Deterministic Scheduling of Real-Time Tasks on Heterogeneous Multicore Platforms

Waqar Ali

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Committee members

Dr. Heechul Yun, Chairperson

Dr. Prasad Kulkarni

Dr. Esam Eldin Mohamed Aly

Dr. Drew Davidson

Dr. Shawn Keshmiri

Date defended: December 15, 2020

The Program Committee for Waqar Ali certifies
that this is the approved version of the following dissertation :

Deterministic Scheduling of Real-Time Tasks on Heterogeneous Multicore Platforms

Dr. Heechul Yun, Chairperson

Date approved: December 15, 2020

Abstract

In recent years, the problem of real-time scheduling has increasingly become more important as well as more complicated. The former is due to the proliferation of safety critical systems into our day-to-day life; such as autonomous vehicles, fueled by the recent advances in artificial intelligence. The latter is caused by the increasing demand for high performance which is driving the adoption of highly integrated complex heterogeneous system-on-chip (SoC) processors to deliver the performance while meeting strict size, weight, power (SWaP) and cost constraints. Motivated by these trends, this dissertation tackles the following main question: how can we guarantee predictable real-time execution on heterogeneous multicore SoCs while preserving high utilization?

The fundamental problem in preserving the determinism of the real-time system realized on a heterogeneous multicore SoC is ensuring that the worst-case execution time (WCET) of each task, measured in isolation, will stay within a reasonable bound during the actual execution of the system. The primary challenge in achieving this goal—tightly bounding task WCETs—is that the execution time of a task can be highly non-deterministic, often varying significantly depending on which tasks are co-scheduled and how they contend on various shared hardware resources in the memory hierarchy. The particular scheduling requirements (e.g., non-preemption) of the different computing resources (e.g., integrated GPU) in the heterogeneous SoC and the possible cross-contention among their workloads can also exacerbate this problem.

In light of these considerations, this dissertation presents new real-time scheduling techniques for predictable and efficient scheduling of mixed criticality workloads on heterogeneous SoCs. The contributions of this dissertation include the following: 1) A novel CPU-GPU scheduling framework that ensures predictable execution of critical GPU kernels on integrated CPU-GPU platforms. 2) A novel gang scheduling framework which guarantees deterministic execution of parallel real-time tasks on the multicore CPU cluster of a heterogeneous SoC. 3) Optimal and heuristic algo-

rithms for gang formation that increase real-time schedulability under the RT-Gang framework and their extension to incorporate scheduling on accelerators in a heterogeneous SoC. 4) Concrete evaluation results using simulated tasksets as well as real-world workloads that demonstrate the analytical and practical benefits of the proposed techniques.

To my parents, my wife and my daughter

Acknowledgements

I am thankful to a number of people for helping me get through the doctoral program at the University of Kansas. First and foremost, I want to thank my mentor Dr. Heechul Yun for diligently advising me throughout the course of my PhD program. I would not have reached this point in my pursuit of knowledge—had it not been for your guidance. I also want to sincerely thank the members of my Ph.D. program committee; Dr. Prasad Kulkarni, Dr. Drew Davidson, Dr. Esam Eldin Mohamed Aly and Dr. Shawn Keshmiri, for bearing with me in the scheduling of my Ph.D. exams in these trying times. Special thanks are due to the EECS graduate program coordinator Joy Grisafe-Gross and her predecessor Pam Shadoin for always being there for me whenever I needed guidance with respect to a particular milestone of the KU doctoral program.

I would like to express my thanks and appreciation to all the current and former students, Farzad Farshchi, Prathap Kumar Valsan, Prasanth Vivekanandan Veerapan Chattir, Jacob Michael Fustos, Michael Garrett Bechtel and Ahmet Soyyigit, in the Computer Systems Lab (CSL) at ITTC, KU. Specially, I want to thank Michael for first creating the DeepPicar test-bed and then meticulously conducting the RT-Gang case-study with DeepPicar that I have used extensively in my work, as well as in the writing of this dissertation. Also, I want to thank Ahmet for taking up my work with the RT-Gang framework and keeping it going into the future. I hope that this work is fruitful to you as it has been for me.

I am specially indebted to Dr. Rodolfo Pellizzoni, from University of Waterloo, for helping me with a particularly difficult problem in my research. Without your input,

a major portion of this dissertation would be missing or would not have reached the level of quality that it has in its current form.

Finally, I am indubitably thankful to my wife, for following me all the way around the world and for never doubting me through the ups and downs of my student life. This would not have been possible without you.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

With the recent advances in the field of artificial intelligence, a new class of real-time applications has emerged which simultaneously demands high performance as well as high safety, predictability, and determinism. Examples of such applications, among others, include autonomous driving cars, unmanned aerial vehicles and robotics. To meet the performance demand of these applications, modern *commercial-of-the-shelf* (COTS) multicore platforms integrate a variety of computing resources in a small form factor system-on-a-chip (SoC). Hence, these days, it is common to come across embedded computing platforms which, in addition to the multicore CPU cluster, contain one or more GPUs as well as special deep learning accelerators and FPGAs. An example of such a platform is NVIDIA's Jetson TX2 [1] board which contains a multicore CPU cluster and an integrated GPU in a single SoC. Due to the varied nature of the computing resources present in such platforms, they are termed as heterogeneous computing platforms and their architecture is called heterogeneous system architecture (HSA).

Although heterogeneous computing platforms offer plenty of raw performance, guaranteeing predictable execution timing of real-time tasks on these platforms is extremely challenging. The primary goal of this dissertation is to investigate the nature of the aforementioned challenges and propose a practically viable solution that can enable efficient usage of these platforms in safety critical real-time applications. We begin by briefly describing the aforementioned challenges in the following.

1.1 The Performance Isolation Challenge

The performance isolation challenge, in using heterogeneous multicore platforms for real-time use-cases, refers to the difficulty in ensuring that the runtime performance of a real-time task will not get affected—or the effect will be within a measurable bound—by the execution of *corunning* tasks in the system. Due to the size, weight, power (SWaP) and cost constraints inherent from their target use-cases, heterogeneous computing platforms contain a memory subsystem that is commonly shared among all the computing elements integrated into the SoC. However, the memory subsystem is often optimized for average performance and in the worst-case, can exhibit extremely poor bottleneck behaviors [2, 3, 4]. For this reason, the runtime performance of applications which are executing simultaneously in such a system, is dependent on how they collectively make use of the shared memory subsystem. This makes it difficult to reliably, and without excessive pessimism, determine the worst-case execution time of the real-time applications which make use of a heterogeneous computing platform; that is instrumental in performing a schedulability test of the system as explained below.

For a critical real-time system, determining the worst-case execution time (WCET) of critical tasks is important and often required for certification [5, 6]. In unicore based systems, the standard method is a two-step approach: (1) obtain the WCET of each task independently from the rest of the system either by using static analysis tools or experimental measurements; (2) perform schedulability analysis based on the obtained WCETs. Applying this approach to heterogeneous multicore SoCs, however, is problematic because of the interference among corunning tasks in the shared memory hierarchy, as mentioned above.

To tackle this problem without requiring modifications to the underlying hardware architecture of the COTS multicore platform, a common approach adopted in a plethora of real-time literature is to partition the most common shared resources in the memory subsystem, such as the last level cache (LLC), the main memory bandwidth and the DRAM etc. However, there is a fundamental limitation of the partitioning based solutions; not all shared resources can be partitioned through software only approaches. Due to intellectual property related issues, the vendors of the COTS

multicore platforms do not disclose all the implementation details of their respective platforms. As a result, the users of these platforms have to treat most of the memory subsystem as a black-box. To safely use such a platform in realizing hard real-time systems, it is imperative to consider all the possible ways in which the tasks in the system can interfere with each other through the shared memory subsystem. This makes the execution of the real-time tasks coupled with each other through the scheduling policy and the underlying hardware architecture of the heterogeneous multicore platform. This coupling can necessitate extremely pessimistic estimation of the interference aware WCET, e.g., up-to 300x of the solo WCET [4], of real-time tasks which can, in turn, severely reduce the overall schedulability of the system. For this reason, in the use-cases where hard real-time guarantees are a must (e.g., avionics), it is recommended to disable all but one core of a multicore processor [6], which obviously defeats the purpose of using a multicore platform in the first place.

1.2 The Nuances of Heterogeneous Computing

In a heterogeneous computing platform, the presence of an accelerator such as an integrated GPU introduces a new facet of complications to the real-time scheduling problem. First of all, the method of using the accelerator can be different from the well-understood CPU usage model. For example, in the case of traditional GPU scheduling, an application running on the CPU acts as a master and drives all the computations on the GPU by copying data between CPU and GPU memories (in the case of discrete GPUs) and triggering computations on the GPU as needed. Moreover, due to the often data driven nature of the computation on the accelerators, the computation may have to be performed non-preemptively; as opposed to the conventional CPU scheduling in which a high priority task, barring any locking protocol [7] related requirements, can preempt a low priority task at any time, to take control of the CPU. These nuances necessitate the use of specialized task models which are usually significantly more complicated than the historical periodic task model for CPU only tasksets[1].

[1] An overview of relevant real-time task models is provided in Section 2.

The presence of an accelerator also complicates the performance isolation challenge which is already quite difficult to tackle. In a typical heterogeneous platform, the main memory bus and the main memory itself is shared among all the computing resources i.e., multicore CPU cluster and the accelerators. Due to the limited bandwidth and capacity of the main memory, it becomes possible for workloads running on accelerators to contend for the main memory with those running on the CPU cluster and vice-versa e.g., our work [8] shows that real-time workloads running on the accelerator (i.e., integrated GPU) in NVIDIA's Jetson TX2 platform can suffer up-to 3.3x slowdown due to co-executing memory intensive CPU tasks. However, ensuring performance isolation in this case is even more difficult (as compared to the multicore CPU only case) because of the particular scheduling requirements of different accelerators (e.g., non-preemption) may make it difficult or even impossible to apply traditional performance isolation techniques to solve this problem. We discuss the accelerator scheduling problem in depth in Sec. 6.2 of this dissertation.

1.3 Thesis Statement

Motivated by the aforementioned discussion, we present the following thesis statement which forms the basis of this dissertation:

> *The problem of deterministic scheduling of real-time tasks on heterogeneous computing platforms demands a novel scheduling framework which can eliminate shared resource interference by design. Such a framework must also allow deterministic usage of the accelerators, such as the integrated GPU, in the heterogeneous computing platforms and must be accompanied with an applicable task model, system model and an easy to use schedulability test that can cater to the diverse range of real-time applications that can run on these platforms. Moreover, to be of practical use, the implementation of such a framework must be light-weight and easily portable to the commodity operating systems.*

1.4 Contributions

In the following, we summarize the main contributions of this research.

1.4.1 Safe Real-Time Execution on the Integrated GPU

We present the **BWLOCK++** framework which provides a mechanism to automatically protect real-time GPU kernels in a heterogeneous SoC while minimizing the throughput impact to CPU tasks. This is done by adopting a restrictive scheduling scheme which allows execution of real-time tasks only on a single core in the multicore CPU cluster and using a bandwidth throttling technique to limit the interference from best-effort CPU tasks to the real-time GPU kernel. We make the following contributions in this work: 1) We apply memory bandwidth throttling technique to the problem of protecting GPU accelerated real-time tasks from memory intensive CPU tasks on integrated CPU-GPU architecture. 2) We identify a negative feedback effect of memory bandwidth throttling when used with Linux's CFS [9] scheduler. We propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS), to mitigate the problem. 3) We introduce an automatic GPU kernel instrumentation method that eliminates the need of manual programmer intervention to protect GPU kernels. 4) We implement the proposed framework on a real platform, NVIDIA Jetson TX2, and present detailed evaluation results showing practical benefits of the framework. 5) We show how the proposed framework can be integrated into the existing CPU focused real-time schedulability analysis framework.

1.4.2 Deterministic Real-Time Task Scheduling on Multicore CPUs

We present **RT-Gang**: a novel real-time gang scheduling framework that enforces a *one-gang-at-a-time* policy. Our goal, in designing RT-Gang, is to enable analyzable and practical parallel real-time task scheduling on multicore platforms. We make the following contributions in this work: 1) A novel gang scheduling algorithm which enables predictable execution on multicore platforms while also providing simpler analysis. 2) Integration of memory bandwidth throttling technique into the

gang scheduler to allow safe co-execution of best-effort CPU tasks. 3) Implementation of our framework on top of the commodity Linux kernel and thorough evaluation on two representative embedded multicore platforms that shows dramatic improvement in the overall predictability of the system under RT-Gang and very low over-head.

1.4.3 Virtual Gang Scheduling of Parallel Real-Time Tasks

We present the **RTG-Sync** framework which allows static grouping of real-time tasks into discrete scheduling entities—called virtual gangs—for scheduling under the one-gang-at-a-time policy of RT-Gang. Our goal, in this case, is to improve the real-time schedulability under the RT-Gang framework when real-time tasks are not perfectly parallelized while preserving the analysis simplicity and runtime determinism proffered by RT-Gang. We make the following contributions in this work: 1) We consider the scheduling of real-time gang tasks with precedence constraints and present the virtual gang abstraction to group certain tasks for co-execution as discrete schedulable units. 2) We present optimal and greedy algorithms for forming virtual gangs from a set of real-time tasks and show how to perform schedulability analysis of a set of virtual gangs using unicore response time analysis. 3) We conduct thorough schedulability study using simulated tasksets and compare the results under our approach against state-of-the-art multicore real-time task scheduling techniques. The results show significant improvement in schedulability under our approach as compared to the competition. 4) We further investigate the requirements of supporting the virtual gang abstraction in a practical system and show that *synchronous release* of the virtual gang members is necessary to gain practical benefits from virtual gangs. 5) We extend the implementation of the RT-Gang framework to support the virtual gang abstraction and create a middleware framework to enforce the *synchronous release* of the virtual gang member tasks. 6) We evaluate the extended RT-Gang framework—called RTG-Sync—on the Jetson TX2 platform with a realistic case-study and demonstrate the practical benefits of our approach. Our work is the first one which enables schedulability analysis of real-time gang tasks that are bound by precedence constraints.

1.5 Organization

The rest of this dissertation is organized as follows. We present necessary background and a summary of related prior work in Chapter 2. We describe the BWLOCK++ framework and its design, implementation and evaluation results in Chapter 3. In Chapter 4, we present the RT-Gang framework and demonstrate, with an illustrative example as well as a real-world case-study using an autonomous driving test-bed, the benefits of the one-gang-at-a-time scheduling policy in improving the determinism of a real-time system realized on a multicore CPU based platform. We also concretely describe the implementation details of the RT-Gang framework inside the Linux kernel and present the evaluation results on two representative embedded multicore platforms. We begin Chapter 5 by elucidating the need of virtual gang scheduling for improving the real-time schedulability under the RT-Gang framework. We then expressly discuss the practical requirements of virtual gang scheduling with illustrative examples and describe the design of RTG-Sync framework and how it fulfills the said requirements. We also present the virtual gang formation algorithms and present analytical schedulability results from a simulation study as well as empirical evaluation results showcasing the benefits of the virtual gang scheduling in improving the schedulability of an RT-Gang managed real-time system. We provide a summary of our contributions, discuss the limitations of our techniques, the extensions we have already implemented to ameliorate some of those limitations and the possible directions of future research based on our work in Chapter 6. Finally, we conclude our discussion in Chapter 7.

Chapter 2

Background and Prior Work

In this chapter, we present prior work, in real-time scheduling theory and its practical applications, performance isolation and real-time scheduling on GPU.

2.1 Real-Time Task Models

Liu and Layland's Recurrent Task Model: It can be said that the modern research into real-time systems began with Liu and Layland's seminal work [10] in which, among other things, they presented the recurrent real-time task model. As per this model, a real-time task (τ) comprises an infinite sequence of jobs ($J_1, J_2, \cdots, J_i, \cdots$), each of which is characterized by the time it takes to complete its execution on a target hardware platform. The task itself is characterized by its maximum execution time C, period T and deadline D. The maximum execution time, also called the worst-case execution time (WCET), is equal to the longest duration any job of the task can take to complete its execution on a target hardware platform. The value of the period describes the separation between any two consecutive jobs of the task and the deadline quantifies the time interval, measured from the instant when a job arrives, during which the job should finish its execution; for the task to be deemed schedulable. The task is considered a *hard real-time* task if it must always meet its deadline; for the timing correctness of the system. If the task can tolerate the deadline misses to some extent, then it is called a *soft real-time* task. In this dissertation, when we refer to a real-time task, we mean a hard real-time task unless explicitly stated otherwise.

In Liu and Layland's original proposal, the value of the period is fixed, in the sense that the jobs of the real-time task are released exactly T time-units apart and the resulting task model is

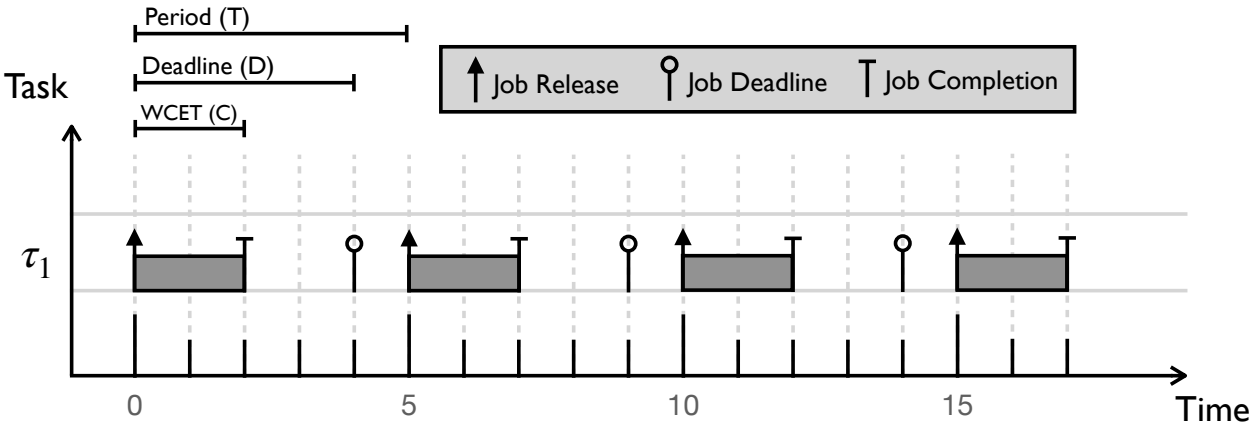


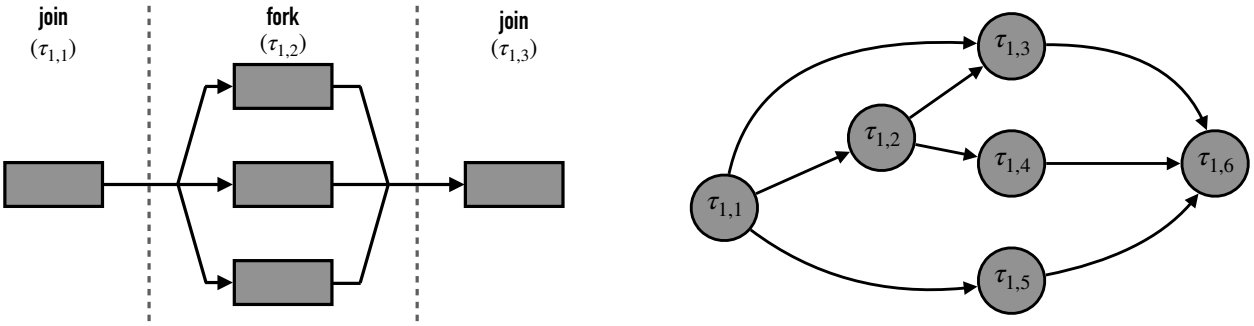
Figure 2.1: Activation diagram of a periodic real-time task

called the *periodic task model*. In a later relaxation of this model, the value T delineates the minimum separation between any two jobs of the task and the task model is called the *sporadic task model* [11]. Moreover, based on the relationship between the period and the deadline, the task model is further classified as *implicit deadline* or *constrained deadline*. In the former, for each job of the task, the deadline is equal to the period whereas for the latter, the deadline must be less than or equal to the period value. An activation diagram showing a prototypical constrained deadline periodic real-time task, with a WCET of 2, a deadline of 4 and a period of 5, is shown in Figure 2.1.

Despite several decades that have passed since the time it was presented, Liu and Layland's work is still relevant today and almost every major work in real-time scheduling theory makes use of their presented task model, in one form or another, based on some specific use-case. In that sense, their work is a canon in the subject area.

2.1.1 Task Models for Multicore Systems

The classical periodic task model applies to systems that contain a single processing element e.g., a unicore CPU based platform. By virtue of that, in such a system, each job of the real-time task must execute entirely on the single processing entity. In a modern embedded computing system, there can be multiple processing elements, such as a multicore CPU cluster, an integrated GPU and additional purpose built accelerators, all integrated into a compact System-on-a-Chip (SoC)



(a) A fork-join task

(b) A DAG task

Figure 2.2: Parallel real-time task models

configuration. Consequently, real-time applications which make use of these platforms can have diverse computing requirements which cannot be expressed adequately using the classical periodic task model. In this section, we discuss three representative task models, from real-time scheduling literature, that can be used to express real-time workloads that make use of a multicore CPU based computing platforms. Importantly, we describe the real-time gang task model in detail which is used extensively throughout this dissertation.

2.1.1.1 The Fork-Join Model

In the fork-join model of parallel real-time tasks [12, 13, 14, 15, 16], the execution of each job of a real-time task consists of a sequence of serial and parallel execution phases. In the serial (*join*) phase, the job requires only a single CPU core to execute whereas in the parallel (*fork*) phase, the job can execute simultaneously on multiple CPU cores. The different phases of execution of a fork-join task have a linear precedence relationship among them: phase (i+1) can begin execution only after phase (i) has completed. Figure 2.2a shows an illustration of a task that can be expressed using the fork-join model. Despite its simplicity, the fork-join model is quite popular and supported by a number of parallel programming frameworks [17, 18, 19].

2.1.1.2 The DAG Model

In the DAG model [20, 21, 13], each job of the periodic real-time task is represented as a directed acyclic graph (DAG). The nodes of the DAG represent the execution phases of the job and the edges represent the precedence constraint relationship among the execution phases. Each execution phase is itself sequential i.e., it executes entirely on a single CPU core. Figure 2.2b shows an illustration of a task that can be expressed using the DAG model. In essence, the DAG model is a generalization of the fork-join model and it is suitable to express real-time workloads which have more complicated relationship among the phases of execution that cannot be expressed using the linear scheme of the fork-join model.

2.1.1.3 The Gang Model

In the gang model of parallel real-time tasks [22, 23, 24, 25], each job of the task is characterized by the number of cores m it needs to execute; in addition to its execution demand c and the period p value as per the classical periodic task model. Based on the nature of the core requirement parameter m, the gang task model is further divided into a number of categories which are described in the following.

Rigid Gang: In the rigid gang model, the number of cores required to execute the gang are determined off-line and they stay the same for all of the jobs of the gang throughout its execution. Figure 2.3 shows the activation diagram of an implicit deadline periodic rigid gang task.

Moldable Gang: If the number of cores required to execute a gang are determined on-line, on a per job basis, by the scheduler but once determined, the core requirement stays the same throughout the job's execution, then the resulting model is called the moldable gang model.

Malleable Gang: If the number of cores required to execute the gang can change during the execution of the job, then such a gang task model is called the malleable gang model.

Bundled Gang: The bundled gang model, introduced recently [25], is a generalization of the rigid gang model. As per this model, the execution of each job of a rigid gang task is sub-divided into

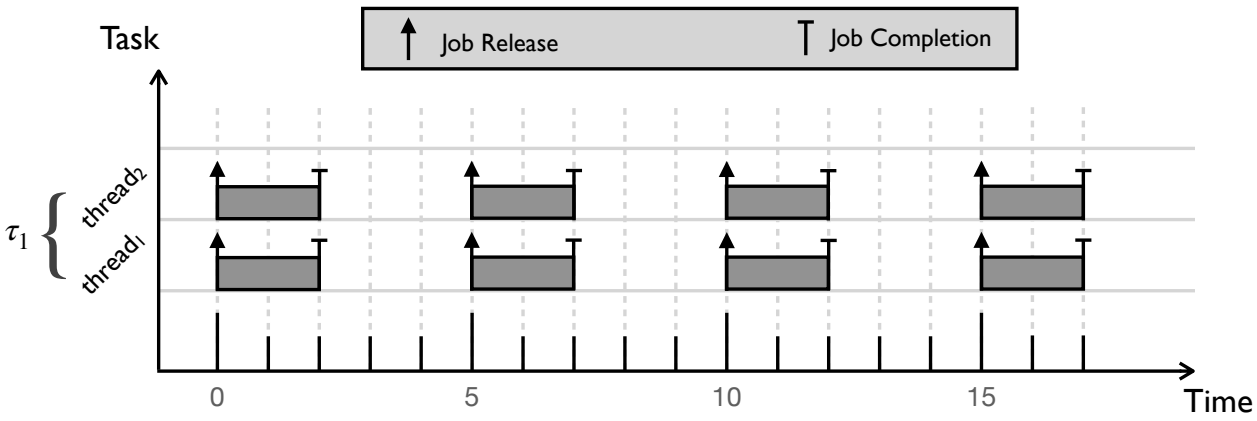


Figure 2.3: Activation diagram of an implicit deadline rigid gang task

multiple execution segments—or bundles—each of which can have a distinct core requirement, determined off-line, that can be different from one bundle to another. The bundles are bound by a linear precedence requirement i.e., bundle (i+1) cannot start before bundle (i) has finished its execution.

For the purpose of this dissertation, the rigid gang model is most relevant and will be discussed in more detail, in the context of real-time gang scheduling, in Chapter 4 and Chapter 5.

2.2 Response Time Analysis

In real-time scheduling, it is of utmost importance to know *a-priori* whether a real-time task will meet its deadlines in a particular use-case. A large body of literature in real-time scheduling theory revolves around devising offline analytical tests that, given a set of tasks parameterized as per a suitable task model, can be used to determine whether each task in the taskset will be able to meet its deadline. The process of analytically determining the schedulability of a real-time taskset is called *response-time analysis* [26, 27, 28]; for the reason that the tests involved in this procedure determine the response-time of the real-time tasks which is defined as the time interval since the arrival of a job of a task to the time that the job completes its execution. Integral to the response-time analysis is the scheduling policy of the target system which determines which task(s) to execute on the available computing resources from a group of tasks that are ready to

execute, based on the notion of *priority*. In the following, we discuss most widely used priority based scheduling policies from real-time scheduling literature that are relevant to the discussion presented in the later chapters of this dissertation.

2.2.1 Priority based Scheduling Policies

Priority based scheduling of real-time tasks can be broadly divided into two categories, based on how the priority of each job of a given task is determined. In fixed priority scheduling, the priority of each job of the task is statically determined offline and does not change throughout the task's execution. An example of fixed priority scheduling is *rate-monotonic* (RM) policy [28], in which, a task with a smaller period is given a higher static priority value. In contrast, in dynamic priority scheduling, the priority of each job of the task is determined online dynamically before the job starts executing. An example of such scheduling scheme is *earliest deadline* first (EDF) policy [10] in which, as the name implies, a job with the closest deadline is given the highest scheduling priority.

In comparing different policies for scheduling real-time tasks, the notion of *feasibility* and *optimality* are of vital importance. A taskset is said to be feasible under a specific scheduling policy if it can be analytically proven that all tasks in the taskset will always be able to meet their deadlines if scheduled according to that policy. A scheduling policy, for a specific priority assignment scheme, is considered optimal if it can schedule all feasible tasksets under that scheme. An optimal scheduling scheme is highly desirable in scheduling hard real-time tasks. For single-core based platforms, it has been proven that rate-monotonic policy is optimal for fixed priority scheduling and EDF policy is optimal for dynamic priority scheduling [10]. For platforms with multiple cores, the problem is considerably more complicated as explained in the next section.

2.2.2 The Multicore Scheduling Problem

In a multicore system, a significant source of complication to the real-time task scheduling problem arises because of the presence of multiple computing cores that can be used to run a ready task.

Moreover, the task models used to describe multicore workloads can be considerably more intricate and hence conceptually difficult to handle than the classical sequential periodic tasks, as described in the earlier sections of this chapter. Based on the scope of problem, the multicore scheduling is divided into three types; each of which is briefly discussed in the following.

2.2.2.1 Global Scheduling

In global scheduling [29, 15, 30], real-time tasks are opportunistically scheduled on the first available core in a multicore platform. If no such core is available, then the lowest priority task running in the platform is preempted in favor of the highest priority ready task, if the priority of the latter is greater than the former. From implementation point of view, global scheduling requires a single ready-queue to keep all the ready tasks in the system and it can select tasks from the ready-queue based on their assigned priority value; as per a fixed priority or dynamic priority assignment scheme. Although conceptually simple, global scheduling is usually avoided in commodity operating systems due to the runtime overheads e.g., cache management and synchronization, associated with this policy in multicore platforms.

2.2.2.2 Partitioned Scheduling

Contrary to the global scheduling paradigm, in partitioned scheme [31, 32], the scheduling problem is divided into two parts. In the first (offline) part, the tasks are statically assigned to the different cores available in the target hardware platform. In the second (online) part, each core in the target platform executes the task present in its ready-queue using an appropriate priority assignment scheme, just like a unicore system. From the implementation and analysis point of view, partitioned scheduling is straight forward because in a partition scheduled system, each core can operate locally on the tasks in its ready-queue without considering the other cores and the schedulability of the tasks in a core's ready-queue can be determined using well-known unicore response-time analysis techniques. However, partitioned scheduling has its own caveats due to its offline task-to-core mapping phase which requires solving a bin-packing like problem that is

known to be NP-hard in the strong sense [33].

2.2.2.3 Clustered Scheduling

This scheme is a generalization of the global and partitioned scheduling policies. In clustered scheduling [34, 35], the multicore platform is divided into clusters of computing cores. Tasks are statically partitioned among the clusters and inside the clusters, the assigned tasks are scheduled using the global policy. Based on the size of the clusters, this scheduling policy transforms into partitioned scheduling if the number of clusters is equal to the number of cores in the target platform and it becomes equivalent to global scheduling when there is only one cluster containing all the cores.

Each of the scheduling policy described above, when coupled with an appropriate task model and a priority assignment scheme, has a known response-time analysis formula for determining the schedulability of a given set of real-time tasks and each such combination has an associated set of trade-offs which make it more suitable for a particular real-time scheduling use-case than others. *Common to all these scheduling schemes, is the fundamental characteristic that any real-time task can conceptually get co-scheduled with any other real-time task present in the system; there is no inherent restriction that limits the possible co-scheduling of particular real-time tasks across different cores of a multicore platform.* In the following section, we will elaborate why this can be problematic and from there, we will motivate the need for a novel *interference-aware* scheduling policy that is one of the main contributions of this dissertation.

2.3 Performance Isolation in Multicore Platforms

The type of multicore platforms that we focus on in this dissertation can be characterized as shared-memory based multicore platforms, due to the reason that in the architecture of these platforms, the memory hierarchy is shared to varying extent among all the computing resources. An example

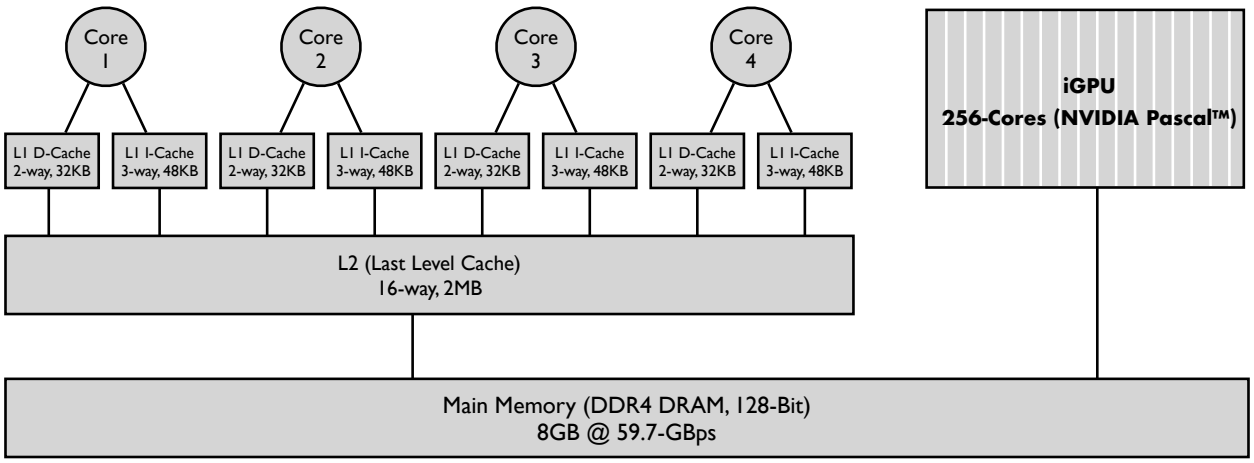


Figure 2.4: Illustration of the shared memory subsystem in NVIDIA's Jetson TX-2 heterogeneous computing board

of such a platform is NVIDIA's Jetson TX-2[1] board [1], which contains a quad-core CPU complex and an integrated GPU, as shown in Figure 2.4. Due to the size, weight, power (SWaP) and cost constraints, in these platforms, the hardware resources are at a premium and there is a strong desire to utilize all these resources to their fullest potential. Unfortunately, this results in a memory subsystem that is geared to optimize the most common use-cases and usually cannot provide any determinism guarantees in situations when the system is heavily loaded due to hardware level interference among co-scheduled applications. In the following, we describe the most important and well-studied sources of this *shared resource interference* in the use of shared-memory based multicore platforms in real-time systems; along with the state-of-the-art techniques proposed in prior works to mitigate the respective problems that arise from such interference.

2.3.1 The Shared Cache Hierarchy

In modern computer architecture, a processor cache is a fast memory block that provides handy and quick access to the most frequently accessed memory items [36, 37]. Caches are usually structured in levels in the form of a hierarchy; with the level closest to the processor containing caches that are the fastest, relatively smallest and local to the processor and the caches in subsequent levels being

[1]Technically, the TX-2 board contains 6 CPU cores (2 Denver + 4 Cortex A57). However, we only use the Cortex A57 cores when scheduling real-time tasks on TX-2 and the block diagram also reflects the same.

increasingly larger in size, slower in speed and shared to a certain extent among multiple processing cores in the SoC. For example, in Figure 2.4, the multicore CPU cluster in TX-2 contains a 32-KB core local data cache, a 48-KB core local instruction cache and a 2-MB L2 cache that is shared among all the four cores. From a performance isolation point-of-view, the shared last level cache (L2 in the case of TX-2) is important because a *cache miss* in the LLC requires a data item to be fetched from main memory which is an order of magnitude slower than the cache hierarchy [38]. Unfortunately, since the LLC has limited space and it is shared among all the CPU cores, it is possible for tasks executing on one core to evict the data cached in the LLC of a task executing in another core. When the task whose data items got evicted, tries to access that data again, it will incur *cache misses* and its execution time will get adversely affected because of that. This is one source of interference that can arise because of how the LLC is utilized simultaneously by co-executing tasks in a multicore platform.

Apart from conflict misses, co-executing task in a multicore platforms can interfere with each other in more ways based on how they utilize the shared LLC. One such interference arises due to a limited resource inside one type of LLC common in modern COTS multicore platforms—a non-blocking cache—which comes into play when there are multiple outstanding LLC misses that need to be serviced from the main memory. These structures are called *miss-status holding registers* (MSHRs) and they are usually limited in number. It has been shown [3] that interference due to MSHRs can cause up-to 21x slowdown of real-time tasks.

Yet another source of interference due to the shared LLC arises due to the limited capacity of the write-back buffers inside the cache. Like MSHRs, write-back buffers are used in a *write-back* cache to keep track of outstanding memory write requests. When a write-back buffer fills up, it can cause lock-up of the LLC and can adversely delay all the cache requests—even those for whom the data is already present in the LLC i.e., cache hits. Interference due to write-back buffer can be extremely severe; it has been shown [4] to cause more than 300x slowdown to the performance of real-time tasks in a well-known and widely used COTS multicore platform.

Mitigating LLC Interference: Due to the importance and severity of the interference that can

arise due to the shared LLC in a multicore system, both software [39, 40, 41, 42, 43, 44, 45, 46, 47, 48] and hardware based techniques [49, 50, 51, 52] have been extensively studied in prior works to mitigate the extent of this problem. Most such techniques employ some form of LLC space partitioning, through purely software based mechanisms or with software / hardware co-design, to isolate the data of co-executing tasks inside the LLC. Among software based techniques, *page-coloring* is a well-known mechanism to enforce LLC space partitioning and has been shown to be effective in a number of existing studies [53, 40, 47, 48]. In hardware based techniques, the lock-down by line feature in the older generation ARM architecture based CPUs [50] and the recently introduced cache allocation technology (CAT) feature [51] in the Intel architecture are noteworthy. Existing work that makes use of the former to address LLC interference includes [54] whereas for the latter, [55] is a framework that makes use of Intel CAT to ensure quality-of-service (QoS) among real-time tasks. The authors of [3], who first identified the MSHR contention problem, also proposed a hardware based solution to mitigate it and demonstrated its effectiveness in a cycle accurate full-system simulator. Similarly, the authors of [4] proposed a rate-limiting approach using hardware performance monitoring counter (PMCs) to mitigate the interference due to write-back buffers inside the LLC.

2.3.2 Bandwidth and Main-Memory

After the shared cache hierarchy, modern COTS multicore platforms contain a large main memory that is connected with the last level cache through a high speed memory bus. In addition to the CPU cores in the embedded multicore platforms, the main memory is usually shared among all the computing resources present on the chip. For example, in the NVIDIA's Jetson TX-2 board shown in Figure 2.4, the main memory is used by all the cores in the CPU cluster as well as the integrated GPU. This introduces another facet of challenge in deterministically using this resource in real-time scheduling ; in addition to the co-executing CPU applications, interference at the level of main memory can happen among the workloads running on the accelerators such as integrated GPU and those running on the CPU cores. The authors of [56] demonstrate the performance

deterioration of the workloads running on the CPU from the interfering memory traffic of the GPU using workloads in NVIDIA Jetson boards. In [57] and [8], the converse effect—the interference on the performance of real-time GPU applications due to CPU side memory traffic—is shown. Due to these challenges, deterministically utilizing the main memory for real-time applications in modern multicore SoCs can be extremely arduous.

Mitigating Memory Level Interference: Similar to the solutions for mitigating shared LLC interference, proposed approaches to address main memory level interference also fall into software based and hardware based techniques. In the software based approaches, *page-coloring* has been shown to be effective [48, 58, 59, 54] in enforcing space partitioning of the main memory banks among co-executing applications; albeit the fact that this method is only applicable if certain architectural details of the main memory are known or can be inferred through reverse engineering. Moreover, space partitioning cannot ameliorate the interference that can arise due to the limited bandwidth of the main memory bus. To address the latter, a fundamental approach used in a number of prior works is memory bandwidth throttling [2, 60, 54] which uses hardware performance monitoring counters (PMCs) to limit the memory usage quota of co-executing applications. Although effective to a certain extent, memory bandwidth throttling has two fundamental limitations. First, it is difficult to analyze the impact of throttling on the execution of real-time tasks; an analysis [61, 62] that takes the effect into account is bound to be severely pessimistic. For this reason, a throttling based mechanism is commonly used to isolate the performance of real-time tasks from lower criticality best-effort tasks only. Second, a software based throttling mechanism requires the system designer to have the ability to enact preemption of an offending workload; irrespective of where it is executing. Although preemption of CPU using workloads is straight forward, the same is not true for accelerators such as GPU where preempting a currently executing workload can incur unacceptable overheads and hence software based preemption control is traditionally not made part of the GPU's programming model. In such a case, memory bandwidth throttling is not possible to ensure performance isolation among applications that are simultaneously using the main memory.

In the hardware based solutions to the memory level performance isolation challenge, the deterministic memory abstraction [63] is notable in which, the authors design a new memory abstraction for expressing the criticality of memory requests. To enforce deterministic access to the critical memory areas, they propose fundamental changes to the OS and the hardware. Although effective, the deterministic memory solution is not applicable to COTS multicore platforms in which, making changes to the hardware after it has been designed and shipped, is not feasible. Recently, Intel introduced the memory bandwidth allocation (MBA) technology [64] to enforce quality-of-service measure in accessing the main memory on Intel platforms where this feature is available [65]. At the time of this writing, we are not aware of an authoritative work that makes use of MBA to effectively address the main memory level performance isolation.

2.3.3 Performance Isolation on Integrated GPUs

Integrated GPU based platforms have recently gained much attention in the real-time systems community. In [66, 67], the authors investigate the suitability of NVIDIA's Tegra X1 platform for use in safety critical real-time systems. With careful reverse engineering, they have identified undisclosed scheduling policies that determine how concurrent GPU kernels are scheduled on the platform. In SiGAMMA [56], the authors present a novel mechanism to preempt the GPU kernel using a high-priority spinning GPU kernel to protect critical real-time CPU applications. Their work is orthogonal to ours (in Chapter 3) as it solves the problem of protecting CPU tasks from GPU tasks while our work solves the problem of protecting GPU tasks from CPU tasks.

More recently, GPUGuard [68] and HePREM [69] have been presented to provide a mechanism for deterministically arbitrating memory access requests between CPU cores and GPU in heterogeneous platforms containing integrated GPUs. They extend the PREM execution model [70], in which a (CPU) task is assumed to have distinct computation and memory phases, to model GPU tasks. The fundamental approach, in these works, to provide deterministic memory access to real-time GPU kernels, is to ensure that only a single PREM memory phase is in execution at any given time. Although this approach can provide strong isolation guarantees, the drawback is that it

may require significant restructuring of application source code to be compatible with the PREM model.

2.4 Summary

In addition to the most widely studied sources of interference in real-time scheduling literature and presented in this chapter, there can be other hardware structures which can cause runtime coupling among simultaneously executing tasks in a shared-memory based multicore platform such as the translation look-aside buffer (TLB) [71], the main memory controller, the hardware prefetchers etc. Unfortunately, there is no comprehensive list of all the hardware resources that can potentially become sources of shared resource interference in modern COTS multicore platforms; due to the proprietary and often closed-source nature of the underlying hardware architecture. This means that even in a meticulously designed use-case where all the known sources of hardware level shared resource interference are fully partitioned—a requirement that is almost impossible to meet due to the aforementioned challenges—all the possible co-schedules of hard real-time tasks have to be thoroughly investigated; to make sure that a previously unknown hardware resource suddenly does not become a bottleneck in a particular situation.

Unfortunately, in any multicore scheduling policy presented in real-time literature so far and available in a commodity operating system, there are numerous ways in which real-time tasks can potentially get co-scheduled. Hence even after rigorously testing a multicore system scheduled under an optimal multicore scheduling policy, a system designer can not be sure that the timing guarantees will be met 100% of the time in the actual deployment of the system. This fact makes it imperative to turn off all but one cores in a multicore system deployed in an extremely safety critical use-case (e.g., avionics); to meet with safety certification guidelines [5] which makes it all but futile to use a multicore platform in the first place; a phenomenon that has been termed as the *one-out-of-m* [72] core problem in existing real-time literature. This realization has motivated us to design a novel *interference-aware* scheduling policy for hard real-time systems that can guarantee determinism—even at the potential cost of a possible CPU utilization loss—in the presence of

shared hardware resource interference on COTS multicore platforms which will be discussed in the subsequent chapters of this dissertation.

Chapter 3

Real-Time Execution on Integrated CPU-GPU SoC Platforms[1]

Heterogeneous SoC based multicore platforms, containing an integrated GPU in addition to the multicore CPUs, have become the go-to choice for running high performance embedded workloads these days. In this chapter, the challenges inherent in deterministically using these SoCs for realizing real-time systems are discussed in detail. A restrictive scheduling and software level throttling framework, called BWLOCK++, is described which can guarantee deterministic execution on the integrated GPU in a heterogeneous SoC. The implementation of the framework, on top of the commodity Linux kernel, is explained and its evaluation results are elucidated on NVIDIA's Jetson TX2 platform which is an exemplar of heterogeneous embedded computing platforms.

3.1 Introduction

As mentioned in the Chapter 1, heterogeneous computing platforms contain one or more accelerators in addition to the CPU; to speed up the processing of specialized workloads. A characteristic

[1]The following publication inspired the performance isolation technique presented in this chapter:
[60] Heechul Yun, **Waqar Ali**, Santosh Gondi and Siddhartha Biswas (2017). BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Transactions on Computers (TC)*, Vol: 66, Issue: 7, pages 1247–1252

Contents of this chapter have previously appeared in the following publications:
[73] **Waqar Ali** and Heechul Yun (2017). Work-In-Progress: Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms. *Proceedings of the IEEE International Conference on Real-Time and Embedded Technology and Applications Symposium Work-In-Progress (RTAS-WIP)*
[8] **Waqar Ali** and Heechul Yun (2018). Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 19:1–19:22

The following work was inspired by the framework presented in this chapter:
[74] Homa Aghilinasab, **Waqar Ali**, Heechul Yun, Rodolfo Pellizzoni (2020). Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms. In *Proceedings of the ACM/IEEE International Conference on Embedded Software (EMSOFT)*, pages 3348–3360

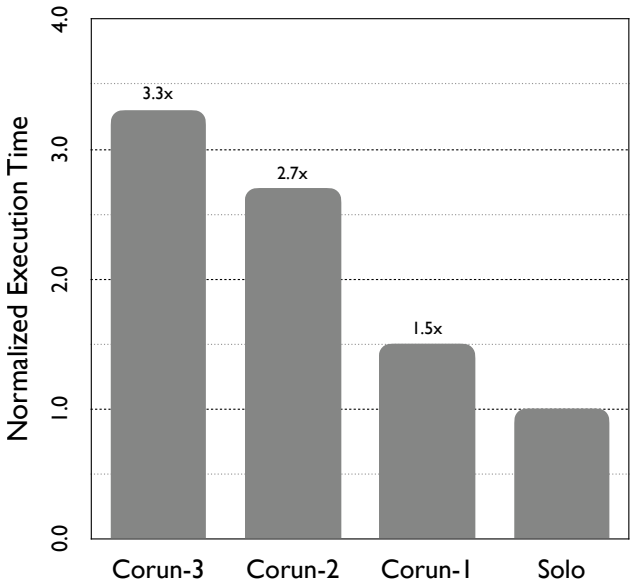


Figure 3.1: Performance of *histo* benchmark on NVIDIA Jetson TX2 with CPU corunners

of many such workloads—e.g., robotics and artificial intelligence—is their data-parallel nature and it has been shown [66] that graphical processing units (GPUs) are tremendously efficient in processing this type of workloads. Consequently, a number of heterogeneous embedded computing platforms (e.g., NVIDIA's Jetson [75] series) these days integrate a GPU alongside the multicore CPU cluster to accelerate the processing of data-parallel applications. The integration of CPU and GPU into a single SoC with a certain level of resource sharing in the memory hierarchy is attractive for the target embedded use-cases where size, weight, power (SWaP) and cost is at a premium. However, this also makes it more challenging to guarantee deterministic execution on heterogeneous computing platforms which is a must for safety critical real-time systems.

The challenge in deterministically using heterogeneous SoCs for real-time applications arises due to contention in the shared hardware resources (e.g., memory bandwidth) which can significantly alter the applications' timing characteristics. On an integrated CPU-GPU platform, such as NVIDIA Jetson TX2, the CPU cores and the GPU typically share a single main memory subsystem. This allows memory intensive batch jobs running on the CPU cores to significantly interfere with the execution of critical real-time GPU tasks (e.g., vision based navigation and obstacle detection) running in parallel due to memory bandwidth contention.

To illustrate the significance of the problem stated above, we evaluate the effect of co-scheduling memory bandwidth intensive synthetic CPU benchmarks on the performance of a GPU benchmark *histo* from the parboil benchmark suite [76] on the NVIDIA Jetson TX2 platform (See Table 3.3 in Section 3.5 for the detailed time breakdown of *histo*.)

We first run the benchmark alone and record the solo execution statistics. We then repeat the experiment with an increasing number of interfering memory intensive benchmarks on the idle CPU cores to observe their impact on the performance of the *histo* benchmark. As can be seen in the Figure 3.1, co-scheduling the memory intensive tasks on the idle CPU cores significantly increase the execution time of the GPU benchmark—an up-to 3.3$\times$ increase—despite the fact that the benchmark has exclusive access to the GPU. The main cause of the problem is that, in the Jetson TX2 platform, both CPU and GPU share the main memory and its limited memory bandwidth becomes a bottleneck. As a result, even though the platform offers plenty of raw performance, no real-time execution guarantees can be provided if the system is left on its own.

Our solution to the problem just described is a software framework called BWLOCK++ which is designed to mitigate the memory bandwidth contention problem in heterogeneous system architecture based embedded computing platforms. More specifically, we focus on protecting real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks. BWLOCK++ dynamically instruments GPU tasks at run-time and inserts a *memory bandwidth lock* while critical GPU kernels are being executed on the GPU. When the bandwidth lock is being held by the GPU, the OS throttles the maximum memory bandwidth usage of the CPU cores to a certain threshold value to protect the GPU kernels. The threshold value is determined on a per GPU task basis and may vary depending on the GPU task's sensitivity to memory bandwidth contention. Throttling CPU cores inevitably negatively affects the CPU throughput. To minimize the throughput impact, we propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS). TFS favors CPU intensive tasks over memory intensive ones while the GPU is busy executing critical tasks in order to minimize CPU throttling. Our evaluation shows that BWLOCK++ can provide good performance isolation for bandwidth intensive GPU tasks in the presence of memory

intensive CPU tasks. Furthermore, the TFS scheduling algorithm reduces the CPU throughput loss by up to 75%. We further show how BLWOCK++ can be incorporated in existing CPU focused response time analysis frameworks to analyze schedulability of real-time tasksets, utilizing both CPU and GPU. Finally, we discuss the limitations of our approach, its existing as well as possible extensions and how it can incorporate heterogeneous SoCs containing accelerators other than the integrated GPU.

3.2 System Model

Although the design philosophy of BWLOCK++ is applicable to any heterogeneous computing platform that contains a single on-chip accelerator; in order to have a concrete discussion about the design and implementation of the framework in this chapter, we primarily limit our discussion to heterogeneous SoCs containing an integrated GPU as an accelerator. Hence we assume an integrated CPU-GPU architecture based platform, which is composed of multiple CPU cores and a single GPU that share the same main memory subsystem. We consider independent periodic real-time tasks with implicit deadlines and best-effort tasks with no real-time constraints.

3.2.1 Task Model

In our system, we assume that each real-time task is composed of at least one CPU execution segment and zero or more GPU execution segments. We assume that *GPU execution is non-preemptible* and we do not allow concurrent execution of multiple GPU kernels from different tasks at the same time. Simultaneously co-scheduling multiple kernels is called GPU co-scheduling, which has been avoided in most prior real-time GPU management approaches [77, 78, 79] as well due to unpredictable timing. According to [66], preventing GPU co-scheduling does not necessarily hurt—if not improve—performance because concurrent GPU kernels from different tasks are executed in a time-multiplexed manner rather than being executed in parallel. [2]

[2]Another recent study [80] finds that GPU kernels can only be executed in parallel if they are submitted from a single address space. In this work, we assume that a task has its own address space, whose GPU kernels are thus

Executing GPU kernels typically requires copying considerable amount of data between the CPU and the GPU. In particular, synchronous copy directly contributes to the task's execution time, while asynchronous copy can overlap with GPU kernel execution. Therefore, we model synchronous copy separately. Lastly, we assume that a task is single-threaded with respect to the CPU. With these assumptions, we can model a real-time task as follows:

$$\tau_i := (C_i, G_i^m, G_i^e, P_i)$$

where:

- C_i is the cumulative WCET of CPU-only execution
- G_i^m is the cumulative WCET of synchronous memory operations between CPU and GPU
- G_i^e is the cumulative WCET of GPU kernels
- P_i is the period

Note that the goal of BWLOCK++ is to reduce G_i^m and G_i^e under the presence of memory intensive best-effort tasks running in parallel.

3.2.2 CPU Scheduling

We assume that a fixed-priority preemptive real-time scheduler is used for scheduling real-time tasks and a virtual run-time based fair sharing scheduler (e.g., Linux's Completely Fair Scheduler (CFS) [9]) is used for best-effort tasks. For simplicity, **we assume a single dedicated real-time core schedules all real-time tasks**, while any core can schedule best-effort tasks. Because GPU kernels are executed serially on the GPU, as mentioned above, for GPU intensive real-time tasks, which we focus on in this work, this assumption does not significantly under-utilize the system, especially when there are enough co-scheduled best-effort tasks, while it enables simpler analysis.

time-multiplexed with other tasks' GPU kernels at the GPU-level.

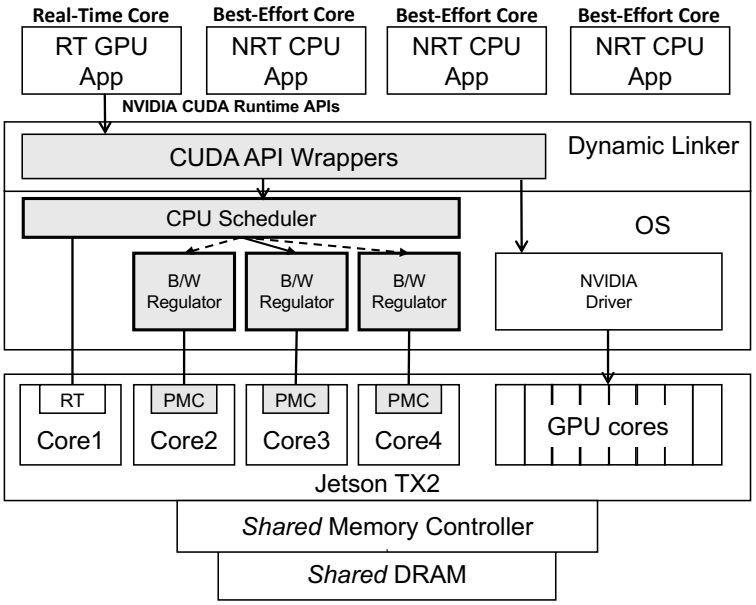


Figure 3.2: BWLOCK++ System Architecture

3.3 BWLOCK++

In this section, we provide an overview of BWLOCK++ and discuss its design details.

3.3.1 Overview

BWLOCK++ is a software framework to protect GPU applications on integrated CPU-GPU architecture based SoC platforms. We focus on the problem of the shared memory bandwidth contention between GPU kernels and CPU tasks in integrated CPU-GPU architectures. More specifically, we focus on protecting GPU execution intervals of real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks.

In BWLOCK++, we exploit the fact that each GPU kernel is executed via explicit programming interfaces from a corresponding host CPU program. In other words, we can precisely determine when the GPU kernel starts and finishes by instrumenting these functions.

To avoid memory bandwidth contention from the CPU, we notify the OS before a GPU application launches a GPU kernel and after the kernel completes with the help of a system call. Apart from acquiring the bandwidth lock on the task's behalf, this system call also implements

the non-preemptive locking protocol [81] to prevent preemption of the GPU using task. While the bandwidth lock is being held by the GPU task, the OS regulates memory bandwidth consumption of the best-effort CPU cores to minimize bandwidth contention with the GPU kernel. Concretely, each best-effort core is periodically given a certain amount of memory bandwidth budget. If the core uses up its given budget for the specified period, the (non-RT) CPU tasks running on that core are throttled. In this way, the GPU kernel suffers minimal memory bandwidth interference from the best-effort CPU cores. However, throttling CPU cores can significantly lower the overall system throughput. To minimize the negative throughput impact, we propose a new CPU scheduling algorithm, which we call the Throttle Fair Scheduler (TFS), to minimize the duration of CPU throttling without affecting memory bandwidth guarantees for real-time GPU applications.

Figure 3.2 shows the overall architecture of the BWLOCK++ framework on an integrated CPU-GPU architecture (NVIDIA Jetson TX2 platform). BWLOCK++ is comprised of three major components: (1) Dynamic run-time library for instrumenting GPU applications; (2) the Throttle Fair Scheduler; (3) Per-core B/W regulator . Working together, they protect real-time GPU kernels and minimize CPU throughput reduction. We will explain each component in the following.

3.3.2 Automatic Instrumentation of GPU Applications

To eliminate manual programming efforts, we automatically instrument the program binary at the dynamic linker level. We exploit the fact that the execution of a GPU application using a GPU runtime library such as NVIDIA CUDA typically follows fairly predictable patterns. Figure 3.3 shows the execution timeline of a typical synchronous GPU application that uses the CUDA API.

In order to protect the runtime performance of a GPU application from co-running memory intensive CPU applications, we need to ensure that the GPU application automatically holds the memory bandwidth lock while a GPU kernel is executing on the GPU or performing a memory copy operation between CPU and GPU. Upon the completion of the execution of the kernel or memory copy operation, the GPU application again shall automatically release the bandwidth lock. This is done by instrumenting a small subset of CUDA API functions that are invoked when

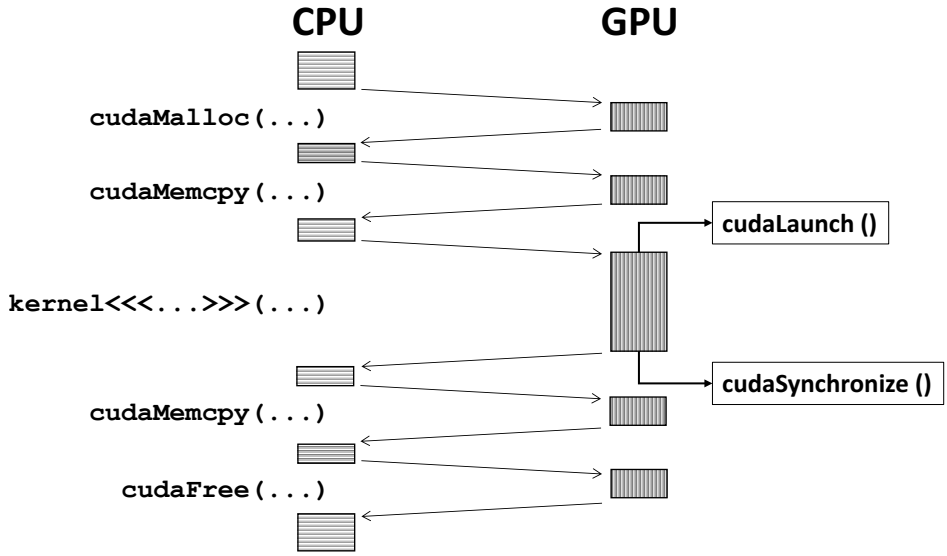


Figure 3.3: Phases of GPU Application under CUDA Runtime

launching or synchronizing with a GPU kernel or while performing a memory copy operation. These APIs are documented in Table 3.1. More specifically, we write wrappers for these functions of interest which request/release bandwidth lock on behalf of the GPU application before calling the actual CUDA library functions. We compile these functions as a shared library and use Linux' `LD_PRELOAD` mechanism [82] to force the GPU application to use those wrapper functions whenever the CUDA functions are called. In this way, we automatically throttle CPU cores' bandwidth usage whenever real-time GPU kernels are being executed so that the GPU kernels' memory bandwidth can be guaranteed.

A complication to the automatic GPU kernel instrumentation arises when the application uses CUDA streams to launch multiple GPU kernels in succession in multiple streams and then waits for those kernels to complete. In this case, the bandwidth lock acquired by a GPU kernel launched in one stream can potentially be released when synchronizing with a kernel launched in another stream. In our framework, this situation is averted by keeping track of active streams and associating bandwidth lock with individual streams instead of the entire application whenever stream based CUDA APIs are invoked. A stream is considered active if:

- A kernel or memory copy operation is launched in that stream

API	Action	Description
cudaConfigureCall	Update active streams	Specify the launch parameters for the CUDA kernel
cudaMemcpy	Acquire BWLOCK++ (*Before*) Release BWLOCK++ (*After*)	Perform synchronous memory copy between CPU and GPU
cudaMemcpyAsync	Acquire BWLOCK++ and update active streams	Perform asynchronous memory copy between CPU and GPU
cudaLaunch	Acquire BWLOCK++	Launch a GPU kernel
cudaDeviceSynchronize cudaThreadSynchronize	Release BWLOCK++ and clear active streams	Block the calling CPU thread until all the previously requested tasks in a specific GPU device have completed
cudaStreamSynchronize	Update active streams and release BWLOCK++ if there are no active streams	Block the calling CPU thread until all the previously requested tasks in a specific CUDA stream have completed

Table 3.1: CUDA APIs instrumented via LD_PRELOAD for BWLOCK++

- The stream has not been explicitly (using cudaStreamSynchronize) or implicitly (using cudaDeviceSynchronize or cudaThreadSynchronize) synchronized with

Our framework ensures that a GPU application continues holding the bandwidth lock as long as it has one or more active streams.

The obvious drawback of throttling CPU cores is that the CPU throughput may be affected especially if some of the tasks on the CPU cores are memory bandwidth intensive. In the following sub-section, we discuss the impact of throttling on CPU throughput and present a new CPU scheduling algorithm that minimizes throughput reduction.

3.3.3 Throttle Fair CPU Scheduler (TFS)

As described earlier in this section, BWLOCK++ uses a throttling based approach to enforce memory bandwidth limit of CPU cores at a regular interval. Although effective in protecting critical GPU applications in the presence of memory intensive CPU applications, this approach runs into the risk of severely under-utilizing the system's CPU capacity; especially in cases when there are multiple best-effort CPU applications with different memory characteristics running on the best-effort CPU cores. In the throttling based design, once a core exceeds its memory bandwidth quota and gets throttled, that core cannot be used for the remainder of the period. Let us denote the regulation period as T (i.e., $T = 1ms$) and the time instant at which an offending core exceeds its bandwidth budget as t. Then the wasted time due to throttling can be described as $\delta = T - t$. The value of δ is a direct quantifier of the wasted system throughput. Since the regulation period T is fixed, δ is entirely dependent on the instant t at which throttling comes into effect i.e., the smaller the value of t (i.e., throttled earlier in the period) the larger the penalty to the overall system throughput. The value of t, on the other hand, depends on the rate at which a core consumes its allocated memory budget and that in turn depends on the memory characteristics of the application executing on that core. **To maximize the overall system throughput, the value of δ should be minimized**—that is if throttling never occurs, $t \geq T \Rightarrow \delta = 0$, or occurs late in the period, throughput reduction will be less. This is the design goal of the Throttle Fair Scheduler.

3.3.3.1 Negative Feedback Effect of Throttling on CFS

In a system where multiple best-effort tasks are available—with different memory bandwidth usage characteristics—to utilize the slack intervals left by real-time tasks, one way to reduce CPU throttling is to schedule less memory bandwidth demanding tasks on the best-effort CPU cores during the regulated intervals i.e., while the GPU is holding the bandwidth lock. Assuming that each best-effort CPU core has a mix of memory bandwidth intensive and CPU intensive tasks, then scheduling the CPU intensive tasks while the GPU is holding the lock would reduce CPU throttling or at least delay the instant at which throttling occurs, which in turn would improve CPU

throughput. Unfortunately, Linux's default scheduler CFS [9] actually aggravates the possibility of early and frequent throttling when used with BWLOCK++'s throttling mechanism.

The CFS algorithm tries to allocate fair amount of CPU time among tasks by using each task's weighted virtual runtime (i.e., weighted execution time) as the scheduling metric. Concretely, a task τ_i's virtual runtime V_i is defined as :

$$V_i = \frac{E_i}{W_i} \tag{3.1}$$

where E_i is the actual runtime and W_i is the weight of the task. At each scheduling instant, the CFS algorithm simply picks the task with the smallest virtual runtime.

The problem with memory bandwidth throttling under CFS arises because the virtual run-time of a memory intensive task, which gets frequently throttled, increases more slowly than the virtual run-time of a compute intensive task which does not get throttled. Due to this, the virtual runtime based arbitration of CFS tends to schedule the memory intensive tasks more than the CPU intensive tasks while bandwidth regulation is in place.

3.3.3.2 TFS Approach

In order to reduce the throttling overhead while keeping the undesirable scheduling of memory intensive tasks quantifiable, TFS modifies the throttled task's virtual runtime to take the task's throttled duration into account. Specifically, at each regulation period, if there exists a throttled task, we scale the throttled duration of the task by a factor, which we call *TFS punishment factor* (ρ), and add it to its virtual runtime.

Under TFS, a throttled task τ_i's virtual runtime V_i^{new} at the end of j^{th} regulation period is expressed as:

$$V_i^{new} = V_i^{old} + \delta_i^j \times \rho \tag{3.2}$$

where δ_i^j is the throttled duration of τ_i in the j^{th} sampling period.

The more memory intensive a task, the more likely it is that it will get throttled in each regulation period for a longer duration of time (i.e., higher δ_i). By adding the throttled time back to the task's virtual runtime, we make sure that the memory intensive tasks are not favored by the scheduler. Furthermore, by adjusting the TFS punishment factor ρ, we can further penalize memory intensive tasks in favor of CPU intensive ones. This in turn reduces the amount of throttled time and improves overall CPU utilization. On the other hand, the memory intensive tasks will still be scheduled (albeit less frequently so) according to the adjusted virtual runtime.

Scheduling of tasks under TFS is fair with respect to the adjusted virtual runtime metric but it can be considered unfair with respect to the CFS's original virtual runtime. A task τ_i's "lost" virtual runtime Δ_i^{TFS} (due to TFS's inflation) over J regulation periods can be quantified as follows:

$$\Delta_i^{TFS} = \sum_{j=0}^{J} \delta_i^j \times \rho. \tag{3.3}$$

3.3.3.3 Illustrative Example

In the following, we elucidate the misbehavior of the CFS algorithm under throttling and the benefit of our TFS extension with a concrete illustrative example.

Let us consider a small integrated CPU-GPU system, which consists of two CPU cores and a GPU. We further assume, according to our system model, that Core-1 is a real-time core, which may use the GPU, and Core-2 is a best-effort core, which doesn't use the GPU.

Table 3.2 shows a taskset to be scheduled on the system. The taskset is composed of a GPU using real-time task, which needs to be protected by our framework for the entire duration of its

Task	Compute Time (C)	Period (P)	Description
τ_{RT}	4	15	Real-time task
τ_{MEM}	4	N/A	Memory intensive best-effort task
τ_{CPU}	4	N/A	CPU intensive best-effort task

Table 3.2: Taskset for Example

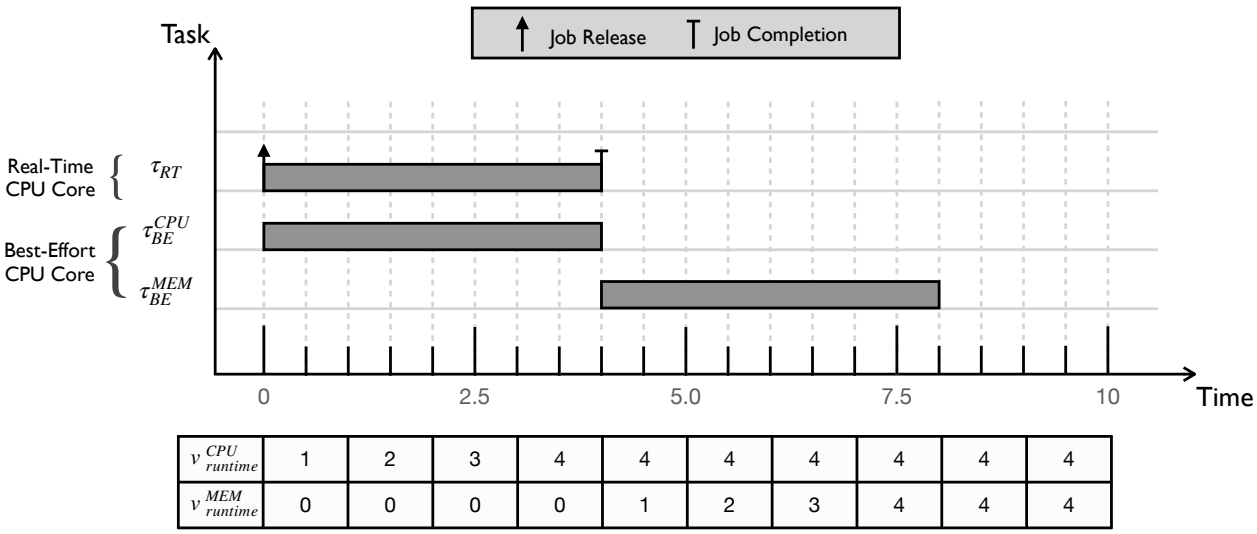
$v_{runtime}^{CPU}$	1	1	1	1	1	2	3	3	3	4
$v_{runtime}^{MEM}$	0	0.33	0.67	1	2	2	2	3	4	4

Figure 3.4: Example schedule under CFS with 1-msec scheduling tick

execution; and two best-effort tasks (of equal CFS priority), one of which is CPU intensive and the other is memory intensive.

Figure 3.4 shows how the scheduling would work when CFS is used to schedule best-effort tasks τ_{CPU} and τ_{MEM} on the best-effort core with its memory bandwidth is throttled by our kernel-level bandwidth regulator. Note that in this example, both OS scheduler tick timer interval and the bandwidth regulator interval are assumed to be 1-msec. At time 0, τ_{CPU} is first scheduled. Because τ_{CPU} is CPU bound, it doesn't suffer throttling. At time 1, the CFS schedules τ_{MEM} as its virtual runtime 0 is smaller than τ_{CPU}'s virtual runtime 1-msec. Shortly after τ_{MEM} is scheduled, however, it gets throttled at time 1.33 as it has used the best-effort core's allowed memory bandwidth budget for the regulation interval. When the budget is replenished at time instant 2, at the beginning of the new regulation interval, τ_{MEM}'s virtual runtime is 0.33-msec whereas for τ_{CPU}, it is 1-msec. So, the CFS picks τ_{MEM} (smaller of the two) again, which gets throttled again. This pattern continues until τ_{MEM}'s virtual runtime finally catches up with τ_{CPU} at time instant 4 by which point the best-effort core has been throttled 66% of the duration between time instants 1 and 4. As can be seen in this example, CFS favors memory intensive tasks as their virtual runtimes increase more slowly than CPU intensive ones when memory bandwidth throttling is used.

Figure 3.5 shows a hypothetical schedule in which the execution of τ_{MEM} is delayed in favor of

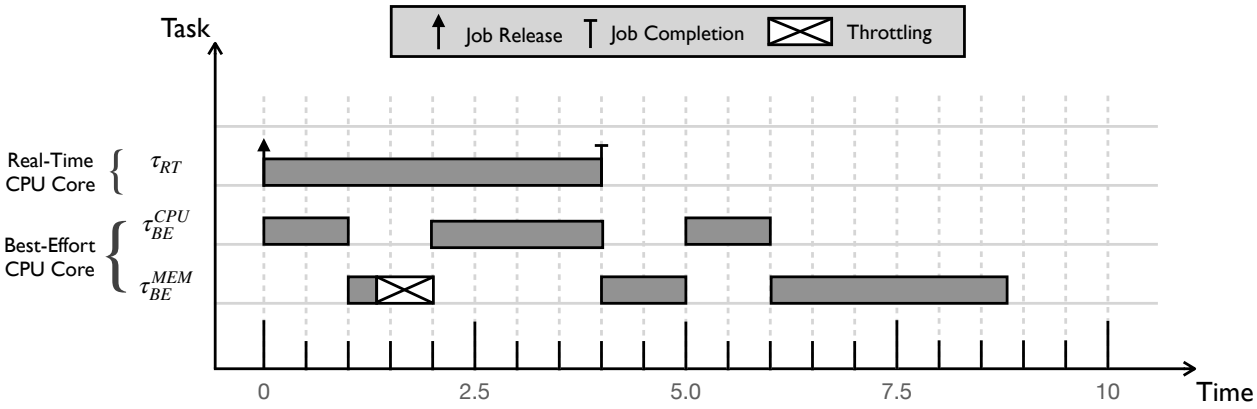


$v_{runtime}^{CPU}$	1	2	3	4	4	4	4	4	4	4
$v_{runtime}^{MEM}$	0	0	0	0	1	2	3	4	4	4

Figure 3.5: Example schedule with zero throttling

τ_{CPU} while τ_{RT} is running (thus, memory bandwidth regulation is in place.) In this case, because τ_{CPU} never exhausts the memory bandwidth budget, throttling does not take place. As a result, the best-effort core gets fully utilized and thus is able to achieve high throughput. While this is ideal behavior from maximizing throughput, it is not ideal for τ_{MEM} as it can suffer starvation.

Figure 3.6 shows the schedule under TFS (with a TFS punishment factor $\rho = 3$). The TFS works identical to CFS until at time 2, when the BWLOCK++'s periodic timer is called. At this point, τ_{MEM}'s virtual runtime (V^{MEM}) is 0.33-msec. However, because it has been throttled for 0.67-msec during the regulation period ($\delta = 0.67$), according to Equation 3.2, TFS increases the task's virtual runtime to 2.34-msec ($V^{MEM} + \delta \times \rho = 0.33 + 0.67 \times 3 = 2.34$). Because of the increased virtual runtime, the TFS scheduler then picks τ_{CPU} as its virtual runtime is now smaller than that of τ_{MEM} ($1 < 2.34$). Later, when τ_{CPU}'s virtual runtime becomes 3-msec at time instant 4, the TFS scheduler can finally re-schedule τ_{MEM}. In this manner, TFS favors CPU intensive tasks over memory-intensive ones, while preventing starvation of the latter. Note that TFS works at each regulation period (i.e., 1-msec) independently and thus automatically adapts to the task's changing behavior. For example, if a task is memory intensive only for a brief period of time, the task will be throttled only for the memory intensive duration, and the throttled time will be added back to the task's virtual runtime at each 1-msec regulation period. Furthermore, even for a period when a task



$v_{runtime}^{CPU}$	1	1	2	3	3	4	4	4	4	4
$v_{runtime}^{MEM}$ (TFS-3x)	0	2.34	2.34	2.34	3.34	3.34	4.34	5.34	6	6
$v_{runtime}^{MEM}$ (Actual)	0	0.33	0.33	0.33	1.33	1.33	2.33	3.33	4	4

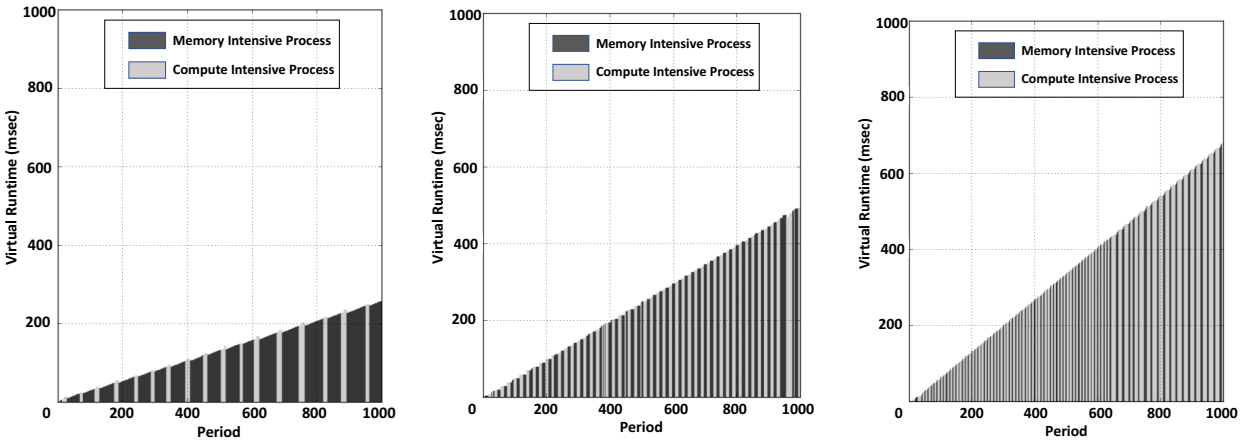
Figure 3.6: Example schedule under TFS with $\rho = 3$

is throttled, the task always makes small progress as allowed by the memory bandwidth budget for the period. Therefore, no task suffers complete starvation for an extended period of time.

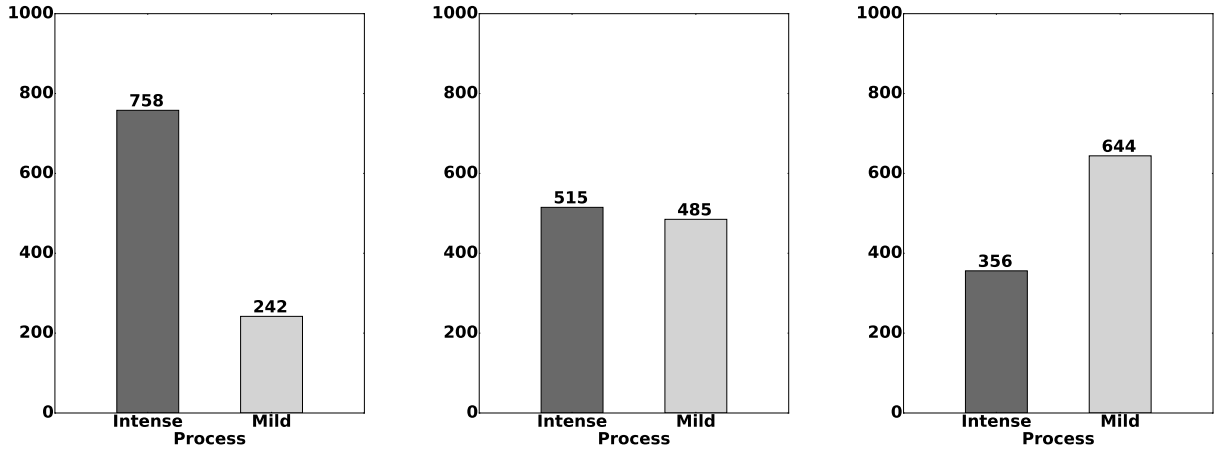
3.3.3.4 Effects of TFS using Synthetic Tasks

We experimentally validate the effect of TFS in better scheduling best-effort tasks on a real system. In this experiment, we use two synthetic tasks: one is CPU intensive and the other is memory-intensive. We use *bandwidth* benchmark from the IsolBench suite [83] for both of these tasks. The *bandwidth* benchmark creates a contiguous array in memory of a specific size (i.e., the working-set size) and then sequentially accesses it for a configurable number of iterations and duration. Based on the size of the array, the benchmark can be used to overload different levels of the memory hierarchy in the underlying hardware platform. In order to make *bandwidth* memory intensive, we configure its working-set size to be twice the size of LLC on our platform. Similarly, to make *bandwidth* compute (CPU) intensive, we make its working set size one half the size of L1 data cache in our platform. We assign these two best-effort tasks to the same best-effort core, which is regulated with a 100-MB/s memory bandwidth budget.

Figure 3.7 shows the virtual runtime progression over 1000 sampling periods of the two tasks



(a) CFS (b) TFS (c) TFS-3x



(d) CFS (e) TFS (f) TFS-3x

Figure 3.7: Virtual runtime progress (top) of two synthetic tasks and the number of periods during which the two tasks are scheduled (bottom). One is cpu-intensive (*Mild*) and the other is memory-intensive (*Intense*).

under three scheduler configurations: CFS, TFS ($\rho = 1$), and TFS-3x ($\rho = 3$). In CFS, the memory intensive process gets preferred by the CFS scheduler at each scheduling instance, because its virtual run-time progresses more slowly. In TFS and TFS-3X, however, as memory-intensive task's virtual runtime increases proportionally to the time it gets throttled, CPU-intensive task is scheduled more frequently.

This can be seen more clearly in the bottom part of the Figure 3.7, which shows the number of periods utilized by each task on the CPU core, over the course of one thousand sampling periods.

Algorithm 1: BWLOCK++ System Call

```
1  syscall sys_bwlock(bw_val)
2      if smp_processor_id () == RT_CORE_ID ∧ rt_task (current) then
3          rt_core_data := get_rt_core_data ()
4          rt_core_data → current_task := current
5          if bw_val ≥ 1 then
6              /* Task is acquiring bwlock */
7              current → bwlock_val := 1
8              current → bw_old_priority := current → rt_priority
9              current → rt_priority := MAX_USER_RT_PRIO - 1
10         else
11             /* Task is releasing bwlock */
12             current → bwlock_val := 0
13             current → rt_priority := current → bw_old_priority
14         end
15     end
16 return;
```

Under CFS, out of all the sampling periods, 75% are utilized by the memory intensive process and only 25% are utilized by the compute intensive process. With TFS, the two tasks get to run in roughly the same number of sampling periods whereas in TFS-3x, the CPU intensive task gets to run more than the memory intensive task.

3.4 Implementation

In this section, we describe the implementation details of BWLOCK++.

We add a new system call `sys_bwlock` to the Linux kernel v4.4.38. The system call serves two purposes. 1) It acquires or releases the memory bandwidth lock on behalf of the currently running task on the real-time core; and 2) it implements the non-preemptive priority protocol, which boosts the calling task's priority to the system's ceiling priority, to prevent preemption. We introduce two new integer fields, `bwlock_val`, `bw_old_priority`, into the task control block: `bwlock_val` stores the current status of the memory bandwidth lock and `bw_old_priority` keeps track of the original real-time priority of the task while it is holding the bandwidth lock.

3.4.1 BWLOCK++ System Call

Algorithm 1 shows the implementation of the system call. To acquire the memory bandwidth lock, the system call must be invoked from the real-time system core and the task currently scheduled on the real-time core must have a real-time priority (*line 2*). At the time of acquisition of bandwidth lock, the priority of the calling task, which is tracked by the globally accessible `current` pointer in the Linux kernel, is raised to the maximum allowed real-time priority value (the ceiling priority) for any user-space task to prevent preemption (*line 7*). The real-time priority value of the the task is restored to its original priority value when the bandwidth lock is released (*line 10*). In this manner, the system call updates the state of the currently scheduled real-time task on the real-time system core, which is then used by the memory bandwidth regulator on best-effort cores to enforce memory usage thresholds, as explained in the following subsection.

3.4.2 Per-Core Memory Bandwidth Regulator

The per-core memory bandwidth regulator is composed of a periodic timer interrupt handler and a performance monitoring counter (PMC) overflow interrupt handler. Algorithm 2 shows the implementation of the memory bandwidth regulator.

The periodic timer interrupt handler is invoked at a regular interval (currently every 1-msec) using a high resolution timer on each best-effort core. The timer handler begins a new bandwidth lock regulation period and performs the following operations:

- Unthrottle the core if it was throttled in the last regulation period (*line 4*)
- Scale the virtual runtime of the task currently scheduled on the core based on the throttling time in the last period and the TFS punishment factor (*lines 5, 6*)
- Determine the new memory usage budget based on the bandwidth lock status of the task currently scheduled on the real-time system core (*lines 9-14*)
- Program the performance monitoring counter on the core based on the new memory usage

Algorithm 2: Memory Bandwidth Regulator

```
1  procedure periodic_interrupt_handler(core_data)
2      /* Unthrottle the core if it was previously throttled */
3      if core_is_throttled (core_data → core_id) == TRUE then
4          unthrottle_core (core_data → core_id)
5          record_throttling_end_time (core_data → current_task)
6          scale_virtual_runtime (core_data → current_task)
7      end

8      /* Update the core's budget based on the current rt task */
9      rt_core_data := get_rt_core_data ()
10     if rt_core_data → current_task → bwlock_val == 1 then
11         core_data → new_budget := rt_core_data → throttle_budget
12     else
13         core_data → new_budget := MAX_BANDWIDTH_BUDGET
14     end

15     /* Reprogram the PMC overflow interrupt */
16     program_pmc (core_data → new_budget)
17 return;

18 procedure pmc_overflow_handler(core_data)
19     record_throttling_start_time (core_data → current_task)
20     throttle_core (core_data → core_id)
21 return;
```

budget for the current regulation period (*line 16*). We use the `L2D_CACHE_REFILL` event for measuring the memory bandwidth traffic in the ARM Cortex-A57 processor core

The PMC overflow interrupt occurs when the core at hand exceeds its memory usage budget in the current regulation period. The interrupt handler prevents further memory transactions from this core by scheduling a high priority idle kernel thread on it for the remainder of the regulation period (*lines 18-21*).

3.5 Evaluation

In this section, we present the experimental evaluation results of BWLOCK++.

Benchmark	Dataset	Copy (KBytes)	Timing Breakdown (msec)			
			Kernel (G^e)	Copy (G^m)	Compute (C)	Total (E)
histo	Large	5226	83409	18	0	83428
sad	Large	709655	152	654	53	861
bfs	1M	62453	174	72	0	246
spmv	Large	30138	69	51	10	131
stencil	Default	196608	749	129	9	888
lbm	Long	379200	43717	358	2004	46080

Table 3.3: GPU execution time breakdown of selected benchmarks

3.5.1 Setup

We evaluate BWLOCK++ on NVIDIA Jetson TX2 platform. We use the Linux kernel version 4.4.38, which is patched with the changes required to support BWLOCK++. The CUDA runtime library version installed on the platform is 8.0. In all our experiments, we place the platform in maximum performance mode by maximizing GPU and memory clock frequencies and disabling the dynamic frequency scaling of CPU cores. We also shutdown the graphical user interface and disable the network manager to avoid run to run variation in the experiments.

As per our system model, we designate the Core-0 in our system as real-time core. The remaining cores execute best-effort tasks only. All the tasks are statically assigned to their respective cores during the experiment. While NVIDIA Jetson TX2 platform contains two CPU islands, a quad-core Cortex-A57 and a dual-core Denver, we only use the Cortex-A57 island for our evaluation and leave the Denver island off because we were unable to find publicly available documentation regarding the Denver cores' hardware performance counters, which is needed to implement throttling. In order to evaluate BWLOCK++, we use six benchmarks from parboil suite which are listed as memory bandwidth sensitive in [76].

3.5.2 Effect of Memory Bandwidth Contention

In this experiment, we investigate the effect of memory bandwidth contention due to co-scheduled memory intensive CPU applications on the evaluated GPU kernels.

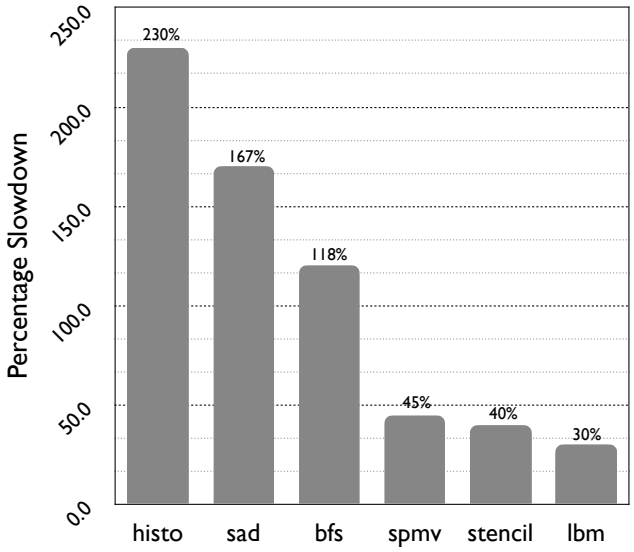


Figure 3.8: Slowdown of the total execution time of GPU benchmarks due to three *Bandwidth* corunners.

First, we measure the execution time of each GPU benchmark in isolation. From this experiment, we record the GPU kernel execution time (G^e), memory copy time for GPU kernels (G^m) and CPU compute time (C) for each benchmark. The data collected is shown in Table 3.3. We then repeat the experiment after co-scheduling three instances of a memory intensive CPU application as co-runners. We use the *bandwidth* benchmark [3] for this purpose. The sequential write access pattern of *bandwidth* can cause worst-case interference on several multicore platforms [84]. The results of this experiment are shown in Figure 3.8 and they demonstrate how much the total execution time of GPU benchmarks ($E = G^e + G^m + C$) suffers from memory bandwidth contention due to the co-scheduled CPU applications.

From Figure 3.8, it can be seen that the worst case slowdown, in case of *histo* benchmark, is more than 250%. Similarly, for SAD benchmark, the worst case slowdown is more than 150%. For all other benchmarks, the slowdown is non-zero and can be significant in affecting the real-time performance. These results clearly show the danger of uncontrolled memory bandwidth sharing in an integrated CPU-GPU architecture as GPU kernels may potentially suffer severe interference from co-scheduled CPU applications. In the following experiment, we investigate how this problem can be addressed by using BWLOCK++.

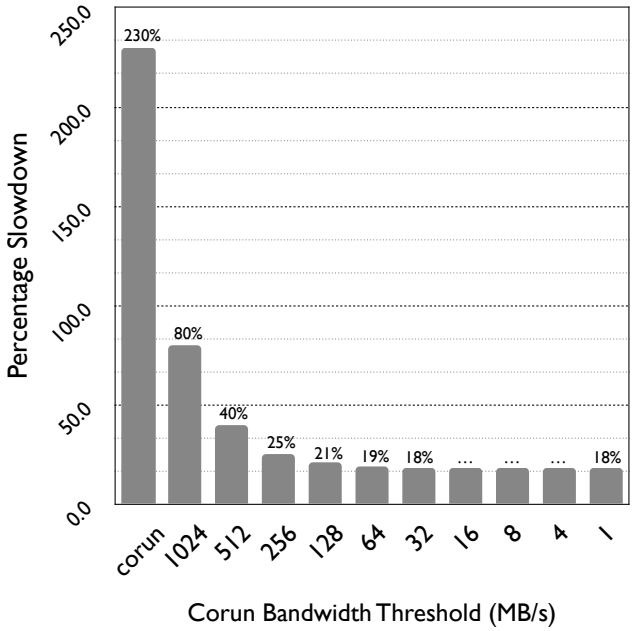


Figure 3.9: Effect of corun bandwidth threshold on the execution time of histo benchmark.

3.5.3 Determining Memory Bandwidth Threshold

In order to apply BWLOCK++, we first need to determine safe memory budget that can be given to the best-effort CPU cores in the presence of GPU applications. However, an appropriate threshold value may vary depending on the characteristics of individual GPU applications. If the threshold value is set too high, then it may not be able to protect the performance of the GPU application. On the other hand, if the threshold value is set too low, then the CPU applications will be throttled more often and that would result in significant CPU capacity loss.

We calculate the safe memory budget for best-effort CPU cores by observing the trend of slowdown of the total execution time of GPU application as the allowed memory usage threshold of CPU co-runners is varied. We start with a threshold value of 1-GB/s for each best-effort CPU core. We then continue reducing the threshold value for best-effort cores by half and measure the impact of this reduction on the slowdown of execution time (E) of the benchmark.

Figure 3.9 shows this trend for the execution time of *histo* benchmark from the parboil suite. From the figure, it can be seen that after `64-MBps` threshold value for best-effort CPU cores, further reduction of threshold value does not yield significant improvement in reducing the slowdown of

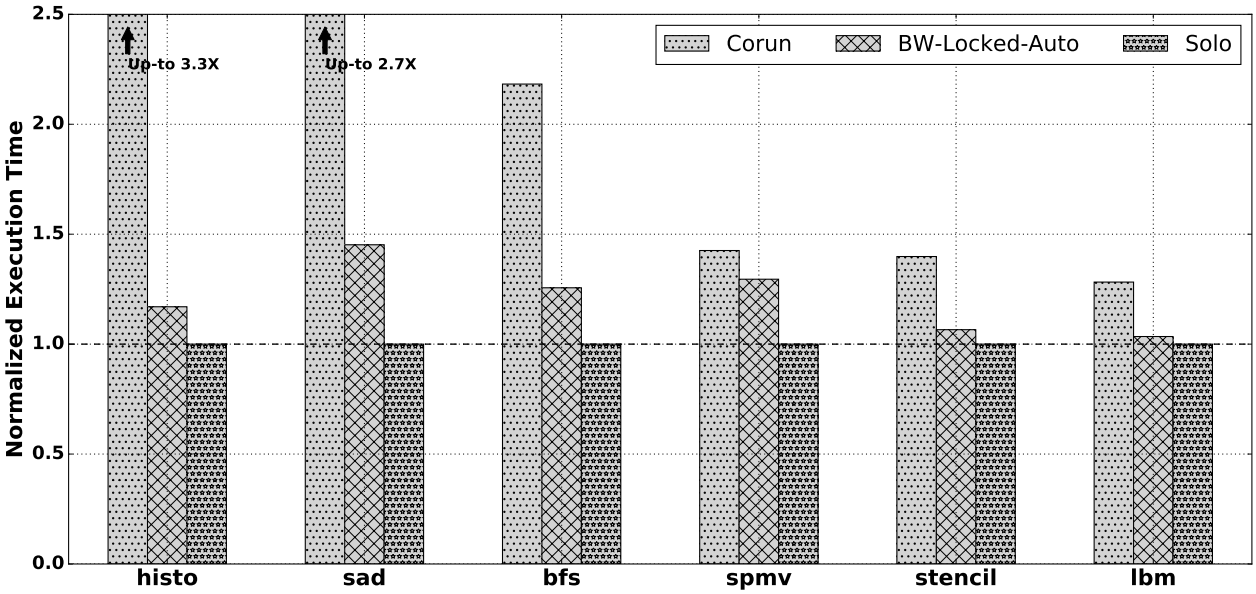


Figure 3.10: BWLOCK++ Evaluation Results

benchmark. Hence, for *histo* benchmark, we select `64-MBps` as the threshold value for the best-effort CPU cores. In a similar fashion, we plot this trend for all the selected benchmarks and determine the value of corun threshold for the best-effort CPU cores.

3.5.4 Effect of BWLOCK++

In this experiment, we evaluate the performance of BWLOCK++. Specifically, we record the corun execution of GPU benchmarks with the automatic instrumentation of BWLOCK++. We call this scenario *BW-Locked-Auto*. We compare the performance under *BW-Locked-Auto* against the *Solo* and *Corun* execution of the GPU benchmarks which represent the measured execution times in isolation and together with three co-scheduled memory intensive CPU applications, respectively.

To get the data-points for *BW-Locked-Auto*, we configure BWLOCK++ according to the allowed memory usage threshold of the benchmark at hand and use our dynamic GPU kernel instrumentation mechanism to launch the benchmark in the presence of three *bandwidth* benchmark instances (write memory access pattern) as CPU co-runners. The results of this experiment are plotted in Figure 3.10. In this figure, we plot the total execution time of each benchmark for the above mentioned scenarios. All the time values are normalized with respect to the total execution

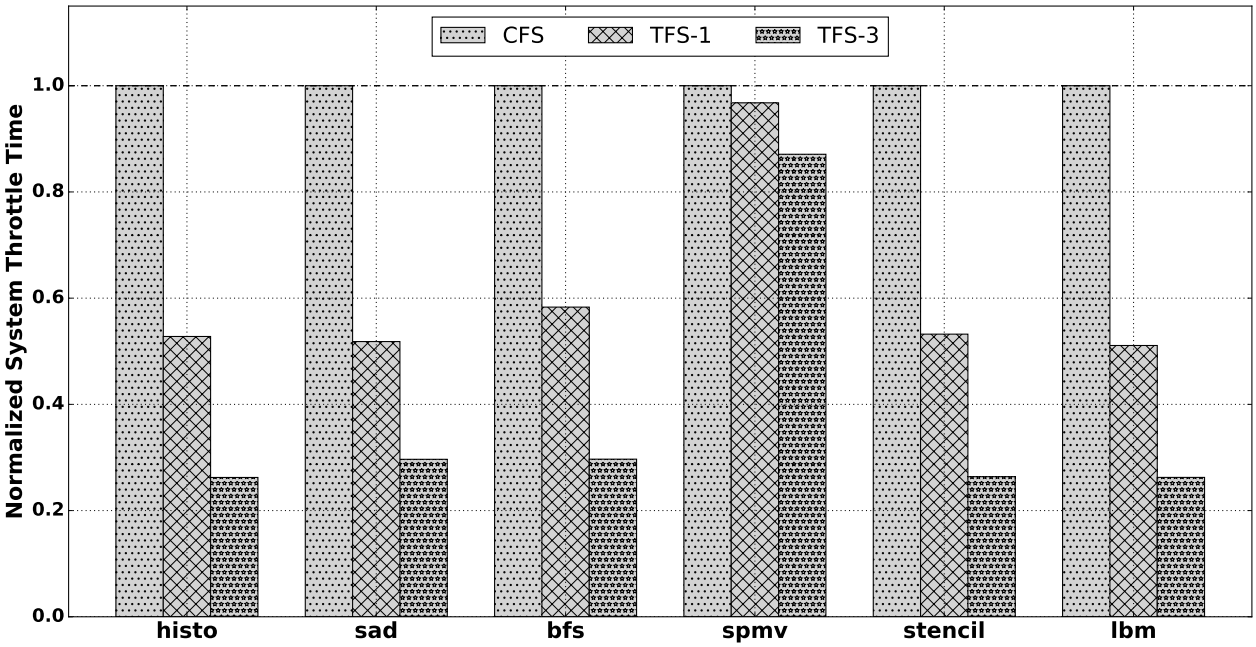


Figure 3.11: Comparison of total system throttle time under different scheduling schemes

time ($E_{solo} = C_{solo} + G^e_{solo} + G^m_{solo}$) of the benchmark in isolation. As can be seen, execution under *BW-Locked-Auto* incurs significantly less slowdown of the total execution time of GPU benchmarks due to reduction of both GPU kernel execution and memory copy operation times.

3.5.5 Throughput improvement with TFS

As explained in Section 3.3.3, throttling under CFS results in significant system throughput reduction. In order to illustrate this, we conduct an experiment in which the GPU benchmarks are executed with six CPU co-runners. Each CPU core, apart from the one executing the GPU benchmark, has a memory intensive application and a compute intensive application scheduled on it. For both of these applications, we use the *bandwidth* benchmark with different working set sizes. In order to make *bandwidth* memory intensive, we configure its working set size to be twice the size of LLC on our evaluation platform. Similarly for compute intensive case, we configure the working set size of *bandwidth* to be half of the L1-data cache size. We record the total system throttle time statistics with BWLOCK++ for all the GPU benchmarks. The total system throttle time is the sum of throttle time across all system cores. We then repeat the experiment with our

Throttle Fair Scheduling scheme. In *TFS-1*, we configure the TFS punishment factor as one for the memory intensive threads and in *TFS-3*, we set this factor to three. We plot the normalized total system throttle time for all the scheduling schemes and present them in Figure 3.11. It can be seen that TFS results in significantly less system throttling (On average, 39% less with *TFS-1* and 62% less with *TFS-3*) as compared to CFS.

3.5.6 Overhead due to BWLOCK++

The overhead incurred by real-time GPU applications due to BWLOCK++ comes from the following sources:

- `LD_PRELOAD` overhead for CUDA API instrumentation
- Overhead due to BWLOCK++ system call

The overhead due to `LD_PRELOAD` is negligible since we cache CUDA API symbols for all the instrumented functions inside our shared library; after searching for them only once through the dynamic linker. We calculate the overhead incurred due to BWLOCK++ system call by executing the system call one million times and taking the average value. In NVIDIA Jetson TX2, the average overhead due to each BWLOCK++ system call is 1.84-usec. Finally, we experimentally determine the overhead value for all the evaluated benchmarks by running the benchmark in isolation with and without BWLOCK++. Our experiment shows that for all the evaluated benchmarks, the total overhead due to BWLOCK++ is less than 1% of the total solo execution time of the benchmark.

3.6 Schedulability Analysis

As we limit the scheduling of real-time tasks on a single real-time core, our system can be analyzed using the classical unicore based response time analysis for preemptive fixed priority scheduling with blocking [26], because we model each GPU execution segment as a critical section, which is

protected by acquiring and releasing the bandwidth lock. The bandwidth lock serializes GPU execution and regulates memory bandwidth consumption of co-scheduled best-effort CPU tasks. The bandwidth lock implements the non-preemptive locking protocol [81], which boosts the priority of the lock holding task (i.e., the task executing a GPU kernel) to the ceiling priority of the system, which is the highest possible real-time priority, so as to prevent preemption. With this constraint, a real-time task τ_i's response time is expressed as:

$$R_i^{n+1} = E_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil E_j \tag{3.4}$$

where $hp(i)$ represents the set of higher priority tasks than τ_i and B_i is the longest GPU kernel or copy duration—protected by the memory bandwidth lock—of one of the lower priority tasks.

The benefit of BWLOCK++ lies in the reduction of worst-case GPU kernel execution and GPU memory copy interval of real-time tasks (which would in turn reduce E_i and B_i terms in Equation 3.4). As shown in Section 3.5.2, without BWLOCK++, GPU execution of a task can suffer severe slowdown (up-to 230% in our evaluation), which results in pessimistic WCET estimation for GPU kernel and copy execution times, reducing schedulability of the system. BWLOCK++ helps reduce pessimism of GPU execution time estimation and thus improves schedulability.

3.7 Discussion

Due to the choices and assumption inherent in its design, BWLOCK++ has the following limitations. First, we assume that all real-time tasks are scheduled on a single dedicated real-time core while the rest of the cores only schedule best-effort tasks. In addition, we assume only real-time tasks can utilize the GPU while best-effort tasks cannot. While restrictive, recall that scheduling multiple GPU using real-time tasks on a single dedicated real-time core does not necessarily reduce GPU utilization because multiple GPU kernels from different tasks (processes) are serialized at the GPU hardware anyway [66] as we already discussed in Section 3.2. Also, due to the capacity limitation of embedded GPUs, it is expected that a few GPU using real-time task can easily achieve

high GPU utilization in practice. We claim that our approach is practically useful for situations where a small number of GPU accelerated tasks are critical, for example, a vision-based automatic braking system.

Second, we assume that GPU applications are given a priori and they can be profiled in advance so that we can determine proper memory bandwidth threshold values. If this assumption cannot be satisfied, an alternative solution is to use a single threshold value for all GPU applications, which eliminates the need of profiling. But the downside is that it may lower the CPU throughput because the memory bandwidth threshold must be conservatively set to cover all types of GPU applications.

3.8 Conclusion

In this chapter, we presented BWLOCK++, a software based mechanism for protecting the performance of GPU kernels on platforms with integrated CPU-GPU architectures. BWLOCK++ automatically instruments GPU applications at run-time and inserts a memory bandwidth lock, which throttles memory bandwidth usage of the CPU cores to protect performance of GPU kernels. We identified a side effect of memory bandwidth throttling on the performance of Linux default scheduler CFS, which results in the reduction of overall system throughput. In order to solve the problem, we proposed a modification to CFS, which we call Throttle Fair Scheduling (TFS) algorithm. Our evaluation results have shown that BWLOCK++ effectively protects the performance of GPU kernels from memory intensive CPU co-runners. Also, the results showed that TFS improves system throughput, compared to CFS, while protecting critical GPU kernels.

Chapter 4

Real-Time Gang Scheduling on Multicore CPUs[1]

In this chapter, we present RT-Gang: a novel real-time gang scheduling framework that enforces a *one-gang-at-a-time* policy. We find that, in a multicore platform, co-scheduling multiple parallel real-time tasks would require highly pessimistic worst-case execution time (WCET) and schedulability analysis—even when there are enough cores—due to contention in shared hardware resources such as cache and DRAM controller.

In RT-Gang, all threads of a parallel real-time task form a real-time gang and the scheduler globally enforces the one-gang-at-a-time scheduling policy to guarantee tight and accurate task WCET. To minimize under-utilization, we integrate a state-of-the-art memory bandwidth throttling framework to allow safe execution of best-effort tasks. Specifically, any idle cores, if exist, are used to schedule best-effort tasks but their maximum memory bandwidth usages are strictly throttled to tightly bound interference to real-time gang tasks.

We implement RT-Gang in the Linux kernel and evaluate it on two representative embedded multicore platforms using both synthetic and real-world DNN workloads. The results show that RT-Gang dramatically improves system predictability while incurring negligible runtime overhead.

4.1 Introduction

In a safety-critical real-time system, the use of high-performance multicore platforms is challenging because shared hardware resources, such as cache and memory controllers, can cause extremely

[1] Contents of this chapter have previously appeared in the following publication:
[85] **Waqar Ali** and Heechul Yun (2019). RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems. In *Proceedings of the 25th IEEE International Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 143–155

high timing variations [86, 3]. The timing unpredictability is a serious problem in both automotive and aviation industries. For example, Bosch, a major automotive supplier, recently announced "predictability on high-performance platforms" as a major industrial challenge for which the industry is actively seeking solutions from the research community [87]. In aviation, the problem was dubbed as **"one-out-of-m"** problem [72] because the current industry practice is to disable all but one core as recommended by the Federal Aviation Administration (FAA) for certification, which requires *evidence of bounded interference* [6].

Prior efforts to address the timing predictability problem have been largely based on two fundamental ideas: (1) designing simpler time-predictable architectures and (2) partitioning shared resources. Unfortunately, simpler architectures tend to trade-off too much performance in favor of predictability, which we can no longer afford. Partitioning shared resources improves predictability but cannot guarantee tight worst-case timing in high-performance multicore architectures because there are too many important but unpartitionable shared resources [3]. Moreover, not only partitioning generally reduces efficiency and maximum achievable performance, but also it is very difficult to partition properly for parallel tasks, while many emerging real-time workloads, such as deep neural network (DNN) [88, 89], are highly parallel.

Motivated by these trends, we present RT-Gang: a novel real-time gang scheduling framework that can *efficiently* and *predictably* utilize modern high-performance multicore architectures for safety-critical real-time systems. We focus on supporting emerging *parallel* real-time workloads, such as DNN-based real-time sensing and control tasks [88, 89]. *Our key observation is that, from the worst-case execution time (WCET) standpoint, scheduling one parallel real-time task at a time is better than co-scheduling multiple parallel real-time tasks*, because the latter case must assume highly pessimistic WCETs on multicore architecture (see Section 4.2).

In RT-Gang, all threads of a parallel real-time task form a real-time gang and the scheduler globally enforces a one-gang-at-a-time scheduling policy to guarantee tight and accurate task WCET. Assuming all real-time tasks are parallelized and each parallel task can fully take advantage of all the available computing resources on the platform, this one-gang-at-a-time approach

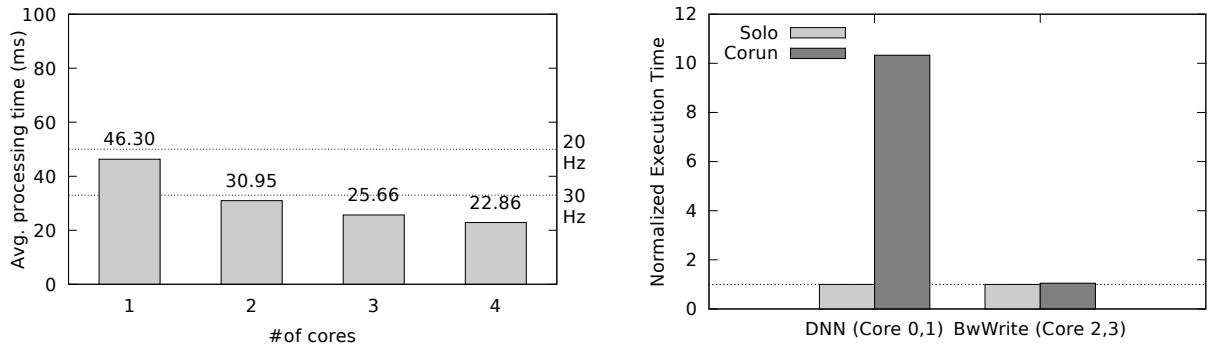
essentially renders parallel real-time task scheduling on multicore into the well-understood single-core real-time scheduling problem [90]. The most significant benefit of this transformation is that we no longer need to worry about shared resource interference because all scheduled threads are part of a single real-time task and that resource sharing is *constructive* rather than destructive. Assuming the WCET of a parallel real-time task is estimated in isolation, RT-Gang guarantees that the WCET will be respected regardless of other tasks on the system. Furthermore, we can apply well-understood single-core based real-time task scheduling policies and analysis methodologies [28, 91, 26] without making strong assumptions on the underlying multicore architectures.

Assuming all real-time tasks are perfectly parallelized is, however, unrealistic. Many real-time tasks are difficult or impossible to parallelize. Also, parallelized code often does not scale well. Therefore, our one-gang-at-a-time policy can significantly under-utilize computing resources when imperfectly parallelized or single-threaded real-time tasks are scheduled one-at-a-time. We mitigate this problem by allowing co-scheduling of best-effort tasks on any of the available idle system cores but with a condition that the cores are strictly regulated by a memory bandwidth throttling mechanism [60]. Each real-time gang defines its tolerable maximum memory bandwidth, which is then strictly enforced by the throttling mechanism to ensure bounded interference to the real-time gang task.

We implement RT-Gang in Linux kernel and evaluate it on two representative embedded multicore platforms, NVIDIA Jetson TX-2 and Raspberry Pi 3, using both synthetic and real-world workloads. The results show that RT-Gang dramatically improves system predictability while the measured overhead is negligible. In the following, we begin our discussion by describing a case-study to illustrate the severity of the shared-resource contention problem in COTS multicore platforms; to motivate the need of the restrictive scheduling of RT-Gang framework.

4.2 Case-Study: Non-Determinism in Multicores

In this section, we provide evidence that shows why scheduling one real-time gang at a time can be better from the perspective of task WCETs through a case-study.



(a) Effect of DNN parallelization

(b) Effect of co-scheduling

Figure 4.1: (a) Average control loop execution time vs. # of CPU cores; (b) performance impact of co-scheduling (DNN on Core 0,1; BwWrite, a memory benchmark [3], on Core 2,3)

We use a deep neural network (DNN) based real-time control task of DeepPicar [92] as our workload. The control loop uses a single DNN to produce the car's steering angle control output from raw images of the car's front-facing camera in real-time. Importantly, its DNN architecture is the same as the one used in NVIDIA's real self-driving car called DAVE-2 [88].

Note that DNN processing is highly compute and data intensive, which is thus often *parallelized* to improve performance. Figure 4.1a shows the average execution times of the control loop while varying the number of CPU cores used on a quad-core embedded platform (Raspberry Pi 3). It can be seen that as we assign more cores for DNN processing, the performance improves—from 46.30-msec on a single core to 22.86-msec on four cores. If the real-time requirement is 30-Hz, one might want to parallelize the DNN using two cores, while *co-scheduling* other tasks on the other two remaining cores. Figure 4.1b shows the timing impact of such co-scheduling, where the DNN control task and a memory intensive task are scheduled first in isolation (*Solo*) and then together (*Corun*). The interesting, and troubling, observation is that the two tasks experience dramatically different timing impact: the DNN control task suffers 10.33x slowdown, while the memory benchmark suffers only 1.05x slowdown.

For safety-critical real-time systems, *this means that extremely pessimistic task WCETs must be assumed to be safe*. The potential timing impact of interference highly depends on task's memory access patterns and the underlying hardware. For example, we observed more than 100x slowdown

(two orders of magnitudes) using a linked-list iteration task on the same computing platform used above, *even after* we partition core as well as the shared cache among the tasks. Similar degrees of timing impacts of interference have been reported in recent empirical studies on contemporary embedded multicore platforms [84, 3, 86, 92, 4].

When a task's WCET has to be assumed 10x or 100x of its solo execution time, we can see why in aviation industry, it makes sense to disable all but one core [72] and why this practice is recommended by the certification authorities [6, 5]. However, disabling cores obviously defeats the purpose of using multicore platforms in the first place—the need of more performance.

In our DNN control-task case-study above, a better approach is to use all four cores just for the parallelized DNN processing task—without co-scheduling—which would result in quicker completion of the control task. More importantly, because there would be no other competing co-scheduled tasks, there's no need to pessimistically estimate the task's WCET. This, in turn, will achieve better overall schedulability. In this sense, from the WCET standpoint, scheduling fully parallelized tasks one-at-a-time might be better than co-scheduling multiple of them at the same time. Generally applying this approach, however, has two major issues. First, not all real-time tasks can be easily parallelized. Second, parallelization often does not scale due to synchronization and other overheads. Therefore, the danger is that some cores may be under-utilized under the one-at-a-time scheduling policy.

In summary, we have made a case why, from the WCET standpoint, scheduling one parallel real-time task at a time can be better than co-scheduling multiple parallel tasks simultaneously, although possible under-utilization of the computing resources is a concern.

4.3 Design Overview

In this section, we describe the overall design of the RT-Gang framework.

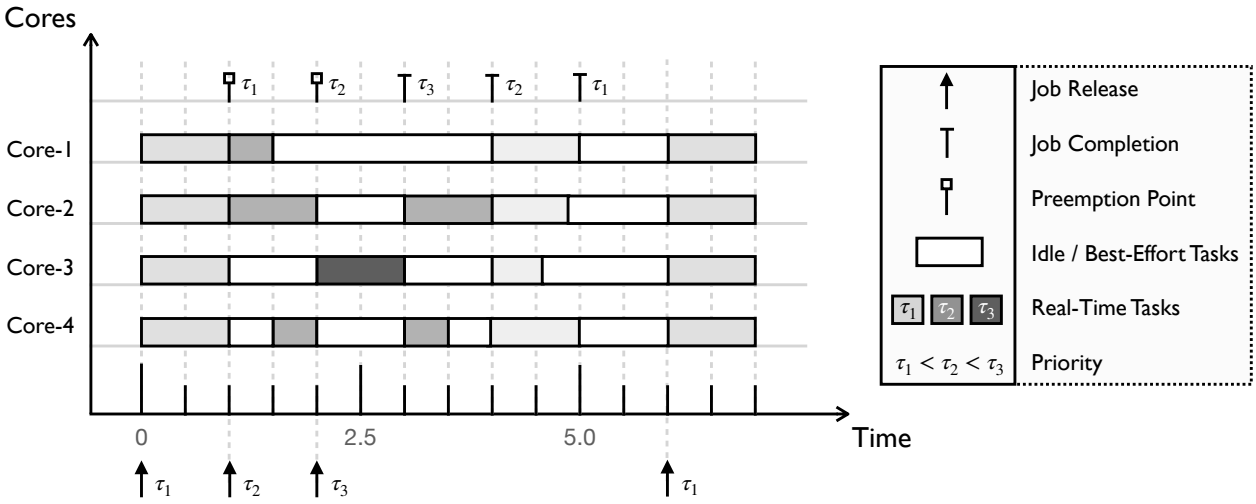


Figure 4.2: Illustration of the one-gang-at-a-time scheduling policy of RT-Gang

4.3.1 One-Gang-at-a-Time Policy

RT-Gang is based on a simple idea: *schedule one real-time task—parallel or not—at a time.* When a real-time task is released, all of its threads, called a *gang*, shall be scheduled all at once—if the task is the highest priority one—or not at all—if the task's priority is lower than the currently scheduled real-time task—even if that leaves some cores idle. In other words, we implement a version of gang scheduler, but unlike prior gang scheduler designs [22, 23, 24], we do not allow co-scheduling of other real-time tasks even when there are idle cores. We do allow, however, co-scheduling of best-effort tasks with strictly imposed limits on their allowed memory access rates by integrating an existing memory bandwidth throttling mechanism [60, 8].

In our approach, each real-time task declares its maximum allowed interfering memory traffic from the best-effort tasks on different cores, which is then enforced by the OS at runtime for the cores that schedule best-effort tasks, if such cores exist. In this way, the parallel real-time task—i.e., a real-time gang—is guaranteed to be able to use all available computing resources and the maximum interference is strictly limited to a certain threshold value, determined by the task itself in advance. If the real-time gang needs maximum isolation, it can set its interference threshold value to be zero, preventing any co-scheduling of best-effort tasks.

Figure 4.2 shows an example schedule under RT-Gang framework. In this example, three real-

time tasks τ_1, τ_2, and τ_3 (in increasing priority) are scheduled. The task τ_1 has four threads, while τ_2 and τ_3 have three threads and one thread, respectively.

At first, τ_1 is scheduled. When τ_2 is released, it preempts τ_1 because it has higher priority. Note that even if Core 3 and 4 are idle at the time, τ_1 cannot use the cores. When τ_3, a single-threaded task, becomes ready, all cores except Core 3 become idle to guarantee that τ_3 is the only real-time task in the entire system. In this way, our real-time gang scheduler strictly enforces the one real-time gang at a time policy.

Note that the solid white rectangles are *slack-durations* during which best-effort tasks can be scheduled, using a general purpose scheduler, such as Linux's Completely Fair Scheduler (CFS) [93], but they will be throttled based on each real-time task's declared tolerable threshold value.

RT-Gang's design approach offers several major benefits. First, by scheduling one real-time gang at a time, we no longer need to worry about interference from other real-time tasks. Possible interference from best-effort tasks is strictly regulated by the threshold value determined by the task itself. Thus, we can obtain accurate WCET of a real-time task (e.g., measure the timing while injecting the threshold amount of memory traffic). Also, as shown in [61], we can obtain better analytic memory interference bounds when we control the amount of competing memory traffic. In other words, we no longer need to deal with highly pessimistic 10x or 100x WCETs. An equally desirable aspect of this approach is that it renders the problem of scheduling parallel real-time tasks on multicore as the simple, well-understood classical real-time scheduling problem on single-core systems [28, 91]. Thus, we can directly apply classical single-core analysis methods [26].

4.3.2 Safe Best-Effort Task Co-Scheduling

Because our real-time gang scheduling approach strictly disallows concurrent real-time tasks, which are not part of the currently scheduled real-time gang, it is possible that some cores may be idle. As we discussed earlier, we allow scheduling of best-effort tasks on those idle cores with a condition that their interference is strictly bounded by integrating a memory bandwidth throttling

mechanism as found in [60].

The throttling mechanism in [60] uses per-core hardware performance counters to bound the maximum number of memory transactions within a given time interval (e.g., `1-msec` period) to a certain programmable threshold (budget). When the core reaches the programmed threshold, the hardware counter generates an overflow interrupt, which is then handled by the OS to stop the core until the next time period begins.

Assuming that the currently scheduled real-time gang is actively using k cores out of m cores ($k \leq m$), there are $m-k$ idle cores on which we can schedule best-effort tasks—i.e., those that do not have hard real-time requirements. The best-effort tasks scheduled on the idle cores are given strict limits in terms of their memory bandwidth usage so that their impact to the real-time gang is bounded. The bandwidth limit of the best-effort tasks is determined by the currently scheduled real-time gang in the system. When the real-time gang is being scheduled on k cores, all the $m-k$ cores are throttled according to the bandwidth threshold of the gang.

4.4 Illustrative Example

In this section, we provide an illustrative example to compare scheduling under one-gang-at-a-time policy with a traditional co-scheduling approach on a multicore platform.

Task	WCET (C)	Period (P)	# of Threads
τ_1^{RT}	2	10	2
τ_2^{RT}	4	10	2
τ_3^{BE}	∞	N/A	4

Table 4.1: Taskset parameters of the illustrative example

We assume that our multicore platform has four homogeneous CPU cores. We want to schedule a parallel taskset, shown in Table 4.1, in the system. τ_1^{RT} and τ_2^{RT} are real-time tasks. τ_3^{BE} is a best-effort task, which is scheduled under the CFS scheduler of Linux. For the sake of simplicity, we assume that all threads of a task have the same compute time and are released simultaneously at the

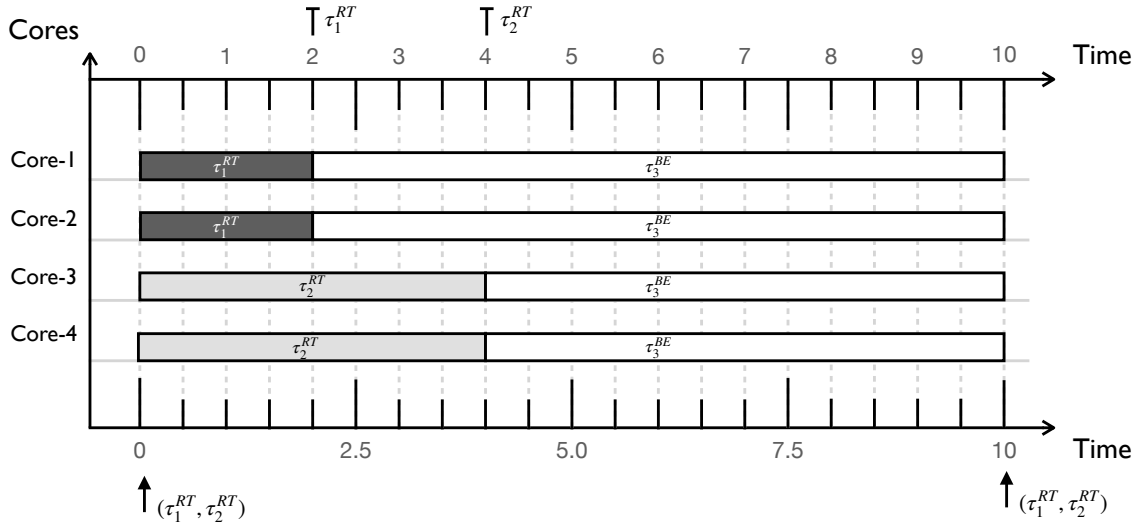


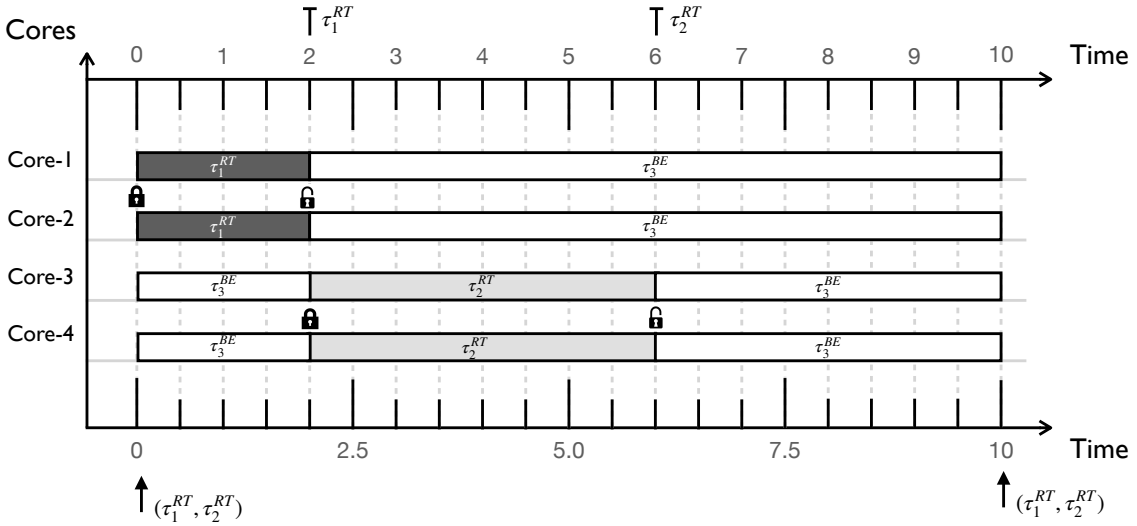
Figure 4.3: Example schedule of a co-scheduling scheme (w/o interference)

start of the period. We assume that all threads are statically pinned to specific CPU cores and they do not migrate during their execution. The OS scheduler tick interval is assumed to be 1-msec.

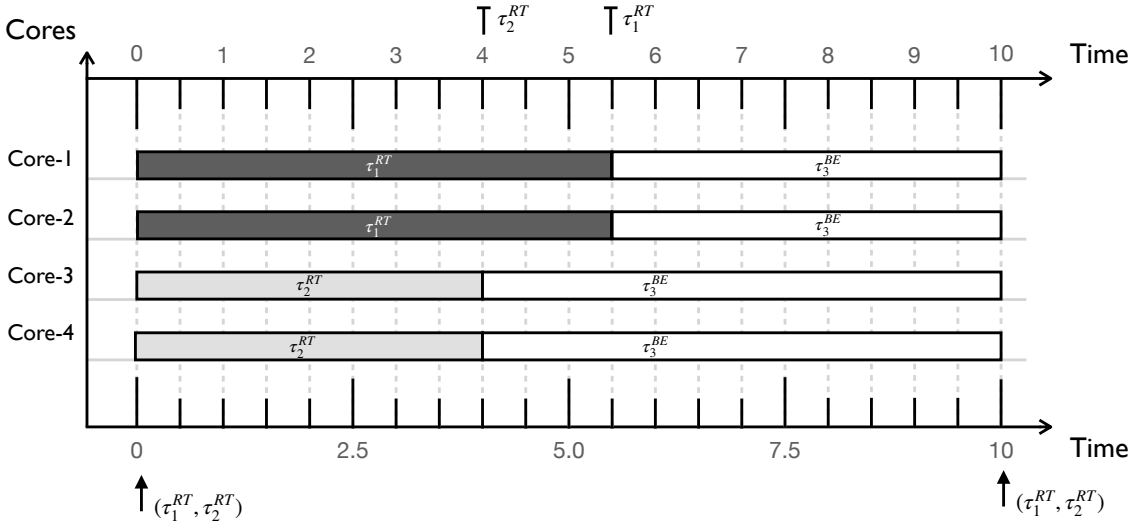
Figure 4.3 shows the scheduling timeline under a traditional co-scheduling approach. For this figure, we assume that tasks on different cores do not suffer interference due to contention in shared hardware resources. Under co-scheduling, τ_1^{RT} completes its execution at 2-msec. This leaves 8-msec *slack duration* on the two cores on which this task was executing. Similarly, τ_2^{RT} leaves a slack duration of 6-msec on its cores. Considering the system as a whole, the total slack duration in this schedule is 28-msec which can be used to schedule the best-effort task τ_3^{BE}.

Figure 4.4a shows the scheduling timeline under the one-gang-at-a-time policy of RT-Gang. Under this schedule, τ_1^{RT} gets to run first. While τ_1^{RT} is executing, τ_2^{RT} is blocked. Once τ_1^{RT} finishes its execution at 2-msec mark, τ_2^{RT} starts executing and completes at 6-msec mark. Under this scheme, the total slack duration left in the system is again 28-msec, assuming that τ_3^{BE} is not throttled (i.e., its memory bandwidth usage is less than the allowed threshold).

Now we consider the case where the real-time tasks can destructively interfere with each other due to resource contention. Borrowing from the DNN case-study introduced in previous subsection, we assume that the execution time of τ_1^{RT} increases 10x, when it is co-scheduled with τ_2^{RT}. τ_2^{RT}, on the other hand, stays unaffected under co-scheduling. Figure 4.3 shows the scheduling



(a) Example schedule under one-gang-at-a-time policy



(b) Example schedule of a co-scheduling scheme (with interference)

Figure 4.4

timeline for this case under co-scheduling scheme. As can be seen from the figure, while τ_2^{RT} is executing, the progress of τ_1^{RT} would be slowed due to interference. At 4-msec mark when τ_2^{RT} finishes its execution, τ_1^{RT} has only made 20% progress and it still has 1.6-msec of its original compute time left. For this reason, τ_1^{RT} completes its execution at 5.6-msec. In this case, the total slack duration for best-effort tasks is 20.8-msec.

Under one-gang-at-a-time policy of RT-Gang, the scheduling timeline of the real-time tasks remains **the same** as the one shown in Figure 4.3 because τ_1^{RT} and τ_2^{RT} can never run at the same

time. In other words, *regardless of task and hardware characteristics, real-time tasks' execution times would remain the same*. The slack duration remains unchanged as well, which can be utilized for scheduling best-effort tasks although they are strictly regulated with a memory bandwidth throttling mechanism, shown as the "locked" duration in Figure 4.4a.

4.5 Implementation

Data-Structure 1 RT-Gang in Linux

```
struct glock
    spinlock_t        lock;
    bool              held_flag;
    bitmask           locked_cores;
    bitmask           blocked_cores;
    task_struct_t*    leader;
    task_struct_t*    gthreads[NR_CPUS];
```

The implementation of RT-Gang, in Linux, revolves around the data-structure `struct glock` shown in Listing 1. This data-structure is used for the following main purposes:

- Quickly check whether the gang scheduling lock is currently being held (*held_flag*)
- Track the cores, which are currently running real-time thread using a bitmask (*locked_cores*)
- Track the blocked cores, which have real-time tasks in their queues but cannot execute them due to the gang scheduling policy (*blocked_cores*)

To simplify the implementation complexity, we assume that each real-time gang in our system has a distinct real-time priority value. In the following sections, we explain the main parts of the algorithm in detail and then describe the entire algorithm.

4.5.1 Gang Lock Acquisition

Before a real-time task can get scheduled, it needs to acquire the gang scheduling lock. Algorithm 3 shows the pseudo-code of the lock acquisition function. In this function, the gang-scheduling lock

Algorithm 3: RT-Gang Lock Acquisition Protocol

```
1  function acquire_gang_lock(task_struct_t *next)
2      glock→held_flag = true;
3      set_bit (this_cpu, glock→locked_cores);
4      glock→gthreads [this_cpu] = next;
5      glock→leader = next;
6  return
```

is marked as held by setting the flag in the global `glock` data-structure. The CPU core on which this function is invoked, is marked "locked" by setting its bit inside the `locked_cores` bitmask. The task that acquires the gang lock is marked as the gang-leader and its task pointer is tracked inside an array, which is later used at the time of lock release.

4.5.2 Gang Lock Release

This function is called to release the gang-scheduling lock on behalf of a task which is going out of schedule. The pseudo-code of this function is shown in Algorithm 4. Upon entering this function, it is checked whether the thread going out of execution is part of the currently executing gang. If this condition is true, the current CPU core is marked as unlocked, by clearing its bit from the `locked_cores` bitmask.

Algorithm 4: RT-Gang Lock Release Protocol

```
1  function try_glock_release(task_struct_t *prev)
2      for_each_locked_core (cpu, glock→locked_cores)
3          if (gthreads [cpu] == prev) then
4              clear_bit (cpu, glock→locked_cores);
5              gthreads [cpu] = null;
6              if (mask_is_zero (glock→locked_cores)) then
7                  glock→held_flag = false;
8                  reschedule_cpus (glock→blocked_cores);
9                  clear_mask (glock→blocked_cores);
10             end
11         end
12 return
```

The next condition checked in this function is to make sure if all of the threads belonging to

current gang have finished their execution, which would imply that the gang-lock is now free. This is done by checking if the `locked_cores` bitmask is zero. If this is the case, the gang-scheduling lock is marked free and rescheduling inter-processor interrupt (IPI) is sent to the CPU cores, which have blocked real-time tasks in their ready queues by using the `blocked_cores` bitmask.

4.5.3 Gang Preemption

The purpose of this function is to preempt all threads, which are part of the currently running gang, so that a new higher priority gang may start its execution. The pseudo-code of this function is shown in Algorithm 5. In this function, the `locked_cores` bitmask is traversed to send rescheduling IPIs to all the cores, which are currently marked as locked. Once this is done, the `locked_cores` bitmask is cleared and the threads being tracked in the `gthreads` array are removed.

Algorithm 5: Gang Preemption Protocol under RT-Gang

```
1 function do_gang_preemption()
2     for_each_locked_core (cpu, glock→locked_cores)
3         gthreads [cpu] = null;
4     reschedule_cpus (glock→locked_cores);
      clear_mask (glock→locked_cores);
5 return
```

4.5.4 Main Gang Scheduling Algorithm

The gang-scheduling algorithm resides in the critical path of the main scheduler entry point function (`__schedule`) in Linux and it works by modifying the task selection heuristics of the real-time scheduling class. Algorithm 6 shows the main scheduling function of RT-Gang. The algorithm starts by checking whether gang-scheduling lock is currently being held by any real-time task. If that is the case, the algorithm tries to release the gang-scheduling lock on behalf of the `prev` task which is going out of schedule (*Line-11*).

Algorithm 6: RT-Gang Scheduling Algorithm

```
1  function __schedule(task_struct_t *prev)
2      for_each_sched_class (class)
3          next = class→pick_next_task (prev);
4          if (next) then
5              context_switch (prev, next);
6          end
7  return

8  function *pick_next_task_rt(task_struct_t *prev)
9      spin_lock (glock→lock);
10     if (glock→held_flag) then
11         try_glock_release (prev);
12     end
13     next = rt_queue [this_cpu]→next_task;
14     if (glock→held_flag == false) then
15         acquire_gang_lock (next);
16     else if (next→prio > glock→leader→prio) then
17         do_gang_preemption ();
18         acquire_gang_lock (next);
19     else
20         set_bit (this_cpu, glock→blocked_cores);
21         next = null;
22     end
23     spin_unlock (glock→lock);
24 return next
```

If the gang-scheduling lock is currently free, the algorithm acquires the lock on the current core on behalf of the `next` real-time task (*Line-15*). If, on the other hand, the lock is not free, it is checked whether this task has a higher priority than the gang in execution (*Line-16*). If that is the case, the currently executing gang is preempted (*Line-17*) and the gang-scheduling lock is acquired on behalf of the `next` task (*Line-18*).

If all of the above conditions fail—i.e., the gang-scheduling lock is not free and the `next` real-time task has lower priority, then the `next` task is deemed ineligible for scheduling. In this case, the current CPU core is marked as blocked by setting its bit in the `blocked_cores` bitmask (*Line-20*) and the `next` task pointer is set to null so that no real-time task is returned to the scheduler by the real-time scheduling class. Finally, the spinlock is released (*Line-23*) and control is returned to the

scheduler (*Line-24*); to either schedule the next real-time task (if any) or go to the next scheduling class (CFS) to pick a best-effort task.

4.5.5 Memory Bandwidth Throttling of Best-Effort Tasks

We incorporated the memory bandwidth throttling mechanism of BWLOCK++, as described in Section 3.4, with certain modifications, into RT-Gang to allow safe co-scheduling of best-effort tasks while a real-time gang is executing. We update the BWLOCK++ system call (Section 3.4.1) such that instead of providing a binary value to start/stop throttling, the caller provides the acceptable memory threshold value for the real-time gang in execution. This value is stored in the task structure of the RT-Gang leader as an integer. We also modify the framework such that in every regulated interval, the memory bandwidth threshold value of the executing gang is automatically enforced on all CPU cores executing best-effort tasks. In this manner, we ensure that the real-time gang is protected from unbounded interference from best-effort tasks.

4.6 Evaluation

We evaluate RT-Gang on two embedded platforms: Raspberry Pi 3 and NVIDIA Jetson TX2. Raspberry Pi 3 is equipped with a Cortex-A53 based quad-core CPU, which is representative of an energy efficient low-cost embedded multicore processor, while NVIDIA Jetson TX2 is equipped with a six-core heterogeneous CPU (4X Cortex-A57 and 2X Denver [2]), which is representative of a high-end embedded processor.

On both platforms, we use Linux 4.4 kernel and implement RT-Gang by modifying its real-time scheduler (`kernel/sched/rt.c`). Our modification is about 500 lines of architecture neutral C code. In all our experiments, we place the platform in the maximum performance mode and disable the dynamic frequency scaling of CPU cores. We also shutdown the graphical user interface and disable networking to minimize run to run variation in the experiments.

[2]We do not use the Denver complex in our experiments due to its lack of hardware counter support needed to implement throttling mechanism [8]

4.6.1 Synthetic Workload

In this experiment, we show the benefits of RT-Gang using a synthetic taskset on Raspberry Pi 3. The taskset is composed of two periodic multi-threaded real-time tasks (τ_1 and τ_2) and two single-threaded best-effort tasks (τ^{BE}_{mem} and τ^{BE}_{cpu}). Using this taskset, we explore task execution time impacts to the real-time tasks and throughput impacts to the best-effort tasks.

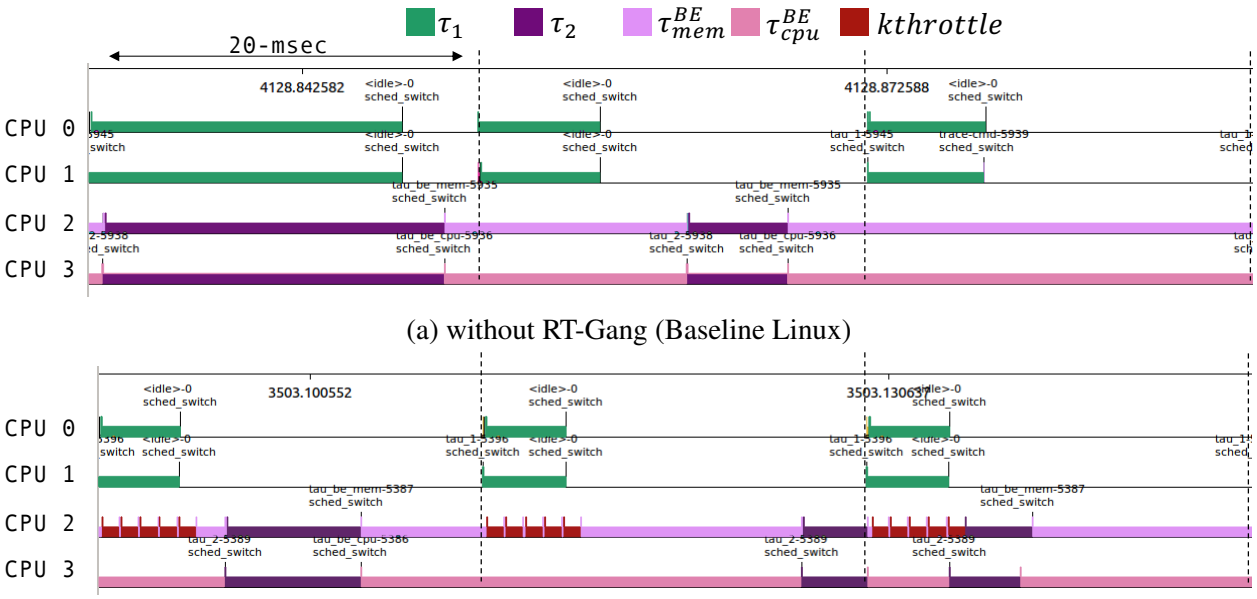
We use a modified version of the *bandwidth* benchmark—referred to as BwRead due to its memory read access pattern—from the IsolBench [3] benchmark suite to construct the taskset [3]. Concretely, τ_1 creates two threads and is configured to use three quarters of the last-level cache size (384KB out of 512KB L2 cache of the Pi 3) as its working-set (which is shared between the two threads). It is periodically released at a 20-msec interval and each job takes about 3.5-msec in isolation. Similarly, τ_2 is also a dual-threaded periodic real-time task with the same working-set size (384KB), but differs in its period (30-msec) and job execution times (6.5-msec in isolation). We set the priority of τ_1 higher than that of τ_2, and schedule τ_1 on Core 0,1 and τ_2 on Core 2,3 (pinned using CPUSET utility).

Note that τ_1 and τ_2 are scheduled by SCHED_FIFO real-time scheduler with and without RT-Gang enabled [4]. On the other hand, τ^{BE}_{mem} and τ^{BE}_{cpu} are both single-threaded best-effort tasks, which are scheduled by the CFS scheduler [93]; they differ in that τ^{BE}_{mem}'s working-set size is two times bigger than the L2 cache size, while τ^{BE}_{cpu}'s working-set is smaller than the core's private L1 cache size. Thus, τ^{BE}_{mem} may cause interference to co-scheduled real-time tasks (if any) on different cores, due to contention in the shared L2 cache, while τ^{BE}_{cpu} may not. We collect the execution traces of the taskset without and with RT-Gang for a duration of 10 seconds using the `trace-cmd` utility and the KernelShark [94] tool in Linux.

Figure 4.5 shows the execution traces. In inset (a), during the first 20-msec duration, τ_1 and τ_2 were overlapped with each other, which resulted in significant job execution time increase for both tasks because their working-sets could not fit into the shared L2 cache simultaneously. During the

[3] Our additions include support for multi-threaded and periodic task invocation. The code can be found in the IsolBench repository [83].

[4] RT-Gang can be enabled or disabled at runtime via: `/sys/kernel/debug/sched_features`



(a) without RT-Gang (Baseline Linux)

(b) with RT-Gang

Figure 4.5: Task execution traces. $\tau_1(C_1 = 3.5, P_1 = 20)$, $\tau_2(C_2 = 6.5, P_2 = 30)$: parallel RT tasks (2 threads / task); τ^{BE}_{mem}, τ^{BE}_{cpu}: memory and cpu intensive best-effort tasks respectively; *kthrottle*: injected kernel thread for throttling

next 20-msec duration, although τ_1 and τ_2 did not overlap, τ_1 was overlapped with the two best-effort tasks, τ^{BE}_{mem} and τ^{BE}_{cpu}, which also resulted in a significant increase in τ_1's job execution time (albeit to a lesser degree). In inset (b), on the other hand, RT-Gang almost completely eliminates job execution time variance. In the first 20-msec duration, τ_1 is overlapped with the two best-effort tasks. However, τ^{BE}_{mem} (the memory intensive one) was throttled most of the time, which protected the execution time of τ_1. At around 40-msec in the timeline, τ_1 preempted τ_2, the real-time task. In place of τ_2, two best-effort tasks were scheduled, of which τ^{BE}_{mem} was again throttled as per τ_1's specified memory bandwidth threshold setting.

Note that in RT-Gang, the execution times of τ_1 is deterministic. Furthermore, τ_2 is also highly predictable as we only need to consider the preemption by the higher priority τ_1, according to the classic response time analysis (RTA)[26]. Furthermore, because the two real-time tasks do not experience significant execution time increases, there are more "slacks" left for the best-effort tasks—compared to without using RT-Gang in inset (a)—which improves throughput.

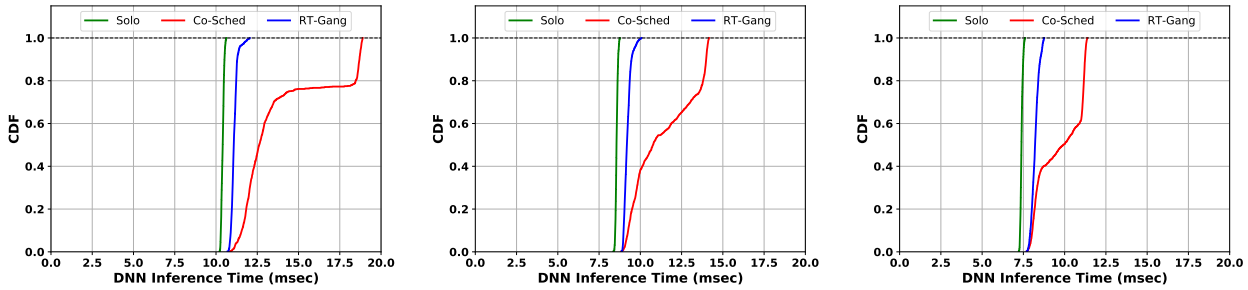
4.6.2 DNN Workload

To establish the practicality of RT-Gang for real-world safety critical applications, we used the DNN workload introduced in Section 4.2 and executed it under different configurations on Raspberry Pi 3 and NVIDIA Jetson TX2, with and without RT-Gang. The Cortex-A53 cores in Raspberry Pi 3 are much less capable than the four Cortex-A57 cores in Jetson TX2 platform. Moreover, the memory system in Raspberry Pi 3 offers significantly less bandwidth than the one in Jetson TX2. Consequently, co-scheduled workloads in Raspberry Pi 3 are much more prone to the negative effects of resource contention. By evaluating RT-Gang on these two disparate platforms, we demonstrate its applicability to the range of embedded devices available today.

Task	WCET (C ms)	Period (P ms)	# Threads
Common			
τ^{BE}_{cutcp}	∞	N/A	4
τ^{BE}_{lbm}	∞	N/A	4
Jetson TX2			
τ^{RT}_{bww}	40.0	100.0	4
$\tau^{RT}_{dnn(2)}$	10.7	24.0	2
$\tau^{RT}_{dnn(3)}$	8.8	19.0	3
$\tau^{RT}_{dnn(4)}$	7.6	17.0	4
Raspberry Pi 3			
τ^{RT}_{bww}	47.0	100.0	4
$\tau^{RT}_{dnn(2)}$	34.0	78.0	2
$\tau^{RT}_{dnn(3)}$	27.90	65.0	3
$\tau^{RT}_{dnn(4)}$	24.81	56.0	4

Table 4.2: Taskset parameters for the DNN experiment.

The taskset parameters for this experiment are shown in Table 4.2. First, we use a multi-threaded DNN application as the high priority periodic real-time task ($\tau^{RT}_{dnn(c)}$ where c denotes the number of cores used). The period of DNN inference operation is selected to keep the per-core utilization of DNN threads close to 45%. Second, as a lower priority real-time task, we use a periodic multi-threaded BwWrite benchmark (τ^{RT}_{bww}). The working set size of BwWrite was chosen to be twice the size of LLC in each platform so as to stress memory subsystem (due to cache



(a) DNN (2 Core) (b) DNN (3 Cores) (c) DNN (4 Cores)

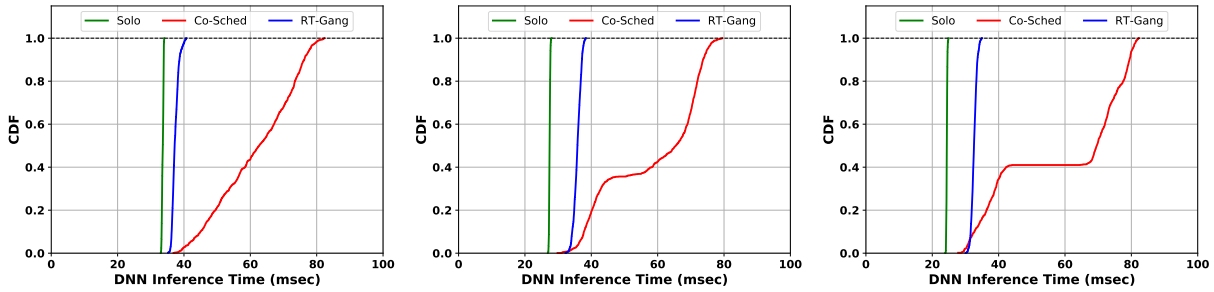
Figure 4.6: Performance of DNN inference loop on Jetson TX2 with RT-Gang

misses). The compute time (C) of BwWrite is selected to keep its per-core utilization less than 50% in all experiments. Lastly, we use two benchmarks from Parboil suite [76], `lbm` (τ^{BE}_{lbm}) and `cutcp`, (τ^{BE}_{cutcp})) as best-effort tasks, which represent memory and CPU intensive parallel OpenMP applications respectively.

Drawing from taskset example shown in Section 4.6.1, LBM represents a memory intensive application that can potentially interfere with the execution of high priority DNN task. Cutcp, on the other hand, represents a compute intensive application which is innocuous to the DNN inferencing operation. We use OpenMP versions of these benchmarks so that they may utilize any slack duration left on any system cores. We vary the thread count (= # of assigned CPU cores) of the DNN task while keeping the thread count of τ^{RT}_{bww} and the best-effort tasks (τ^{BE}_{lbm} and τ^{BE}_{cutcp}) fixed at four. For the experiment, we first measure the performance of $\tau^{RT}_{dnn(c)}$ in isolation, then co-scheduled with the rest of the taskset on baseline Linux, and finally using RT-Gang.

Figure 4.7 shows the cumulative distribution function (CDF) of the per-frame DNN inference time in each configuration (*Solo*: alone in isolation, *Co-Sched*: co-scheduled under baseline Linux, *RT-Gang*: co-scheduled under RT-Gang enabled Linux). Note first that execution times of the DNN task vary significantly under the co-scheduling scheme (*Co-Sched*). On Raspberry Pi 3, the WCET across all configurations is more than 2x of its solo WCET. On TX2, the co-scheduling graphs again show deteriorated performance, albeit to a lesser extent than Raspberry Pi 3.

Under RT-Gang, on the other hand, the execution times of the DNN workload are highly deterministic and match closely with its solo execution times in all tested configurations on both



(a) DNN (2 Core) (b) DNN (3 Cores) (c) DNN (3 Cores)

Figure 4.7: Performance of DNN inference loop on Raspberry Pi 3 with RT-Gang

platforms. On Raspberry Pi 3, however, although DNN's performance is deterministic (i.e., low variance), noticeable performance constant increase is observed under RT-Gang when compared to the solo execution of the DNN task. We believe that this is caused by Cache Related Preemption Delay (CRPD) [95], as the memory system of Raspberry Pi 3 is significantly less powerful than that of Jetson TX2.

Note that taking CRPD into analysis is well known in the context of single-core processors, but its applications have been difficult in multicore due to cache contention among the co-scheduled tasks on different cores, as shown in the CDF plots of *Co-Sched*. The determinism that RT-Gang brings thus would make CRPD analysis valid on multicore processors again, enabling effective timing analysis.

4.6.3 Overhead

There are two main sources of overhead in our implementation of RT-Gang: 1) The serialization overhead associated with the critical section of our gang scheduling algorithm in selecting next real-time task. 2) The overhead involved in sending cross-core interrupts (IPIs) and acquiring ready-queue locks for gang preemption.

The serialization overhead of RT-Gang is only incurred during the selection of real-time tasks due to the use of a spinlock. However, the length of the critical section of RT-Gang is small—comparable to existing spinlocks used in the various parts of the Linux kernel scheduler. On the other hand, the overhead associated with gang-preemption due to the IPIs can pose a scalability

problem as the number of necessary IPIs can be as many as all the rest of the cores.

In order to estimate both these overheads, we conducted an experiment on the NVIDIA Jetson TX2 platform in which a high priority real-time gang preempts a multi-threaded low priority real-time gang, a fixed number of times (100000), with and without RT-Gang. We also varied the number of threads of the low priority gang to see the effect of gang size on the gang preemption overhead. The result from this experiment is shown in Table 4.3.

Scenario	Context Switch Cost (usec)
1-Thread-Lowprio (Linux)	6.81
1-Thread-Lowprio (RT-Gang)	7.19
2-Thread-Lowprio (RT-Gang)	7.37
3-Thread-Lowprio (RT-Gang)	7.55
4-Thread-Lowprio (RT-Gang)	7.72

Table 4.3: RT-Gang Overhead in Linux

As can been seen from the table, RT-Gang adds very small overhead to the overall cost of a context-switch under Linux; considering the fact that for a well-designed system, a context-switch is not supposed to happen too frequently. The findings from this experiment also match the results seen during evaluation with DNN workloads; in which, we saw that the performance of these workloads remain completely unaffected under RT-Gang.

4.7 Discussion

In this section, we briefly discuss potential use-cases of RT-Gang. We believe that our simple design and practical implementation leveraging existing real-time scheduler in Linux offer broader practical use-cases in many application domains that concern timing predictability and need to run parallelized multi-thread applications. Time critical parallel real-time applications in automotive and aviation domains (e.g., perception and control applications in a self-driving car) are our main target use-cases. Also, barrier based scientific applications in high-performance computing (HPC) can potentially benefit from using RT-Gang as they are sensitive to thread imbalance, and thus motivated original gang scheduling research [96] in the first place. Although we mainly used

embedded multicore processors (typically having 4-8 cores) in our description throughout this chapter, we recently were able to apply our RT-Gang kernel patch on a 12 hardware thread (6 core) x86 PC, successfully performing gang scheduling across the 12 hardware threads.

4.8 Conclusion

In this chapter, we presented RT-Gang: a novel real-time gang scheduling framework for predictable and efficient parallel real-time scheduling on multicore. RT-Gang implements a novel gang scheduling policy that eliminates inter-task interference by enforcing an invariant that only one (parallel) real-time task (gang) can be scheduled at any given time. This enables tighter task WCET estimation and simpler schedulability analysis. RT-Gang also provides additional mechanisms, namely best-effort task throttling, which can help maximize system utilization while providing strong time predictability to real-time tasks. We implemented RT-Gang in Linux and evaluated it on two embedded multicore platforms. The evaluation results show the predictability and efficiency benefits of RT-Gang.

Chapter 5

Virtual Gang Scheduling of Parallel Real-Time Tasks[1]

In Chapter 4, we presented the *RT-Gang* framework, which implements a restrictive form of gang scheduling policy for parallel real-time tasks, to address this problem of shared resource contention, by scheduling only one real-time gang task at a time, even if there are enough cores left to accommodate other real-time tasks. This design philosophy of RT-Gang, although solves the problem of shared-resource contention among different real-time tasks, constrains the overall real-time schedulability of the system. Given that parallelization of a task often does not scale well, and more cores are being integrated in modern multicore processors, it is unlikely to be a general solution for many systems.

In this chapter, we introduce the notion of virtual gang—a group of real-time tasks which are treated by the scheduler as a single schedulable entity; just like a real-gang—to mitigate this shortcoming of RT-Gang framework. We rigorously examine the requirements for virtual-gang creation and life-cycle management in a modern operating system and show that synchronous release of all member tasks of a virtual-gang is required for it to be modeled as a single real-time gang. To fulfill this requirement, we design a light-weight intra-gang synchronization framework integrated with OS level page-coloring and memory bandwidth throttling techniques and implement it on top of Linux kernel to provide an end-to-end performance isolation stack for real-time tasks on multicore platforms. We also describe optimal and heuristic algorithms for forming virtual gangs from a given set of real-time tasks. We call the modified RT-Gang framework, equipped with the virtual gang abstraction, the RTG-Sync framework.

[1] Contents of this chapter will appear in the following publication:
[97] **Waqar Ali**, Rodolfo Pellizzoni and Heechul Yun (2021). Virtual Gang based Scheduling of Parallel Real-Time Tasks. In *Proceedings of the 25th IEEE International Conference on Design, Automation and Test in Europe (DATE)*

RTG-Sync provides the following *guarantees* to the members of each virtual gang task: (1) all member tasks are statically determined and do not change over time; (2) no other real-time tasks can be co-scheduled; (3) best-effort tasks can be co-scheduled on any idle cores, but with the following restrictions: (i) their usage of LLC is limited to a static partition via page-coloring so that they do not pollute the working-sets of real-time tasks; (ii) their maximum memory bandwidth usages are strictly regulated to a certain threshold value set by the virtual gang task. These properties greatly simplify the process of determining task WCETs, because once a virtual gang is created, other tasks that do not belong to the virtual gang cannot interfere with the member tasks, regardless of the OS scheduling policy, and the effect of shared hardware resource contention is strictly bounded. In short, RTG-Sync enables compositional timing analysis on multicore platforms[2].

We present evaluation results of RTG-Sync both analytically, with generated task sets against state-of-the-art approaches in a simulation, and empirically with a case-study involving real-world workloads on a real embedded multicore platform. The results from simulation show that schedulability of real-time tasks under RTG-Sync, outperforms state-of-the-art multicore schedulability results in terms of total number of task sets deemed schedulable when shared-resource interference is considered. Even in the absence of interference, the RTG-Sync framework performs better than the state-of-the-art approaches for highly parallel task sets. This observation and the results from the case-study lead us to conclude that our approach provides simple but powerful compositional analysis framework, achieves better analytic schedulability, especially when the effect of interference is considered, and is a practical solution for multicore platforms.

5.1 Requirements for Virtual Gang Scheduling

In this section, we describe the requirements for virtual gang scheduling—synchronous release of gang members and the gang formation problem—and the challenges involved in the process.

[2]Timing analysis of a real-time system is *compositional* if analysis of a component can be carried out independently of other components.

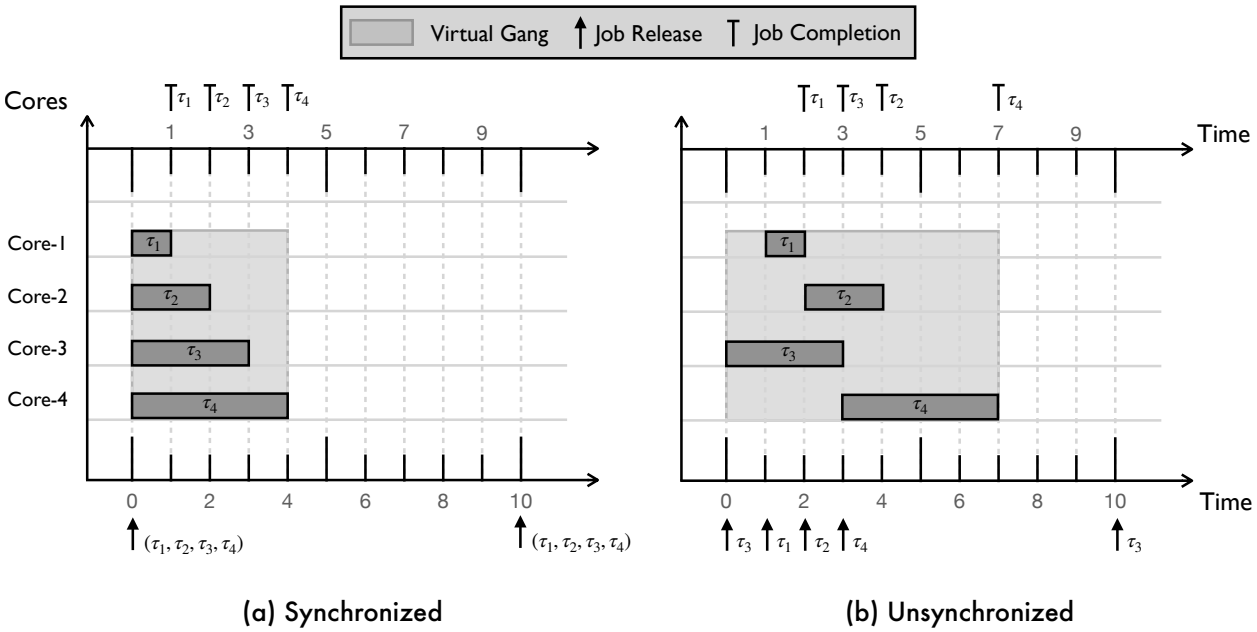


Figure 5.1: Example virtual gang schedules with and w/o synchronization

5.1.1 Need of Synchronization

The first major requirement for virtual gang scheduling is that all member tasks must have equal period and that they must be released synchronously. For the same period requirement, if the selected member tasks of a virtual gang do not share the same period, then the consolidated gang task will not be effectively modeled as a single periodic task from the analysis point of view. Therefore, a virtual gang can only be created when all of its member tasks have a common period.

For the synchronous release requirement, consider a taskset $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ comprising 4 single-threaded tasks, where their WCETs are 1, 2, 3, 4 ms respectively and all of them have the same period of 10-msec, and suppose it is scheduled on a quad-core platform under the one-gang-at-a-time policy. Assuming that these are the only tasks that share the same period in our system and there is no inter-task interference, an intuitive grouping of these tasks into a virtual-gang would be to run them at the same time across all four cores in the system. This results in the execution timeline shown in Figure 5.1(a). In this scheme, the virtual gang completes in just 4-msec, after which other real-time tasks can be scheduled.

However, the execution of the taskset in the virtual gang scheme assumes that the jobs of the

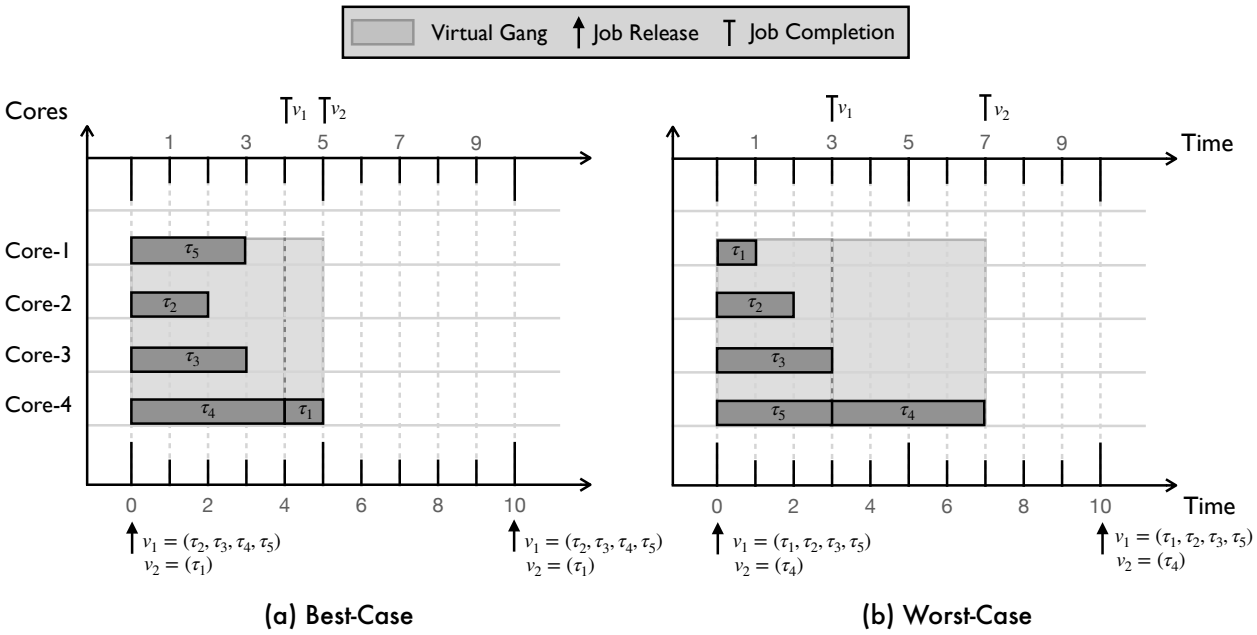


Figure 5.2: Example schedules under different gang formations.

members are perfectly aligned. If this is not the case, then the virtual gang task's execution time will be increased, as shown in Figure 5.1(b), and in the worst-case, it can be as bad as the sequential execution of the tasks under one-gang-at-a-time policy without using virtual gang at all.

5.1.2 Gang Formation Problem

Another major challenge, when creating virtual gangs, is to decide which tasks to group together for concurrent execution.

In the example taskset used in Section 5.1.2, if we add one more task $\tau_5 = (h:1, C:3, P:10)$, then the taskset will have to be split into at least two virtual gangs since all five tasks in the taskset cannot execute simultaneously in our target quad-core system. Hence the problem is to find an optimal grouping of tasks into virtual gangs such that the execution time of the taskset is minimized. For the simple taskset considered here, it can be seen, with a little trial and error, that a virtual gang comprising τ_2, τ_3, τ_4, τ_5 and another one comprising just τ_1 will achieve this goal, resulting in the execution timeline shown in the inset (a) of Figure 5.2.

However if the tasks in a virtual gang are not carefully selected, the execution time of the

taskset can increase significantly. In the example taskset, a virtual gang comprising τ_1, τ_2, τ_3, τ_5 and the other one comprising τ_4 leads to an execution time of 7 time units as compared to 5 time units in the previous case; as can be seen in inset (b) of Figure 5.2.

Given a taskset, the problem of selecting the tasks which should be run together as virtual gangs so that the execution time of the entire taskset is minimized, is non-trivial. The problem is further complicated by the fact that the tasks in a virtual gang can interfere with each other when run concurrently due to shared hardware resource contention, which may require some degree of pessimism in estimating the virtual gang's WCET. Another complication is the possible precedence relationship among the tasks in the taskset, commonly found among the real-time tasks in practical real-world applications, which can constrain the creation of virtual gangs.

Without taking the synchronization and gang formation problems into account, a strategy to improve system utilization via virtual gangs under RT-Gang may not lead to the desired results and may actually deteriorate the system's performance and real-time schedulability.

5.2 System Model

Before describing the design and implementation of the RTG-Sync framework, we present a new system and task model that is suitable for tackling the considered real-time scheduling problems.

We consider a multicore processor based platform π, which contains m unit-speed CPU cores. We consider a system comprising a set Γ of n periodic, rigid gang real-time tasks with implicit deadlines: $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task $\tau_i = (c_i, h_i, r_i, T_i)$ is characterized by its WCET c_i in isolation, the number of cores $h_i \leq m$ needed to execute, the shared resource demand factor r_i in the range $[0, 1]$, and the period T_i. The system also comprises a set of k DAGs $\{G_1, G_2, ..., G_k\}$. Each DAG $G_i = (v_i, e_i, T_i)$ expresses a set of precedence constraints among tasks. The node set $v_i \subseteq \Gamma$ consists of a subset of the tasks in Γ; we assume that all tasks in v_i must have the same period T_i, and no task in Γ can belong to the node set of more than one DAG. The edge set $e_i : v_i \times v_i$ consists of ordered pairs of the form (τ_p, τ_q) describing the precedence constraints among the tasks in v_i: formally, this means that the j-th job of τ_p must finish before the j-th job of τ_q can start executing.

5.2.1 Virtual Gangs and Scheduler

We assume that the number of distinct periods q within Γ is small relative to n. In other words, multiple tasks may share the same period, which is common in practice (e.g., [98]). All tasks that share the same period T forms a candidate-set $\Delta_T = \{\forall \tau_i \in \Gamma \mid T_i = T\}$. A virtual gang w_l is a subset of the tasks within the candidate-set Δ_T that are statically grouped together as a schedulable unit. Each virtual gang $w_l = (C_l, H_l, R_l, T_l)$ is characterized by its WCET C_l, the core requirement H_l and the resource demand R_l; the latter two are equal to the sum of the respective parameters of all of its member tasks.

For the virtual gangs of Δ_T, there must exist a linear ordering between them such that all precedence constraints $\{G_1, G_2, ..., G_k\}$ are satisfied. The virtual gangs of all candidate sets are scheduled according to a preemptive fixed-priority gang scheduling scheme, which schedules one virtual gang at a time on π, subject to the linear ordering of each candidate set. We require all tasks in a candidate-set to be synchronously released and all member tasks of a virtual gang to be scheduled in parallel under the gang scheduler.

5.2.2 Interference Model

As noted earlier, the member tasks of a virtual gang are scheduled simultaneously on π. As such, they can suffer from interference on shared hardware resources (e.g., cache, memory bandwidth). In general, it is difficult to precisely model the impact of interference on a COTS hardware platform. For analysis purpose, we use a simple interference model in which the impact of interference to a virtual gang w_l is incorporated in its WCET C_l by scaling the length of its longest constituent task as follows: $C_l = \max_{\forall \tau_k \in w_l}\{c_k\} \times \max(R_l, 1)$; intuitively, we assume that w_l suffers no interference until the resource is over-utilized, after which we apply a linear scaling.

We note that this simple interference model is based on our experimental evaluation on two real embedded platforms (a NVIDIA Jetson Nano and a Raspberry Pi 4) using a set of synthetic benchmarks where they compete for memory bandwidth of the evaluated platforms. We do not, however, claim the general correctness of the interference model. If a different interference model

exists for a given hardware platform, it can be used instead. Furthermore, we stress that regardless of the used interference model and its accuracy, it stands to reason that the static nature of virtual gangs enables low timing variability, effective shared resource partitioning (e.g, [2, 48]), and accurate WCET estimations.

5.3 The RTG-Sync Framework

RTG-Sync is a software framework to enable virtual-gang based parallel real-time task scheduling on multicore platforms. As explained in the Section 5.1, synchronization between the members of each virtual gang is a key requirement for effective virtual gang based scheduling. For a typical multi-threaded process (task), synchronization between the threads of the process can be achieved by using a barrier mechanism available in the parallel programming library it uses (e.g., OpenMP [18] barrier). However, such a barrier mechanism is tied to the particular parallel programming framework, which is used by the particular parallel task, and is not designed to be used by disparate tasks for system-level scheduling.

RTG-Sync provides a cross-process synchronization mechanism, by utilizing existing OS-level inter-process communication (IPC) mechanisms. In addition, it provides APIs to manage virtual gangs and their membership. Furthermore, it integrates shared cache partitioning and memory bandwidth throttling mechanisms to bound the impact of interference in hardware resources.

Figure 5.3 shows the high level architecture of RTG-Sync. The user-level component of RTG-Sync provides a specially designed system-wide barrier to each virtual gang so that all its member tasks can be synchronously released and scheduled by the kernel-level gang scheduler simultaneously. At the kernel-level, the modifications we have implemented ensure that the virtual gang execution is protected from interference by best-effort tasks through partitioning of LLC via page-coloring and memory level bandwidth throttling framework. In the figure, τ_1 (on Core-1 and 2) and τ_2 (on Core-3) are periodic real-time tasks of a virtual gang under RTG-Sync. In the configuration shown in Figure 5.3, the LLC has eight distinct partitions (colors), of which four are given to τ_1, two are given to τ_2, and the rest are reserved for best-effort tasks by RTG-Sync. The LLC parti-

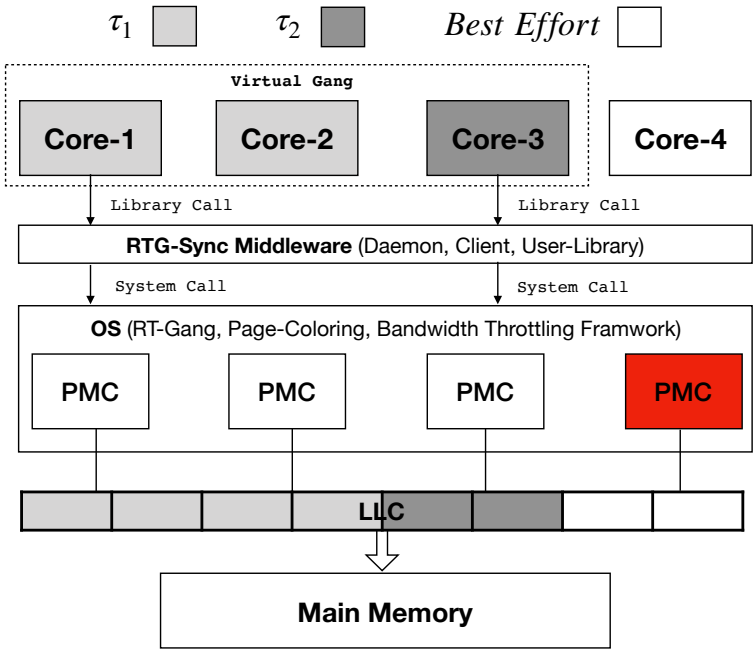


Figure 5.3: High level architecture of RTG-Sync framework with a sample partitioning setup. In this figure, τ_1 and τ_2 are real-time tasks and the resources allocated to them via RTG-Sync framework are color-coded. Note that this is just one possible partitioning of resources under RTG-Sync; the framework is highly configurable.

tioning scheme can be changed or disabled completely as needed. In addition to the coloring based LLC partitioning, RTG-Sync additionally throttles the maximum memory bandwidth of Core-4, which can schedule best-effort tasks, to the virtual gang determined bandwidth thresholds so that the interference impact of best-effort tasks to the virtual-gang is bounded.

5.3.1 Middleware

RTG-Sync middleware consists of a server daemon and a client program. The primary service provided by the server is creating virtual gangs and initializing their associated resources. The server receives the number of processes, which need to be run as a single virtual gang, and creates a new memory mapped file in a predefined location, which is used for creating a system-wide barrier. When creating a virtual gang, we also specify the maximum memory bandwidth thresholds and LLC partitions (colors) for the best-effort cores (i.e., the cores that are not used to schedule the gang member tasks and thus can schedule any best-effort tasks). These parameters are enforced

by the kernel-level mechanisms (Section 5.3.2) in order to bound the impact of co-scheduled best-effort tasks, if exist, to the virtual gang. A unique ID value is generated for each virtual gang, which is then used by the member tasks. Each member task shall make RTG-Sync user-library calls, to map the barrier file into its own address space and synchronize with other virtual gang members through the barrier.

The API call to register a process as a virtual gang member takes the virtual gang ID value issued by the RTG-Sync server along with the shared-resource requirement information for the calling process. Currently, each gang member task can specifies a portion of the LLC space, in terms of colors, the task is allowed to use. Internally, RTG-Sync makes a system-call to record the passed in parameters into the calling process's task-structure in the kernel. Furthermore, it maps the system-wide barrier registered against the passed in virtual gang id into the calling process' address-space. Once a process is registered as part of a virtual gang, the call to synchronize gang members is simple. It takes the barrier pointer returned by the aforementioned API call and uses it to synchronize on the barrier. This call must be made by all member tasks at the start of their periodic execution. As soon as the waiter count for the barrier is reached, the member tasks are unblocked simultaneously; leading to desired alignment of their periodic execution. Because RTG-Sync requires all member tasks of a virtual gang to share the same period, no additional synchronization is necessary after they are simultaneously released at the beginning.

5.3.2 Kernel Modification

RTG-Sync uses the RT-Gang framework [85] to provide gang scheduling inside the Linux kernel. We have made several changes to the RT-Gang framework. First, to support virtual gangs, we have modified Linux's `task_struct` to include the virtual gang ID and created related system calls to register/destroy virtual gangs. The RT-Gang's gang scheduler is then modified to utilize the stored virtual gang ids in making scheduling decisions.

Second, we have integrated PALLOC [48], which is a page-coloring framework, into RTG-Sync and modified it to allocate pages to a task based on the LLC color-map information stored

in the task's process control block (i.e., `task_struct`). We use PALLOC to perform two-level partitioning of the LLC. The first level statically partition's the LLC into two regions which are assigned to real-time tasks and best-effort tasks via Linux's Cgroups. The second level of partitioning is used to divide LLC between member tasks of a virtual gang as per the resource requirement information stored in the task's control-block.

Lastly, we have extended RT-Gang's memory bandwidth throttling framework to support separate read and write throttling capabilities [4]. This can be used to improve performance of the best-effort tasks without impacting the virtual gang's real-time performance.

5.4 Virtual Gang Formation

Problem Statement: For a given candidate-set Δ_T of N tasks with the same period T and a given multicore platform with m unit-speed CPU cores with a known interference model, we want to partition the N tasks into a set of virtual gangs such that the total completion time of the virtual gangs is minimized, while respecting all the precedence constraints among the original tasks. In the following, we first describe a satisfiability modulo theories (SMT) based optimal algorithm and then explain a heuristic solution.

5.4.1 Optimal Virtual Gang Formation via SMT

In the SMT based solution for virtual gang formation, we write the constraints for our optimization problem in a form that can be understood by an SMT solver; based on the candidate-set. The resulting SMT script is then fed to the SMT solver which declares the problem as either satisfiable or unsatisfiable. For a satisfiable problem, we also obtain a model for the input parameters that satisfies the constraints of the problem. We use quantifier free linear integer arithmetic logic (QF-LIA) of SMT. In the following, we describe the parameters of our problem and the constraints for a feasible solution.

Parameters: For each task τ_i in the considered candidate-set Δ_T, we use the variable x_i to denote

the index of the virtual gang that τ_i is assigned to in a feasible solution. Note that for a candidate-set with N tasks, at-most N virtual gangs can be formed. We assume that the virtual gangs are indexed in the linear order in which they are required to execute. Consistent with our system model, we use the variable C_i to denote the length (WCET) and the variable R_i to denote the resource demand of each virtual gang. The parameters x_i, C_i and R_i are subject to the following constraints:

Constraint 1 $\forall \tau_i \in \Delta_T : 1 \leq x_i \leq N$

The value of each x_i must be between 1 and N; because we can have at-most N virtual gangs.

Constraint 2 $\forall j = 1...N : \sum_{\forall \tau_i \in \Delta_T | x_i = j} h_i \leq m$

The combined core demand of all the tasks assigned to each virtual gang must not exceed the total number of cores m.

Constraint 3 $\forall G_p \mid T_p = T, \forall (\tau_i, \tau_j) \in e_p : x_i < x_j$

If τ_i has a precedence constraint with τ_j, the virtual gang x_i containing τ_i must execute before the virtual gang x_j.

Constraint 4 $\forall j = 1...N : R_j \geq \sum_{\forall \tau_i \in \Delta_T | x_i = j} r_i$

The combined resource demand R_j of the j-th virtual gang must be greater than or equal to the sum of the resource demands of its constituent tasks.

Constraint 5 $\forall j = 1...N, \forall \tau_i \in \Delta_T \mid x_i = j : C_j \geq c_i \wedge C_j \geq c_i \times R_j$

The length C_j of the j-th virtual gang must be greater than or equal to the length of each of its constituent tasks, as well as the length of each constituent task multiplied by the combined resource demand R_j. In essence, this means that C_j must be at least equal to the length of the longest constituent task multiplied by $\max(1, R_j)$; the above formulation ensures that the constraints are expressed in linear arithmetic by removing the max.

Constraint 6 $\sum_{j=1}^{N} C_j = \mathbb{C}$

Finally, the combined length of all virtual gangs must be equal to a specified value $\mathbb{C}$ whose minimum possible assignment needs to be found.

Since we use quantifier free logic, in our SMT formulation, we remove the universal quantifier ($\forall$) by repeating each constraint for every task τ_i, index j and/or edge (τ_i, τ_j) in the corresponding constraint formula. To find the virtual gang combination with minimum collective length, we conduct binary search on the combined gang length value $\mathbb{C}$; starting with the maximum possible length equal to the sum of the length of all the tasks in the candidate-set i.e., $\mathbb{C}_{init} = \sum_{\forall \tau_i \in \Delta_T} c_i$. In each step of the binary search, the SMT script is re-run with a new value of $\mathbb{C}$; to check if a model of input parameters (x_i, R_j and C_j) can be found which satisfies the constraints. If a satisfiable solution is found, the maximum combined gang length is reduced (search down); otherwise it is increased (search up). The process is repeated until the maximum combined gang length cannot be changed any further; in which case, the last solution, that was found satisfiable, is taken as the optimal solution. Note that a satisfiable solution can be found for any value of $\mathbb{C}$ equal to or greater than the optimal because Constraint 5 requires the values of C_j to be greater than or equal, rather than exactly equal, to the adjusted length of the longest constituent task. This allows the SMT solver to find feasible solutions quicker. The values of x_i in the optimal solution are used to create a new taskset of virtual gangs by combining the tasks in the candidate-set.

5.4.2 Virtual Gang Formation Heuristic

Due to the combinatorial nature of the optimization problem of virtual gang formation, the time required for obtaining the optimal solution via SMT quickly becomes intractable with increasing candidate-set size; as can be seen in Sec 5.5.3. For this reason, we design a fast running heuristic for virtual gang formation, which is shown in Algorithm 7.

At the high-level, the algorithm tries to group tasks with similar WCET values so long as their combined shared resource utilization is not too high; to make the virtual gang's WCET as small as

Algorithm 7: Virtual Gang Formation Heuristic

```
1  Input: Candidate Set (Δ_T), Number of Cores (m)
2  Output: Taskset comprising virtual gangs
3  function gang_formation(Δ_T, m)
4      pq = sort_tasks_by_wcet(Δ_T)
5      virtualGangs = ()
6      while not_empty(pq) do
7          τ_i = pq.pop()
8          f_i = family(τ_i)
9          partners = ()
10         for τ_j ∈ pq do
11             if τ_i.h + τ_j.h ≤ m ∧ τ_j ∉ f_i then
12                 partners ← partners ∪ {τ_j}
13             end
14         end
15         pq_i = score_partners(partners)
16         while not_empty(pq_i) do
17             τ_p = pq_i.pop()
18             τ_i = merge(τ_i, τ_p)
19             pq.remove(τ_p)
20             update_partners(τ_i, pq_i)
21         end
22         virtualGangs ← virtualGangs ∪ {τ_i}
23     end
24 return virtualGangs
```

possible while fully utilizing the cores. The first step in the heuristic is to create a priority queue of the tasks in the candidate-set by sorting the tasks based on their WCETs (*line-4*), using the intuition that in a virtual gang, the longest running task subsumes the WCET of its corunning tasks. The next step is to remove the longest task τ_i from the front of the queue and identify all tasks τ_j which can be paired with τ_i; under the following constraints: 1) The combined core demand of τ_i and τ_j must be less than m. 2) τ_i and τ_j must not be related by precedence constraints (*lines-10:12*).

To check for precedence constraints, we introduce the notion of the $family$ of a node τ_i in our DAG which comprises all nodes τ_k that are connected with τ_i in an ancestor or descendant relationship i.e., $family(\tau_i) = \{\tau_k \mid \tau_k \in ancestor(\tau_i) \vee \tau_k \in descendant(\tau_i)\}$. With this, the precedence constraint check between τ_i and τ_j becomes $\tau_j \notin family(\tau_i)$ i.e., τ_j cannot be paired with τ_i if it is

a member of τ_i's family.

In the list of potential corunners of τ_i, we score each task τ_p based on the *net advantage* that can be obtained by pairing it with τ_i using the following idea. The *advantage* obtained by pairing τ_p with τ_i is equal to the length of τ_p since it is the shorter running task in the potential virtual gang. The *disadvantage* of this pairing is the potential *increase* in the length of τ_i if $\tau_i.r + \tau_p.r > 1$. The *net advantage* is then equal to the difference between these two values; which we use to score all the potential partners of τ_i (*line-13*).

Once all the partners are scored, we keep removing the partner with the currently highest score from the partner list and merge it with τ_i to create a virtual gang; until no more pairing is possible (*lines-15:16*). In each step, we keep track of the precedence constraint in the DAG from forming virtual gang (since merging tasks can change the precedence constraint relationships in the DAG) and update the partner list to remove any tasks that can no longer be paired due to the new precedence constraint requirements (*line-18*). A task that is paired off is removed from the priority queue as well. Once a virtual gang is finalized, we put it in a separate list and start the process again by selecting the next τ_i from priority queue until the queue is empty. The final virtual gangs and the transformed DAGs of the candidate-set are returned once the heuristic finishes.

5.5 Schedulability Analysis

The schedulability analysis of a taskset comprising virtual gangs $\{w_1, w_2, \cdots, w_l\}$ under the rate-monotonic priority assignment scheme [26] can be done by performing fixed point iteration on the response time equation:

$$\mathbb{R}_i^{k+1} = \overline{C_i} + \sum_{\forall w_j \in hp(w_i)} \left\lceil \frac{\mathbb{R}_i^k}{T_j} \right\rceil C_j, \tag{5.1}$$

where: $\mathbb{R}_i^k$ is the response-time of w_i at the k-th iteration; $\overline{C_i}$ is the sum of the WCET of w_i itself and all the virtual gangs with the same period which come before w_i in the linear execution order;

and $hp(w_i)$ represents the set of all virtual gangs which have higher priority than w_i (i.e., smaller period). The taskset is deemed schedulable if for each w_i, the final response time $\mathbb{R}_i$ is less than the period T_i [3].

In the following, we compare schedulability results with virtual gang formation with other parallel real-time task scheduling approaches with synthetically generated tasksets.

5.5.1 Simulation Study

Taskset Generation: For the real-time taskset generation, we first uniformly select a period T_i in the range $[10, 1500]$. For each T_i, N tasks $\tau_{i,j}$, where N is randomly picked from the interval $[2, m]$, are generated by selecting a WCET $c_{i,j}$ in the range $[T/10, T/5]$, a resource demand factor $r_{i,j}$ in the interval $[0, 1]$ and a parallelism level $h_{i,j}$. The utilization $u_{i,j}$ of each $t_{i,j}$ is then calculated using the relation: $u_{i,j} = (c_{i,j} \times h_{i,j})/T_i$. If $u_{i,j}$ is less than the remaining utilization for the taskset, $c_{i,j}$ is adjusted so that $\tau_{i,j}$ fills the remaining utilization. Otherwise, taskset generation continues until the desired level of utilization is reached. We generate 1000 tasksets for each data point.

Precedence Constraints: Once a taskset is generated, we model precedence constraint by adding edges among tasks, which have the same period T_i, based on an edge probability value $P(e)$ which represents the average chance of an outgoing edge from one task to another. To simplify the creation of a DAG without explicitly checking for a cycle, we assume that an edge can only exist between $\tau_{i,j}$ and $\tau_{i,k}$ if $j < k$; hence, task $\tau_{i,N}$ has no outgoing edges. Under this scheme, the tasks with smaller index values have potentially more *neighbors*, to have an edge with, than the tasks with larger index values. To have a balanced edge generation scheme, we divide $P(e)$ value by the number of potential neighbors of a task ; except for the last task $\tau_{i,N}$ which does not have any neighbors e.g., for N = 8 and $\tau_{i,1}$, the number of potential neighbors of $\tau_{i,1}$ is $N-1=7$ and $\tau_{i,1}$ has $P(e)/7$ chance of having an edge with each of these neighbors.

Taskset Types: Similarly to [25], we consider three types of tasksets in our simulation, based on the allowed level of parallelization $h_{i,j}$ for the tasks in the taskset. For a *lightly-parallel* taskset,

[3] Note that technically, it suffices to check the last virtual gang in the linear order for each period.

$h_{i,j}$ is uniformly selected in the range $[1, \lceil 0.3 \times m \rceil]$. For a *heavily-parallel* taskset, the value of $h_{i,j}$ is picked from the range $[\lceil 0.3 \times m \rceil, m]$. Finally, for *mixed* taskset, $h_{i,j}$ is selected randomly from the interval $[1, m]$.

Scheduling Policies: We conduct two separate experiments to understand the impact of virtual gang formation on system schedulability. In the first experiment, we consider precedence constraints among tasks using an edge probability $P(e) = 0.25$. Due to the absence of an existing schedulability analysis for gang tasks that can handle precedence constraints to the best of our knowledge, we only compare schedulability with virtual gangs in this experiment against RT-Gang [4]. Concretely, we consider three scheduling scenarios in this experiment. Under the *RT-Gang* scheme, the unicore response time analysis using Equation 5.1 is applied to calculate schedulability of the taskset under the one-gang-at-a-time scheduling. For *Virtual Gang (SMT)*, we first form virtual gangs from the given taskset using the optimal SMT algorithm and then use Equation 5.1 to calculate schedulability of the new taskset comprising the virtual gangs. Under *Virtual Gang (Heuristic)*, we use the heuristic from Sec 5.4.2 to form virtual gangs and then calculate the schedulability results.

In the second experiment, we assume that there are no precedence constraints among the tasks i.e., $P(e) = 0$ which allows us to consider more multicore scheduling policies for comparing against virtual-gang scheduling. In this case, for each taskset type, we calculate schedulability results under four scheduling policies. We retain the RT-Gang and virtual-gang (SMT) schemes from the first experiment. In addition, we consider *Gang-FTP* policy and use the analysis in [25] to calculate schedulability of the taskset under gang fixed-priority scheduling [5]. We also consider the *Threaded* scheme which models the scheduling of parallel tasks under vanilla Linux real-time scheduler, where the $h_{i,j}$ threads of each task $\tau_{i,j}$ are independently scheduled. In this case, we assess schedulability based on the state-of-the-art analysis for fixed-priority scheduling of DAG

[4] Since RT-Gang implements one-gang-at-a-time scheduling policy, its analysis does not need to be changed if there are precedence constraints among same period tasks. For a schedulable taskset under RT-Gang, a feasible schedule can be found by applying topological sort on the DAGs of the candidate-sets.

[5] Although the analysis in [25] uses a bundled gang model, it is still applicable here since bundled model generalizes rigid gang model.

tasks in [30]; here $\tau_{i,j}$ is simply modeled as a DAG of $h_{i,j}$ nodes with the same execution time.

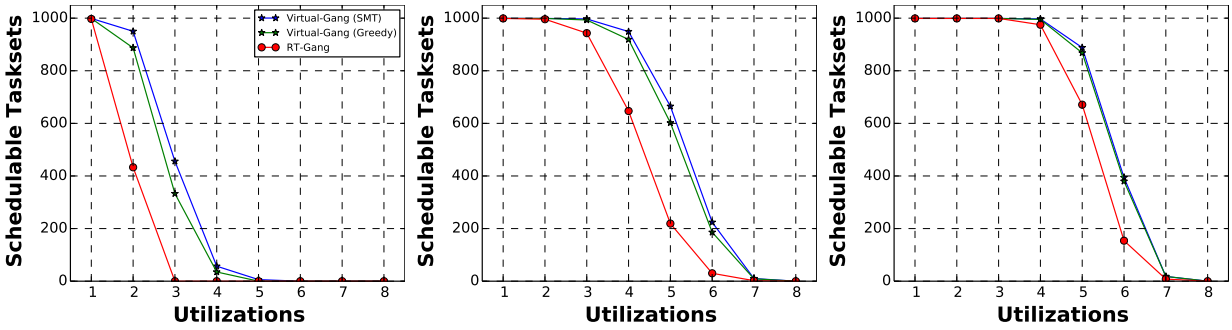
Interference Model: For each scheduling policy considered in the second experiment, we calculate schedulability with and without taking the interference between corunning tasks into account. In virtual gang scheduling, the gang formation algorithms already incorporate the interference model described in Sec 5.2.2 in creating virtual gangs. For *Gang-FTP*, for each $\tau_{i,j}$, we enumerate all possible sets of co-running tasks based on the remaining number of cores $m - h_{i,j}$, and pick the set with the maximal combined resource demand $R_{i,j}$ which we then use to scale the execution time $c_{i,j}$ of $\tau_{i,j}$ by multiplying it with $\max(1, R_{i,j})$. For *Threaded*, we assume that each independently scheduled thread of $\tau_{i,j}$ has a resource demand of $r_{i,j}/h_{i,j}$, and pick the $m-1$ other threads (either of the same or different task) with maximal demands. While the described procedure for *Gang-FTP* and *Threaded* can be pessimistic, we are not aware of any better mechanism to safely account for the effect of resource interference under such scheduling policies. Furthermore, we point out that results for *Threaded* can still be optimistic, since we do not account for the extra synchronization overheads that could be incurred when scheduling threads independently rather than as a gang.

Finally, in creating plots without interference, we set the resource demand of each task to zero and redo all our calculations (e.g., SMT virtual gang formation) before calculating schedulability. Concerning other schedulability analyses for the rigid gang model, we do not employ [23, 24] because they assume Gang EDF rather than FTP; nor we compare against [99] as it requires creating static execution patterns over a hyper-period which significantly complicates the runtime.

Priority Assignment: We consider the rate-monotonic based priority assignment scheme: $prio(\tau_i) > prio(\tau_j)$ if $T_i < T_j$. For tasks with the same period, we assign priorities based on task's WCET: $prio(\tau_{i,j}) > prio(\tau_{i,k})$ if $c_{i,j} < c_{i,k}$.

5.5.2 Schedulability Results

Figure 5.4 shows the schedulability plots for the first experiment with $P(e) = 0.25$ from our simulation for 8 cores ($m = 8$). For all taskset types, virtual gang formation provides noticeable



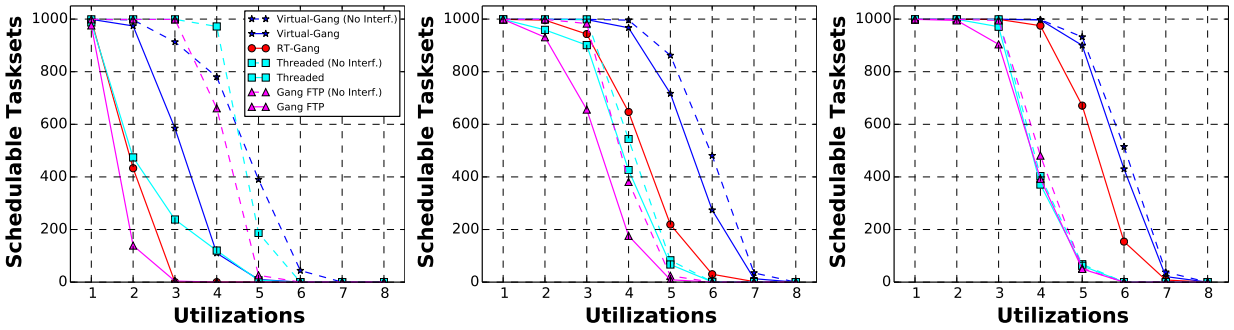
(a) Lightly Parallel (b) Mixed (c) Heavily Parallel

Figure 5.4: Schedulability plots for tasksets with precedence constraints ($P(e) = 0.25$) on 8 cores

improvement in schedulability as compared to RT-Gang. Moreover, the virtual gang formation heuristic does a good job in giving comparable performance to the optimal SMT algorithm; in terms of total number of schedulable tasksets under both schemes. It can also be seen from this figure that the difference in the performance between the heuristic and the optimal virtual gang formation decreases as the parallelization level of the tasksets increases. This is expected since for lightly parallel tasksets, there are much greater possibilities for virtual gang formation which can be missed by the greedy local optimization criteria of the heuristic.

When the tasksets comprise independent tasks (i.e., $P(e) = 0$), Fig 5.5 shows the schedulability results for the considered scheduling policies of the second experiment. For lightly parallel tasksets, the *Threaded* and *Virtual Gang* schemes give the best schedulability results, followed closely by the *Gang-FTP* policy, if interference model is not used (dashed lines). However, when interference is considered (solid lines), the schedulability under *Threaded* and *Gang FTP* policies deteriorates rapidly as compared to the *Virtual Gang* scheme. This is due to the fact that under the *Virtual Gang* scheme, only the tasks of the same virtual gang can possibly interfere with each other, while a lot more tasks must be considered in Gang-FTP and Threaded. As expected, *RT-Gang* suffers the most for lightly parallel tasks as it under-utilizes the cores.

For mixed and heavily parallel tasksets, the *Virtual Gang* scheme outperforms the rest regardless whether interference is considered or not. For these taskset types, *RT-Gang* performs considerably better as well since a single parallel task can utilize more cores in the platform, though it still



(a) Lightly Parallel (b) Mixed (c) Heavily Parallel

Figure 5.5: Schedulability plots for tasksets containing independent tasks ($P(e) = 0$) on 8 cores. Dashed lines are used when interference is not considered. For RT-Gang, because only one task can be scheduled at a time, interference cannot occur by design (thus no dashed lines)

lags behind Virtual Gang scheme. On the other hand, *Gang-FTP* and *Threaded* are significantly worse than the Virtual Gang scheme and the RT-Gang for both mixed and heavily parallel tasksets. This can be attributed to the analysis pessimism needed to handle carry-in jobs in their schedulability tests [25, 30], which becomes more pronounced as the parallelism of the tasks increases. Because both Virtual Gang and RT-Gang can use exact unicore-based fixed-priority schedulability techniques, they do not suffer from such analysis pessimism.

Finally, in all cases, interference impact becomes less prominent as the parallelization of the taskset increases. This is because with highly parallel tasks, the opportunity of getting co-scheduled with other resource intensive tasks decreases, leading to improved schedulability.

In summary, our simulation results show that the Virtual Gang scheme significantly outperforms the rest when interference is considered, and is competitive even when interference is not considered.

5.5.3 SMT and Heuristic Gang Formation Runtime

In this experiment, we compare the time required to form virtual gangs from a given candidate-set using the SMT and the heuristic algorithms. We use the Z3 SMT solver [100]. We vary the candidate-set size (N) from 4 tasks up-to 9 tasks and measure the time taken by each algorithm in generating virtual gangs. For each N, we generate 75 candidate-sets and process them through the

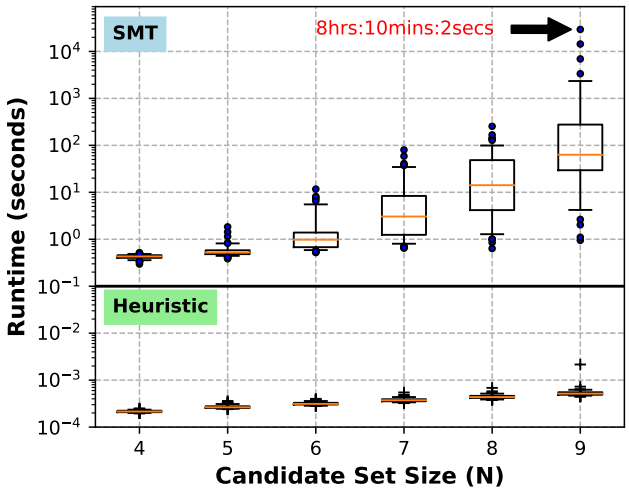


Figure 5.6: Comparison of SMT and heuristic virtual gang formation runtime. In each box, the orange line represents the median value. The box represents the interquartile range (Q2-Q3). The lower and upper whiskers mark the 5 percentile and the 95 percentile values respectively.

gang formation algorithms. We retain all the other simulation parameters from the first experiment in the previous section. We plot the time taken by each algorithm for all the candidate-sets in Fig 5.6. It can be seen that the runtime of obtaining the optimal solution via SMT increases with an exponential trend and quickly becomes unmanageable. The runtime of the heuristic, on the other hand, remains relatively stable (within 10-msec) for all candidate-set sizes.

5.6 Evaluation

In this section, we describe the evaluation results of RTG-Sync on a real multicore platform.

5.6.1 Setup

We use NVIDIA's Jetson TX-2 [1] board for our evaluation experiments with RTG-Sync. The Jetson TX-2 board has a heterogeneous multicore cluster comprising six CPU cores (4 Cortex-A57 + 2 Denver[6]). On the software side, we use the Linux kernel version 4.4 and patch it with the modified version of RT-Gang [101] to enable real-time gang scheduling at the kernel level

[6]We do not use the Denver cores because of their lack of support for necessary hardware performance counters to implement the throttling mechanism.

along with best-effort task throttling and page-coloring frameworks, adding ∼3000 lines to the architecture neutral part of the Linux kernel. In all our experiments, we put our evaluation platform in maximum performance mode which involves statically maximizing the CPU and memory bus clock frequencies and disabling the dynamic frequency scaling governor. We also turn off the GUI and networking components and lower the run-level of the system ($5 \rightarrow 3$) to keep the background system services to a minimum.

5.6.2 Case-Study

In this case-study, we demonstrate the effectiveness of using virtual gangs to improve system utilization, compared to the "one-gang-at-a-time" scheduling and Linux's default scheduler.

Task	WCET (ms)	Period (ms)	# of Threads	Priority
τ^{RT}_{BWT}	50.0	100.0	4	5
τ^{RT}_{DNN-1}	8.2	50.0	2	10
τ^{RT}_{DNN-2}	8.2	50.0	2	10
τ^{BE}_{cutcp}	∞	N/A	2	N/A
τ^{BE}_{lbm}	∞	N/A	2	N/A

Table 5.1: Taskset parameters for case-study

The taskset for the case-study is shown in Table 5.1. It consists of three real-time tasks and two best-effort ones. For real-time tasks, we use the DNN workload from DeepPicar [92] as two of the real-time tasks τ^{RT}_{DNN-1} and τ^{RT}_{DNN-2}. Both DNN tasks use two threads each and have the same period of 50 ms. We use the synthetic bandwidth-rt benchmark as the third real-time task τ^{RT}_{BWT}, which uses 4 threads and has a period of 100 ms. τ^{RT}_{BWT} is designed to be oblivious to shared resource interference but it creates significant shared hardware resource contention to DNN tasks under co-scheduling. As per the RMS priority assignment, we assign higher real-time priority to DNN tasks than the bandwidth-rt task.

For best-effort tasks, we use two benchmarks from the Parboil benchmark suite [76]. Among the best-effort tasks, τ^{BE}_{lbm} is significantly more memory intensive than τ^{BE}_{cutcp}. Both best-effort tasks

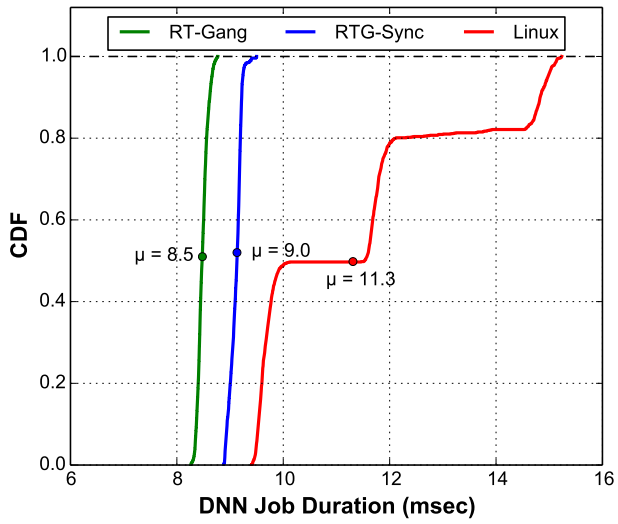


Figure 5.7: Distribution of job duration for τ^{RT}_{DNN-1}

use two threads each and are pinned to disjoint CPU cores.

We evaluate the performance of this taskset on Jetson TX-2 under three scenarios. The *Linux* scenario represents the scheduling of the taskset under the vanilla Linux kernel. In *RT-Gang* scheme, the real-time tasks are gang scheduled with the one-gang-at-a-time policy. Finally, under *RTG-Sync*, we create a virtual gang, which is comprised of the two real-time DNN tasks. We assign 3/4th of the LLC to the virtual gang (two DNN tasks) and the rest 1/4th of the cache to the best-effort tasks. We do not, however, apply partitioning between the DNN tasks as sharing the cache space is beneficial in this case.

Figure 5.7 shows the cumulative distribution function of the job execution times of τ^{RT}_{DNN-1} under the three compared schemes. Note that this task has the highest real-time priority in our case-study. In this figure, the performance of τ^{RT}_{DNN-1} remains highly deterministic under both RT-Gang and RTG-Sync. In both cases, the observed WCET of τ^{RT}_{DNN-1} stays within 10% of its solo WCET—i.e., measured WCET in isolation—from Table 5.1. However, under the baseline Linux kernel (denoted as *Linux*), the job execution times of τ^{RT}_{DNN-1} vary significantly, with the observed WCET approaching 2X of the solo WCET.

The difference among the observed performance of τ^{RT}_{DNN-1} under the three scenarios can be better explained by analyzing the execution trace of the taskset in one hyper-period of 100 ms,

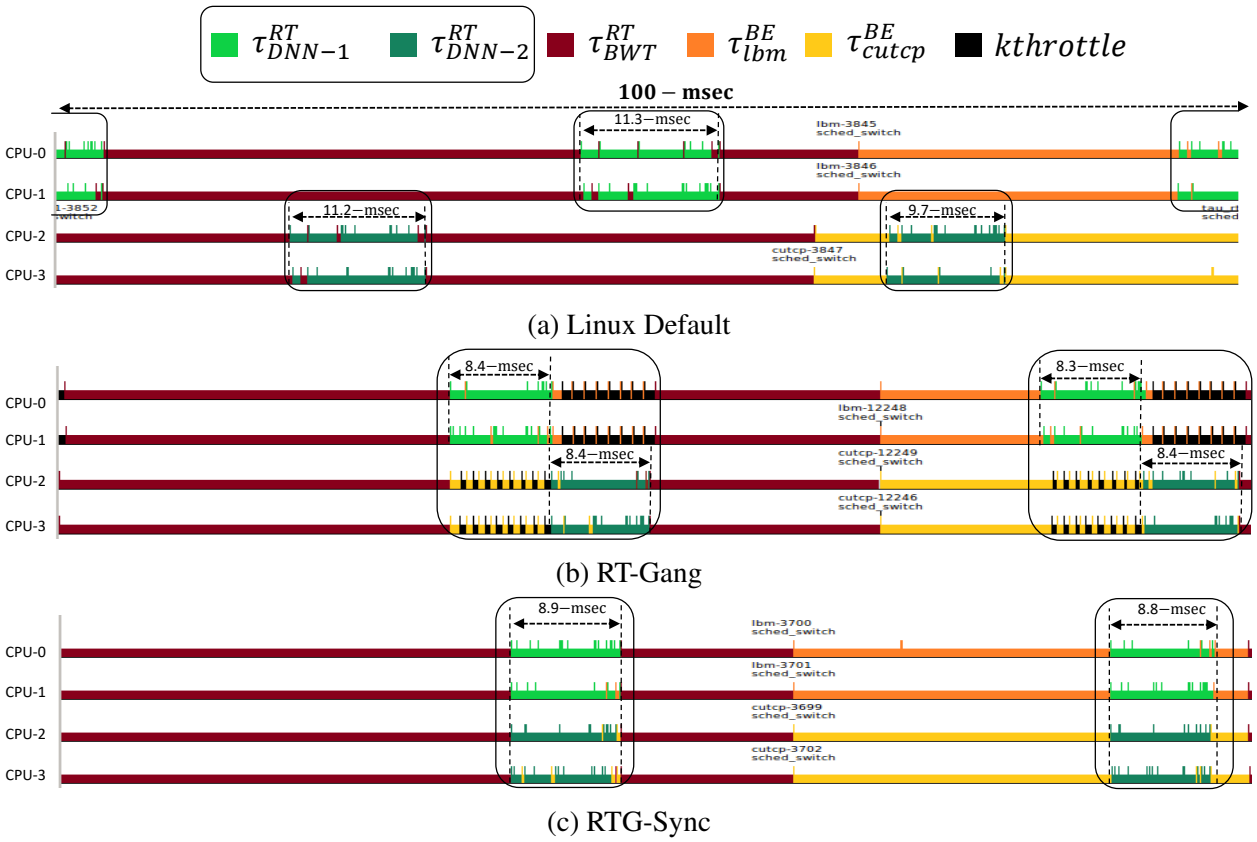


Figure 5.8: Annotated KernelShark trace snapshots of case-study scenarios for one hyper period

which is shown in Figure 5.8. Inset 5.8a displays the execution timeline under vanilla Linux. It can be seen that the DNN tasks suffer from two main sources of interference in this scenario. Whenever the execution of the DNN tasks overlaps with the execution of τ^{RT}_{BWT}, the execution time of the task increases. The execution time also increases when the DNN tasks get co-scheduled with best-effort tasks. Note that the system is not regulated in any way in this scenario. Therefore, the effect of shared resource interference is difficult to predict, as evidenced in the CDF plot of Figure 5.7, which shows highly variable timing behavior. Under RT-Gang, on the other hand, the execution of DNN tasks is almost completely deterministic. Due to the restrictive one-gang-at-a-time scheduling policy, co-scheduling of DNN tasks with τ^{RT}_{BWT} is not possible. Moreover, the shared resource interference from the best-effort tasks is strictly regulated due to LLC partitioning and the kernel level throttling framework.

However, under RT-Gang, each DNN task executes as a separate gang by itself, which means

that two system cores are left unusable for real-time tasks while the DNN tasks are executing because of the one-gang-at-a-time policy. This reduces the share of total system utilization of the multicore platform, which can be used by other real-time tasks. Although the idle cores are utilized by best-effort tasks, the strict regulation imposed by DNN tasks means that the best-effort tasks are mostly throttled when they are co-scheduled with DNN tasks. Under RTG-Sync, both of these problems are solved by pairing τ^{RT}_{DNN-1} and τ^{RT}_{DNN-2} into a single virtual gang. In this case, the system is fully utilizable by real-time tasks. The execution of virtual DNN gang is completely deterministic due to the synchronization framework of RTG-Sync. Moreover, since there is no co-scheduling of best-effort tasks with real-time tasks, the throttling framework does not get activated and any slack duration left by real-time tasks can be utilized completely by best-effort tasks without imposing throttling.

5.6.3 Overhead

The runtime overhead due to RTG-Sync can be broken down into two parts. First is the overhead due to synchronization. This overhead is incurred only once during the setup phase of the real-time tasks which are members of the same virtual gang and it does not contribute to the WCETs of the periodic jobs. Second, the kernel level overhead is incurred due to the simultaneous scheduling of real-time tasks by the gang-scheduler. Since we use the RT-Gang framework for this purpose, the kernel level overhead is the same as reported in [85], which showed negligible overhead on a quad-core platform.

5.7 Discussion

One limitation of our virtual-gang based approach is that, to create a virtual gang task, all member tasks of the virtual gang must share the same period. If each and every task in the real-time taskset has a different period value, a virtual gang can have only one real-time task as its member and no schedulability gain can be achieved with our approach. In practice, however, it is common that

a small number of periods are shared by multiple real-time tasks [102]. As such, we believe our approach can be applied in such applications and provide tangible schedulability benefits.

5.8 Conclusion

In this chapter, we introduced a virtual gang based parallel real-time task scheduling approach for multicore platforms. Our approach is based on the notion of virtual gang, a group of parallel real-time tasks that are statically linked and scheduled together as a single scheduling entity. We presented an intra-gang synchronization framework and virtual gang formation algorithms that enable strong temporal isolation and high real-time schedulability in scheduling parallel real-time tasks on COTS multicore platforms. We evaluated our approach both analytically and empirically on a real embedded multicore platform using real-world workloads. Our evaluation results showed the effectiveness and practicality of our approach.

Chapter 6

Extensions and Future Directions

In this chapter, we discuss possible extensions of the techniques presented in this dissertation. We begin by summarizing our contributions till now and describing their limitations. We then discuss possible ways to ameliorate some of those limitations.

6.1 Summary of Contributions

In the previous chapters of this dissertation, we presented three scheduling frameworks that solve part of the real-time scheduling problem on heterogeneous multicore platforms. We first presented the **BWLOCK++** framework in Chapter 3 which enables deterministic scheduling of real-time tasks on an accelerator, such as the integrated GPU, in the heterogeneous SoC. Due to the nuances of accelerator use, which will be discussed in more detail in this chapter, BWLOCK++ uses a restrictive system model in which real-time tasks are constrained to use a single-core of the multicore platform. Although this restriction makes BWLOCK++ effective in enabling deterministic use of an accelerator by real-time tasks, it is very limiting for those real-time tasks that do not require use of the accelerator. For such tasks, the single-core scheduling requirement of BWLOCK++ can severely reduce the real-time schedulability of the system realized on the heterogeneous SoC.

We next presented the **RT-Gang** framework in Chapter 4. The RT-Gang framework enables deterministic scheduling of real-time tasks on the multicore CPU cluster of a heterogeneous SoC. The one-gang-at-a-time scheduling policy of RT-Gang breaks the possibly unrestricted *coupling* between all the real-time tasks in the system and enables tight estimation of WCET that can be relatively easily enforced at runtime through OS level shared hardware resource partitioning tech-

niques. This results in a system that can provide high real-time schedulability for highly parallel CPU using real-time tasks through non-pessimistic schedulability analysis that is also easy to understand. We implemented the RT-Gang framework on top of the open-source Linux kernel and showed, with real-world benchmarks, that our framework is useful and practical in enabling deterministic real-time scheduling on multicore CPUs.

A primary limitation of the RT-Gang framework is that it severely reduces the schedulability of the system when real-time tasks are not sufficiently parallelizable. In Chapter 5, we presented the **RTG-Sync** framework which extends RT-Gang by employing virtual gang scheduling on top of the one-gang-at-a-time scheduling policy. A virtual gang is a group of real-time tasks which, although not the threads of the same process as in a *real* gang, are treated as a single schedulable entity and all its member tasks are synchronously released and simultaneously scheduled under the one-gang-at-a-time scheduling policy. The virtual gang scheduling is a compromise—between the strictly one real gang task scheduling of RT-Gang and the unrestricted scheduling of all the real-time tasks in a globally scheduled system[1]—it limits the *coupling* among the real-time tasks to only those tasks which are members of the same virtual gang. Moreover, due to the synchronous release of the gang members, the said coupling is very limited; there is only one co-schedule of the member tasks that is easy to verify and evaluate on a target hardware platform.

In the form it is presented in Chapter 5, the RTG-Sync framework does not cater to real-time tasks that require the use of an accelerator for part of their execution. In the following, we will discuss the challenges involved in handling accelerator using real-time tasks under the RTG-Sync framework. We will then present a new system model and task model for the RTG-Sync framework that is suitable for the modified real-time scheduling problem and discuss the theoretical and practical extensions to the framework for resolving the said problem.

[1]Please see Sec. 2.2.2.1 for a discussion on global scheduling

6.2 The Accelerator Scheduling Problem

There are two main challenges in integrating the scheduling on accelerators into the RTG-Sync framework: 1) how to handle the proprietary hardware level scheduling implemented in an accelerator (e.g., integrated GPU) into the virtual gang scheduling of RTG-Sync? 2) how to tackle possible interference to the real-time CPU tasks from co-executing tasks on the accelerators? In the following, we discuss both these challenges in detail.

6.2.1 Hardware Level Scheduling in the Accelerator

An accelerator in a heterogeneous SoC can come equipped with a hardware level scheduler that controls, among other things, which workloads to execute on the accelerator at any given time and whether simultaneous use of the accelerator is allowed by multiple workloads and if so, then how to arbitrate the division of resources (e.g., registers, computing cores, caches etc.) present inside the accelerator among the co-executing workloads. Since these aspects of hardware level scheduling impact the timing characteristics of the real-time workloads utilizing the accelerator, it is of utmost importance for the real-time system designers to know these details. Unfortunately, there is reluctance from the vendors of the heterogeneous SoCs to disclose the details of the internal scheduling policies of the hardware e.g., this is the case for the integrated GPUs present in all the heterogeneous SoCs from NVIDIA in the Jetson family. Although efforts have been made in the real-time community to reverse engineer these hidden details [80, 103], the available information is too limited and can easily get out-dated when a new generation of SoCs comes into the market. In addition to the above, an accelerator e.g., iGPU, may not allow software controlled preemption of an executing workload i.e., a workload must run to completion once it starts executing on the accelerator. For this reason, it is a common practice in real-time systems to treat the accelerator as a non-preemptible resource that must be accessed sequentially by real-time workloads.

There are a number of pros and cons associated with treating an accelerator as a non-preemptible resource. On one hand, it uncouples the timing behavior of the workloads that want to use the ac-

celerator from each other since only one workload can be running on the accelerator at any given time. On the other hand, it can potentially result in under-utilization of the accelerator if the individual workloads cannot utilize the accelerator to the fullest extent possible. Another complication arises due to the fact that the programming model of the accelerator may not, by default, allow non-preemptive use of the accelerator e.g., in the CUDA programming model of the NVIDIA iGPUs, kernels launched concurrently are time-multiplexed on the GPU. To enforce non-preemptive accelerator use, modifications may be needed in the OS kernel or in the accelerator programming model to make the concurrent workloads use the GPU sequentially. Despite these cons, in our opinion, assuming the non-preemptibility of the accelerator workloads will be the accepted practice in the foreseeable future until official programming models e.g., CUDA streamline support for software controlled preemption for their respective accelerators and disclose the implementation details of the hardware level scheduler.

6.2.2 Performance Isolation between Accelerators and CPU Workloads

As discussed in Chapter 2, the memory hierarchy in the embedded heterogeneous computing platforms is shared among all the on-chip computing resources; including CPU cluster and the accelerators. This sharing can create coupling between co-executing workloads on CPUs and the accelerators i.e., workloads on CPU can change the execution time of the real-time accelerator applications by creating too much memory traffic that can bottleneck the underlying memory subsystem and vice-versa. Our BWLOCK++ framework solves this problem by restrictive scheduling and bandwidth throttling. The restrictive scheduling of BWLOCK++ constrains all the real-time tasks to use a single core of the CPU cluster and use the accelerator[2] non-preemptively and only by real-time tasks. Due to this, only one real-time task can be in execution at any given time and there can be no coupling between real-time CPU and accelerator workloads. The throttling framework of BWLOCK++ allows best-effort CPU tasks to utilize the remaining CPU cores as long as their

[2]Although in Chapter 3, we describe BWLOCK++ for heterogeneous systems that have integrated GPU as the accelerator, the design principles of BWLOCK++ are general and can be applied to any heterogeneous system that contains a single accelerator that must be used non-preemptively.

memory traffic stays within a safe threshold that is acceptable to the executing real-time task.

As explained earlier, BWLOCK++ limits real-time schedulability due to the restrictions it imposes on the target system. These restrictions are:

1. The accelerator must be used non-preemptively.

2. Only one real-time task can be in execution, in the entire system, at any given time.

3. Best-effort tasks are not allowed to use the accelerator.

In the following, we discuss the implications of relaxing each of these restrictions in detail. The first restriction is fundamental to accelerator scheduling for the reason described in Sec. 6.2.1. If preemptive execution on the accelerator is made possible, in the manner CPU scheduling is preemptive, then most of the problems associated with real-time scheduling on accelerators—including the second and third restrictions enumerated above—would become trivial. However, we do not consider this scenario since it is beyond our control.

For the second restriction, if multiple real-time tasks are allowed to execute concurrently in the system, then it is possible for simultaneously executing real-time tasks on the CPU and the accelerator to interfere with each other. This, in turn, would re-introduce the uncontrolled coupling problem among all real-time tasks in the system that makes the WCET estimation and response-time analysis pessimistic. Hence it is unwise to relax this restriction unless there is an alternative scheduling scheme, such as RTG-Sync, in place to ameliorate this problem.

For the third restriction, allowing best-effort tasks to utilize the accelerator may be useful if real-time tasks do not make adequate use of the accelerator. However, this introduces two complications: 1) The blocking time for real-time tasks in Eq. 3.4 would have to include possible blocking from best-effort tasks. 2) The best-effort tasks executing on the accelerator may interfere with real-time tasks on the CPU cores. The first complication is fundamental—it is not possible to circumvent it unless accelerator scheduling is made preemptive. The second complication can conceptually be resolved by using memory bandwidth regulation techniques to throttle accelerator using best-effort tasks; similar to how we regulate best-effort tasks on the CPU cores. However,

such a regulation mechanism requires periodic monitoring of certain hardware performance counters that can indicate, on a periodic basis with an acceptable time granularity (e.g., 1-msec), the memory traffic generated by accelerator using tasks. Given the secrecy surrounding the implementation details of the accelerators, such information is hard to come-by. In addition to this, a more cardinal problem is that the regulation mechanism such as [2] requires the ability to idle an offending computing resource if it exceeds its given budget by immediately scheduling a high priority thread on it for the remainder of the regulation interval. This, in turn, requires the ability to enact quick preemption of the currently executing workload which is not possible for accelerators.

In light of the above discussion, the only restriction of BWLOCK++ that can be relaxed, to a certain extent, is allowing multiple real-time tasks to execute in the system in a controlled manner. In fact, this was our prime motivation when designing the RT-Gang and the RTG-Sync frameworks. As stated in Sec. 6.1, RTG-Sync enables deterministic scheduling of real-time tasks on multicore CPUs; by grouping tasks into virtual gangs and scheduling virtual gangs one-at-a-time on all cores of the CPU cluster. In the following, we discuss the RTG-Sync extension to include accelerator using tasks into its design and the implementation changes required to support this extension in a practical system.

6.3 Virtual Gang Scheduling with Accelerator using Tasks

In this section, we describe a new system model and task model that can enable the inclusion of accelerator using real-time tasks into the virtual gang scheduling scheme of RTG-Sync framework.

6.3.1 System Model

We consider a multicore processor based platform π, which contains m unit-speed CPU cores and k accelerators $(A_1, A_2, \cdots, A_k)$. We assume that each accelerator must be used in a non-preemptive manner i.e., once a task starts executing on the accelerator A_i, it must run to completion before the next task, that also needs to use A_i, is allowed to access it. Moreover, only real-time tasks

are allowed to use the accelerators. We consider a system comprising a set Γ of n periodic, rigid gang real-time tasks with implicit deadlines: $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task $\tau_i = (c_i, b_i, \kappa_i, r_i, T_i)$ is characterized by its WCET c_i in isolation, its maximum blocking time while b_i, its demand of the on-chip computing resources κ_i, the shared resource demand factor r_i in the range $[0,1]$, and the period T_i. The WCET c_i of the task describes the total execution time of the task; including possible execution on any of the accelerators in the SoC. The maximum blocking time b_i is equal to the longest duration for which the task uses any of the accelerators that it requires to complete its execution; this is the longest duration for which the task can potentially delay the execution of higher priority tasks in the system. The compute demand factor $\kappa_i = (h_i, a_{i,1}, a_{i,2}, \cdots, a_{i,k})$ is a tuple of values. The h_i in κ_i denotes the number of cores required to run τ_i and $h_i \leq m$; as per the rigid gang model. The subsequent values $a_{i,j} \in (0,1)$ in κ_i denote whether the task needs to use the j^{th} accelerator A_j for its execution; $a_{i,j} = 1$ if the task needs the j^{th} accelerator for its execution and $a_{i,j} = 0$ otherwise. The resource demand factor r_i describes the maximum instantaneous demand of the task τ_i at any point during its execution for a critical hardware resource in the shared memory subsystem e.g., the main memory bandwidth.

The system also comprises a set of k DAGs $\{G_1, G_2, ..., G_k\}$. Each DAG $G_i = (v_i, e_i, T_i)$ expresses a set of precedence constraints among tasks. The node set $v_i \subseteq \Gamma$ consists of a subset of the tasks in Γ; we assume that all tasks in v_i must have the same period T_i, and no task in Γ can belong to the node set of more than one DAG. The edge set $e_i : v_i \times v_i$ consists of ordered pairs of the form (τ_p, τ_q) describing the precedence constraints among the tasks in v_i: formally, this means that the j^{th} job of τ_p must finish before the j^{th} job of τ_q can start executing.

Example: Consider the block diagram, shown in Figure 6.1, of a non-trivial real-time taskset with precedence constraints that needs to be executed on a heterogeneous computing platform which has 8 CPU cores ($m = 8$) and three accelerators: an integrated GPU (A_0) and two custom-built deep learning accelerators (A_1, A_2). A taskset like this can be found in a self-driving application suite such as [98]. When expressed in the form of our task model, the taskset comprises a single DAG $G_1 = (v_1, e_1, T_1)$. The period of the DAG is $T_1 = 100$-msec because of the 10-Hz frequency of the

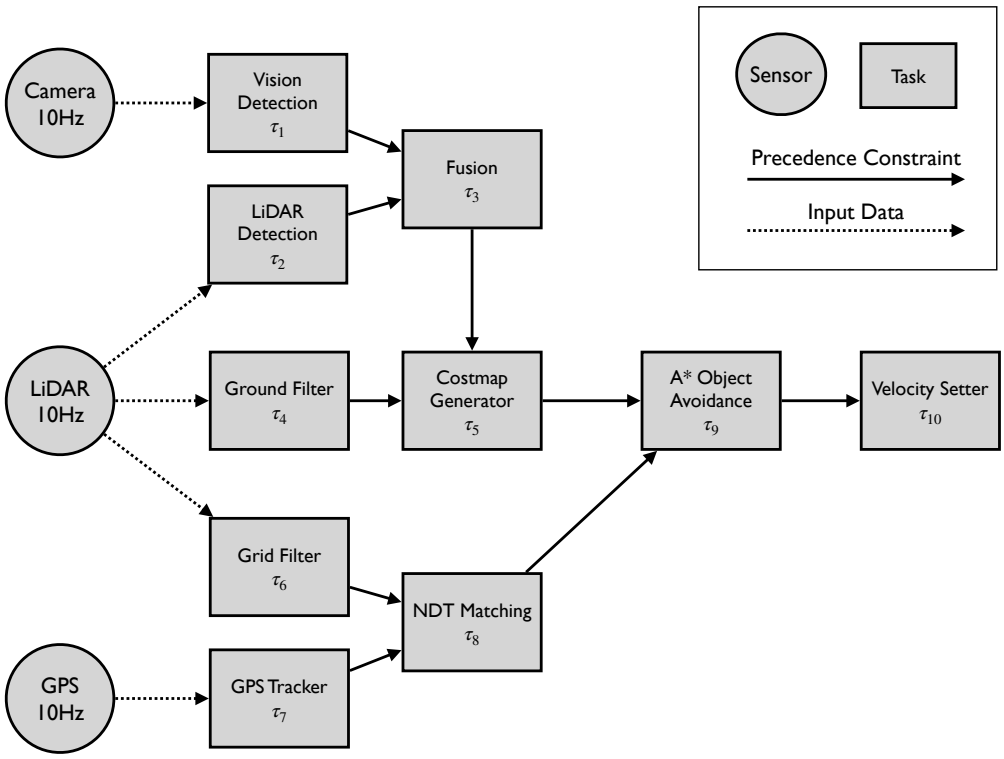


Figure 6.1: Block diagram of an example taskset that can be specified by the new task model.

data coming from the input sensor nodes. The node-set comprises 10 tasks $v_1 = (\tau_1, \tau_2, \cdots, \tau_{10})$ that are part of the DAG. The edge-set e_1 describing the precedence constraints of the DAG is the following:

$$e_1 = ((\tau_1,\tau_3),(\tau_2,\tau_3),(\tau_3,\tau_5),(\tau_4,\tau_5),(\tau_5,\tau_9),(\tau_6,\tau_8),(\tau_7,\tau_8),(\tau_8,\tau_9),(\tau_9,\tau_{10}))$$

Table 6.1 shows the parameters of each task in the taskset. From this table, it can be seen that the tasks in the example taskset have varied demand for the on-chip compute resources and have diverse execution requirements. For example, the vision detector task $\tau_1 = (13, 5, (2,1,0,0), 90, 100)$ can be characterized as a *light* utilization task, with a total execution time of 13-msec, that needs 2 CPU cores and the GPU for its execution with a 5-msec of max. blocking time. However, this task has high demand for the shared memory subsystem—it may slowdown the execution time of its co-executing tasks—as its resource demand factor is 0.9. In contrast, the task τ_9 that runs the A* object avoidance algorithm can be characterized as a *heavy* utilization task with an 80-msec total

Name	Task	WCET (c-msec)	Blocking Time (b-msec)	Compute Demand ($\kappa = (h, a_0, a_1, a_2)$)	Resource Demand (r)
Vision Detector	τ_1	13	5	(2, 1, 0, 0)	0.9
LiDAR Detector	τ_2	70	20	(2, 0, 1, 0)	0.75
Fusion	τ_3	2	0	(4, 0, 1, 0)	0.1
Ground Filter	τ_4	75	30	(1, 1, 0, 0)	0.3
Costmap Generator	τ_5	35	10	(2, 0, 0, 1)	0.6
Grid Filter	τ_6	28	0	(6, 0, 0, 0)	0.95
GPS Tracker	τ_7	5	0	(1, 0, 0, 0)	0.05
NDT Matching	τ_8	3	1	(1, 1, 0, 0)	0.2
A* Object Avoidance	τ_9	80	50	(4, 0, 1, 0)	0.7
Velocity Setter	τ_{10}	10	0	(3, 0, 0, 0)	0.4

Table 6.1: Taskset parameters for the example illustrating the new task model. Period of the taskset is 100-msec. In the compute demand tuple $\kappa = (h, a_0, a_1, a_2)$, h denotes the number of cores used by the task, a_0 denotes whether the task needs the integrated GPU to run and a_1, a_2 denote the same for DLA-1 and DLA-2 respectively. For example, τ_5, with $\kappa = (2, 0, 0, 1)$, needs 2-cores and the DLA-2 for its execution.

execution time and 50-msec max. blocking time.

6.3.2 Virtual Gang Formation

Given the new system model, we rephrase the virtual gang formation problem from Sec. 5.4 and then discuss the changes needed in the gang formation algorithms to tackle the new problem.

Problem Statement: For a given candidate-set Δ_T of N tasks with the same period T and a given multicore platform with m unit-speed CPU cores, k accelerators $(A_1, A_2, \cdots, A_k)$ and a known interference model, we want to partition the N tasks into a set of virtual gangs such that the total completion time of the virtual gangs is minimized, while respecting all the precedence constraints among the original tasks.

6.3.2.1 Changes in the Gang Formation Algorithms

The SMT based algorithm for virtual gang formation is described in the form of constraints that must be satisfied by any feasible grouping of tasks in the candidate-set into a set of virtual gangs. The constraints of the original algorithm can be seen in Sec. 5.4.1; in this section, we only discuss

the changes required in the original parameters and constraints to handle the modified virtual gang formation problem. In the new problem, it is possible for tasks in the candidate-set to require an accelerator for a portion of their execution in a non-preemptive (*blocking*) manner. Hence we add a new parameter B_j to the virtual gang formation problem which denotes the maximum blocking time of the j-th virtual gang.

If the original SMT based algorithm is used to form virtual gangs, then it is possible to obtain gang combinations in which multiple tasks require the use of the same accelerator in the SoC. However, we want to avoid this situation because of the complications related to accelerator scheduling; as discussed earlier in this chapter. This results in an additional constraint in the SMT based virtual gang formation algorithm as stated in the following:

Constraint 7 $\forall j = 1...N, \forall l = 1...k : \sum_{\forall \tau_i \in \Delta_T | x_i = j} a_{i,l} \leq 1$

For all tasks assigned to the same virtual gang, the accelerator required by each task must be different from the accelerator required by every other task in the virtual gang. In other words, no two tasks in the same virtual gang should require execution on the same accelerator in the SoC.

Under this constraint, for the taskset in example in Sec. 6.3.1, τ_1 with $\kappa_1 = (h_1 = 2, a_{1,0} = 1, a_{1,1} = 0, a_{1,2} = 0)$ can potentially be paired with τ_2 that has $\kappa_2 = (h_2 = 2, a_{2,0} = 0, a_{2,1} = 1, a_{2,2} = 0)$ but it cannot be paired with τ_4 that has $\kappa_4 = (h_4 = 1, a_{4,0} = 1, a_{4,1} = 0, a_{4,2} = 0)$ since that would make $\sum_{i \in (1,4)} a_{i,0} = 2 \nleq 1$; violating the Constraint 7. In other words, the constraint would disallow pairing of τ_1 and τ_4 into the same virtual gang because both these tasks require integrated GPU for their execution.

Since tasks in the candidate-set can use the accelerators in a non-preemptive manner which can result in blocking of the higher priority tasks, when these tasks are combined into virtual gangs, the question arises as to the blocking time of the virtual gang that contains multiple accelerator using tasks. Due to Constraint 7, this question is easy to answer from a conceptual perspective: the maximum blocking time of a virtual gang is equal to maximum blocking time of any of its constituent tasks. This is specified by the following constraint:

Algorithm 8: Updated Virtual Gang Formation Heuristic

```
1  Input: Candidate Set (Δ_T), Number of Cores (m)
2  Output: Taskset comprising virtual gangs
3  function gang_formation(Δ_T, m)
4      pq = sort_tasks_by_wcet(Δ_T)
5      virtualGangs = ()
6      while not_empty(pq) do
7          τ_i = pq.pop()
8          f_i = family(τ_i)
9          partners = ()
10         for τ_j ∈ pq do
11             if τ_i.h + τ_j.h ≤ m ∧ τ_j ∉ f_i ∧ ∀ l ∈ (1,2,···,k) a_{i,l} = 1 ⟺ a_{j,l} ≠ 1 then
12                 partners ← partners ∪ {τ_j}
13             end
14         end
15         pq_i = score_partners(partners)
16         while not_empty(pq_i) do
17             τ_p = pq_i.pop()
18             τ_i = merge(τ_i, τ_p)
19             pq.remove(τ_p)
20             update_partners(τ_i, pq_i)
21         end
22         virtualGangs ← virtualGangs ∪ {τ_i}
23     end
24 return virtualGangs
```

Constraint 8 $\forall j = 1...N, \forall \tau_i \in \Delta_T \mid x_i = j : B_j \geq b_i$

The maximum blocking time B_j of the j-th virtual gang must be greater than or equal to the maximum blocking time of each of its constituent tasks; the above formulation ensures that the constraints are expressed in linear arithmetic by removing the max. Although this constraint is easy to understand, it requires additional implementation consideration to ensure that the blocking time of the virtual gang does not exceed the calculated value in the actual system. We will discuss these changes later on in this chapter.

Similar to the modified SMT based algorithm, the gang formation heuristic 5.4.2 needs to be modified to tackle the new problem. The updated heuristic is shown in Algorithm 8. Compared to before, the only change in the algorithm is in *line-11* in which all possible partners τ_j of a task

τ_i are checked. The check is modified such that τ_j is cleared for pairing with τ_i only if both these tasks use a disjoint set of accelerators.

6.4 Implementation Changes in the RTG-Sync Framework

In order to support virtual gang scheduling with accelerator using tasks, we need to make certain implementation changes in the RTG-Sync framework, to ensure that the maximum blocking time of a gang does not exceed the longest blocking duration of any of its constituent tasks; as per the Constraint 8 described in the previous section. In the following, we first illustrate the need of this implementation change with the help of an example and then explain the non-preemptive locking protocol that we have integrated into the RTG-Sync framework to address this issue.

Task	WCET (c-msec)	Blocking Time (b-msec)	Compute Demand ($\kappa = (h, a_0, a_1)$	Resource Demand (r)	Period (T-msec)
τ_1	20	8	(1, 1, 0, 0)	40	100
τ_2	22	7	(1, 0, 1, 0)	30	100
τ_3	8	0	(1, 0, 0, 0)	50	50

Table 6.2: Taskset parameters for the example illustrating the calculation of virtual gang blocking time under RTG-Sync.

Example: Consider the simple taskset shown in Table 6.2; comprising three single-threaded tasks. As per the RMS priority assignment policy [26], τ_1 and τ_2 have the same priority whereas τ_3 has higher priority than both these tasks. In this taskset, a single virtual gang v is formed by pairing τ_1 and τ_2. The blocking time of this virtual gang, as per the Constraint 8 of SMT gang formation algorithm, should be $B_v = \max(8,7) = 8$ i.e., in the worst-case, a higher priority task can expect to be blocked a maximum of 8-msec by this virtual gang.

Figure 6.2 shows an example scheduling timeline of this taskset. In this timeline, the virtual gang comprising (τ_1, τ_2) starts executing at $t = 0$. At $t = 4$, τ_2 starts its non-preemptive execution (e.g., on an accelerator). At $t = 6$, the higher priority task τ_3 arrives but it cannot start its execution due to the non-preemptive section of τ_2. Hence it gets blocked. At $t = 10$, τ_3 begins its

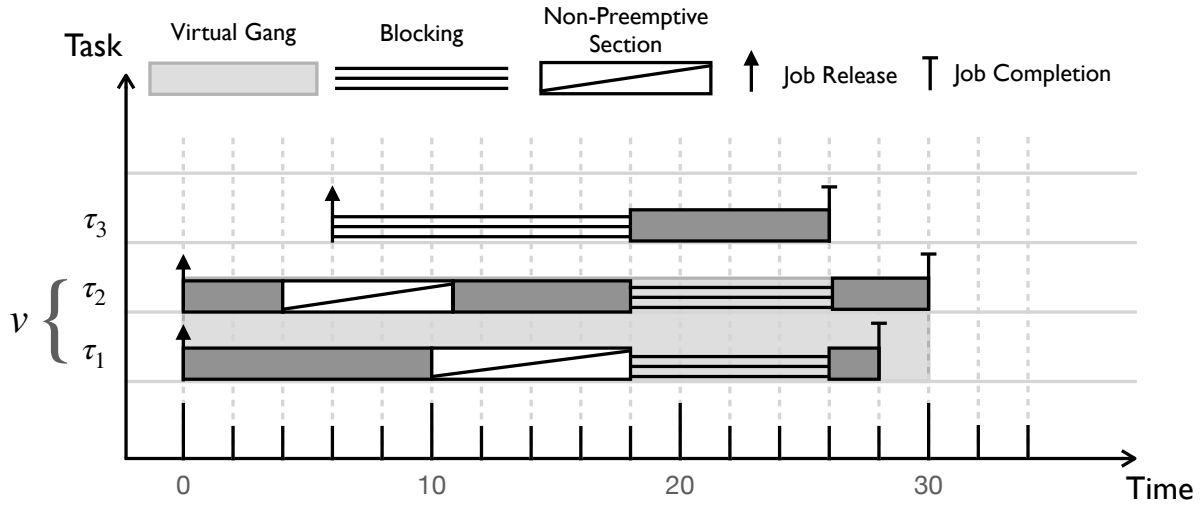


Figure 6.2: Scheduling diagram illustrating a problematic locking behavior, of the virtual gang from the taskset in Table 6.2, for executing non-preemptive sections. In this case, the blocking time experienced by the higher priority task τ_3 is 12-msec; which is greater than the expected value of 8-msec (the maximum blocking time among the constituent tasks of the virtual gang).

non-preemptive execution (e.g., on a different accelerator than τ_2's) because there is no existing mechanism in the RTG-Sync framework which prevents it from doing that. At $t = 11$, τ_2 completes its non-preemptive execution; however, τ_3 cannot get scheduled since τ_1 has already started its non-preemptive section. At $t = 18$, τ_1 finishes its non-preemptive section at which time, τ_3 gets unblocked and preempts the virtual gang of τ_1 and τ_2. τ_3 completes its execution at $t = 26$. Then τ_1 and τ_2 get unblocked and are able to complete their execution at $t = 28$ and $t = 30$ respectively. In this example, the blocking time experienced by the higher priority task τ_3 due to the virtual gang $v = (\tau_1, \tau_2)$ is 12-msec which is greater than the maximum blocking time $B_v = 8$-msec of the virtual gang.

The desired locking behavior of the virtual gang, for the taskset in Table 6.2, can be seen in Figure 6.3. In this figure, at $t = 10$, τ_1 is not allowed to begin its non-preemptive section because a higher priority task τ_3 has arrived and it is waiting for the current non-preemptive section of τ_2 to finish. At $t = 11$, when τ_2 completes its non-preemptive execution, τ_3 gets unblocked and immediately preempts τ_2; to start its own execution. At $t = 19$, τ_3 finishes at execution, at which time, the virtual gang $v = (\tau_1, \tau_2)$ resumes execution. In this case, the blocking time of τ_3 is 5-msec which is within the analytically computed maximum value of $B_v = 8$-msec.

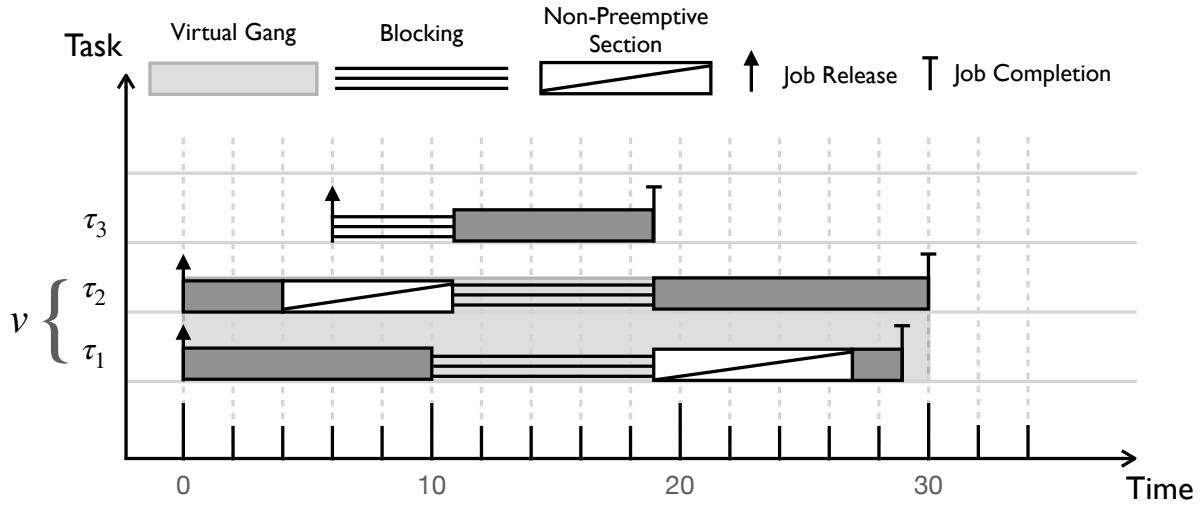


Figure 6.3: Scheduling diagram illustrating the desired locking behavior for the taskset in Table 6.2. In this case, the virtual gang member τ_1 is not allowed to execute the non-preemptive section at $t = 10$ since the higher priority task τ_3 is already waiting for the virtual gang to finish its current non-preemptive section of τ_2.

Having established the desired behavior of the member tasks of a virtual gang for executing non-preemptive sections, in the following, we discuss the changes that we have made to the RTG-Sync framework to elicit this behavior.

6.4.1 Gang Scheduling Data-Structure

Data-Structure 2 Modified Gang Lock Data-Structure

```
struct glock
    spinlock_t        lock;
    bool              held_flag;
    bool              hp_waiting;
    int               no_preempt;
    bitmask           locked_cores;
    bitmask           blocked_cores;
    task_struct_t*    leader;
    task_struct_t*    gthreads[NR_CPUS];
```

In order to track the currently executing non-preemptive sections by the virtual gang and any high priority blocked tasks that are waiting to for those sections to complete, we add two new fields to our kernel level gang lock data-structure. `hp_waiting` is a binary flag: if it is set to

Algorithm 9: Non-Preemption System Call

```
1  syscall sys_npp_lock(val)
2      if val ∈ (0,1) ∧ RT_Task(current) then
3          spin_lock(glock → lock)
4          if val == 0 ∧ glock → no_preempt > 0 then
5              /* Task is finishing a non-preemptive section. */
6              glock → no_preempt −= 1
7              if glock → no_preempt == 0 ∧ glock → hp_waiting == True then
8                  /* Execute the waiting high priority task. */
9                  glock → hp_waiting = False
10                 resched_cpus(glock → blocked_cores)
11                 clear_mask(glock → blocked_cores)
12             else if val == 1 ∧ glock → hp_waiting == False then
13                 /* Allow task to enter non-preemptive section. */
14                 glock → no_preempt += 1
15             else
16                 /* The action being attempted is not allowed.
17                  Return an appropriate status code to the caller. */
18             end
19         end
20     end
21 return
```

true, then it means a higher priority task has arrived but cannot begin its execution due to the non-preemptive segment of the executing gang task. `no_preempt` is an integer value: it indicates how many non-preemptive segments are currently being executed by the scheduled gang task. If it is zero, then it means the scheduled gang task is not executing any non-preemptive sections and it can be preempted in favor of higher priority tasks. We will now describe how these values are updated and used by our gang scheduling algorithm.

6.4.2 Non-Preemption System Call

We add a new system call to the Linux kernel to let a real-time task inform the kernel level gang scheduler when it wants to starts executing a non-preemptive section of code (e.g., accelerator execution segment). The pseudo-code of this system call is shown in Algorithm 9. The call receives a single integer value *val* from the caller which can be either 0 or 1. If the caller is informing the

Algorithm 10: Updated Gang Preemption Protocol

```
1  function do_gang_preemption()
2      if glock → no_preempt > 0 then
3          /* Preemption is not allowed. */
4          set_bit(this_cpu, glock → blocked_cores)
5          glock → hp_waiting = True
6      else
7          /* Perform gang preemption. */
8          for_each_locked_core (cpu, glock → locked_cores)
9              gthreads[cpu] = null
10         reschedule_cpus(glock → locked_cores)
11         clear_mask(glock → locked_cores)
12     end
13 return
```

kernel about the end of non-preemptive execution, then the `no_preempt` field of the gang lock data-structure is decremented (*line-6*). After the subtraction, if `no_preempt` value is 0 and a high priority task is waiting i.e., `hp_waiting` is true (*line-7*), then a rescheduling interrupt is sent to all the cores that have tasks blocked on them; so that the highest priority blocked real-time task may immediately start its execution (*lines-9:11*).

If the caller is informing the kernel about starting a non-preemptive section i.e., $val = 1$, then it is checked whether a high priority task is currently waiting for an on-going non-preemptive section to finish. If a high priority task is not waiting, then the caller is allowed to begin its non-preemptive execution and the `no_preempt` filed in the gang lock data-structure is incremented to reflect that (*line-14*). If the caller is attempting an action that does not fall in the categories described above, then an appropriate status is returned (*line-15*); without making any changes to the gang scheduling data-structure.

6.4.3 Gang Preemption Protocol

We make a minor change to the gang preemption protocol mentioned in Algorithm 5 to enforce the new locking behavior. The updated algorithm is shown in Algorithm 10. In this algorithm, during the gang preemption call, we first check whether the scheduled gang is executing a non-preemptive

segment (*line-2*); using the `no_preempt` value in the gang lock data-structure. If this is the case, then the preemption request is denied. Instead, the `hp_waiting` flag is set to indicate that a higher priority task is currently blocked due to the non-preemptive execution of the scheduled gang. The `blocked_cores` mask is also updated to reflect the same (*lines-4:5*). If a non-preemptive section is not currently being executed i.e., `no_preempt == 0`; then the gang preemption request is allowed as before (*lines:8-11*).

6.5 Future Work

There are multiple directions in which the work presented in this dissertation can be extended in the future. One of the limitations of our novel gang scheduling techniques is their applicability to sporadic tasks. Although we admit that we have focused our discussion on periodic tasks, we do not see a conceptual or implementation barrier in extending our techniques to the more general sporadic task model; although it will require careful consideration of the nuances associated with analyzing sporadic task systems. Extending our gang scheduling techniques so that our analysis methodologies can be used to analyze sporadic task systems can be an immediately useful extension of our work.

A limitation of our virtual gang formation technique presented in Sec. 5.4[3] is that we only pick tasks that have the same period to form virtual gangs. A relaxation of this limitation can be to allow virtual gang formation among tasks that have harmonic periods. According to our understanding, before this relaxation is allowed, it will have to be carefully analyzed whether the optimization criterion, used in the current virtual gang formation algorithms, stays valid when tasks in the candidate-set do not have the same period. Moreover, the schedulability analysis of virtual gang tasks will have to be revisited for this case.

Yet another extension can be to devise the blocking aware schedulability analysis of the extended RTG-Sync framework presented in this chapter. It may be assumed that the blocking-aware response-time analysis of a set of virtual gangs $\{w_1, w_2, \cdots, w_l\}$, parameterized according to our

[3]The same limitation applies to the extension proposed in Sec. 6.3.2.

new task model from Sec. 6.3.1, under the rate-monotonic priority assignment scheme [26], can be done using the following iterative relationship:

$$\mathbb{R}_i^{k+1} = \overline{C_i} + \max_{\forall w_j \in lp(w_i)} B_i + \sum_{\forall w_j \in hp(w_i)} \left\lceil \frac{\mathbb{R}_i^k}{T_j} \right\rceil C_j \tag{6.1}$$

where $\mathbb{R}_i^k$ is the response-time of w_i at the k-th iteration; $\overline{C_i}$ is the sum of the WCET of w_i itself and all the virtual gangs with the same period which come before w_i in the linear execution order; $lp(w_i)$ represents the set of all virtual gangs which have lower priority than w_i and $hp(w_i)$ represents the set of all virtual gangs which have higher priority than w_i (i.e., smaller period).

However, this analysis can hold only if the non-preemptive execution segments of the member tasks of a virtual gang do not partially overlap with each other; as in Figure 6.2. If the non-preemptive sections partially overlap, then the *shape* of the virtual gang can change when one of its non-preemptive sections is delayed due to a waiting high priority task. This can cause the normal and non-preemptive sections of the virtual gang to get overlapped differently than their empirically evaluated form under which the virtual gang was parameterized. This situation can be seen in Figure 6.3. One way to enforce this behavior i.e., no partial overlap of the non-preemptive sections of a virtual gang, is to use a synchronization scheme—similar to the one in Sec. 5.3 that is used for synchronous release of virtual gang member tasks—to align the non-preemptive sections of a virtual gang. This would allow the non-preemptive sections of the virtual gang to be fully contained within each other which would in turn cause the shape of the virtual gang to stay the same irrespective of the activation pattern of the high priority tasks. Updating the implementation of the RTG-Sync framework to include this feature can also be a useful extension of our work.

Chapter 7

Conclusion

In this dissertation, we analyzed the problem of ensuring deterministic execution of hard real-time tasks on heterogeneous multicore platforms. We first established the importance and complexity of the problem by discussing the effects of shared hardware resource interference which creates a *coupling* among all the real-time tasks in the system—through the OS level scheduling policy—that increases pessimism in the estimation of the worst-case execution time of the real-time tasks. We also explained how the presence of accelerators (e.g., integrated GPU), in the heterogeneous SoC, exacerbates this problem.

We presented three frameworks in this dissertation to address this non-determinism problem: 1) A novel CPU-GPU scheduling framework, called BWLOCK++, that ensures predictable execution of critical GPU kernels on integrated CPU-GPU platforms. 2) A novel gang scheduling framework, called RT-Gang, which guarantees deterministic execution of parallel real-time tasks on the multicore CPU cluster of a heterogeneous SoC. 3) The RTG-Sync framework which introduces the notion of virtual gangs, along with algorithms for virtual gang formation, that increases real-time schedulability under the RT-Gang framework. We also described extensions to these algorithms to incorporate scheduling on accelerators in a heterogeneous SoC. Throughout our discussion, we presented concrete evaluation results using simulated tasksets as well as real-world workloads that demonstrate the analytical and practical benefits of the proposed techniques.

We believe that our presented techniques introduce a new direction in tackling the problem of non-deterministic execution of real-time tasks due to interference in the shared hardware resources in the heterogeneous SoCs. We have also identified future directions for extending our work; to make our techniques more generally applicable to well-known real-time task models. Using our

techniques, the designers of real-time systems have a way to circumvent the **one-out-of-m** core problem without sacrificing the determinism of the overall system. We consider this the main achievement of our work.

References

[1] "Jetson TX2 Module." `https://developer.nvidia.com/embedded/jetson-tx2`.

[2] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[3] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[4] M. G. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[5] Certification Authorities Software Team, "CAST-32: Multi-core Processors (Rev 0)," tech. rep., Federal Aviation Administration (FAA), May 2014.

[6] Certification Authorities Software Team, "CAST-32A: Multi-core Processors," tech. rep., Federal Aviation Administration (FAA), November 2016.

[7] L. Sha, R. R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[8] W. Ali and H. Yun, "Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[9] I. Molnar, "Modular scheduler core and completely fair scheduler." `https://lwn.net/Articles/230501`.

[10] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, p. 46–61, 1973.

[11] A. K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[12] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS)*, pp. 259–268, IEEE, 2010.

[13] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[14] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 321–330, IEEE, 2012.

[15] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for synchronous parallel tasks on multicore platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 25–34, IEEE, 2013.

[16] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Döbel, and H. Härtig, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *2013 25th Euromicro Conference on Real-Time Systems*, pp. 215–224, IEEE, 2013.

[17] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, pp. 36–43, 2000.

[18] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[19] M. Ojail, R. David, Y. Lhuillier, and A. Guerre, "Artm: A lightweight fork-join framework for many-core embedded systems," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1510–1515, IEEE, 2013.

[20] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Real-Time Systems Symposium (RTSS)*, pp. 63–72, IEEE, 2012.

[21] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of DAGs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 12, pp. 3242–3252, 2014.

[22] J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," in *International Conference on Real-Time Networks and Systems (RTNS)*, pp. 189–196, 2010.

[23] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *Real-Time Systems Symposium (RTSS)*, pp. 459–468, IEEE, 2009.

[24] Z. Dong and C. Liu, "Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems," in *Real-Time Systems Symposium (RTSS)*, pp. 128–138, 2017.

[25] S. Wasly and R. Pellizzoni, "Bundled scheduling of parallel real-time tasks," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 130–142, IEEE, 2019.

[26] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[27] S. Baruah and A. Burns, "Sustainable Scheduling Analysis," in *RTSS*, pp. 159–168, 2006.

[28] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Real-Time Systems Symposium (RTSS)*, pp. 166–171, 1989.

[29] B. Andersson, S. Baruah, and J. Jonsson, "Static-Priority Scheduling on Multiprocessors," in *RTSS*, pp. 193–202, 2001.

[30] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic dag tasks for global fp scheduling," in *International Conference on Real-Time Networks and Systems (RTNS)*, p. 28–37, 2017.

[31] S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem," *Operations Research*, vol. 26, no. 1, p. 127–140, 1978.

[32] S. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms," in *Real-Time Applications Symposium (RTAS)*, pp. 536–543, IEEE, 2004.

[33] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & amp; Co., 1979.

[34] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms," in *ECRTS*, pp. 247–258, 2007.

[35] T. P. Baker and S. K. Baruah, "Schedulability Analysis of Multiprocessor Sporadic Task Systems," *Handbook of Real-Time and Embedded Systems*, pp. 3–1, 2007.

[36] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[37] D. Thiebaut and H. S. Stone, "Footprints in the Cache," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 4, pp. 305–329, 1987.

[38] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2009.

[39] F. Mueller, "Compiler Support for Software-Based Cache Partitioning," *ACM Sigplan Notices*, vol. 30, no. 11, pp. 125–133, 1995.

[40] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-Controlled Cache Predictability for Real-Time Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 213–224, 1997.

[41] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 367–378, 2008.

[42] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pp. 89–102, 2009.

[43] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 258–269, 2008.

[44] X. Ding, K. Wang, and X. Zhang, "Srm-buffer: An os buffer management technique to prevent last level cache from thrashing in multicores," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys, pp. 243–256, 2011.

[45] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 157–167, 2013.

[46] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical os-level cache management in multi-core real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 80–89, 2013.

[47] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: A dynamic cache partitioning system using page coloring," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 381–392, 2014.

[48] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, 2014.

[49] D. B. Kirk, "SMART (Strategic Memory Allocation for Real-Time) Cache Design," in *RTSS*, pp. 229–230, IEEE, 1989.

[50] I. V. Devereux, "Management of caches in a data processing apparatus," 2003. US Patent 6,671,779.

[51] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 657–668, IEEE, 2016.

[52] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 45–54, 2013.

[53] D. M. Durham, R. L. Sahita, D. C. Larson, and R. S. Yavatkar, "Page coloring to associate memory pages with programs," July 12 2016. US Patent 9,390,031.

[54] N. Kim, J. Erickson, and J. H. Anderson, "Mixed-criticality on multicore (mc2): A status report," in *Proceedings of the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 45–50, 2014.

[55] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 33–47, 2016.

[56] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: Server based gpu arbitration mechanism for memory accesses," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2017.

[57] B. Forsberg, A. Marongiu, and L. Benini, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *Design, Automation & Test in Europe (DATE)*, (3001 Leuven, Belgium, Belgium), pp. 318–321, European Design and Automation Association, 2017.

[58] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 367–375, 2012.

[59] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *IEEE International Conference on Computational Science and Engineering (CSE)*, pp. 685–692, 2013.

[60] H. Yun, W. Ali, S. Gondi, and S. Biswas, "BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms," *IEEE Transactions on Computers (TC)*, vol. PP, no. 99, pp. 1–1, 2016.

[61] R. Pellizzoni and H. Yun, "Memory servers for multicore systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–12, 2016.

[62] A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler, "Analysis of dynamic memory bandwidth regulation in multi-core real-time systems," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 230–241, IEEE, 2018.

[63] F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun, "Deterministic Memory Abstraction and Supporting Multicore System Architecture," *arXiv preprint arXiv:1707.05260*, 2017.

[64] Intel, "Improving real-time performance by utilizing cache allocation technology." `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology`.

[65] Y. Xiang, C. Ye, X. Wang, Y. Luo, and Z. Wang, "Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor," in *Proceedings of the 48th International Conference on Parallel Processing*, pp. 1–12, 2019.

[66] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. C. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[67] N. Otterness, M. Yang, S. Rust, and E. Park, "Inferring the scheduling policies of an embedded cuda gpu," in *Workshop on Operating Systems Platforms for Embedded Real Time Systems Applications (OSPERT)*, 2017.

[68] B. Forsberg, A. Marongiu, and L. Benini, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *Design, Automation & Test in Europe (DATE)*, 2017.

[69] B. Forsberg, L. Benini, and A. Marongiu, "HePREM: Enabling predictable GPU execution on heterogeneous SoC," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 539–544, IEEE, 2018.

[70] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[71] S. A. Panchamukhi and F. Mueller, "Providing task isolation via tlb coloring," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 3–13, 2015.

[72] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," *Real-Time Systems*, vol. 53, no. 5, pp. 709–759, 2017.

[73] W. Ali and H. Yun, "Work-in-progress: Protecting real-time gpu applications on integrated cpu-gpu soc platforms," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 141–144, IEEE, 2017.

[74] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni, "Dynamic memory bandwidth allocation for real-time gpu-based soc platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3348–3360, 2020.

[75] "Nvidia jetson platforms." `https://developer.nvidia.com/embedded-computing`.

[76] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," tech. rep., University of Illinois at Urbana-Champaign, 2012.

[77] S. Kato, M. McThrow, C. Maltzahn, and B. Scott, "Gdev: First-class gpu resource management in the operating system," in *USENIX Annual Technical Conference (ATC)*, 2012.

[78] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[79] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server based approach for predictable gpu access control," in *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[80] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[81] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 24. Springer Science & Business Media, 2011.

[82] G. K. Hartman, "Modifying a dynamic library without changing the source code | linux journal." `http://www.linuxjournal.com/article/7795`.

[83] "IsolBench code repository." `https://github.com/CSL-KU/IsolBench`.

[84] P. K. Valsan, H. Yun, and F. Farshchi, "Addressing isolation challenges of non-blocking caches for multicore real-time systems," *Real-Time Systems*, vol. 53, no. 5, pp. 673–708, 2017.

[85] W. Ali and H. Yun, "Rt-gang: Real-time gang scheduling framework for safety-critical systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[86] H. Yun and P. K. Valsan, "Evaluating the isolation effect of cache partitioning on cots multicore platforms," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2015.

[87] A. Hamann, "Industrial challenges: Moving from classical to high performance real-time systems," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2018.

[88] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *CoRR*, vol. abs/1604.07316, 2016.

[89] NVIDIA, "NVIDIA BB8 Self-Driving Car." `https://blogs.nvidia.com/blog/2017/01/04/bb8-ces/`, 2017.

[90] L. Sha, T. Abdelzaher, K.-E. AArzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, no. 2-3, pp. 101–155, 2004.

[91] B. Sprunt, *Aperiodic Task Scheduling for Real-time Systems*. PhD thesis, Carnegie Mellon University, 1990. AAI9107570.

[92] M. G. Bechtel, E. McEllhiney, and H. Yun, "DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car," in *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018.

[93] I. Molnar, "Cfs scheduler." `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[94] S. Rostedt, "Using KernelShark to analyze the real-time scheduler," *Linux Weekly News (LWN)*, 2011.

[95] C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE transactions on computers*, vol. 47, no. 6, pp. 700–713, 1998.

[96] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and distributed Computing*, vol. 16, no. 4, pp. 306–318, 1992.

[97] H. Y. Waqar Ali, Rodolfo Pellizzoni, "Virtual Gang Scheduling of Parallel Real-Time Tasks," in *Design, Automation & Test in Europe (DATE)*, European Design and Automation Association, 2021.

[98] S. Kato *et al.*, "An Open Approach to Autonomous Vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[99] J. Goossens and P. Richard, "Optimal scheduling of periodic gang tasks," *Leibniz Transactions on Embedded Systems (LITES)*, vol. 3, no. 1, pp. 04:1–04:18, 2016.

[100] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008.

[101] "RT-Gang code repository." https://github.com/CSL-KU/RT-Gang.

[102] WATERS, "WATERS 2018 Challenge RESSAC Use-Case." https://github.com/AdaCore/RESSAC_Use_Case, 2018.

[103] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017.