4 = 5$, $E_i^5 = 3$, $E_i^6 = 4$, $E_i^7 = 2$, $E_i^8 = 4$, $E_i^9 = 1$, $E_i^{10} = 1$. Hence, $C_i = 30$, $P_i = 14$. Let period $T_i = 21$.

To perform the decomposition, we first represent DAG τ_i as a *timing diagram* τ_i^∞ (Figure 10.2(a)) that shows its execution time on an infinite number of unit-speed processor cores. Specifically, τ_i^∞ indicates the earliest start time and the earliest finishing time (of the worst case execution requirement) of each node when $m = \infty$. For any node W_i^j that has no

parents, the *earliest start time* and the *earliest finishing time* are 0 and E_i^j, respectively. For every other node W_i^j, the *earliest start time* is the latest finishing time among its parents, and the *earliest finishing time* is E_i^j time units after that. For example, in τ_i of Figure 10.1, nodes W_i^1, W_i^2, and W_i^3 can start execution at time 0, and their earliest finishing times are 4, 2, and 4, respectively. Node W_i^4 can start after W_i^1 and W_i^2 complete, and finish after 5 time units at its earliest, and so on. Figure 10.2(a) shows τ_i^∞ for DAG τ_i. Next, based on τ_i^∞, the calculation of D_i^j and Φ_i^j for each node W_i^j involves the following two steps. In Step 1, for each node, we estimate the time requirement at different parts of the node. In Step 2, the total estimated time requirements at different parts of the node is assigned as the node's deadline.

As stated before, our resource augmentation analysis is based on the densities of the decomposed tasks. As density of any task is the ratio of its execution requirement to its deadline, density of a sequential task is directly related to scheduling difficulty. The smaller the density, the looser the deadline, and hence is easier to schedule the task. Since, upon decomposition, we have a set of sequential subtasks, we analyze their schedulability based on their densities. The efficiency of the analysis is largely dependent on the total density (δ_{sum}) and the maximum density (δ_{max}) of the decomposed tasks. Namely, we need to keep both δ_{sum} and δ_{max} bounded and as small as possible to minimize the resource augmentation requirement. Therefore, the objective of the decomposition algorithm is to split the entire task deadline into node deadlines and to keep their densities small so that each node (subtask) has enough slack. The *slack* of any task represents the extra time beyond its execution requirement and is defined as the difference between its deadline and execution requirement.

Estimating Time Requirements of the Nodes

In DAG τ_i, a node can execute with different numbers of nodes in parallel at different times. Such a degree of parallelism can be estimated based on τ_i^∞. For example, in Figure 10.2(a), node W_i^5 executes with W_i^1 and W_i^3 in parallel for the first 2 time units, and then executes with W_i^4 in parallel for the next time unit. In this way, we first identify the degrees of parallelism at different parts of each node. Intuitively, the parts of a node that may execute with a large number of nodes in parallel demand more time. Therefore, different parts of a node are assigned different amounts of time considering these degrees of parallelism and

execution requirements. Later, the total time of all parts of a node is assigned to the node as its deadline.

To identify the degree of parallelism for different portions of a node based on τ_i^∞, we assign time units to a node in different (consecutive) segments. In different segments of a node, the task may have different degrees of parallelism. In τ_i^∞, starting from the beginning, we draw a vertical line at every time instant where a node starts or ends (as shown in Figure 10.2(b)). This is done in linear time using a breadth-first search over the DAG. The vertical lines now split τ_i^∞ into segments. For example, in Figure 10.2(b), τ_i is split into 7 segments (numbered from left to right).

Once τ_i^∞ is split into segments, each segment consists of an equal amount of execution by the nodes that lie in the segment. Parts of different nodes in the same segment can now be thought of as *threads of execution* that run in parallel, and the threads in a segment can start only after those in the preceding segment finish. We denote this synchronous form of τ_i^∞ by τ_i^{syn}. We first allot time to the segments, and finally add all times allotted to different segments of a node to calculate its deadline. Note that *τ_i is never converted to a synchronous model; the procedure only identifies segments to estimate time requirements of nodes, and does not decompose τ_i in this step.*

We split T_i time units among the nodes based on the number of threads and execution requirement of the segments where a node lies in τ_i^{syn}. We first estimate time requirement for each segment. Let τ_i^{syn} be a sequence of s_i segments numbered as $1, 2, \cdots, s_i$. For any segment j, we use m_i^j to denote the *number of threads in the segment*, and e_i^j to denote the *execution requirement of each thread* in the segment (see Figure 10.2(b)). Since τ_i^{syn} has the same critical path and total execution requirements as those of τ_i,

$$P_i = \sum_{j=1}^{s_i} e_i^j; \qquad C_i = \sum_{j=1}^{s_i} m_i^j . e_i^j$$

For any segment j of τ_i^{syn}, we calculate a value d_i^j, called the *segment deadline*, so that the segment is assigned a total of d_i^j time units to finish all its threads. Now we calculate the value d_i^j that minimizes both thread density and segment density that would lead to minimizing δ_{sum} and δ_{max} upon decomposition.

Since segment j consists of m_i^j parallel threads, with each thread having an execution requirement of e_i^j, the total execution requirement of segment j is $m_i^j e_i^j$. Thus, the segments with larger numbers of threads and with longer threads are computation-intensive, and demand more time to finish execution. Therefore, a reasonable way to assign the segment deadlines is to split T_i proportionally among the segments by considering their total execution requirement. Such a policy assigns a segment deadline of $\frac{T_i}{C_i} m_i^j e_i^j$ to segment j. Since this is the deadline for each parallel thread of segment j, by Equation (10.1), the density of a thread becomes $\frac{C_i}{m_i^j T_i}$ which can be as large as m. Hence, such a method does not minimize $\delta_{\max}$, and is not useful. Instead, we classify the segments of τ_i^{syn} into two groups based on a threshold θ_i on the number threads per segment: each segment j with $m_i^j > \theta_i$ is called a *heavy segment*, and each segment j with $m_i^j \leq \theta_i$ is called a *light segment*. Among the heavy segments, we allocate a portion of time T_i that is no less than that allocated among the light ones. Before assigning time among the segments, we determine a value of θ_i and the fraction of time T_i to be split among the heavy and light segments.

We show below that choosing $\theta_i = \frac{C_i}{2T_i - P_i}$ helps us keep both thread density and segment density bounded. Therefore, each segment j with $m_i^j > \frac{C_i}{2T_i - P_i}$ is classified as a *heavy segment* while other segments are called *light segments*. Let H_i denote the *set of heavy segments*, and L_i denote the *set of light segments* of τ_i^{syn}. This raises three different cases: when $L_i = \emptyset$ (i.e., when τ_i^{syn} consists of only heavy segments), when $H_i = \emptyset$ (i.e., when τ_i^{syn} consists of only light segments), and when $H_i \neq \emptyset,\ L_i \neq \emptyset$ (i.e., when τ_i^{syn} consists of both light segments and heavy segments). We use three different approaches for these three scenarios.

Case 1: when $H_i = \emptyset$. Since each segment has a smaller number ($\leq \frac{C_i}{2T_i - P_i}$) of threads, we only consider the length of a thread in each segment to assign time for it. Hence, T_i time units is split proportionally among all segments according to the length of each thread. For each segment j, its deadline d_i^j is calculated as follows.

$$d_i^j = \frac{T_i}{P_i} e_i^j \tag{10.4}$$

Since the condition $T_i \geq P_i$ must hold for every task τ_i to be schedulable,

$$d_i^j = \frac{T_i}{P_i} e_i^j \geq \frac{T_i}{T_i} e_i^j = e_i^j \tag{10.5}$$

Hence, the maximum density of a thread in any segment is at most 1. Since a segment has at most $\frac{C_i}{2T_i - P_i}$ threads, and $T_i \geq P_i$, the segment's density is at most

$$\frac{C_i}{2T_i - P_i} \leq \frac{C_i}{2T_i - T_i} = \frac{C_i}{T_i} \tag{10.6}$$

Case 2: when $L_i = \emptyset$. All segments are heavy, and T_i time units is split proportionally among all segments according to the work (i.e. total execution requirement) of each segment. For each segment j, its deadline d_i^j is given by

$$d_i^j = \frac{T_i}{C_i} m_i^j e_i^j \tag{10.7}$$

Since for every segment j, $m_i^j > \frac{C_i}{2T_i - P_i}$, we have

$$d_i^j = \frac{T_i}{C_i} m_i^j e_i^j > \frac{T_i}{C_i} \frac{C_i}{2T_i - P_i} e_i^j = \frac{2T_i}{2(2T_i - P_i)} e_i^j \geq \frac{e_i^j}{2} \tag{10.8}$$

Hence, the maximum density of any thread is at most 2. The total density of segment j is at most

$$\frac{m_i^j e_i^j}{d_i^j} = \frac{m_i^j e_i^j}{\frac{T_i}{C_i} m_i^j e_i^j} = \frac{C_i}{T_i} \tag{10.9}$$

Case 3: when $H_i \neq \emptyset$ and $L_i \neq \emptyset$. The task has both heavy segments and light segments. A total of $(T_i - P_i/2)$ time units is assigned to heavy segments, and the remaining $P_i/2$ time units is assigned to light segments. $(T_i - P_i/2)$ time units is split proportionally among heavy segments according to the work of each segment. The total execution requirement of heavy segments of τ_i^{syn} is denoted by C_i^{heavy}, defined as

$$C_i^{\text{heavy}} = \sum_{j \in H_i} m_i^j . e_i^j$$

For each heavy segment j, the deadline d_i^j is

$$d_i^j = \frac{T_i - \frac{P_i}{2}}{C_i^{\text{heavy}}} m_i^j e_i^j \tag{10.10}$$

Algorithm 7: Decomposition Algorithm

Input: a DAG task τ_i with period and deadline T_i, total execution requirement C_i, critical path length P_i;
Output: deadline D_i^j, offset Φ_i^j for each node W_i^j of τ_i;

for *each node* W_i^j *of* τ_i **do** $\Phi_i^j \leftarrow 0$; $D_i^j \leftarrow 0$; **end**;
Represent τ_i as τ_i^{syn};
$\theta_i \leftarrow C_i/(2T_i - P_i)$; /* heavy or light threshold */
$total\_heavy \leftarrow 0$; /* total heavy segments */
$total\_light \leftarrow 0$; /* total heavy segments */
$C_i^{\text{heavy}} \leftarrow 0$; /* total work of heavy segments */
$P_i^{\text{light}} \leftarrow 0$; /* light segments' critical path length */
for *each j-th segment in* τ_i^{syn} **do**
  if $m_i^j > \theta_i$ **then** /* it is a heavy segment */
    $total\_heavy \leftarrow total\_heavy + 1$;
    $C_i^{\text{heavy}} \leftarrow C_i^{\text{heavy}} + m_i^j e_i^j$;
  else /* it is a light segment */
    $total\_light \leftarrow total\_light + 1$;
    $P_i^{\text{light}} \leftarrow P_i^{\text{light}} + e_i^j$;
  end
end
if $total\_heavy = 0$ **then** /* all segments are light */
  for *each j-th segment in* τ_i^{syn} **do** $d_i^j = \frac{T_i}{P_i} e_i^j$;;
else if $total\_light = 0$ **then** /* all seg. are heavy */
  for *each j-th segment in* τ_i^{syn} **do** $d_i^j = \frac{T_i}{C_i} m_i^j e_i^j$;;
else /* τ_i^{syn} has both heavy and light segment */
  for *each j-th segment in* τ_i^{syn} **do**
    if $m_i^j > \theta_i$ **then** /* for heavy segment */
      $d_i^j = \frac{T_i - P_i/2}{C_i^{\text{heavy}}} m_i^j e_i^j$;
    else /* for light segment */
      $d_i^j = \frac{P_i/2}{P_i^{\text{light}}} e_i^j$;
    end
  end
end
/* Remove segments. Assign node deadlines */
for *each node* W_i^j *of* τ_i *in breadth-first search order* **do**
  if W_i^j *belongs to segments* k *to* r *in* τ_i^{syn} **then**
    $D_i^j = d_i^k + d_i^{k+1} + \cdots + d_i^r$; /* node deadline */
  $\Phi_i^j \leftarrow \max\{\Phi_i^l + D_i^l | W_i^l \text{ is a parent of } W_i^j\}$;
end

Since for each heavy segment j, $m_i^j > \frac{C_i}{2T_i - P_i}$, we have

$$d_i^j = \frac{(T_i - \frac{P_i}{2}) m_i^j e_i^j}{C_i^{\text{heavy}}} > \frac{(T_i - \frac{P_i}{2}) \frac{C_i}{2T_i - P_i} e_i^j}{C_i^{\text{heavy}}} \geq \frac{e_i^j}{2} \tag{10.11}$$

Hence, maximum density of a thread in any heavy segment is at most 2. As $T_i \geq P_i$, the total density of a heavy segment becomes

$$\frac{m_i^j e_i^j}{d_i^j} = \frac{m_i^j e_i^j}{\frac{T_i - \frac{P_i}{2}}{C_i^{\text{heavy}}} m_i^j e_i^j} = \frac{C_i^{\text{heavy}}}{T_i - \frac{P_i}{2}} \leq \frac{C_i}{T_i - \frac{T_i}{2}} = \frac{2C_i}{T_i} \tag{10.12}$$

Now, to distribute time among the light segments, $P_i/2$ time units is split proportionally among light segments according to the length of each thread. The critical path length of light segments is denoted by P_i^{light}, and is defined as follows.

$$P_i^{\text{light}} = \sum_{j \in L_i} e_i^j$$

For each light segment j, the deadline d_i^j is

$$d_i^j = \frac{\frac{P_i}{2}}{P_i^{\text{light}}} e_i^j \tag{10.13}$$

The density of a thread in any light segment is at most 2 since

$$d_i^j = \frac{\frac{P_i}{2}}{P_i^{\text{light}}} e_i^j \geq \frac{\frac{P_i}{2}}{P_i} e_i^j = \frac{e_i^j}{2} \tag{10.14}$$

Since a light segment has at most $\frac{C_i}{2T_i - P_i}$ threads, and $T_i \geq P_i$, the total density of a light segment is at most

$$\frac{2C_i}{2T_i - P_i} \leq \frac{2C_i}{2T_i - T_i} = \frac{2C_i}{T_i} \tag{10.15}$$

Calculating Deadline and Offset for Nodes

We have assigned segment deadlines to (the threads of) each segment of τ_i^{syn} in Step 1 (Equations (10.4), (10.7), (10.10), (10.13)). Since a node may be split into multiple (consecutive) segments in τ_i^{syn}, now we have to remove all segment deadlines of a node to reconstruct (restore) the node. Namely, we add all segment deadlines of a node, and assign the total as the node's deadline.

Now let a node W_i^j of τ_i belong to segments k to r ($1 \leq k \leq r \leq s_i$) in τ_i^{syn}. Therefore, the deadline D_i^j of node W_i^j is calculated as follows.

$$D_i^j = d_i^k + d_i^{k+1} + \cdots + d_i^r \tag{10.16}$$

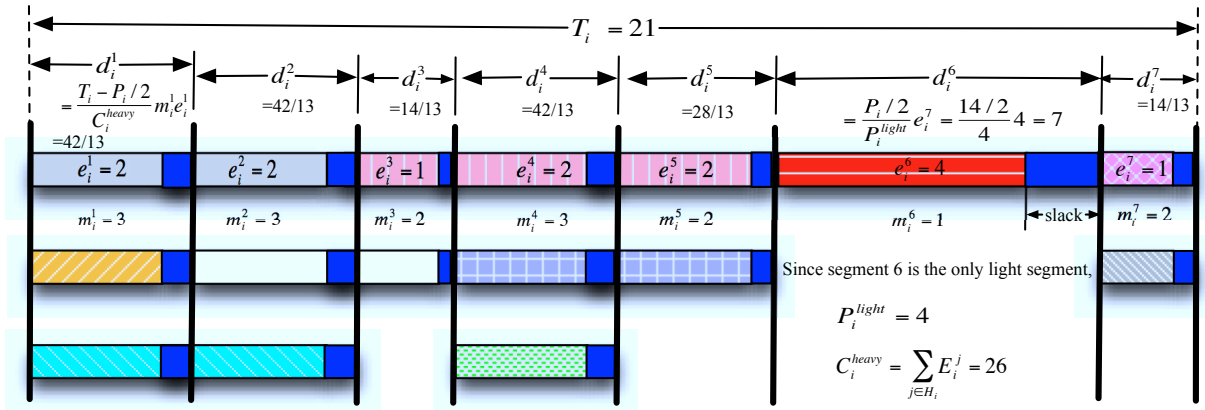
Note the execution requirement E_i^j of node W_i^j is

$$E_i^j = e_i^k + e_i^{k+1} + \cdots + e_i^r \tag{10.17}$$

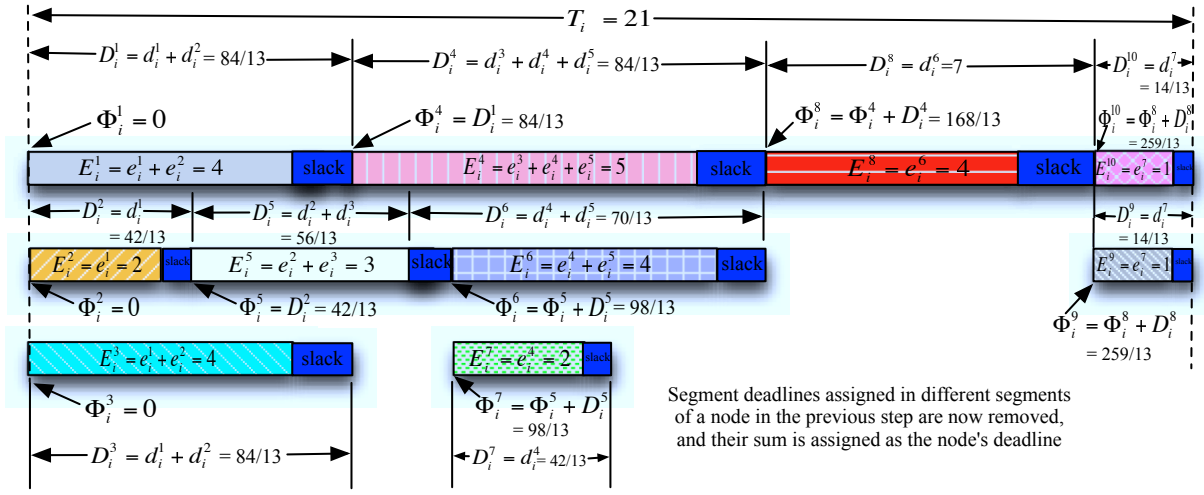
Node W_i^j cannot start until all of its parents complete. Hence, its release offset Φ_i^j is determined as follows.

$$\Phi_i^j = \begin{cases} 0; \text{ if } W_i^j \text{ has no parent} \\ \max\{\Phi_i^l + D_i^l | W_i^l \text{ is a parent of } W_i^j\}; \text{ otherwise.} \end{cases}$$

Now that we have assigned an appropriate deadline D_i^j and release offset Φ_i^j to each node W_i^j of τ_i, the DAG τ_i is now decomposed into nodes. Each node W_i^j is now an individual (sequential) multiprocessor subtask with an execution requirement E_i^j, a constrained deadline D_i^j, and a release offset Φ_i^j. Note that the period of W_i^j is still the same as that of the original DAG which is T_i. The release offset Φ_i^j ensures that node W_i^j can start execution no earlier than W_i^j time units following the release time of the original DAG. Our method guarantees that for a general DAG no node is split into smaller subtasks to ensure node-level non-preemption. Thus, the (node-level) non-preemptive behavior of the original task is preserved in scheduling the nodes as individual tasks, where nodes of the DAG are never



(a) Calculating segment deadlines of τ_i^{syn}



(b) Removing segment deadlines, and calculating node deadlines and offsets

Figure 10.3: Decomposition of τ_i (shown in Figure 10.1) when $T_i = 21$

preempted. The entire decomposition method is presented as Algorithm 7 which runs in linear time (in terms of the DAG size i.e., number of nodes and edges). Figure 10.3 shows the complete decomposition of τ_i.

10.4.3 Density Analysis after Decomposition

After decomposition, let τ_i^{dec} denote all subtasks (i.e., nodes) that τ_i generates. Note that the densities of all such subtasks comprise the density of τ_i^{dec}. Now we analyze the density of τ_i^{dec} which will later be used to analyze schedulability.

Let node W_i^j of τ_i belong to segments k to r ($1 \leq k \leq r \leq s_i$) in τ_i^{syn}. Since W_i^j has been assigned deadline D_i^j, by Equations (10.16) and (10.17), its density δ_i^j after decomposition is

$$\delta_i^j = \frac{E_i^j}{D_i^j} = \frac{e_i^k + e_i^{k+1} + \cdots + e_i^r}{d_i^k + d_i^{k+1} + \cdots + d_i^r} \tag{10.18}$$

By Equations (10.5), (10.8), (10.11), (10.14), $d_i^k \geq \frac{e_i^k}{2}$, $\forall i, k$. Hence, from (10.18),

$$\delta_i^j = \frac{E_i^j}{D_i^j} \leq \frac{2e_i^k + 2e_i^{k+1} + \cdots + 2e_i^r}{e_i^k + e_i^{k+1} + \cdots + e_i^r} = 2 \tag{10.19}$$

Let τ^{dec} be the set of all generated subtasks of all original DAG tasks, and $\delta_{\max}$ be the *maximum density* among all subtasks in τ^{dec}. By Equation (10.19),

$$\delta_{\max} = \max\left\{\delta_i^j \middle| 1 \leq j \leq n_i,\ 1 \leq i \leq n\right\} \leq 2 \tag{10.20}$$

We use $D_{\min}$ to denote the minimum deadline among all subtasks in τ^{dec}. That is,

$$D_{\min} = \min\left\{D_i^j \middle| 1 \leq j \leq n_i,\ 1 \leq i \leq n\right\} \tag{10.21}$$

Theorem 30. *Let a DAG τ_i, $1 \leq i \leq n$, with period T_i, critical path length P_i where $T_i \geq P_i$, and maximum execution requirement C_i be decomposed into subtasks (nodes) denoted τ_i^{dec} using Algorithm 7. The density of τ_i^{dec} is at most $\frac{2C_i}{T_i}$.*

Proof. Since we decompose τ_i into nodes, the densities of all decomposed nodes W_i^j, $1 \leq j \leq n_i$, comprise the density of τ_i^{dec}. In Step 1, every node W_i^j of τ_i is split into threads in different segments of τ_i^{syn}, and each segment is assigned a segment deadline. In Step 2, we remove all segment deadlines in the node, and their total is assigned as the node's deadline.

If τ_i is scheduled in the form of τ_i^{syn}, then each segment is scheduled after its preceding segment is complete. That is, at any time at most one segment is active. By Equations (10.6), (10.9), (10.12), (10.15), a segment has density at most $\frac{2C_i}{T_i}$ (considering $T_i \geq P_i$). Hence, the overall density of τ_i^{syn} never exceeds $\frac{2C_i}{T_i}$. Therefore, it is sufficient to prove that removing segment deadlines in the nodes does not increase the task's overall density. That is, it is sufficient to prove that the density δ_i^j (Equation (10.18)) of any node W_i^j after removing its segment deadlines is no greater than the density $\delta_i^{j,\text{syn}}$ that it had before removing its segment deadlines.

Let node W_i^j of the original DAG task τ_i be split into threads in segments k to r ($1 \leq k \leq r \leq s_i$) in τ_i^{syn}. Since the total density of any set of tasks is an upper bound on its load (as proven in [79]), the load of the threads of W_i^j must be no greater than the total density of these threads. Since each of these threads is executed only once in the interval of D_i^j time units, based on Equation (10.2), the DBF of the thread, $thread_i^l$, in segment l, $k \leq l \leq r$, in the interval of D_i^j time units is expressed as

$$\text{DBF}(thread_i^l, D_i^j) = e_i^l$$

Therefore, using Equation (10.3), the load, denoted by $\lambda_i^{j,\text{syn}}$, of the threads of W_i^j in τ_i^{syn} for interval D_i^j is

$$\lambda_i^{j,\text{syn}} \geq \frac{e_i^k}{D_i^j} + \frac{e_i^{k+1}}{D_i^j} + \cdots + \frac{e_i^r}{D_i^j} = \frac{E_i^j}{D_i^j} = \delta_i^j$$

Since $\delta_i^{\text{j, syn}} \geq \lambda_i^{\text{j, syn}}$, for any W_i^j, we have $\delta_i^{\text{j, syn}} \geq \delta_i^j$. □

Let δ_{sum} be the *total density* of all subtasks τ^{dec}. Since, from Theorem 30, the density of each τ_i^{dec} is at most $\frac{2C_i}{T_i}$ where $T_i \geq P_i$,

$$\delta_{\text{sum}} \leq \sum_{i=1}^{n} \frac{2C_i}{T_i} = 2\sum_{i=1}^{n} \frac{C_i}{T_i} \tag{10.22}$$

10.4.4 Implementation Considerations

This paper provides the algorithmic foundation for building a real-time parallel scheduler for parallel tasks. We now provide a sketch (Figure 10.4) on how it be implemented on a real system. In principle, one can use any parallel language such as OpenMP [17] and Cilk-Plus [12] that provides parallel programming support through library routines and directives. For example, OpenMP directives are compiler **pragma** statements that indicate where and how parallelization can occur within a program. One such directive converts a regular **for** loop to a **parallel-for** loop, by prefacing the loop with **#pragma omp parallel for**. The programmer can specify their task set as a set of parallel programs written in some such parallel language. To make these tasks real-time tasks, the programmer must also specify task deadlines and periods. We assume that these things are specified using a separate task specification file that is also an input to the scheduler. In addition, the decomposition algorithm also needs the execution requirements of each node in task, which can also be either specified in the task specification file, or measured using a profiler.

Using the task specification file and the task set, the scheduler computes the intermediate deadlines and release times for each node. In addition, the compiler decomposes the task into individual nodes/subtasks. Once the intermediate deadlines are known, we can use a global priority queue to keep the subtasks sorted by priorities according to EDF. Now at runtime, the scheduler schedules these subtasks on m processors using OS support for scheduling priorities, runtime preemption, and synchronization. When a subtask becomes available, if a worker is free, it is simply scheduled on this worker. If no worker is free, it is added to the priority queue. In the preemptive scheduler, we can use Linux support for preemption to preempt tasks with lower priorities when a high-priority task becomes available. For non-preemptive scheduling, we disable preemption and add yield points after each node[3]. When a subtask yields at its yield point, the scheduler can schedule the highest priority task that is available in the priority queue.

[3]Most parallel languages already have this support, since this is when control returns to the scheduler. For others, these can be added by the compiler.

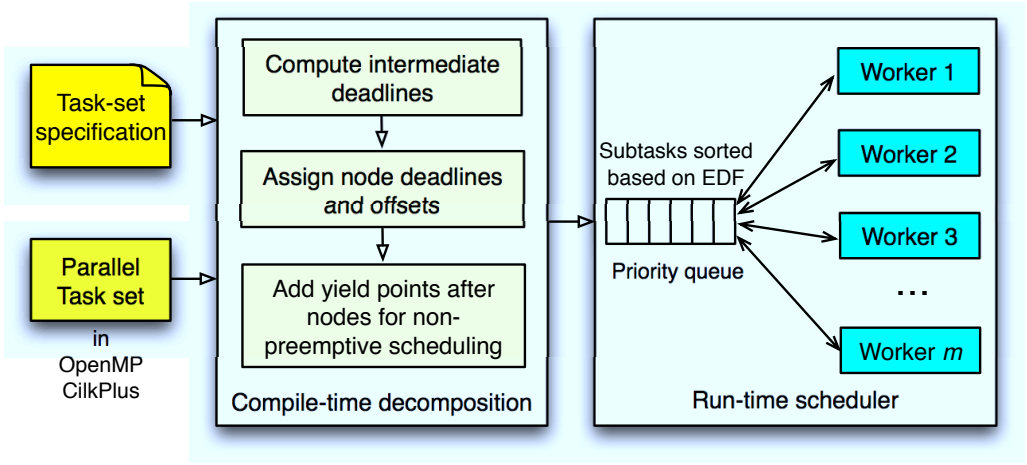


Figure 10.4: Scheduler components

10.5 Preemptive EDF Scheduling

Once all DAG tasks are decomposed into nodes (i.e., subtasks), we consider scheduling the nodes. Since every node after decomposition becomes a sequential task, we schedule them using traditional multiprocessor scheduling policies. In this section, we consider the preemptive global EDF policy.

Lemma 31. *For any set of DAGs $\tau = \{\tau_1, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set. If τ^{dec} is schedulable under some preemptive scheduling, then τ is preemptively schedulable.*

Proof. In each τ_i^{dec}, a node is released only after all of its parents finish execution. Hence, the precedence relations in original task τ_i are retained in τ_i^{dec} (that represent all subtasks of τ_i). Besides, the time by which the last subtask of τ_i^{dec} has to finish is equal to the deadline of the original task τ_i, and the sum of the execution requirements of these subtasks is equal to the execution requirement of the original task τ_i. Hence, if τ^{dec} is preemptively schedulable, a preemptive schedule must exist for τ where each task in τ meets its deadline. □

To schedule the decomposed subtasks τ^{dec}, the EDF policy is the same as the traditional global EDF policy where jobs with earlier absolute deadlines have higher priorities. Due to the *preemptive* policy, a job can be suspended (preempted) at any time by arriving higher-priority jobs, and is later resumed with (in theory) no cost or penalty. Under preemptive

global EDF, we now present a schedulability analysis for τ^{dec} in terms of a resource augmentation bound which, by Lemma 31, is also a sufficient analysis for the original DAG task set τ. For a task set, a *resource augmentation bound* ν of a scheduling policy $\mathbb{A}$ on an m-core machine is a processor speed-up factor. That is, if there exists any way to schedule the task set on m identical unit-speed processor cores, then $\mathbb{A}$ is guaranteed to successfully schedule it on an m-core processor with each core being ν times as fast as the original.

Our analysis hinges on a result (Theorem 32) for preemptive global EDF scheduling of constrained deadline sporadic tasks on a traditional multiprocessor platform [41]. This result is a generalization of the result for implicit deadline tasks [85].

Theorem 32. *(From [41]) Any constrained deadline sporadic sequential task set π with total density $\delta_{sum}(\pi)$ and maximum density $\delta_{\max}(\pi)$ is schedulable using preemptive global EDF policy on m unit-speed processor cores if*

$$\delta_{sum}(\pi) \leq m - (m-1)\delta_{\max}(\pi)$$

Note that τ^{dec} consists of constrained deadline (sub)tasks that are periodic with offsets. If they do not have offsets, then the above condition directly applies. Taking the offsets into account, the execution requirement, the deadline, and the period (which is equal to the period of the original DAG) of each subtask remains unchanged. The release offsets only ensure that some subtasks of the same original DAG are not executed simultaneously to preserve the precedence relations in the DAG. This implies that both δ_{sum} and $\delta_{\max}$ of the subtasks with offsets are no greater than δ_{sum} and $\delta_{\max}$, respectively, of the same set of tasks with no offsets. Hence, Theorem 32 holds for τ^{dec}. We now use the results of density analysis from Subsection 10.4.3, and prove that τ^{dec} is guaranteed to be schedulable with a resource augmentation of at most 4 in Corollary 3 that follows Theorem 33. The proof of Theorem 33 in this paper is similar to the proof used in [153] since both papers use the same prior result [41] to schedule the sequential subtasks. Note that our key contribution lies in decomposing DAGs to subtasks, not in scheduling of subtasks.

Theorem 33. *For any set of DAGs $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set. If every DAG τ_i satisfies the condition $T_i \geq P_i$, and the DAG set τ satisfies the condition $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m$ on m identical unit-speed processor cores, then the decomposed task set τ^{dec}*

is guaranteed to be schedulable under preemptive global EDF on m processor cores, each of speed 4.

Proof. If each DAG τ_i satisfies the condition $T_i \geq P_i$, then the total density δ_{sum} of the decomposed task set τ^{dec} is at most $2\sum_{i=1}^{n} \frac{C_i}{T_i}$ (Equation (10.22)), and the maximum density δ_{max} of τ^{dec} is at most 2 (Equation (10.20)) on unit-speed processors. To be able to schedule the decomposed tasks τ^{dec}, let each processor core be of speed ν, where $\nu > 1$. On an m-core platform where each core has speed ν, let the total density and the maximum density of task set τ^{dec} be denoted by $\delta_{\text{sum},\nu}$ and $\delta_{\text{max},\nu}$, respectively.

Considering that the condition $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m$ holds for τ, the total density of decomposed tasks τ^{dec} from Equation (10.22) is derived as follows on ν-speed cores.

$$\delta_{\text{sum},\nu} = \frac{\delta_{\text{sum}}}{\nu} \leq 2\sum_{i=1}^{n} \frac{\frac{C_i}{\nu}}{T_i} = \frac{2}{\nu}\sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{2m}{\nu} \tag{10.23}$$

On ν-speed cores, the maximum density of τ^{dec} is derived from Equation (10.20) as follows.

$$\delta_{\text{max},\nu} = \frac{\delta_{\text{max}}}{\nu} \leq \frac{2}{\nu} \tag{10.24}$$

Using Conditions (10.24) and (10.23) in Theorem 32, τ^{dec} is schedulable under preemptive EDF policy on m processor cores each of speed ν if

$$\frac{2m}{\nu} \leq m - (m-1)\frac{2}{\nu} \;\Leftrightarrow\; \frac{4}{\nu} - \frac{2}{m\nu} \leq 1$$

From the above condition, τ^{dec} must be schedulable if

$$\frac{4}{\nu} \leq 1 \quad \Leftrightarrow \quad \nu \geq 4.$$

□

Corollary 3. *For any set of DAGs $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set. If there exists any algorithm that can schedule τ on m unit-speed processor cores, then the decomposed task set τ^{dec} is guaranteed to be schedulable under preemptive global EDF on m cores, each of speed* 4.

Proof. If there exists any algorithm that can schedule τ on m unit-speed processor cores, then the following two conditions must hold.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \tag{10.25}$$

$$T_i \geq P_i, \text{ for each } \tau_i \tag{10.26}$$

If the above two conditions hold, then by Theorem 33 the decomposed task set τ^{dec} must be schedulable under preemptive global EDF on m cores, each of speed 4. Therefore, the corollary holds. □

Since Theorem 33 holds, we have the following straightforward schedulability test based on the resource augmentation bound of 4 for any set of DAGs: *For any set of DAGs* $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$*, if the total utilization* $u_{sum}(\tau) \leq \frac{m}{4}$ *and every DAG* τ_i *individually satisfies condition* $P_i \leq \frac{T_i}{4}$*, then the task set is schedulable under preemptive EDF policy upon decomposition.*

10.6 Non-Preemptive EDF Scheduling

We now address non-preemptive global EDF scheduling considering that the original task set τ is scheduled based on node-level non-preemption. In *node-level non-preemptive scheduling*, whenever the execution of a node in a DAG starts, the node's execution cannot be preempted by any task. Most parallel languages and libraries have yield points at the ends of threads (nodes of the DAG), where they allow low cost, user-space preemption. For these languages and libraries, schedulers that switch context only when threads end (i.e. where threads do not preempt each other) can be implemented entirely in user-space (without interaction with the kernel), and hence have low overheads.

The decomposition converts each node of a DAG to a traditional multiprocessor (sub)task. Therefore, we consider fully non-preemptive global EDF scheduling of the decomposed tasks. Namely, once a job of a decomposed (sub)task starts execution, it cannot be preempted by any other job.

Lemma 34. *For any set of DAGs $\tau = \{\tau_1, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set. If τ^{dec} is schedulable under some fully non-preemptive scheduling, then τ is schedulable under node-level non-preemption.*

Proof. Since the decomposition converts each node of a DAG to an individual task, a fully non-preemptive scheduling of τ^{dec} preserves the node-level non-preemptive behavior of task set τ. The rest of the proof follows from Lemma 31. □

Under non-preemptive global EDF, we now present a schedulability analysis for τ^{dec} in terms of a resource augmentation bound which, by Lemma 34, is also a sufficient analysis for the DAG task set τ. This analysis exploits Theorem 35 for non-preemptive global EDF scheduling of constrained deadline periodic tasks on traditional multiprocessor. The theorem is a generalization of the result for implicit deadline tasks [42].

For a task set π, let $C_{\max}(\pi)$ and $D_{\min}(\pi)$ be the maximum execution requirement and the minimum deadline among all tasks in π. In non-preemptive scheduling, $C_{\max}(\pi)$ represents the *maximum blocking time* that a task may experience, and plays a major role in schedulability. Hence, a *non-preemption overhead*, defined in [42], for the task set π is given by $\rho(\pi) = \frac{C_{\max}(\pi)}{D_{\min}(\pi)}$. The value of $\rho(\pi)$ indicates the added penalty or overhead associated with non-preemptivity. In other words, since preemption is not allowed, the capacity of each processor is reduced (at most) by a factor of $\rho(\pi)$. In non-preemptive scheduling, this capacity reduction is recompensed by reducing the cost associated with context-switch, saving state etc.

Theorem 35. *(From [42]) Any constrained deadline periodic task set π with total density $\delta_{sum}(\pi)$, maximum density $\delta_{\max}(\pi)$, and a non-preemption overhead $\rho(\pi)$ is schedulable using non-preemptive global EDF on m unit-speed cores if*

$$\delta_{sum}(\pi) \leq m\big(1 - \rho(\pi)\big) - (m-1)\delta_{\max}(\pi)$$

Let $E_{\max}$ and $E_{\min}$ be the maximum and minimum execution requirement, respectively, among all nodes of all DAG tasks. That is,

$$E_{\max} = \max\left\{E_i^j \middle| 1 \leq j \leq n_i,\ 1 \leq i \leq n\right\} \tag{10.27}$$

$$E_{\min} = \min\left\{E_i^j \middle| 1 \leq j \leq n_i,\ 1 \leq i \leq n\right\} \tag{10.28}$$

In node-level non-preemptive scheduling of the DAG tasks, the processor capacity reduction due to non-preemptivity is at most $\frac{E_{\max}}{E_{\min}}$. Hence, this value is the *non-preemption overhead* of the DAG tasks, and is denoted by ρ [4]:

$$\rho = \frac{E_{\max}}{E_{\min}} \tag{10.29}$$

Theorem 36 derives a resource augmentation bound of $4+2\rho$ for non-preemptive global EDF scheduling of the decomposed tasks.

Theorem 36. *For DAG model parallel tasks $\tau = \{\tau_1, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set with non-preemption overhead ρ. If there exists any way to schedule τ on m unit-speed processor cores, then τ^{dec} is schedulable under non-preemptive global EDF on m cores, each of speed $4 + 2\rho$.*

Proof. After decomposition, $D_{\min}$ (Equation (10.21)) is the minimum deadline among all subtasks in τ^{dec}. Since $E_{\max}$ (Equation (10.27)) represents the maximum blocking time that a subtask may experience, the non-preemption overhead of the decomposed tasks is $\frac{E_{\max}}{D_{\min}}$. From Equations (10.19) and (10.29), the non-preemption overhead of the decomposed tasks

$$\frac{E_{\max}}{D_{\min}} \leq \frac{E_{\max}}{E_{\min}/2} = \frac{2E_{\max}}{E_{\min}} = 2\rho \tag{10.30}$$

Similar to Theorem 33 and Corollary 3, suppose we need each processor core to be of speed ν to be able to schedule the decomposed tasks τ^{dec}. From Equation (10.30), the non-preemption overhead of τ^{dec} on ν-speed cores is

$$\frac{E_{\max}/\nu}{D_{\min}} \leq \frac{2\rho}{\nu} \tag{10.31}$$

Considering a non-preemption overhead of at most $\frac{2\rho}{\nu}$ on ν-speed processor cores, and using Equations (10.24) and (10.23) in Theorem 35, τ^{dec} is schedulable under non-preemptive EDF

[4] Unfortunately, $D_{\min}$ is not an input parameter in our case as its value is only known upon decomposition. Instead, the value of $E_{\min}$ is known from input, and hence we use this in defining ρ.

on m cores each of speed ν if

$$\frac{2m}{\nu} \leq m(1 - \frac{2\rho}{\nu}) - (m-1)\frac{2}{\nu} \;\Leftrightarrow\; \frac{4+2\rho}{\nu} - \frac{1}{m\nu} \leq 1$$

From the above condition, task set τ^{dec} is schedulable if

$$\frac{4+2\rho}{\nu} \leq 1 \;\Leftrightarrow\; \nu \geq 4 + 2\rho.$$

□

A Tighter Bound for Non-Preemptive EDF:

A resource augmentation of $4 + 2\rho$ for non-preemptive EDF is relatively looser than the corresponding bound of 4 for preemptive EDF. This is mainly because non-preemptivity can cause processor capacity reduction of up to ρ. Due to decomposition, this value increases to 2ρ (see Equation (10.30)). However, we can express the augmentation bound in a tighter form by using a tighter bound on non-preemption overhead. As shown in Equation (10.30) the non-preemption overhead of the decomposed task is indeed at most $\frac{E_{\max}}{D_{\min}}$. But we used a pessimistic upper bound of this value by replacing $D_{\min}$ with $E_{\min}$ as the value of $D_{\min}$ is unknown before decomposition. $E_{\min}$ is a lower bound of $D_{\min}$ and is known (from input) before decomposition. Therefore, if we define the bound upon decomposition, we can use $\frac{E_{\max}}{D_{\min}}$ as the maximum non-preemption overhead. Using this value of non-preemption overhead in Theorem 36, our bound will be $4 + \frac{E_{\max}}{D_{\min}}$ which is a lot smaller than $4 + 2\rho$.

Notably, the work in [24] has identified a large class of applications such as high-performance web and data-servers that consist of many real-time tasks, called *liquid tasks*, in which the smallest deadline of any job in the system is orders of magnitude greater than the largest execution requirement of any job. Upon decomposing the liquid tasks, the value of $D_{\min}$ can be very close to $E_{\max}$. Thus the value of $\frac{E_{\max}}{D_{\min}}$ approaches 1, and a resource augmentation of $4 + \frac{E_{\max}}{D_{\min}}$ is tight and quite useful in scheduling liquid parallel tasks. Our result provides the first such bound for non-preemptive real-time scheduling of parallel tasks, and provides the basis for future directions to derive tighter bounds for all classes of real-time tasks.

10.7 Evaluation

The derived resource augmentation bounds provide a sufficient condition for schedulability. Namely, if a set of DAG tasks is schedulable on a unit-speed m-core machine by a (potentially non-existing) ideal scheduler, then the tasks upon our proposed decomposition are guaranteed to be schedulable under global EDF on an m-core machine where each core has a speed of 4 (with preemption) or $4 + 2\rho$ (without preemption).

In this section, we evaluate our scheduler using simulations. We simulate the execution of a set of parallel tasks under scheduling algorithms to observe deadline misses. We developed a simple event-driven simulator where task executions are simulated in parallel as if they executed on m cores. We first randomly create tasks and then calculate subtask deadlines using our proposed decomposition method. We then simulate the execution of these subtasks. The environment consists of m cores and a global priority queue which keeps subtasks in the order of priorities based on EDF. An event occurs when a subtask is released or completed. When a subtask t is released, a preemptive and non-preemptive schedulers behave differently. In a non-preemptive scheduler, two things can occur: (i) If a core is free, then t is scheduled on that core; (ii) If all cores are busy, then the task is added to the priority queue. On a preemptive scheduler, if all cores are busy, but another subtask s with a deadline later than t's deadline is executing, then s is preempted and placed in the priority queue, and t is scheduled instead of s. When a subtask completes, the highest priority subtask from the queue is executed on the core that has just become free. This is a simple simulator which only simulates the task executions and ignores overheads due to migration, cache misses, preemption, and synchronization.

10.7.1 Task and Task Set Generation

We want to evaluate our scheduler using task sets that an optimal scheduler could schedule on 1-speed processors. However, as we cannot determine this ideal scheduler, we assume that an ideal scheduler can schedule any task set whose total utilization is no greater than m, and that each individual task is schedulable in isolation (i.e. its critical path length is no greater than its deadline). Therefore, in our experiments, for each value of m (i.e. the number

of cores), we generate task sets whose utilization is exactly m, fully loading a machine of 1-speed processors.

We use the Erdös-Rényi method $G(n_i, p)$ [65] as presented below to generate task sets for evaluation.

Number of nodes. To generate a DAG τ_i, we pick the number of nodes n_i uniformly at random in range $[50, 350]$. These values would allow us to generate varied task sets within a reasonable amount of time.

Adding edges. We add edges to the graph using the Erdös-Rényi method $G(n_i, p)$ [65]. We scan all the possible edges directing from lower node id to higher node id to avoid introducing a cycle into the graph. For each possible edge, we generate a random value in range $[0, 1]$ and add the edge only if the generated value is less than a predefined probability p. (We will vary p in our experiments to explore the effect of changing p.) Finally, we add an additional minimum number of edges so that each node (except the first and the last node) has at least one incoming and one outgoing edge in order to make the DAG weakly connected. Note that the critical path length of a DAG generated using the pure Erdös-Rényi method increases as p increases. Since our method is slightly modified, the critical path is also large when p is small. Hence, as p increases, the critical path first decreases up to a certain value of p and then increases again.

Execution time of nodes. We assign every node an execution time chosen randomly from a specified range. The range is based on the value and type (continuous or discrete) of the non-preemption overhead ρ (explained in the next subsection).

At this point, we have the DAG structure and the execution times for its nodes. For each DAG τ_i, we now assign a period T_i that is no less than the critical path length P_i. We consider two types of task sets:

Task sets with harmonic periods. These deadlines are carefully picked so that they are multiples of each other, so as to ensure that we can run our experiments up to the hyper-period of the task sets. In particular, we pick deadlines that are powers of two. We find the smallest value a such that $P_i \leq 2^a$, and randomly set T_i to be one of 2^a, 2^{a+1}, or 2^{a+2}. We choose such periods because we want some high utilization tasks and some low utilization

tasks. The ratio P_i/T_i of the task is in the range $[1, 1/2]$, (1/2,1/4], or $(1/4, 1/8]$, when its period T_i is 2^a, 2^{a+1}, or 2^{a+2}, respectively.

Task sets with arbitrary periods. We first generate a random number Gamma$(2, 1)$ using the *gamma distribution* [14]. Then we set period T_i to be $(P_i + \frac{C_i}{0.5m}) * (1 + 0.25 * \text{Gamma}(2, 1))$. We choose this formula for three reasons. First, we want to ensure that the assigned value is a *valid* period, i.e., $P_i \leq T_i$. Second, we want to ensure that each task set contains a reasonable number of tasks even when m is small. At the same time, with more cores, we do not want to limit average DAG utilization to a certain small value. Hence the minimum period is a function of m. Third, while we want the average period to be close to the minimum valid period (to have high utilization tasks), we also want some tasks with large periods. Table 10.1 shows the average number of DAGs per task set achieved by the random period generation process.

m \ p	0.01	0.02	0.03	0.05	0.07	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4	4	4	4	4	4	4	5	6	6	7	7	8	8	8
8	4	4	4	4	5	5	7	8	9	10	11	12	13	14
16	4	5	5	6	6	7	10	12	15	17	19	20	22	24
32	5	6	7	8	9	11	17	22	26	30	34	37	41	45

Table 10.1: Number of tasks per task set

To create a task set we combine individual DAGs as follows. We add DAGs to the task set until the total utilization of the set exceeds m. We then remove the last generated DAG. Thus, at this point, the total utilization is smaller than m. To make the total utilization exactly m, we add small DAGs with long periods (i.e., of small utilization). We stop adding small DAGs when the total utilization is larger than 99% of m.

10.7.2 Experimental Methodology

We experiment by varying the following 4 parameters.

Harmonic vs. arbitrary periods. We want to evaluate whether arbitrary periods are better or worse than harmonic ones. For harmonic period task sets, we run simulation up

to their hyper-period. For arbitrary period task sets, the hyper-period can be too long to simulate, and hence we run simulation up to 20 times the maximum period.

Number of cores (m). We want to evaluate if parallel scheduling is easier or harder as the number of cores increases. We run experiments on m: $4, 8, 16$, and 32.

Probability of an edge (p). As stated before, p affects the critical path length, the density, and the structure of the DAG. We test using 14 values of p: $0.01, 0.02, 0.03, 0.05, 0.07, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$.

Non-preemption overhead (ρ). This is the ratio of the maximum node execution length to the minimum node execution length. For non-preemptive EDF scheduling, the resource augmentation bound increases as ρ increases. We want to evaluate whether the effect of increased ρ is really that severe in practice. For all of our experiments, we set the minimum node execution requirement to be 50, and vary the maximum execution requirement. To get $\rho = 1, 2, 5$, and 10, the maximum execution requirements are chosen to be $50, 100, 250$, and 500, respectively. In addition, when we evaluate the performance of non-preemptive EDF, we want to maximize the influence of ρ. Therefore, besides using uniformly generated node execution time between maximum and minimum (called *continuous* ρ), we also generate by choosing from discrete values 50, $2 * 50$, $\cdots$, $\rho * 50$ (called *discrete* ρ).

In all experiments, we simulate 1000 task sets. For each task set, we start by simulating its execution on 1-speed processors, and increase the speed by 0.1 intervals until all task sets are schedulable. Using these different task sets, we conduct two sets of experiments. In our first set, we evaluate the scheduler under preemptive global EDF. Hence, we vary the types of period, m and p, but keep ρ constant at 2, leading to 112 combinations. In the second set, we evaluate under non-preemptive global EDF by varying all four factors, leading to 896 combinations.

10.7.3 Results

Of the 896 combinations of parameters (each having 1000 task sets) we have tested, preemptive EDF has the maximum required speed of 3.2 to meet all deadlines (this data point is not

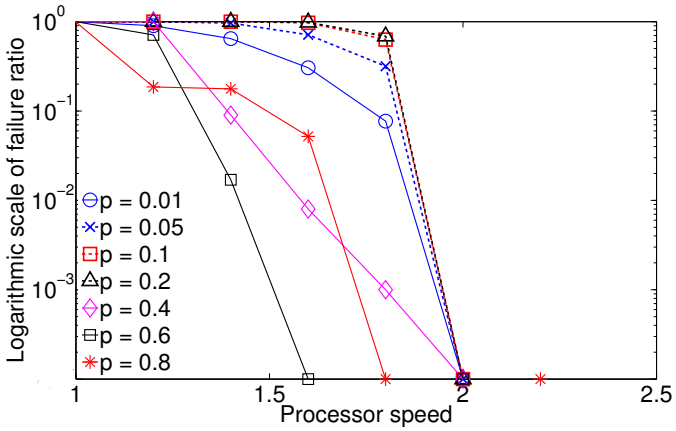


Figure 10.5: Failure ratio in preemptive EDF on 32 cores under different edge probability

shown in figures for better resolution), which is close to our analytical resource augmentation bound of 4. In contrast, among the combinations of parameters with $\rho = 1, 2, 5, 10$, the maximum required speed for non-preemptive EDF are 4.0, 5.8, 8.6, and 12.6, respectively, which look much smaller than the analytical bound of 6, 8, 14, and 24, respectively. These issues are discussed upon presenting the results. For brevity, we present only a subset of the experimental results.

Effect of harmonic vs. arbitrary periods. We find that it is slightly harder to schedule harmonic period tasks using preemptive EDF, and vice-versa for non-preemptive EDF. However, the difference is minor, and the trends are very similar under both. Here we will only show the experiments for arbitrary periods.

Effect of p in preemptive scheduling. For each value of p, Figure 10.5 shows the *failure ratio* defined as the ratio of the number of task sets where some task missed a deadline to the total number of task sets (which is 1000 in our experiment) attempted to be scheduled. To preserve resolution of the figure, we show the results for only 7 (out of 14) values of p. In these experiments, $\rho = 2$, $m = 32$. Note that the failure ratio increases as p increases from 0.01 to 0.1, and then falls again. As we explained in Section 10.7.1, as p increases, the critical-path length first decreases (making the tasks more "parallel" or "DAG-like") and then increases again (making the tasks more sequential). Therefore, for both small and large p, the tasks are largely sequential. These results seem to conform to our intuition that, in

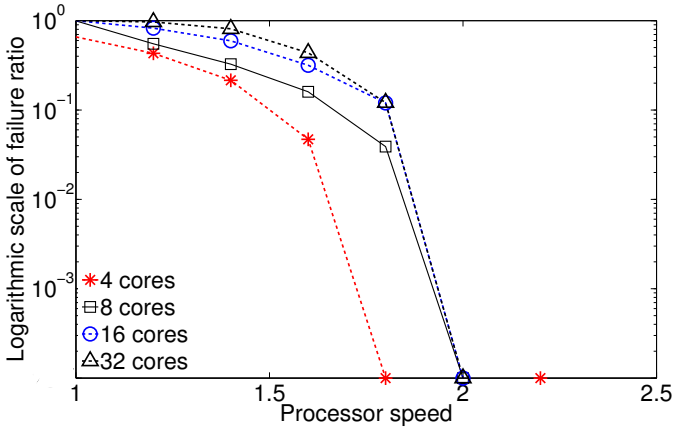


Figure 10.6: Failure ratio in preemptive EDF on different numbers of cores

general, parallel tasks are more difficult to schedule than sequential ones. The results for 4, 8, and 16 cores also follow this trend, and hence are omitted.

Effect of m in preemptive scheduling. Figure 10.6 shows the failure ratio in logarithmic scale for each value of m, when $p = 0.2$ and $\rho = 2$. The failure ratio increases as m increases, indicating that it is harder to schedule on larger numbers of cores. The trend is similar for different values of p, and hence is not shown.

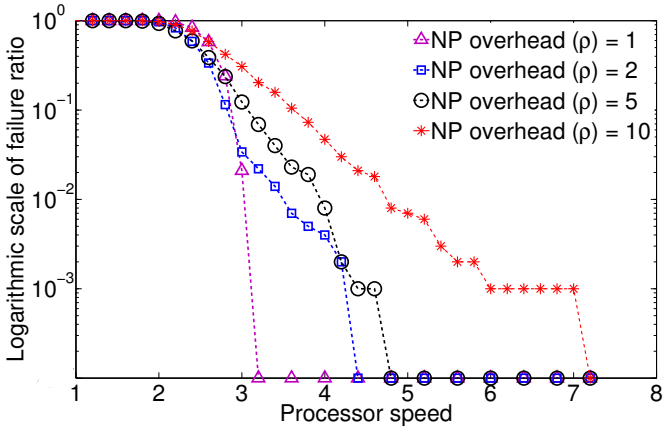


Figure 10.7: Failure ratio in non-preemptive EDF on 8 cores under different non-preemption overhead

Effect of ρ in non-preemptive scheduling. The most important factor to evaluate is the effect of ρ. Figure 10.7 shows the failure ratio for discrete ρ for each value of ρ, with

fixed $p = 0.2$, $m = 8$. With the increase in ρ, the failure ratio becomes much higher, which is expected. However, this trend is not quite strong for continuous ρ, and we omit plotting those results. Following may be the reason for this anomaly. The maximum value of ρ only affects the schedule if a node having the maximum execution interferes with a node having the minimum execution. Since ρ is continuous, a node's execution requirement is assigned from many different values. This causes only a small number of nodes to be at these extremes, thereby reducing the chances of such interference.

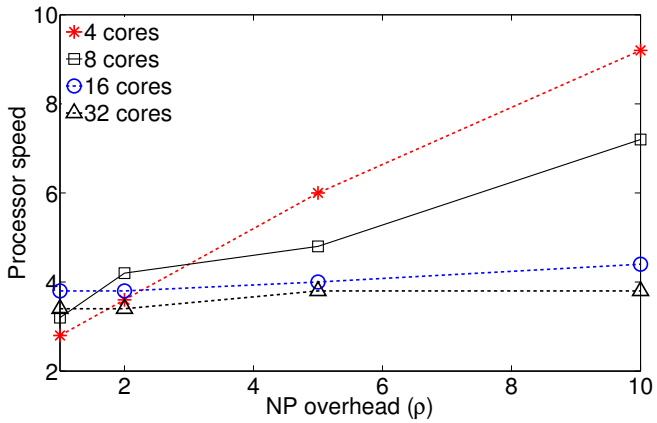


Figure 10.8: Required speed in non-preemptive EDF on different numbers of cores with increasing non-preemption overhead

Effect of m in non-preemptive scheduling. Figure 10.7 shows the required speed for each combination of m and ρ, with $p = 0.2$. This figure is different from the previous ones in that it only shows the speed at which all task sets become schedulable. We can see that for each value of m, when ρ increases, the required speed increases, which is expected. This trend is less visible when m increases. One possible reason is that when there are more cores, the overhead from interference between executing low priority subtask and a newly released higher priority subtask will, on average, be smaller. This happens because the overhead is the minimum remaining work of all m running lower priority subtasks, instead of the average or worst case subtask execution time. When m is higher, the minimum will be much smaller than average, making the system much less influenced when ρ increases.

The simulation results show a maximum speed requirement of 3.2 for preemptive EDF suggesting that our analytical resource augmentation bound of 4 is reasonably tight. The corresponding bounds for non-preemptive EDF sound relatively looser in our simulation results.

This is because, as stated in Section 10.6, non-preemptivity can cause processor capacity reduction of up to ρ in the worst case. However, as our bound holds for all task sets, some task sets are likely to exist for which a bound of $4 + 2\rho$ may still be tight, but our simulation does not encounter those tasks. In other words, our results may be an artifact of our experimental set up and random task generation that is unlikely to generate the worst-case task set.

10.8 Summary

As multi-core technology becomes mainstream in processor design, real-time scheduling of parallel tasks is crucial to exploit its potential. In this paper, we consider a general task model and through a novel task decomposition we prove a resource augmentation bound of 4 for preemptive EDF, and 4 plus a non-preemption overhead for non-preemptive EDF scheduling. To our knowledge, these are the first bounds for real-time scheduling of general DAG model tasks. Through simulations, we have observed that the required augmentation bounds are safe in practice. Our results suggest several possible directions of future work. One direction is to provide better bounds and/or provide lower bound arguments to argue that the bounds are in fact tight. Another possible direction is to study the effect of caches on scheduling overhead. While non-preemption mitigates this problem to some extent, more can be done to optimize cache-locality. Finally, we can generalize our results to models that take into account the effects of non-deterministic synchronization such as locks.

Chapter 11

Conclusion

Wireless sensor-actuator networks represent a new communication infrastructure for wireless cyber-physical systems (CPS). Sensing and control in these systems need to meet stringent real-time performance requirements on communication latency in challenging environments for a large class of CPS applications such as process control, smart manufacturing, and data center management. Real-time transmission scheduling and analysis for wireless sensor-actuator networks requires new methodologies to deal with unique characteristics of wireless communication. Furthermore, the performance of a wireless control involves intricate interactions between real-time communication and control. In this thesis, we have addressed these challenges and made a series of contributions to the theory and system for wireless CPS. (1) We have established a new real-time scheduling theory for wireless sensor-actuator networks. (2) We have developed a scheduling-control co-design approach for holistic optimization of control performance in a wireless control system. (3) We have designed and implemented a wireless sensor-actuator network for CPS in data center power management. (4) We have expanded our research to develop scheduling algorithms and analyses for real-time parallel computing to support computation-intensive CPS.

This thesis research has made important contribution towards, and impacts on the field. It has provided the first set of results on real-time wireless networks, and has kicked off a research domain towards the evolution of wireless CPS. Our work on parallel computing for computation intensive CPS is regarded as one of the pioneering works in the literature of Real-Time Parallel Computing. This research has inspired many follow-up works. The CapNet implementation provides the first proof of concept design for using low cost wireless

for CPS in data center power capping. This thesis also opens a number of future research problems. Specifically, we will address the following research agenda in our future work.

Many wireless CPS may tolerate a certain degree of packet loss or deadline misses at the cost of degraded performance. Hence, we plan to incorporate packet loss, latency, and deadline misses in co-design optimization for CPS. We will make the solutions scalable through local adaptation or hierarchical networking, and closely integrate theoretical research and system experimentation on testbeds. We will also work on multi-core systems which will be heavily exploited by the future CPS. In addition to aggressively utilizing a multi-core, it is critical to ensure efficiency in terms of memory and resource sharing. Hence, in the future, we will consider parallel computing by taking into account cache effects on scheduling overhead (e.g., by optimizing cache-locality), and by considering non-deterministic synchronization such as locks.

Emerging large-scale wireless CPS (e.g., process control, civil structure control) will require many sensors and actuators connected over long distances. With current short-range wireless (e.g., IEEE 802.15.4, 802.11), these applications form many-hop mesh networks and address significant engineering challenges in scalability, time synchronization, and protocol design. For example, the emerging wireless control networks may have tens of thousands of field devices [21] which will require significant wiring to integrate numerous small networks in a large deployment, significantly reducing the benefit of wireless. Similarly, due to short radio range, the current structural monitoring wireless sensor network deployed on Golden Gate Bridge [105] forms a 46-hop network that cannot collect sensor data in real-time. Due to long communication range, using TV spectrum white spaces (i.e., allocated but unused TV spectrum [38, 38]) can overcome these limitations and simplify network design.

While the potential of white spaces is mostly being tapped into for wireless broadband access, we will research on exploiting these for CPS applications that involve wide-area sensing and control. Federal Communications Commission poses strict rules on the white spaces usage policy to ensure no interference with the primary users (TV, wireless microphones). Long ranges also bring forth brand new challenges in energy efficiency, resource contention, and network management for sensor networking, requiring the development of new energy efficient MAC protocols. Another class of challenges rises in spectrum management as the spectrum availability changes with time and geographic location. How often to perform spectrum

sensing and how to share the duty among sensor nodes to sense spectrum availability is an interesting research question. Since a white space network must vacate a channel as soon as a primary user occupies it, new protocols must be developed to handle temporal disruptions in real-time sensor networking. Another challenge lies in handling node joining process as the channel availability at one node is unknown to the other end. Developing efficient protocols to handle mobility also brings new challenges. We will tackle these challenges in our future research. Our vision is to engender *White Space Sensor Networking* that can eventually supersede current wireless sensor network technology for wide-area sensing and control applications

References

[1] http://www.hse.gov.uk/pubns/regindex.htm.

[2] http://mobilab.wustl.edu/testbed.

[3] Private communication with data center operators.

[4] http://www.cdwg.com/shop/products/Digi-Passport-48-console-server/1317701.aspx.

[5] http://www.cdwg.com/shop/search/Servers-Server-Management/Servers/x86-Based-Servers/result.aspx?w=S62&pCurrent=1&p=200008&a1520=002200.

[6] http://www.cdwg.com/shop/search/Cables/Networking-Cables/Category-5-TP-Cables-Ethernet/result.aspx?w=B25&pCurrent=1&ctlgfilter=&key=rj45+cable&searchscope=All&sr=1&x=0&y=0#Brand.

[7] http://www.cdwg.com/shop/search/Networking-Products/Ethernet-Switches/Fixed-Managed-Switches/result.aspx?w=N11&MaxRecords=25&SortBy=TopSellers.

[8] http://www.digikey.com/us/en/techzone/wireless/resources/articles/comparing-low-power-wireless.html.

[9] http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en535967.

[10] http://en.wikipedia.org/wiki/Teraflops_Research_Chip.

[11] www.clearspeed.com.

[12] http://software.intel.com/en-us/articles/intel-cilk-plus.

[13] www.amd.com/us/products/server/processors.

[14] http://en.wikipedia.org/wiki/Gamma_distribution.

[15] HP power capping and HP dynamic power capping for ProLiant servers. Technology brief, 2nd edition. http://h20000.www2.hp.com/bc/docs/support/SupportManual/c01549455/c01549455.pdf.

[16] Intelligent power optimization for higher server density racks a baidu case study with intel intelligent power technology. http://www.intel.com/content/dam/doc/case-study/data-center-efficiency-xeon-baidu-case-study.pdf.

[17] OpenMP. http://openmp.org.

[18] Subgradient solver: SSMS. http://www.searching-eye.com/sanjeevsharma/matlab_solver/subgradient_solver/.

[19] TelosB. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[20] TinyOS Community Forum. http://www.tinyos.net/.

[21] WirelessHART system engineering guide. http://www2.emersonprocess.com/siteadmincenter/PM%20Central%20Web%20Documents/EMR_WirelessHART_SysEngGuide.pdf.

[22] WirelessHART, 2007. http://www.hartcomm2.org.

[23] Rockwell Automation Inc Bulletin 1489 circuit breakers selection guide, 2010. http://literature.rockwellautomation.com/idc/groups/literature/documents/sg/1489-sg001_-en-p.pdf.

[24] T. Abdelzaher, B. Andersson, J. Jonsson, V. Sharma, and M. Nguyen. The aperiodic multiprocessor utilization bound for liquid tasks. In *RTAS '02*.

[25] Tarek F. Abdelzaher, Shashi Prabh, and Raghu Kiran. On real-time capacity limits of multihop wireless sensor networks. In *RTSS '04*.

[26] A. Adya, P. Bahl, J. Padhye, A. Wolman, and Lidong Zhou. A multi-radio unification protocol for ieee 802.11 wireless networks. In *BroadNets '04*.

[27] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive task scheduling with parallelism feedback. In *PPoPP '06*.

[28] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26(3), 2008.

[29] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[30] Mansoor Alicherry, Randeep Bhatia, and Li (Erran) Li. Joint channel assignment and routing for throughput optimization in multi-radio wireless mesh networks. In *MobiCom '05*.

[31] Mansoor Alicherry, Randeep Bhatia, and Li (Erran) Li. Joint channel assignment and routing for throughput optimization in multi-radio wireless mesh networks. In *MobiCom '05*.

[32] R. Alur, D'Innocenzo, Johansson, Pappas, and G. Weiss. Modeling and analysis of multi-hop control network. In *RTAS '09*.

[33] James Anderson and John Calandrino. Parallel real-time task scheduling on multicore platforms. In *RTSS '06*.

[34] Nimar Arora, Robert Blumofe, and C Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*.

[35] N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1), 2001.

[36] Werner Backes and Jared Cordasco. Moteaodv – an aodv implementation for tinyos 2.0. In *Proceedings of the 4th IFIP WG 11.2 international conference on Information Security Theory and Practices: security and Privacy of Pervasive Systems and Smart Devices*, WISTP '10, 2010.

[37] Paramvir Bahl, Ranveer Chandra, and John Dunagan. SSCH: slotted seeded channel hopping for capacity improvement in ieee 802.11 ad-hoc wireless networks. In *MobiCom '04*.

[38] Paramvir Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. White space networking with Wi-Fi like connectivity. In *SIGCOMM '09*.

[39] Nikhil Bansal, Kedar Dhamdhere, Jochen Konemann, and Amitabh Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, 2004.

[40] Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. In *PODC '10*.

[41] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS '07*.

[42] Sanjoy Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Syst.*, 32:9–20, 2006.

[43] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS '12*.

[44] Sanjoy Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS '90*.

[45] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Dist. Sys., IEEE Transactions on*, 20(4):553 –566, 2009.

[46] V. Bhandari and N.H. Vaidya. Capacity of multi-channel wireless networks with random (c, f) assignment. In *MobiHoc '07*.

[47] V. Bhandari and N.H. Vaidya. Connectivity and capacity of multi-channel wireless networks with channel switching constraints. In *INFOCOM '07*.

[48] Arka A. Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. The need for speed and stability in data center power capping. In *IGCC '12*.

[49] E Bini and A Cervin. Delay-aware period assignment in control systems. In *RTSS '08*.

[50] M.S. Branicky, S.M. Phillips, and Wei Zhang. Scheduling and feedback co-design for networked control systems. In *the 41st IEEE Conference on Decision and Control*, 2002.

[51] John Calandrino and James Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *ECRTS '08*.

[52] John Calandrino and James Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *ECRTS '09*.

[53] John Calandrino, James Anderson, and Dan Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *ECRTS '07*.

[54] John Calandrino, Dan Baumberger, Tong Li, Scott Hahn, and James Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *RTAS '07*.

[55] T. W. Carley, M. A. Ba, R. Barua, and D. B. Stewart. Contention-free periodic message scheduler medium access control in wireless sensor/actuator networks. In *RTSS '03*.

[56] D Chen, M Nixon, and A Mok. *WirelessHART^{TM}Real-Time Mesh Network for Industrial Automation*. Springer, 2010.

[57] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, 2008.

[58] Yixin Chen and Minmin Chen. Extended duality for nonlinear programming. *Comput. Optim. Appl.*, 47:33–59, 2010.

[59] Wei Cheng, Xiuzhen Cheng, T. Znati, Xicheng Lu, and Zexin Lu. The complexity of channel scheduling in multi-radio multi-channel wireless networks. In *INFOCOM '09*.

[60] Octav Chipara, Chenyang Lu, and Gruia-Catalin Roman. Real-time query scheduling for wireless sensor networks. In *RTSS '07*.

[61] Octav Chipara, Chenyang Lu, and John Stankovic. Dynamic conflict-free query scheduling for wireless sensor networks. In *ICNP '06*.

[62] Jeonghwan Choi, S. Govindan, B. Urgaonkar, and Anand Sivasubramaniam. Profiling, prediction, and capping of power consumption in consolidated environments. In *MASCOTS '08*.

[63] K.R. Chowdhury, P. Chanda, D.P. Agrawal, and Qing-An Zeng. Dca-a distributed channel allocation scheme for wireless sensor networks. In *PIMRC '05*, volume 2, pages 1297 –1301, sept. 2005.

[64] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Inf. Process. Lett.*, 106(5):180–187, 2008.

[65] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *SIMUTools '10*.

[66] Shi-Lu Dai, Hai Lin, and Shuzhi Sam Ge. Scheduling-and-control codesign for a collection of networked control systems with uncertain delays. *IEEE Transaction on Control Systems Tech.*, 18(1):66 –78, 2010.

[67] Arindam K. Das, Hamed M. K. Alazemi, Rajiv Vijayakumar, and Sumit Roy. Optimization models for fixed channel assignment in wireless mesh networks with multiple radios. In *SECON '05*.

[68] R I Davis and A Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *RTSS '09*.

[69] Robert Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comp. Surv.*, 43, 2011.

[70] Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47:1–40, January 2011.

[71] Xiaotie Deng, Nian Gu, Tim Brecht, and KaiCheng Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *SODA '96*.

[72] Aditya Dhananjay, Hui Zhang, Jinyang Li, and Lakshminarayanan Subramanian. Practical, distributed channel assignment and routing in dual-radio mesh networks. In *SIGCOMM '09*.

[73] Maciej Drozdowski. Real-time scheduling of linear speedup parallel tasks. *Inf. Process. Lett.*, 57(1):35–40, 1996.

[74] Jeff Edmonds, Donald D. Chinn, Timothy Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6(3):231–250, 2003.

[75] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07*.

[76] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07*, 2007.

[77] M. E. Femal and V. W. Freeh. Boosting data center performance through non-uniform power allocation. In *ICAC '05*.

[78] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. A real-time scheduling service for parallel tasks. In *RTAS'13*.

[79] Nathan Fisher, Theodore P. Baker, and Sanjoy Baruah. Algorithms for determining the demand-based load of a sporadic task system. In *RTCSA '06*.

[80] Nathan Fisher, Sanjoy Baruah, and Theodore P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *ECRTS '06*.

[81] Xing Fu, Xiaorui Wang, and Charles Lefurgy. How much power oversubscription is safe and allowed in data centers? In *ICAC '11*.

[82] M.E.M.B. Gaid, A. Cela, and Y. Hamam. Optimal integrated control and scheduling of networked control systems with communication constraints: application to a car suspension system. *IEEE Transactions on Control Systems Technology*, 14(4):776 – 787, 2006.

[83] Lin Gao and Xinbing Wang. A game approach for multi-channel allocation in multi-hop wireless networks. In *MobiHoc '08*.

[84] A. Ghosh, O. Durmaz Incel, V.S. Anil Kumar, and B. Krishnamachari. Multi-channel scheduling for fast aggregated convergecast in wireless sensor networks. In *MASS '09*.

[85] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[86] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming. `http://cvxr.com/cvx/`.

[87] Yu Gu, Tian He, Mingen Lin, and Jinhui Xu. Spatiotemporal delay control for low-duty-cycle sensor networks. In *RTSS '09*.

[88] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *RTSS '09*.

[89] M. Hajiaghayi, Min Dong, and Ben Liang. Optimal channel assignment and power allocation for dual-hop multi-channel multi-user relaying. In *INFOCOM '11*.

[90] Y. Halevi and H. Ray. Performance analysis of integrated communication and control system networks. *J. Dyn. Syst. Meas. Control*, 112:365 – 372, 1990.

[91] Hamilton. Cost of power in large-scale data centers, 2008. `http://perspectives.mvdirona.com`.

[92] C.-C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. In *RTSS '89*.

[93] Song Han, Xiuming Zhu, and Aloysius K. Mok. Reliable and real-time communication in industrial wireless mesh networks. In *RTAS '11*.

[94] Tian He, Brian M. Blum, Qing Cao, John A. Stankovic, Sang H. Son, and Tarek F. Abdelzaher. Robust and timely communication over highly dynamic sensor networks. *Real-Time Syst.*, 37(3), 2007.

[95] Huang-Ming Huang, Terry Tidwell, Christopher Gill, Chenyang Lu, Xiuyu Gao, and Shirley Dyke. Cyber-physical systems for real-time hybrid structural testing: a case study. In *ICCPS '10*.

[96] IPOPT. Interior point optimizer, 2011. `https://projects.coin-or.org/Ipopt`.

[97] Michael Isard. Autopilot: automatic data center management. *Operating Systems Review*, 41:60–67, 2007.

[98] Klaus Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, 2004.

[99] Petr Jurcík, Ricardo Severino, Anis Koubâa, Mário Alves, and Eduardo Tovar. Real-time communications over cluster-tree sensor networks with mobile sink behaviour. In *RTCSA '08*.

[100] V. Kanodia, C. Li, A. Sabharwal, B. Sadeghi, and E. Knightly. Distributed multi-hop scheduling and medium access with delay and throughput constraints. In *MobiCom '01*.

[101] Kyriakos Karenos and Vana Kalogeraki. Real-time traffic management in sensor networks. In *RTSS '06*.

[102] Kyriakos Karenos, Vana Kalogeraki, and Srikanth V. Krishnamurthy. A rate control framework for supporting multiple classes of traffic in sensor networks. In *RTSS '05*.

[103] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *RTSS '09*.

[104] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *ICCPS '13*.

[105] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN '07*.

[106] Youngmin Kim, Hyojeong Shin, and H Cha. Y-MAC: An energy-efficient multi-channel MAC protocol for dense wireless sensor networks. In *IPSN '08*.

[107] M. Kodialam and T. Nandagopal. Characterizing the capacity region in multi-radio multi-channel wireless mesh networks. In *MobiCom '05*.

[108] Vasileios Kontorinisy, Liuyi Eric Zhangy, Baris Aksanliy, and Jack Sampson. Managing distributed UPS energy for effective power capping in data centers. In *ISCA '12*.

[109] H.-J. Korber, H. Wattar, and G. Scholl. Modular wireless real-time sensor/actuator network for factory automation applications. *IEEE Trans. on Industrial Informatics*, 3(2):111–119, 2007.

[110] O Kwon and Kyung Chwa. Scheduling parallel tasks with individual deadlines. *Theor. Com. Sc.*, 215:209–223, 1999.

[111] Pradeep Kyasanur, N. Vaidya, and M. Zorzi. Capacity of multichannel wireless networks: Impact of number of channels and interfaces. In *MobiCom '05*.

[112] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS '10*.

[113] Hieu Le, Henriksson, and Tarek Abdelzaher. A practical multi-channel media access control protocol for wireless sensor networks. In *IPSN '08*.

[114] Hieu Khac Le, Dan Henriksson, and Tarek Abdelzaher. A control theory approach to throughput optimization in multi-channel collection sensor networks. In *IPSN '07*.

[115] Wan Yeon Lee and Heejo Lee. Optimal scheduling for real-time parallel tasks. *IEICE Trans. Inf. Syst.*, E89-D(6):1962–1966, 2006.

[116] Huan Li, Prashant Shenoy, and Krithi Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *RTAS '05*.

[117] Feng-Li Lian, J. Moyne, and D. Tilbury. Network design consideration for distributed control systems. *IEEE Transactions on Control Systems Technology*, 10(2):297 –307, 2002.

[118] Feng-Li Lian, J. Yook, P. Otanez, D. Tilbury, and J. Moyne. Design of sampling and transmission rates for achieving control and communication performance in networked agent systems. In *ACC '03*.

[119] Chieh-Jan Mike Liang, Jie Liu, Liqian Luo, Andreas Terzis, and Feng Zhao. RACNet: a high-fidelity data center sensing network. In *SenSys '09*, 2009.

[120] Harold Lim, Aman Kansal, and Jie Liu. Power budgeting for virtualized data centers. In *USENIXATC '11*.

[121] Xiaojun Lin and S. Rasool. A distributed joint channel-assignment, scheduling and routing algorithm for multi-channel ad-hoc wireless networks. In *INFOCOM '07*.

[122] Jane W.S. Liu. *Real-time Systems*. Prentice Hall, 2000.

[123] Ke Liu, Nael Abu-Ghazaleh, and Kyoung-Don Kang. JiTS: Just-in-time scheduling for real-time sensor data dissemination. In *PERCOM '06*.

[124] S Liu, G Xing, H Zhang, J Wang, J Huang, M Sha, and L Huang. Passive interference measurement in wireless sensor networks. In *ICNP'10*.

[125] Xiangheng Liu and A. Goldsmith. Wireless network design for distributed control. In *43rd IEEE Conference on Decision and Control*, 2004.

[126] Xiangheng Liu and Andrea J. Goldsmith. Cross-layer design of distributed control over wireless network. In *Systems and Control: Foundations and Applications, Birkhauser*, 2005.

[127] Xue Liu, Qixin Wang, Wenbo He, Marco Caccamo, and Lui Sha. Optimal real-time sampling rate assignment for wireless sensor networks. *ACM Trans. Sen. Netw.*, 2:263–295, 2006.

[128] Chenyang Lu, Brian M. Blum, Tarek F. Abdelzaher, John A. Stankovic, and Tian He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *RTAS '02*.

[129] R. Maheshwari, S. Jain, and S. R. Das. A measurement study of interference modeling and scheduling in low-power wireless networks. In *SenSys '08*.

[130] G. Manimaran, C Murthy, and Krithi Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Syst.*, 15(1), 1998.

[131] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *SenSys '04*.

[132] P. Marti, J. Yepez, M. Velasco, R. Villa, and J.M. Fuertes. Managing quality-of-control in network-based control systems by controller and message scheduling co-design. *IEEE Transactions on Industrial Electronics*, 51(6):1159 – 1167, 2004.

[133] S. Merlin, N. Vaidya, and M. Zorzi. Resource allocation in multi-radio multi-channel multi-hop wireless networks. In *INFOCOM '08*.

[134] Arunesh Mishra, Vivek Shrivastava, Dheeraj Agrawal, Suman Banerjee, and Samrat Ganguly. Distributed channel management in uncoordinated wireless environments. In *MobiCom '06*.

[135] Anjum Naveed, Salil S. Kanhere, and Sanjay K. Jha. Topology control and channel assignment in multi-radio multi-channel wireless mesh networks. In *MASS '07*.

[136] Geoffrey Nelissen, Vandy Berten, J Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *ECRTS '12*.

[137] S. Pediaditaki, P. Arrieta, and M.K. Marina. A learning-based approach for distributed multi-radio channel allocation in wireless mesh networks. In *ICNP '09*.

[138] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. Power routing: Dynamic power provisioning in the data center. In *ASPLOS '10*.

[139] N. Pereira, B. Andersson, E. Tovar, and A. Rowe. Static-priority scheduling over wireless networks with multiple broadcast domains. In *RTSS '07*.

[140] Joonas Pesonen, Haibo Zhang, Pablo Soldati, and Mikael Johansson. Methodology and tools for controller-networking co-design in WirelessHART. In *EFTA '09*.

[141] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *STOC '97: Proceedings of the 29th annual ACM symposium on Theory of Computing*, pages 140–149, 1997.

[142] B.T. Polyak. *Introduction to Optimization*. 1987.

[143] Constantine Polychronopoulos and David Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Comp.*, C-36(12):1425–1439, 1987.

[144] Python. Python simulated annealing module, 2009. `http://www-personal.umich.edu/~wagnerr/PythonAnneal.html`.

[145] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No power struggles: coordinated multi-level power management for the data center. In *ASPLOS '08*.

[146] K. N. Ramachandran, E. M. Belding, K. C. Almeroth, and M. M. Buddhikot. Interference-aware channel assignment in multi-radio wireless mesh networks. In *INFOCOM '06*.

[147] Bhaskaran Raman. Channel allocation in 802.11-based mesh networks. In *INFOCOM '06*.

[148] Parthasarathy Ranganathan, Phil Leech, David Irwin, Jeffrey Chase, and Hewlett Packard. Ensemble-level power management for dense blade servers. In *ISCA '06*.

[149] A. Raniwala and T. Chiueh. Architecture and algorithm for an ieee 802.11-based multi-channel wireless mesh network. In *INFOCOM '05*.

[150] Ashish Raniwala and Tzi-cker Chiueh. Architecture and algorithms for an ieee 802.11-based multi-radio wireless mesh networks. In *INFOCOM '05*.

[151] Ashish Raniwala, Kartik Gopalan, and Tzi-cker Chiueh. Centralized channel assignment and routing algorithms for multi-channel wireless mesh networks. *Mob. Comput. Commun. Rev.*, 2004.

[152] Injong Rhee, Ajit Warrier, Mahesh Aia, and Jeongki Min. Z-MAC: a hybrid MAC for wireless sensor networks. In *SenSys '05*, 2005.

[153] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *RTSS '11*.

[154] Abusayeed Saifullah, Chengjie Wu, Paras Tiwari, You Xu, Yong Fu, Chenyang Lu, and Yixin Chen. Near optimal rate selection for wireless control systems. In *RTAS '12*.

[155] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. End-to-end delay analysis for fixed priority scheduling in WirelessHART networks. In *RTAS '11*.

[156] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. Priority assignment for real-time flows in WirelessHART networks. In *ECRTS '11*.

[157] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. Real-time scheduling for WirelessHART networks. In *RTSS '10*.

[158] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. End-to-end communication delay analysis in industrial wireless networks. *IEEE Transactions on Computers*, 2012.

[159] Jens B. Schmitt and Utz Roedig. Sensor network calculus - A framework for worst case analysis. In *DCOSS '05*.

[160] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *RTSS '96*.

[161] L. Sha, X. Liu, M. Caccamo, and G. Buttazzo. Online control optimization using load driven scheduling. In *CDC '00*.

[162] Minho Shin, Seungjoon Lee, and Yoo ah Kim. Distributed channel assignment for multi-radio wireless networks. In *MASS '06*.

[163] Weihuan Shu, Xue Liu, Zonghua Gu, and Sathish Gopalakrishnan. Optimal sampling rate assignment with dynamic route selection for real-time wireless sensor networks. In *RTSS '08*.

[164] Jungmin So and Nitin H. Vaidya. Multi-channel MAC for ad hoc networks: handling multi-channel hidden terminals using a single transceiver. In *MobiHoc '04*.

[165] Pablo Soldati, Haibo Zhang, and Mikael Johansson. Deadline-constrained transmission scheduling and data evacuation in WirelessHART networks. In *ECC '09*.

[166] J. Song, A. K. Mok, D. Chen, and M. Nixon. Challenges of wireless control in process industry. In *Workshop on Research Directions for Security and Net. in Critical Real-Time and Embed. Sys.*, 2006.

[167] Jianping Song, Song Han, A.K. Mok, Deji Chen, M. Lucas, and M. Nixon. Wirelesshart: Applying wireless technology in real-time industrial process control. In *RTAS '08: IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 377–386, April 2008.

[168] K Srinivasan and P Levis. RSSI is under appreciated. In *EmNets '06*.

[169] K Srinivasan and P Levis. RSSI is under appreciated. In *EmNets '06*.

[170] J.A. Stankovic, T.E. Abdelzaher, Chenyang Lu, Lui Sha, and J.C. Hou. Real-time communication and coordination in embedded sensor networks. *Proceedings of the IEEE*, 91(7):1002–1022, 2003.

[171] A. Tzamaloukas and J. G.-Luna-Aceves. A receiver-initiated collision-avoidance protocol for multi-channel networks. In *INFOCOM '01*.

[172] Ramanuja Vedantham, Sandeep Kakumanu, Sriram Lakshmanan, and Raghupathy Sivakumar. Component based channel assignment in single radio, multi-channel ad hoc networks. In *MobiCom '06*.

[173] J. Wang, Y. Fang, and D. Wu. A power-saving multi-radio multi-channel mac protocol for wireless local area networks. In *INFOCOM '06*.

[174] Qingzhou Wang and Kam Hoi Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM J. Comput.*, 21(2), 1992.

[175] Xiaodong Wang, Xiaorui Wang, X Fu, G Xing, and N Jha. Flow-based real-time communication in multi-channel wireless sensor networks. In *EWSN '09*.

[176] Xiaodong Wang, Xiaorui Wang, Xing Fu, Guoliang Xing, and Nitish Jha. Flow-based real-time communication in multichannel wireless sensor networks. In *EWSN '09*.

[177] Xiaorui Wang, Ming Chen, C. Lefurgy, and T.W. Keller. SHIP: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Transactions on Parallel and Distributed Systems*, 23, 2012.

[178] Z Wang, C McCarthy, X. Zhu, P. Ranganathan, and V Talwar. Feedback control algorithm for power management of servers. In *ASPLOS '08*.

[179] Chengjie Wu, Mo Sha, Dolvara Gunatalika, Abusayeed Saifullah, Chenyang Lu, and Yixin Chen. Analysis of EDF scheduling for wireless sensor-actuator networks. In *IWQoS 14*.

[180] D. Wu and P. Mohapatra. From theory to practice: Evaluating static channel assignments on a wireless mesh network. In *INFOCOM '10*.

[181] Yafeng Wu, J.A. Stankovic, He, and Lin. Realistic and efficient multi-channel communications in wireless sensor networks. In *INFOCOM '08*.

[182] Feng Xia and Youxian Sun. Control-scheduling codesign: A perspective on integrating control and computing. *Dynamics of Cont., Discr. and Impulsive Syst.*, 13:1352–1358, 2008.

[183] Lin Xiao, Mikael Johansson, Haitham Hindi, Stephen Boyd, and Andrea Goldsmith. Joint optimization of wireless communication and networked control systems. *Lecture Notes in Computer Science*, 3355:248–272.

[184] Guoliang Xing, Mo Sha, Jun Huang, Gang Zhou, Xiaorui Wang, and Shucheng Liu. Multi-channel interference measurement and modeling in low-power wireless networks. In *RTSS '09*.

[185] Kai Xing, Xiuzhen Cheng, Liran Ma, and Qilian Liang. Superimposed code based channel assignment in multi-radio multi-channel wireless mesh networks. In *MobiCom '07*.

[186] Dejun Yang, Xi Fang, and Guoliang Xue. Channel allocation in non-cooperative multi-radio multi-channel wireless networks. In *INFOCOM '12*.

[187] Qing Yu, Jiming Chen, Yanfei Fan, Xuemin Shen, and Youxian Sun. Multi-channel assignment in wireless sensor networks: A game theoretic approach. In *INFOCOM '10*.

[188] Haibo Zhang, Fredrik Osterlind, Pablo Soldati, Thiemo Voigt, and Mikael Johansson. Rapid convergecast on commodity hardware: Performance limits and optimal policies. In *SECON '10*.

[189] Haibo Zhang, Pablo Soldati, and Mikael Johansson. Optimal link scheduling and channel assignment for convergecast in linear WirelessHART networks. In *WiOpt '09*.

[190] Lei Zhang and Dimitrios Hristu-Varsakelis. Communication and control co-design for networked control systems. *Automatica*, 42(6):953 – 958, 2006.

[191] Yanwei Zhang, Yefu Wang, and Xiaorui Wang. Capping the electricity cost of cloud-scale data centers with impacts on power markets. In *HPDC '11*.

[192] G. Zhou, T. He, J. A. Stankovic, and T. F. Abdelzaher. RID: radio interference detection in wireless sensor networks. In *INFOCOM '05*.

[193] G. Zhou, C. Huang, T. Yan, T. He, J. A. Stankovic, and T. F. Abdelzaher. MMSN: Multi-frequency media access control for wireless sensor networks. In *INFOCOM '06*.

[194] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror mirror on the ceiling: flexible wireless links for data centers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 443–454, New York, NY, USA, 2012. ACM.

[195] Xiuming Zhu, Pei-Chi Huang, Song Han, A.K. Mok, Deji Chen, and M. Nixon. RoamingHART: A collaborative localization system on WirelessHART. In *RTAS '12*.