FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Integrating Dataflow and Non-Dataflow Real-time Application Models on Multi-core Platforms

Hazem Ismail Abdelaziz Ali

Programa Doutoral em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Dr. Luís Miguel Rosário da Silva Pinho

Co-Supervisor: Dr. Kjell Benny Åkesson

June 13, 2017

Integrating Dataflow and Non-Dataflow Real-time Application Models on Multi-core Platforms

Hazem Ismail Abdelaziz Ali

Programa Doutoral em Engenharia Eletrotécnica e de Computadores

Approved by :

June 13, 2017

Abstract

Day by day, gradually and steadily, applications in all segments of computing, including embedded systems, are getting more complex, because of the increased range of functionality they offer. This complexity requires platforms with increased performance that satisfies such growing computational demands. This need has driven the adoption of multi-core processors in embedded systems, since they allow performance to be increased at a reasonable energy consumption.

Future real-time embedded systems will increasingly incorporate mixed application models with timing constraints running on the same multi-core platform. These application models are dataflow applications with timing constraints and traditional real-time applications modelled as independent arbitrary-deadline tasks. Examples of such mixed embedded systems are Autonomous Driving Systems and Unmanned Ariel Vehicles. These systems require guarantees that all running applications execute satisfying their timing constraints. Also, to be cost-efficient in terms of design, they require efficient mapping strategies that maximize the use of system resources to reduce the overall cost.

This work proposes a complete approach with a main goal to integrate mixed application models (dataflow and traditional real-time applications) with timing requirements on the same multi-core platform. This approach guarantees that the mapped applications satisfy their timing constraints and maximize utilization of the platform resources. Three main algorithms to achieve the main goal. The first algorithm is called *slack-based merging*, which is an offline dataflow graph reduction technique that aims to decrease the complexity of dataflow applications, and thereby their analysis time. The algorithm reduces the run-time of our approach with 82% to 90%, compared to when it is not used. The experimental evaluation with real application models from the SDF^3 benchmark shows that the reduced graph: **1)** respects the timing constraints, i.e. *throughput* and *latency*, of the original application graph and **2)** when the throughput constraint is relaxed with respect to the maximal throughput of the graph, the merging algorithm is able to achieve a larger reduction in graph size.

The second algorithm is called *Timing Parameter Extraction*, which extracts timing parameters, i.e. *offsets*, *periods* and *deadlines*, of dataflow applications with timing constraints, i.e. *throughput* and *latency*, converting them into periodic arbitrary-deadline tasks. These tasks execute in a way that preserve the dependencies of the original dataflow application using the *offset* parameter, while satisfying its timing constraints using the *period* and *deadline* parameters. This algorithm is a means to *unify the two mixed application models* into a single real-time task set. The main advantage of this algorithm is that the extraction of the timing parameters is independent of the specific scheduler being used, of other applications running in the system and the details of the particular platform. In addition, the experimental evaluation shows that the reduced-size dataflow graphs generated by the *slack-based merging* algorithm, in particular for applications that do not need to execute at maximum throughput, help speeding up the extraction of the timing parameters.

The third algorithm is called *communication-aware mapping*, which allocates the mixed application models on a 2D-Mesh multi-core platform after unifying them. The mapping process is done considering the timing constraints of the applications and maximizing resource utilization of the platform, while accounting for the communication cost of the dataflow applications. The algorithm is based on a novel mapping heuristic called *Sensitive-Path-First*, which surpasses the well-known First Fit bin-packing heuristic in terms of number of allocated applications and runtime by up to 28% and 22%, respectively. The experimental evaluation reveals a direct relation between the number of allocated applications and the availability of communication resources, which demonstrates the importance of considering communication cost. We also show that ignoring communication cost, as frequently done in existing work, allows 76% more applications to be mapped, although the applications in the system are no longer guaranteed to satisfy their timing constraints.

Together, these three important algorithms successfully achieve the main goal of this thesis and play a part in allowing embedded real-time systems to map and schedule mixed application models. The complete approach and the three algorithms presented in this thesis have been validated through proofs and experimental evaluation.

Resumo

À semelhança do que acontece noutros domínios da computação, os sistemas embebidos estão cada vez mais complexos, devido ao aumento e diversidade das funcionalidades que fornecem, o que tem levado à necessidade de plataformas com maior desempenho. Esta exigência tem levado à cada vez maior adoção de plataformas multi-núcleo de processamento (*multi-core*) neste tipo de sistemas, permitindo o aumento de desempenho com custos razoáveis de energia.

Os sistemas embebidos do futuro integrarão na mesma plataforma multi-núcleo aplicações com diferentes modelos de computação, e com requisitos temporais. Entre estas é expectável a necessidade de integrar aplicações tradicionais de tempo-real (modelizadas por tarefas independentes) com aplicações modelizadas por fluxos de dados (*dataflow*). Exemplos podem ser encontrados em sistemas de condução autónoma ou veículos aéreos sem piloto, sistemas que requerem a garantia de cumprimentos dos prazos temporais de todas as aplicações. Para além disso, são sistemas em que é fundamental a existência de estratégias automatizadas de mapeamento da computação que maximizem a utilização dos recursos disponibilizados pela plataforma.

Esta dissertação propõe uma metodologia completa para a integração numa só plataforma multi-núcleo de aplicações com modelos computacionais distintos (fluxo de dados e tradicionais tempo-real) e com requisitos temporais. Esta metodologia permite garantir que as aplicações cumprem com os seus requisitos temporais, ao mesmo tempo que maximiza a utilização dos recursos do sistema. Para este efeito, a metodologia inclui três algoritmos diferentes.

Num primeiro passo, é utilizado um algoritmo, *slack-based merging*, para reduzir a complexidade dos grafos de fluxo de dados com que são modelizadas as aplicações que utilizam este modelo computacional, o que permite reduzir o tempo de análise das mesmas. Este algoritmo permite reduzir o tempo de processamento do processo de 82% a 90%. A avaliação experimental com modelos de aplicações reais, do benchmark SDF^3 demonstra que o grafo reduzido: **1)** respeita os requisites temporais do grafo original, i.e., o desempenho (*throughput*) e a latência (*latency*), e **2)** quando se relaxa o requisito de desempenho em relação ao máximo permitido pelo grafo, o algoritmo permite uma maior redução do tamanho do grafo.

O segundo algoritmo, *Timing Parameter Extraction*, permite extrair as características temporais tradicionais de uma aplicação de tempo-real, i.e., períodos (*periods*), prazos (*deadlines*) e deslocamentos (*offsets*), a partir dos modelos de fluxo de dados com requisitos de desempenho (*throughput*) e latência (*latency*), convertendo assim estes fluxos em tarefas periódicas independentes. Estas tarefas executam de forma a preservar as dependências do modelo de fluxo de dados original através do deslocamento da ativação de tarefas consequentes, satisfazendo os requisitos de processamento e latência através dos períodos de ativação e prazos temporais. Este algoritmo permite assim unificar os dois modelos distintos de computação, num só conjunto de tarefas de tempo-real. A vantagem principal deste algoritmo é que esta extração de parâmetros é independente do escalonador utilizado, de outras aplicações que executam no sistema, e dos detalhes da plataforma. A avaliação experimental também demonstra que o tempo de processamento desta extração é reduzido pela redução dos grafos obtida pelo algoritmo anterior, particularmente para aplicações que não necessitam executar com o máximo desempenho.

O terceiro algoritmo, *communication-aware mapping*, mapeia as tarefas das aplicações que usam os dois modelos de computação, após unificação, em plataforma multi-núcleo com comunicação em 2 dimensões entre núcleos (*2D-Mesh*). O mapeamento é efetuado considerando os requisites temporais das aplicações, e maximiza a utilização dos recursos computacionais da plataforma, tendo em consideração os potenciais custos de comunicação. Este algoritmo é baseado numa noval heurística, *Sensitive-Path-First*, a qual obtém melhores resultados que a heurística *First-Fit*, tanto em termos de número de aplicações mapeadas como em tempo de processamento (28% e 22% melhor, respetivamente). A avaliação experimental mostra uma relação direta entre o número de aplicações mapeadas e a disponibilizada de recursos de Comunicação, o que demonstra a importância da consideração destes custos durante o mapeamento. Também mostramos que, ignorando os custos de comunicação, como é habitualmente feito em trabalhos semelhantes, permite mapear até 76% mais aplicações, embora sem conseguir garantir a satisfação dos seus requisitos temporais.

Em conjunto, estes três algoritmos importantes permitem atingir com sucesso o objetivo principal desta dissertação, potenciando o mapeamento e integração em sistemas embebidos de tempo-real de aplicações com modelos computacionais distintos. A metodologia complete e os três algoritmos apresentados na dissertação foram validados por provas e avaliação experimental.

Acknowledgements

Undertaking this PhD has been a truly life-changing experience. Like most research work, this PhD is the result of a curious and inquisitive spirit, coupled with plenty of hard work and persistence. Naturally, it was difficult at times, but overall, the fulfilling moments far exceeded the hardship ones. This research would not be possible to do it without the support and guidance that I received from a lot of people, to whom I will always be grateful.

First, I would like to express my sincere gratitude to my supervisor Prof. Luís Miguel Pinho for believing in me and giving me the chance to work with him. His continuous guidance, patience, motivation, and support through my entire PhD studies helped me in all time of research. Also, I wish to extend a sincere and heartfelt thanks to my co-supervisor Dr. Benny Akesson on both professional and personal level. On professional level, for his dedication and comprehensive assistance through my entire research journey. His sharp insights, valuable feedback and detailed discussions with him, helped in shaping up my research till this final outcome. On the personal level, Benny is one of the friendliest persons that you forget that he is actually your supervisor. He always maintains a personal relation with his students where he socialize and involve in different activities. I will never forget our interesting long runs, where we had fun and enjoyable discussions.

Second, a huge thank goes to Dr. Stefan Markus Petters. Although we did not work directly together, he was one of the main reasons to join CISTER research group. He was kind enough to listen to my counter argument, after he sent an email not accepting me for the PhD position. This normally does not happen in applying for PhD positions. I am really grateful for him.

Third and most important, none of this achievements would have been possible without the love and patience of my family. My parents, Ismail Abdelaziz Ali and Somaia Mohamed Elsayad, have been a constant source of love, concern, support and strength all these years. Especially my mother, Somaia Mohamed Elsayad, for the long hours she invested teaching me mathematics, algerbra and geometry that made me like engineering. I owe her what I am right now. Also, I would like to express my heartfelt gratitude to my brothers and sister, Mohamed Abdelaziz, Ahmed and Reham, for their continuous encouragement during my long research journey that started in 2008, going to sweden for doing my masters degree.

Fourth, my dear friend and CISTER companion Borislav Nikolić. We have spent more than six years together at CISTER, where we shared a very memorable moments of happiness, success and lifetime achievements. His valuable advice along with his cheerful and funny spirit made my PhD life easier. I deeply thank him very much. I will never forget such times and I wish you all the best in your life and career.

Fifth, Prof. Eduardo Tovar for creating an outstanding work environment in CISTER Research Center. I have always enjoyed the working environment in our office, with great office mates. Especially Muhammad Ali Awan and Claudio Maia for being good friends and colleagues. During these six years, we have had all the interesting discussions covering a variety of topics, such as technology, sports, culture etc. I would like to add that I feel fortunate to have known

Ricardo Garibay, Hossein Fotouhi, Maryam Vahabi, Artem Burmyakov, Kostiantyn Berezovskyi, Gurulingesh Raravi, Dakshina Dasari, António Barros, Paulo Baltarejo, Syed Aftab Rashid and Harrison Kurunathan during these years. Last but not the least, I extend my gratitude to all the staff members at CISTER Research Center, who have made these years more enjoyable.

This work was partially supported by FCT (Fundação para a Ciência e Tecnologia) under the individual doctoral grant SFRH/BD/79872/2011.

Hazem Ismail Ali

List of Publications

Articles Included in this Thesis

- Hazem Ismail Ali, Luís Miguel Pinho and Benny Akesson, "*Critical-Path-First based allocation of real-time streaming applications on 2D mesh-type multi-cores*," in **IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, Taipei, 2013**, pp. 201-208. doi: 10.1109/RTCSA.2013.6732220, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6732220&isnumber=6732192

- Hazem Ismail Ali, Benny Akesson and Luís Miguel Pinho, "*Generalized Extraction of Real-Time Parameters for Homogeneous Synchronous Dataflow Graphs*," in **23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turku, 2015**, pp. 701-710. doi: 10.1109/PDP.2015.57, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7092796&isnumber=7092002

- Hazem Ismail Ali, Sander Stuijk, Benny Akesson, and Luís Miguel Pinho. "*Reducing the complexity of dataflow graphs using slack-based merging*,". in **ACM Transactions on Design Automation of Electronic Systems**, 22, 2, Article 24 (January 2017), 22 pages. ISSN 1084-4309. doi: 10.1145/2956232. URL: http://dx.doi.org/10.1145/2956232

- Hazem Ismail Ali, Benny Akesson and Luís Miguel Pinho. *Combining dataflow applications and real-time task sets on multi-core platforms*. In **Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES '17)**, Sander Stuijk (Ed.). ACM, New York, NY, USA, 60-63. doi: 10.1145/3078659.3078671 URL: https://doi.org/10.1145/3078659.3078671

Other Articles

- Hazem Ismail Ali and Luís Miguel Pinho. 2011. "*A parallel programming model for ada*". **In Proceedings of the 2011 ACM annual international conference on Special interest group on the ada programming language (SIGAda '11)**. ACM, New York, NY, USA, 19-26. DOI=http://dx.doi.org/10.1145/2070337.2070350

- Borislav Nikolić, Hazem Ismail Ali, Stefan M. Petters, and Luís Miguel Pinho. 2013. "*Are virtual channels the bottleneck of priority-aware wormhole-switched NoC-based many-cores?*". **In Proceedings of the 21st International conference on Real-Time Networks and Systems (RTNS '13)**". ACM, New York, NY, USA, 13-22. DOI=http://dx.doi.org/10.1145/2516821.2516845

*“Science is not going to yield anything unless you devote yourself completely.
But even if you do devote yourself entirely, it remains uncertain whether you will get anything from it.”*

Ibrahim Al-Nazzam

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ADF	Affine Dataflow
BFS	Breadth First Search
bps	bits per second
CPF	Critical-Path-First
CP	Critical Path
CSDF	Cyclo-Static Dataflow
DAG	Directed Acyclic Graph
dbf	demand bound function
DCG	Directed Cyclic Graph
DF	Dataflow
DM	Deadline Monotonic
DSP	Digital Signal Processing
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
FF	First Fit
FIFO	First In First Out
GEDF	Global Earliest Deadline First
HSDF	Homogeneous Synchronous Dataflow
ILP	Integer Linear Programming
IN	Interconnection Network
MCM	Maximum Cycle Mean
MLLF	Modified Least Laxity First
MPAG	Max-Plus Automaton Graph
NDF	Non-Dataflow
NoC	Network on Chip
PEDF	Partitioned Earliest Deadline First
P/C	Production/Consumption
RM	Rate Monotonic
SCC	Strongly Connected Component
SDF	Synchronous Dataflow
SDM	Space Division Multiplexing
SADF	Scenario-Aware Dataflow
SPF	Sensitive-Path-First
SPP	Static-Priority Preemptive
TGFF	Task Graphs For Free
TDM	Time Division Multiplexing
TDMA	Time Division Multiple Access
TPE	Timing Parameters Extraction

WCET	Worst Case Execution Time
XML	eXtensible Markup Language

List of Symbols

Task Parameters

τ	A task set.
U	A task set τ utilization.
τ_i	The i^{th} task.
a_i	The offset of τ_i (seconds).
C_i	The WCET of τ_i (seconds).
T_i	The period of τ_i.
D_i	The relative deadline of τ_i (seconds).
$\overline{D}_i$	The absolute deadline of τ_i (seconds).
S_i	The arrival time of τ_i (seconds).
F_i	The finish time of τ_i (seconds).
U_i	The utilization of τ_i (seconds).
ρ_i	The density of τ_i.
J_i	The job of τ_i.
R_i	The response time of τ_i (seconds).

Feasibility Analysis

dbf (t_0, t_1)	The demand bound function within the time interval $[t_0, t_1]$.
H	Hyperperiod (seconds).
t_a	The upper bound on schedulability (seconds).
t_b	The synchronous busy period of the processor (seconds).
$h(t)$	The processor demand at time t (seconds).

Dataflow Applications

G	A Synchronous Dataflow graph (SDF).
V	Set of nodes (actors) in an SDF graph.
E	Set of edges (channels) in an SDF graph.
G_h	A Homogeneous Synchronous Dataflow graph (HSDF).
V_h	Set of nodes (actors) in an HSDF graph.
E_h	Set of edges (channels) in an HSDF graph.
G_p	A graph representing a pipeline application.
V_p	Set of nodes in a pipeline graph.
E_p	Set of edges in a pipeline graph.
G_m	A merged HSDF graph.
G_{com}	An HSDF graph with message actors.
d	Set of initial tokens in a dataflow (SDF/HSDF) graph.
Γ	Topology matrix of an SDF graph.
$\vec{q}$	Repetition vector of an SDF graph.

v_i	The i^{th} actor of an SDF graph.
v_{i_j}	The j^{th} firing of the i^{th} actor of an SDF graph.
e_{ij}	A channel starting from actor v_i to actor v_j.
ζ	Throughput requirement.
P_i	The i^{th} time-constrained path in an HSDF graph.
P_i^p	Partial path of the time-constrained path P_i.
LP_i^p	List of partial paths in P_i.
γ_i	The sensitivity of path P_i.
$\mathcal{D}$	End-to-end deadline constraint (seconds).
$\mathcal{D}_{xy}$	Latency constraint of a time-constrained path from v_x to v_y (seconds).
ε	laxity on a time-constrained path P (seconds).
δ	Task slack (seconds).
$\Omega(v_{i_j})$	Set of predecessor firings of the firing v_{i_j}.
$\Phi(v_{i_j})$	Set of successor firings of the firing v_{i_j}.
ϑ_{i_j}	Earliest start time of a firing v_{i_j} (seconds).
θ_{i_j}	Latest finish time of a firing v_{i_j} (seconds).
$\hat{V}$	Topologically ordered set of actors.
$\mathcal{V}$	Merged cluster of HSDF actors.
$\mathscr{P}$	Path cover for a DAG component of a G_h.
$\mathscr{O}$	The set of cycles in G_h.
σ_{i_j}	The slack of a firing v_{i_j} (seconds).
$\mathcal{C}_k$	The execution time of a cycle k (seconds).
β	A constant has a value with range $[1,\infty)$.
$\mathcal{P}$	The set of all time-constrained paths between actors with latency constraints in an HSDF graph.
$Succ(v_x)$	The list of successor actors for the actor v_x.

System Model

Ψ	The system.
Π	The multi-core platform.
π_i	The i^{th} core in the platform Π.
n	One of the dimensions of the multi-core platform Π.
l_{sw}	The router switching latency of a single flit (seconds).
l_t	The transfer latency of a single flit (seconds).
$\mathcal{L}$	The link capacity of IN of the platform Π (Gbps).
$\mathcal{R}_i$	The fraction of the link capacity $\mathcal{L}$ reserved for an application A_i (percentage).
h	The number of hops of a packet p.
$\hat{h}$	The maximum number of hops on IN of any packet on the platform Π.
$\mathcal{F}$	The TDM frame size (slots).
$\varkappa_i$	Number of allocated slots for application A_i (slots).
p	The packet size (bits).
f	The flit size (bits).
A	The application set.
A_i	The i^{th} application of the application set A.
A_p	A pipeline application.
m	The size of the application set A.
$C_{i,p}$	The time spent by a packet p of application A_i traversing the IN (seconds).
$C_{i,p}^{iso}$	The isolation time of a packet p of application A_i traversing the IN (seconds).

$\hat{C}_{i,p}$	The initial value of the WCET of a message actor (seconds).
I_i^{TDM}	The TDM interference time of any packet from application A_i traversing the IN (seconds).
$I_i^{TDM.co}$	The TDM interference time of any packet from application A_i traversing the IN, assuming continuous slot assignment policy (slots).
$\mathcal{G}$	The IN frequency.

Chapter 1

Introduction

We are living the golden age of ubiquitous computing. If we look around, we will find ourselves surrounded by computing devices embedded in systems that help or serve us in our daily life. These systems ranges from simple portable gadgets, e.g. smartphones, cameras, gaming consoles, to large complex systems, e.g. airplanes, cars, industrial automation. These systems are called *embedded systems*.

An embedded system can be broadly defined as a computing system that performs a dedicated function within a larger system [Jiménez et al., 2014]. This dedicated function is not designed to be programmed by the end user as functions in general purpose computing [Heath, 2002]. The concept of computing systems performing dedicated functions is old going back in time preceding the concept of a general-purpose computer [Jiménez et al., 2014]. If we look at the earliest forms of computing devices, they adhere better to the definition of an embedded system (in terms of performing a dedicated function) than to that of a general-purpose computer. An example of these devices is the Colossus computer [Copeland, 2006], which refers to a series of computers developed by British code-breakers in 1943-1945. Colossus dedicated function was to help in the cryptanalysis of the German teleprinter messages during World War II.

At early stages, embedded system designs used microcontrollers as a main processing unit, since the application demands were simple. Following the rise in application demands and growing complexity, many embedded systems incorporate multi-core processor architectures for satisfying the increasing demands of its applications, since the need for high processing power at a low power budget is a great concern for such systems [Kim et al., 2010]. A real life example of this trend is the cellular phone. At the beginning, the first generation of cellular phones incorporated a single core digital signal processor chip [PratapSingh and Kumar Jain, 2014], since its main dedicated function was making phone calls. However, the latest generations feature at least a quad-core multi-processor at least, e.g. Samsung Galaxy S7 smartphone incorporating Qualcomm® Snapdragon™ 820 processor [Qualcomm, 2016]. This is because the cellular phone has become a portable computer, multimedia and connectivity device.

The trend of the growing functionality of embedded systems can be demonstrated by the various types of applications that run simultaneously on the system [Jiménez et al., 2014]. These applications may have different requirements, such as computational demands or timing constraints. For example, the cellular phone runs a time-constrained application, which is the phone call, along with computationally intensive ones, such as multimedia and gaming applications. The fact that embedded systems run various applications with different requirements can mean different applications may be represented using different computational models. In such systems running mixed computational models, guarantees are required to assure stratifying requirements (computational demands or timing constraints) and the correct execution of the system, especially in case of safety-critical applications. A current example of such systems is high-end cars, which may run an advanced multimedia entertainment system (that requires huge computational resources) along with the autonomous driving function (safety-critical application) that allow self-driving on the highways, i.e. Tesla Model S, X and 3 [TESLA, 2016].

Embedded system running mixed computational models is an increasing futuristic trend, since embedded systems are included in almost every device. In this thesis, we are concerned with embedded systems that incorporate mixed computational models with timing constraints running on the same multi-core platform. These computational models are dataflow with timing constraints and traditional real-time task sets, since they represent a wide range of applications running on top of embedded systems. The dataflow computational model represents Digital Signal Processing (DSP), Streaming and multimedia applications, while traditional real-time computational model covers a wide range of time-constrained applications with different levels of criticality. Example of future embedded systems that run these two computational modes are Autonomous Driving Systems [Elliott et al., 2014] and Unmanned Air Vehicles [Zhou and Wu, 2006]. These kind of systems require real-time guarantees that all running applications will execute safely without missing their deadlines. Also, they require efficient use of system resources to minimize the overall cost of the system.

We begin this thesis by briefly introducing the two computational models considered in this thesis. They are the real-time computational model (Section 1.1) and the dataflow computational model (Section 1.2), where we detail the parameters and the properties of each model. Then we follow by presenting an overview of processing platforms and architectures in Section 1.3. After these introductory sections, we introduce our problem statement in Section 1.4, followed by a detailed proposed solution explaining its functionality in Section 1.5. Finally, we end this chapter by summarising our thesis contributions and providing the thesis organisation in Sections 1.6 and 1.7, respectively.

1.1 Real-time Computational Model

A **real-time computational model** is *a computing paradigm used to define a certain set of applications that have to respond to externally generated input stimuli within a finite and specified period of time* [Buttazzo, 2004, Krishna, 1996]. The main characteristic that distinguishes real-time computing from other types of computation is time, because the correct execution of the applications of such computational model depends not only on the logical result but also on the time it is delivered. The instant when a result must be produced is called a *deadline*. Failure to respond within the specified timing interval or a delayed response could be useless or even have fatal consequences. Based on these consequences, the real-time computational model classifies its applications into three categories [Buttazzo, 2004, Krishna, 1996]:

Hard real-time: An application is considered **hard real-time** if missing its deadline during execution may cause catastrophic consequences on the system under control, surrounding environment or people.

Firm real-time: An application is considered **firm real-time** if missing its deadline during execution is useless for the system, but does not cause any damage.

Soft real-time: An application is considered **soft real-time** if missing its deadline during execution has still some utility for the system, although causing performance degradation.

These are the three basic categories of applications according to the real-time computational model. There exist other classifications that branch from these basic categories. Whatever their category, all the applications in this computational model are called *real-time applications*. In the following section, we will shed more light on real-time applications and its different criteria classifications.

1.1.1 Real-time Applications

Real-time applications are wide-spread in daily life systems, e.g. telecommunications, aviation, nuclear reactors, autonomous driving systems , industrial automation. A real-time application can be modelled as a finite set of simple, highly repetitive entities that are recurrent in nature called real-time *tasks* [Baruah and Goossens, 2004]. Each instance of a task is a basic unit of work that executes on the processing platform and is called a *job* [Liu, 2000]. A real-time task has different classifications based on its timing parameters. In the following section we discuss that in details.

Real-time task classification:

A real-time task has several classifications that vary based on the criteria used. In this thesis, we are concerned with two criteria in real-time task classification. First, the frequency of which a task instantiates its jobs (task periodicity) classifies a real-time task into three categories [Isović and Fohler, 2000]:

Periodic tasks: A task that releases its jobs periodically after a fixed time interval is defined as a periodic task. The fixed duration between the two consecutive jobs releases is called the period of the task.

Sporadic tasks: A task that releases its jobs at some arbitrary time instant but two consecutive jobs of a task are always separated by at least a predefined time interval called the minimum inter-arrival time.

Aperiodic tasks: Jobs of an aperiodic task are not constrained by a minimum interarrival time or a period, the task can release jobs at any instant.

Periodic tasks are the most well-known model in real-time systems. Sporadic tasks can be converted into periodic tasks with a predefined minimum interarrival time [Buttazzo, 2004]. Aperiodic tasks can be handled using periodic server-based systems with budget. The server is modelled as a periodic task. The server can serve aperiodic tasks until the budget expires. The budget can be replenished every period [Sprunt, 1990].

Second, real-time tasks are always constrained with a timing requirement. A task should complete its execution within a predefined time interval called the *relative deadline*. The *relative deadline* of a task depends on the nature of an application. For example, the object recognition/detection application in an autonomous driving system has a relative deadline in terms of a few microseconds, while a room temperature monitoring application in an air conditioning system can have a relative deadline in terms of a few seconds. The *relative deadline* of a real-time task, whether it is periodic, sporadic or aperiodic, can be categorized into three main categories [Buttazzo, 2004, Krishna, 1996]:

Implicit-deadline task model: has a relative deadline equal to its period or minimum inter-arrival time.

Constrained-deadline task model: may have a relative deadline less than or equal to its period or minimum inter-arrival time.

Arbitrary-deadline task model: has a relative deadline that has no relation with the period or minimum inter-arrival time of a task. This means that the relative deadline can be set to any value regardless the value of the task's period.

In this thesis, we are concerned with real-time systems running *periodic arbitrary-deadline* tasks.

1.1.2 Worst-Case Execution Time

The execution time of a real-time task is an important parameter that defines its temporal behaviour. Different jobs of a task exhibit variation in their execution time depending on the hardware characteristics, structure of the software, input data and different behaviour of the environment with which the jobs are interacting. In order to guarantee the temporal correctness, the upper bound on the execution time of a task, referred to as the Worst-Case Execution Time (WCET), is specified. The WCET of a task is a safe upper bound greater than or equal to the longest execution of any job released by the task, under worst-case input conditions without interference from other tasks. Any miscalculation of WCET may cause a system failure depending on, whether or not, the system is a hard real-time. There are several methodologies and techniques to determine the WCET of a task detailed in [Puschner and Burns, 2000, Wilhelm et al., 2008] for further reading. Real-time system designers consider the WCET of tasks while designing a system to guarantee the timing properties. However, different jobs of a task may execute for less than their WCET

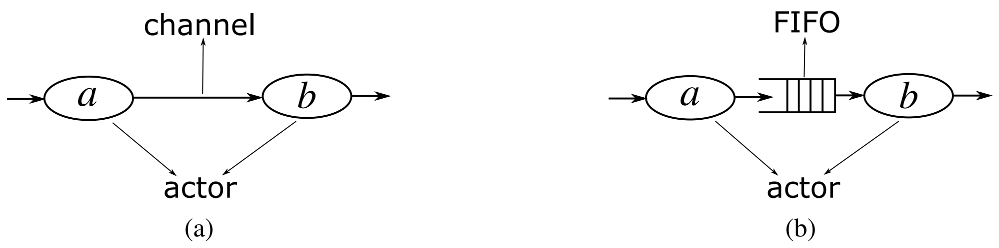


Figure 1.1: Dataflow application.

leaving behind unused computing resources. This bound is almost always pessimistic to be safe. Jobs hence typically execute faster.

1.2 Dataflow Computational Model

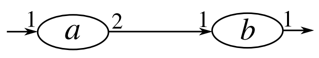
The dataflow computational model [Chamberlin, 1971, Estrin and Turn, 1963, Rodrigues, 1969, Shields, 1997] is a well-known, simple, and powerful model of parallel computation. In this model, there is no notion of a single point or locus of control corresponding to the conventional sequential computing. However, it models an application as a set of tasks with data dependencies. It is a very useful specification mechanism for signal processing systems since it captures the intuitive expressiveness of block diagrams, flow charts, and signal flow graphs, while providing the formal semantics needed for system design and analysis tools.

1.2.1 Dataflow Applications

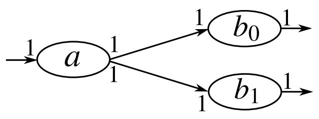
A dataflow application is a directed graph, where the vertices represent computation tasks and edges represent First-In First-Out (FIFO) queues that direct data values from the output port of one computation task to the input port of another. Hence, a dataflow application can be considered a set of computation tasks with dependencies. The graphs' vertices (computation tasks) are called *actors*, while its edges (FIFO queues) are called *channels*. Channels thus represent data dependencies between actors.

A dataflow application executes by performing the functions defined by its actors. An actor can be a single instruction, or a sequence of instructions, since the dataflow model does not imply a limit on the size or complexity of actors. Initially, an actor is an idle task. Its execution is triggered once the required amount of data arrives on its input ports. The amount of input data is specified by each actor according to its functional requirements. Many actors may be ready to execute simultaneously, and thus represent many asynchronous concurrent computation events. An actor starts execution by consuming data from its corresponding input ports, performing computations, and then produce a certain amount of data on its output ports. The execution process of an actor is called a *firing*, while the data produced or consumed in the firing process are referred to as *tokens*.

Figure 1.1 shows an example of a dataflow graph, that consists of actors (a, b) and the channel between them represented as a FIFO queue that direct tokens from the output port of actor a to the input port of actor b. Initially, actors a and b are idle. Once the required tokens are available



(a) SDF graph



(b) HSDF graph

Figure 1.2: Example of SDF and HSDF graphs.

on the input port of actor a, it consumes them, starting the firing process, then produces tokens on its output port. The tokens produced are transferred to the input port of actor b through the FIFO channel, triggering its firing process that results in producing tokens on its output port similar to actor a. The functions performed by the actors define the overall function of the dataflow graph. For example, Figure 1.1 could represent a water level control system, where actor a is measuring the current level of water in a tank and send signals to actor b that controls the operation of the water pump.

A dataflow application has three important timing parameters, they are:

Execution time of its actors: an actor may have different values of execution time. This may be due to different tokens consumed, which triggers different functions to be executed inside the actor. Also, it may be due to the same reasons a real-time task faces that are mentioned previously in Section 1.1.2. However, for predictable execution behaviour and analysis purposes, the execution time determined for each actor represents an upper bound (WCET) to all of its firing modes. The calculation of WCET is mentioned earlier in Section 1.1.2.

Throughput: is an important constraint and crucial indicator of performance for dataflow applications. The throughput of a dataflow application refers to how often an actor produces an output token. To compute throughput, the WCET of the firing of each actor has to be measured and an execution scheme must be defined. The execution scheme is the self-timed execution of actors, where each actor fires as soon as all of its input data are available [Sriram and Lee, 1997].

Latency: is a timing constraint that defines a time bounded interval between firings of two actors in the dataflow application. It can be realised as a relative deadline for the firings that happen between these specific two firings.

There exist several dataflow computational models, e.g. Synchronous Dataflow (SDF), Homogeneous Synchronous Dataflow (HSDF) [Lee and Messerschmitt, 1987b], Cyclo-static Dataflow (CSDF) [Bilsen et al., 1995], Scenario-Aware Dataflow (SADF) [Theelen et al., 2006], where each model have its own specifications and rules that enable capturing wide range of applications. *However, we focus on those that can be described by SDF and HSDF* [Lee and Messerschmitt, 1987b].

SDF: is useful for modelling and analysis of Digital Signal Processing (DSP) and concurrent multimedia applications [Lee and Messerschmitt, 1987b, Poplavko et al., 2003, Sriram and

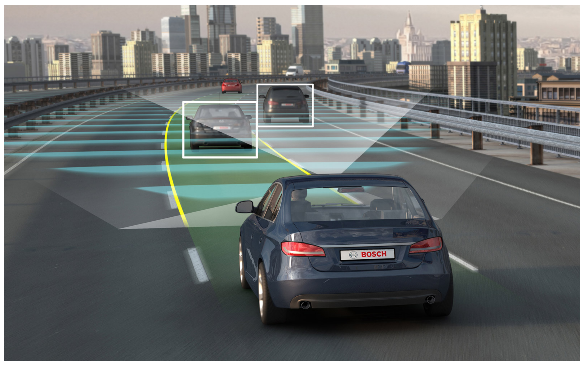
Bhattacharyya, 2000, Wiggers et al., 2007], where they represent computations on an indefinitely long data sequence. This is because of the ability to obtain periodic schedules for the SDF execution where actors fires a determined number of times with a specific order, in a cyclic manner, where each cycle called an *iteration*. Every actor in an SDF graph consumes/produces a fixed number of tokens every time it fires. The SDF graphs are accompanied with several timing analysis techniques, which are used for evaluating performance metrics of such applications, most importantly throughput. Figure 1.2(a) shows an example of an SDF graph that consists of two actors a and b. Actor a represents a source task that produces two tokens every time it fires (denoted on its output port), while actor b represents a sink task that consumes a single token every time it fires (denoted on its input port). The periodic schedule for such SDF graph is (a,b,b), because actor a produces two tokens that triggers actor b to fire twice consuming a single token each.

HSDF: is a more restricted model of SDF, where actors consume/produce a single token every time they fire. Each actor in an HSDF graph fires once during an iteration of the graph. This restriction allows HSDF graph to reveal the parallelism hidden in applications represented using more expressive models, e.g. SDF, CSDF. For example, Figure 1.2(b) shows an HSDF graph representation of the SDF graph shown in Figure 1.2(a). As we notice, the HSDF graph reveals the parallelism hidden in the SDF graph by showing actor b firing twice simultaneously (b_0, b_1). Many dataflow graphs expressive models, e.g. SDF, CSDF, can be converted to an equivalent HSDF graph by using a conversion algorithm, such as the one presented in [Sriram and Bhattacharyya, 2000]. Although transformation to HSDF allows revealing the parallelism in dataflow applications, it can lead to an exponential increase in the size of the original dataflow graph [Lee and Messerschmitt, 1987a, Sriram and Bhattacharyya, 2000], which may result in a significant increase in the run-time of many dataflow analysis algorithms, e.g. throughput analysis, as described in the following chapters. Further details on SDF and HSDF are given in Chapter 3.

1.2.2 Streaming Applications

Streaming applications constitute a huge application space for embedded systems. They are becoming increasingly important and widespread, since they run on many common devices and systems that affect our daily life. A common well-known example of this in daily life is the smartphone, as shown in Figure 1.3(a). It is a multi-purpose (i.e., communication, entertainment, navigator, etc.) embedded system that runs several streaming applications with different purposes that ranges from communication to entertainment. Another example considered as safety-critical is Autonomous driving systems, shown in Figure 1.3(b), that have started to be integrated in many car driving systems (e.g. Google, Tesla, Mercedes, etc.). These systems enable cars to sense their environment, navigate without human input and stay connected to the Internet [Gehrig and Stein, 1999]. Both of these example systems process audio and video streams on which streaming applications perform functions like audio/video encoding and decoding, object recognition, object

(a) Smartphones [Kenya Tech News, 2015].



(b) Autonomous driving systems [Daily Autonomous Car News, 2015] .

Figure 1.3: Examples of embedded systems running streaming applications.

detection and image enhancement on the streams [Elliott et al., 2014, Salunkhe et al., 2014, Siyoum et al., 2011]. These kind of streaming applications have *high processing requirements* and *timing constraints* that must be satisfied, especially in case of safety-critical applications.

The *high processing requirements* raises the need for a parallelization model to enable applications to use massive computational power [Pankratius et al., 2009], which the dataflow model of computation is able to achieve for streaming applications [Lee and Messerschmitt, 1987a]. This is because dataflow model is inherently parallel and can work well in decentralized systems. Furthermore, since these applications are basically a series of transformations that are applied to a data stream, the dataflow model is a natural paradigm for representing them for concurrent implementation on multi-/many-core processors [Lee and Messerschmitt, 1987a].

The streaming applications' *timing constraints* require guarantees that they will be satisfied during applications execution. Recently, several works applied real-time scheduling and analysis techniques on dataflow applications [Bamakhrama and Stefanov, 2011, 2012, Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997, Lipari and Bini, 2011, Liu et al., 2014, Saifullah et al., 2011]. However, they are limited to dataflow applications represented as Directed Acyclic Graphs (DAG) or implicit-deadline task models, which discards a wide range of dataflow applications.

1.3 Processing Platform

This section aims to discuss different processing platform architectures and features of interconnection network. The main goal is to explain the specifications of the processing platform assumed in this thesis.

The processing platform refers to the hardware responsible for running applications in the real-time embedded system. There is a paradigm shift towards multi-/many-cores in the design process of processing platforms. Presently, increasing the number of cores is the current way to improve the performance for high-end processors rather than increasing the clock speed for single processors. One of the reasons why the clock rate gains of the past cannot any more be continued is the unsustainable level of power consumption [Vajda, 2011].

Architecture:

A multi-/many-core platform has more than one core or processor. These cores can be similar or completely different in architecture. Consequently, multi-/many-core platforms can be categorised into two main types based on the relation between the cores on a given platform:

Homogeneous Architecture: in this architecture type all cores in the platform are identical and have exactly the same properties in terms of computation (e.g. instruction set, frequency and cache size) and the cores are interchangeable. The execution time and energy consumption of a task remains the same on all cores on such a platform. These platforms are also sometimes called symmetric multi-processor platforms (SMP). Many platforms manufactured and deployed today in embedded systems fall under this category. For example, Cortex-A17 [Cor] from ARM (used in smart phones, tablets, smart TV's, etc.) has four identical cores on a same die.

Heterogeneous Architecture: this architecture type features at least two different kinds of cores that may differ in both the instruction set architecture, frequency and cache size. The most widespread example of a heterogeneous multi-core architecture is the Cell BE architecture, jointly developed by IBM, Sony and Toshiba [Gschwind et al., 2006] and used in areas such as gaming devices and computers targeting high performance computing.

Interconnection Networks (IN):

Since increasing the number of cores in multi-/many-core platforms is the current trend to increase the performance, there should be an efficient communication network to connect them, called Interconnection Networks (IN). The IN between multiple cores may be a performance bottleneck, since it is responsible for transferring and routing of data between different cores. These data are in the form of packets with headers that contain information about its destination. Data transfer between distant cores can increase latency and consume extra power. In the following paragraphs, we look at traditional IN topologies.

2D-Mesh: shown in Figure 1.4(a), is a common topology that uses routers that are connected to other routers as well as a number of cores. Advantages include design simplicity and short links. Disadvantages include a potentially high number of hops.

Fat Tree: shown in Figure 1.4(b), is a tree topology where the cores are located at leaves of a tree and internal nodes are routers. Data travels upward in the tree until a common ancestor is found between source and destination. The number of links increases towards the root of the tree. Advantages include high bandwidth because of the increased number of links

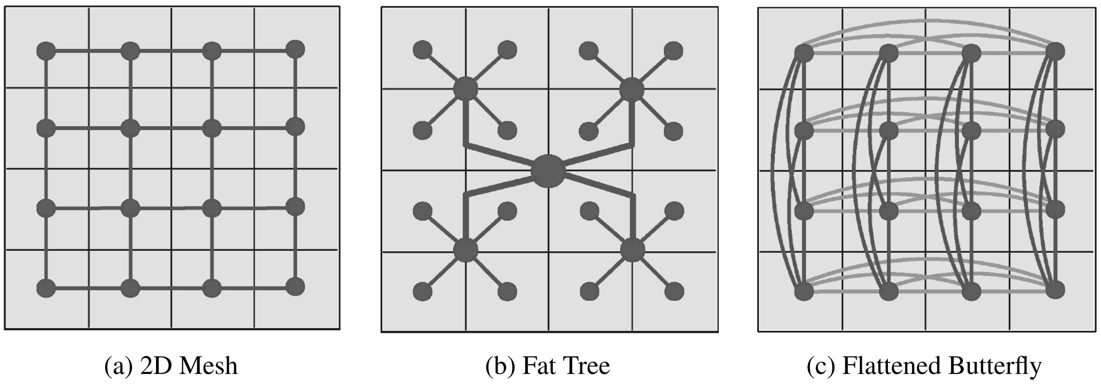


Figure 1.4: Examples of Interconnection Networks (IN) [Sanchez et al., 2010].

as data moves towards the root. Disadvantages include the need for more complex routers, again because of the increased number of connections toward the root.

Flattened Butterfly: shown in Figure 1.4(c), is a modified butterfly network that is essentially a mesh network with additional links. Advantages include a small number of hops. Disadvantages include complex routers and increased chip area due to the large number of links.

Routing:

In all IN topologies, except fully connected topology, not all the router-pairs are directly connected. Therefore, in such cases, depending on the position of the sender and the receiver, packets may need to travel across multiple intermediate links and routers. A set of traversed network elements (routers and links) is called the *route*, while the number of traversed links is usually referred to as the *number of hops*.

The process of transferring packets from source to destination is called *routing*, which is the responsibility of the routers. Once packets reach the router, it decides in which direction they will be forwarded. The logic inside the router that is responsible for making this decision is called the *routing algorithm*. There exist numerous criteria based on which the routing decisions can be made. For example, the *minimal routing* class algorithms [Ni and McKinley, 1993] which aim to minimise the route, and hence derive routing decisions such that the packets always traverses the minimal possible number of hops. Moreover, the *deterministic routing* class algorithms, which always routes packets between the same source and destination on the same path. Alternatively, the *adaptive routing* class algorithms [Bolotin et al., 2004] makes routing decisions at runtime based on the status and load of individual links. Adaptive routing can improve the performance of the system (the average case behaviour) by reducing the average communication time, however, at the expense of predictability. Conversely, deterministic routing is predictable and much easier to implement, but may cause an inefficient utilisation of the NoC resources, where some links may be heavily congested, and others may be completely idle.

The selection of the routing mechanism depends on the purpose of the system. As already mentioned, in the real-time embedded domain the predictability of the system is essential, because it allows to analyse the temporal behaviour of the system with significantly less pessimism. Thus, in the real-time domain, the deterministic routing techniques are a preferable option.

One class of popular minimal deterministic routing algorithms in 2D-mesh IN is the *dimension-ordered routing*. Assuming these schemes, the packets are firstly routed along one dimension of the IN, and after reaching the coordinate of the destination, if needed, continue the transfer along the other dimension. One of the most popular routing algorithms of this class is *X-Y routing*, where the horizontal axis of the platform is usually denoted with the letter X and the vertical axis is denoted with the letter Y. The *X-Y routing* policy is deadlock free [Hu and Marculescu, 2003].

Switching:

Switching defines how packets are transmitted from source to destination. When the IN resources are free, packets traverse routers and links on their route towards the destination. However, in the presence of other traffic, it may happen that one of the links on its route is busy transferring other packets. In such cases, switching mechanisms resolves the situation. One of these mechanisms is the *store-and-forward switching* [Tanenbaum, 2002], where the router stores the full packet before forwarding it to the next router on the route. In this mechanism, one must ensure that the buffer size at each router is sufficient to store the whole packet, otherwise it will be stalled. Another well-known mechanism is *wormhole switching* [Ni and McKinley, 1993], where the router makes the routing decision and forwards the packet as soon as the header arrives. The subsequent payload is split into smaller containers called *flits*. These flits follow the header as they arrive. This reduces the latency within the router, but in case of packet stalling, many links risk to be locked at once.

Arbitration:

The main responsibility of IN is to transfer and route communication data between different cores. During the process of data transfer, significant contention may occur due to accessing the IN shared medium, e.g. links and routers. Several approaches, called *arbitration* mechanisms, have been proposed to manage such contention. These mechanisms are provided by the IN to allow the multiplexing of several streams of data over the same physical medium (link). Common schemes are Space Division Multiplexing (SDM) [Banerjee et al., 2009, Lusala and Legat, 2011, Marchal et al., 2005, Modarressi et al., 2009], Time Division Multiplexing (TDM) [Goossens et al., 2005, Liu et al., 2004, Wang et al., 2008, Zhang et al., 2010] either in the conventional slot allocation approach or in an arbitrated (e.g. round-robin, priority) link time sharing scheme. TDM is a commonly used arbiter for management of communication resources in multi-core platforms. The reasons for its popularity is that it is conceptually easy to understand and analyze and has efficient implementations both in hardware and software [Akesson et al., 2015]. Moreover, it provides temporal isolation between clients when used in a non-work-conserving manner [Goossens et al., 2013a]. Several platforms relying extensively on TDM for a variety of resources management are PRET [Edwards and Lee, 2007] and CompSOC [Akesson et al., 2015].

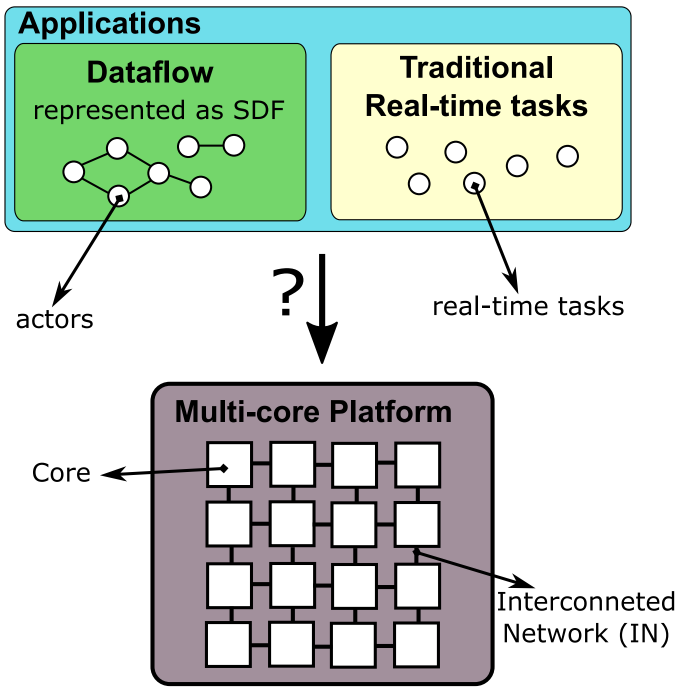


Figure 1.5: Problem to be addressed

In this thesis, we are concerned with *homogeneous architecture* processing multi-core platforms that incorporates a *2D Mesh* IN operated using *X-Y routing*, *wormhole switching* and using *TDM* as arbitration mechanism.

1.4 Problem Statement

In this thesis, we address the problem of real-time embedded systems incorporating mixed application models with timing constraints running on the same multi-core platform. These mixed application models are dataflow applications with timing constraints (latency and throughput) and traditional real-time applications, as shown in Figure 1.5. The design of such systems require guarantees that all running applications mapped on the platform will execute safely satisfying their timing constraints.

As shown in Figure 1.5, the traditional real-time applications are modelled as independent tasks. Each task is characterised with specific parameters, e.g. WCET, deadline and period. In contrast, dataflow applications are basically graphs of communicating tasks, which are actors. These actors are defined by a different set of parameters, e.g. WCET, Production/Consumption rate (P/C) of tokens. A dataflow application has timing constraints, i.e. latency and throughput requirements (Section 1.2.1), that must be satisfied. This leads to the main question of the thesis: *How can future real-time embedded systems safely incorporate mixed application models, dataflow and traditional real-time tasks, with timing constraints onto multi-core platforms, such that their timing constraints are satisfied?*

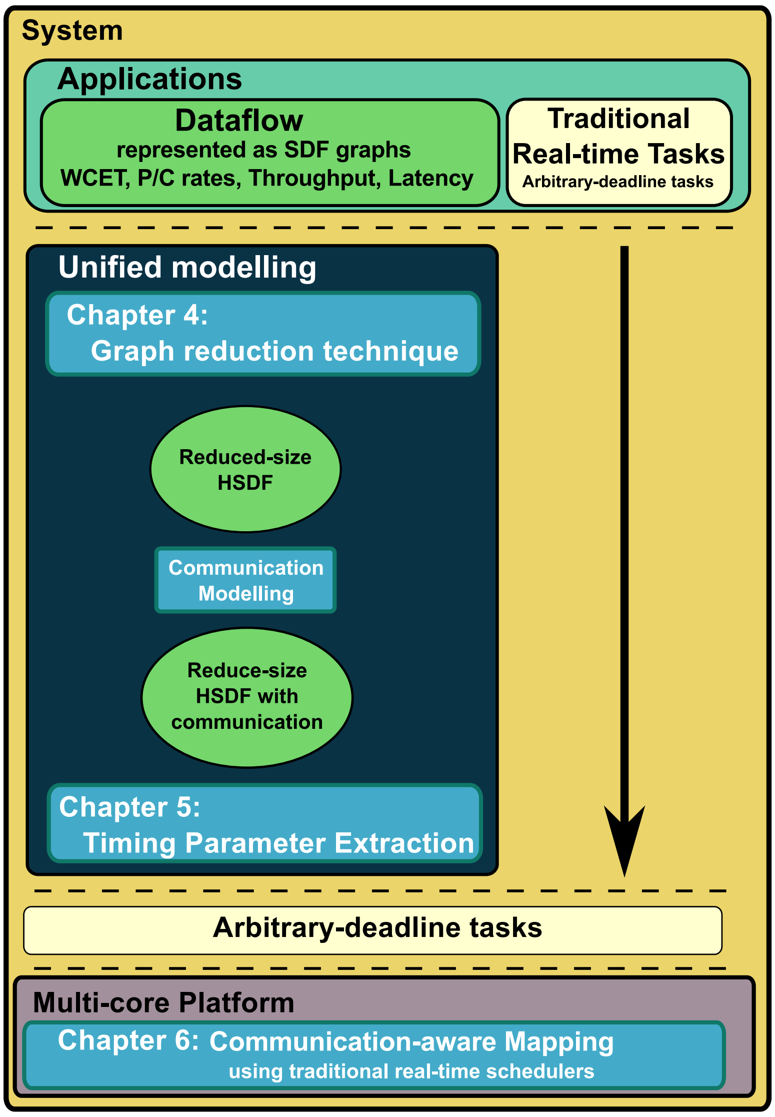

System
Applications
Dataflow
represented as SDF graphs
WCET, P/C rates, Throughput, Latency
Traditional
Real-time Tasks
Arbitrary-deadline tasks
Unified modelling
Chapter 4:
Graph reduction technique
Reduced-size HSDF
Communication Modelling
Reduce-size HSDF with communication
Chapter 5:
Timing Parameter Extraction
Arbitrary-deadline tasks
Multi-core Platform
Chapter 6: Communication-aware Mapping
using traditional real-time schedulers

Figure 1.6: Solution outline.

1.5 Solution Overview

In this section, we present an outline of our proposed solution to the stated problem outlined in Section 1.4. The main goal of this solution is to provide guarantees for the mixed application model executing on the multi-core platform, such that timing constraints are satisfied.

To implement this kind of systems, we have to address how to map and schedule such mixed application model on the multi-core platform. Different solutions in mapping and scheduling have been proposed for each application model independently. The mapping problem has previously been tackled in several works from a high-performance point-of-view [Ennals et al., 2005, Evans and Kessler, 1992, Liu et al., 2007, Lo, 1988, Ma et al., 1982], where all applications are represented either as graphs or independent tasks. However, using these approaches in the mapping of real-time applications does not guarantee satisfying their timing constraints. Another mapping approach uses the First Fit (FF) bin-packing heuristic, since it has been shown to outperform other bin-packing heuristics in terms of achieved throughput [Guo and Bhuyan, 2006]. However, applying approaches that satisfy timing constraints and use FF, such as [Bamakhrama and Stefanov, 2011], results in over-dimensioned systems, as our experimental evaluation shows in [Ali et al., 2013] and Chapter 6. Moreover, such work [Guo and Bhuyan, 2006] does not consider the communication cost and its effect on the schedulability of the system.

The scheduling problem has been studied extensively for traditional real-time applications through introducing several real-time scheduling algorithms either onto uniprocessors, e.g. Fixed Priority (FP) [Liu and Layland, 1973], Earliest Deadline First (EDF) [Liu and Layland, 1973], or multi-processor Partitioned EDF (PEDF) [López et al., 2004] and Hierarchical scheduling [Calandrino et al., 2007, Easwaran et al., 2009, Leontyev and Anderson, 2008, Zhu et al., 2011]. However, dataflow applications mostly use static scheduling, i.e. TDMA. Static scheduling works well in case of systems that only run dataflow applications. In contrast, in case of systems that run mixed real-time applications, a dynamic real-time scheduling algorithm may have a higher schedulability success rate than static scheduling, but it is not currently available for mixed systems. Furthermore, real-time scheduling algorithms can enable efficient real-time analysis techniques for such mixed systems. Recently, several works scheduled dataflow applications using real-time scheduling algorithms [Bamakhrama and Stefanov, 2011, 2012, Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997, Lipari and Bini, 2011, Liu et al., 2014, Saifullah et al., 2011]. However, they are either limited to dataflow applications represented as Directed Acyclic Graphs (DAG), or they are represented as implicit-deadline tasks.

The proposed system runs two types of application models, traditional real-time and dataflow applications. The traditional real-time applications are a set of independent periodic arbitrary-deadline real-time tasks. These tasks are characterised by timing parameters that define their temporal behaviour in execution, e.g. WCET, period and relative deadline. Independent real-time tasks have a set of well-established real-time scheduling and analysis techniques in the literature that allow satisfying their timing constraints. The main idea is to use these techniques and methods and apply them on dataflow applications to get the same guarantees. However, these techniques cannot be applied directly on dataflow applications, because they miss the appropriate task model parameters to allow using them. Therefore, a *unified model* for both types of application models is needed to apply traditional real-time scheduling and analysis techniques on the system, thereby guaranteeing that timing constraints are satisfied.

The *unified modelling* is a process that transforms the dataflow applications into traditional real-time tasks. This transformation is done using the *timing parameter extraction* algorithm shown in Figure 1.6 and detailed in Chapter 5. However, before sending the dataflow graph to the timing parameter extraction algorithm, it has to go through two processes. First, is the *graph reduction* process, discussed in Chapter 4. It generates a *reduced-size HSDF graph* from the original HSDF graph. This is because transformation to HSDF graphs can result in an exponential explosion in the graph size, which slows down the timing parameter extraction algorithm when applied on them. Therefore, the graph reduction process speeds up the overall design process, as the experiments show in Chapter 5. Second, is the *communication modelling* process, where it models the communication in the reduced-size HSDF graph, generating an extended HSDF graph that accounts for the communication cost. The extended communication-aware graph is then used as input to the mapping algorithm, as explained in Chapter 6. Following these two steps, the timing parameter extraction algorithm takes the HSDF graph with modelled communication as an input, transforming it into a set of independent *arbitrary-deadline tasks*.

Now, we reached the stage where we have a unified set of arbitrary-deadline real-time tasks. This enables applying traditional real-time scheduling and analysis techniques while mapping them on the platform. The mapping algorithm, shown in Figure 1.6, allocates the task set on the platform guaranteeing that all applications satisfy their timing constraints. Also, the proposed mapping algorithm is *communication-aware*, which means that it considers the communication overhead resulting from the token exchange between different actors in the dataflow applications. The *communication-aware mapping* algorithm, detailed in Chapter 6, is able to do that because of the communication modelling of the HSDF graph that happened in the early stages in the solution.

1.6 Thesis Contributions

As highlighted in the problem statement (Section 1.4), the main goal of this thesis is to allow future real-time embedded systems to map and schedule mixed application models with timing constraints on the same multi-core platform guaranteeing that timing constraints are satisfied. To achieve this goal we proposed the solution outline, discussed in Section 1.5 and shown in Figure 1.6, that consists of three main contributions. They are:

1. An offline **dataflow graph reduction algorithm**, called *slack-based merging*, that aims to speed-up the process of timing parameter extraction and finding a feasible real-time schedule, thereby reducing the overall design time of the real-time system. To achieve this goal, the algorithm combines two main concepts:

   (a) The *slack*, which is the difference between the WCET of the SDF graph's firings and its timing constraints.

   (b) The *safe merge*, which is a novel merging concept that we prove cannot cause a live HSDF graph to deadlock.

   The output is a reduced-size HSDF graph that satisfies the throughput and latency constraints of the original application graph.

2. A **timing parameter extraction algorithm** that extracts timing parameters of HSDF graphs with timing constraints, converting them into periodic arbitrary-deadline tasks. This algorithm provides a method to unify mixed application models into a single real-time task set. A main advantage of our proposal is that the extraction of the timing parameters is independent of the specific scheduler being used, of other applications running in the system and the details of the particular platform. The proposed algorithm:

(a) Enables applying traditional real-time schedulers and analysis techniques on cyclic or acyclic HSDF applications with periodic sources.

(b) Captures overlapping iterations, which is a main characteristic of the execution of dataflow applications, by modelling actors as tasks with arbitrary-deadlines.

3. A **mapping algorithm**, called *communication-aware mapping*, dedicated for allocating HSDF graphs on 2D-Mesh multi-core platforms. The algorithm is based on a novel mapping heuristic called *Sensitive-Path-First*. This heuristic allocates first, for each HSDF, the most critical paths (a path consists of a set of tasks) in terms of schedulability, maximizing path parallelism when possible. The mapping process is done taking into account satisfying applications time constraints and maximizing resource utilization of the platform, while accounting for the communication cost.

Together, these three important contributions successfully achieve the main goal of this thesis and play a part in allowing embedded real-time systems to map and schedule mixed application models.

1.7 Thesis Organization

This thesis addresses the problem of mapping and scheduling mixed application models with timing constraints running on the same multi-core platform in real-time embedded systems. The thesis is organized as follows:

- Chapter 2 discusses the state of the art in three main topics that represent the three main contributions of this thesis. These three main topics are dataflow graph analysis, timing parameter extraction techniques and mapping methodologies.

- Chapter 3 provides a background on topics and terminology essential for understanding the research problem and the system model.

- Chapter 4 introduces the proposed graph reduction technique for dataflow applications called slack-based merging. It provides a detailed explanation of the algorithm assisted with proofs, examples and experiments that show its validity and functionality.

- Chapter 5 presents the timing parameter extraction algorithm that transforms dataflow applications into independent real-time tasks. The chapter starts by discussing similar mechanisms for timing parameter extraction for pipelines. Then, it shows how these mechanisms are incorporated in the proposed algorithm to extended its functionality to cover dataflow graphs. We present proofs, examples and experiments that shows the validity and functionality of our proposed algorithm. Moreover, the experiments show the speed-up effect of the graph reduction technique on the timing parameter extraction process.

- Chapter 6 describes the proposed mapping algorithm called communication-aware mapping. It begins by presenting the mechanism for communication modelling in dataflow graphs.

Then, it lists and describes the components of the communication-aware mapping algorithm. Especially, its main mapping heuristic called Sensitive-Path-First, which is inspired from the Critical-Path-First (CPF) mapping heuristic proposed in [Ali et al., 2013]. In addition, the chapter provides a full view of our proposed solution by integrating the three algorithms together. This allows experimenting both communication-aware mapping algorithm and the whole system.

- Chapter 7 finishes the thesis with conclusions and future directions of research.

Chapter 2

State of the Art

This chapter gives an overview on the state of the art related to this thesis. It positions our work with respect to the state of the art in three aspects that comprise our proposed solution (previously shown in Figure 1.6). These three aspects are: **1)** graph reduction techniques explained in the context of dataflow analysis (Section 2.1), **2)** extraction of timing parameters that transforms actors of dataflow graphs into traditional real-time tasks that enable applying traditional real-time scheduling and analysis techniques (Section 2.2) and **3)** mapping of dataflow graphs onto multi-/many-core platforms (Section 2.3).

2.1 Dataflow Graph Analysis

The dataflow model of computation is popular for modelling the timing behaviour of real-time embedded hardware and software systems and applications. It is an essential ingredient of several automated design-flows and design-space exploration tools. In this section, we will present the state of the art in dataflow graph analysis techniques concerning certain properties essential for our work, throughput, latency and graph size.

Various analysis techniques have been proposed to determine throughput and latency properties of this computational model. For throughput analysis, there are several methods and tools, e.g. [Damavandpeyma et al., 2012, Ghamarian et al., 2008, Stuijk et al., 2006]. In [Ghamarian et al., 2008], the authors propose three methods to compute throughput of an SDF graph where actor execution times can be parameters. The throughput of these graphs is obtained in the form of a function of these parameters, which can be evaluated for specific parameter values. The three proposed methods are based on different algorithms. The first two algorithms, called HSDF graph and State-Space methods, are variants of the standard throughput analysis algorithms for SDF graphs for parametric actor execution times. The third algorithm, called Divide-and-Conquer Method, is based on a divide-and-conquer strategy. Experimental results show that the divide-and-conquer algorithm performs best. In [Damavandpeyma et al., 2012], the authors propose a new method to determine a tighter throughput bound for applications modelled as Scenario-Aware Dataflow (SADF) Graphs [Theelen et al., 2006]. This method is based on Max-Plus automata that finds throughput expressions for a parametrized SADF graph. The approach extracts a Max-Plus Automaton Graph (MPAG) from an SADF graph and then uses a maximum cycle mean algorithm to determine the critical timing cycle of the extracted MPAG. The timing behaviour of an application depends on several dynamic aspects, e.g. its scheduling, Dynamic Voltage and Frequency Scaling (DVFS), etc. The new technique is able to capture this dynamic timing behaviour by generating throughput expressions for dynamic applications. Experimental results show that the proposed technique outperforms others in terms of run-time, e.g. [Ghamarian et al., 2008].

Throughput analysis can also be obtained through the HSDF graph method by getting the inverse of the Maximum Cycle Mean (MCM) of the equivalent HSDF graph [Karp and Miller, 1966, Sriram and Bhattacharyya, 2000]. The cycle mean of a cycle of an HSDF graph is defined as the total execution time of the cycle over the number of initial tokens in that cycle. There are efficient algorithms for calculating the MCM of an HSDF graph [Dasdan and Gupta, 1998]. However, the HSDF conversion process may lead to an exponential growth in the size of the HSDF graph, which leads to longer throughput analysis time. Another method for throughput analysis called state-space analysis based on the periodic phase of execution of self-timed execution of an SDF, CSDF, etc. graph, where a sequence of actor firings occur in a periodic pattern. The throughput of an actor can be calculated by dividing the length of the period by the number of firings of the actor in one period. An experimental comparison in [Ghamarian et al., 2008] showed that the state-space method outperforms the HSDF graph method in terms of analysis time.

Similar to throughput analysis methods, there are several works on latency analysis [Bamakhrama and Stefanov, 2012, Ghamarian et al., 2007]. In [Ghamarian et al., 2007], the authors propose an algorithm to determine the minimal achievable latency between the execution of any two actors in an SDF graph. Also, they present a heuristic that defines a class of static order schedules that provide minimal latency, while satisfying the throughput constraint. Experimental results show that latency computations are efficient despite the theoretical complexity of the problem. In [Bamakhrama and Stefanov, 2012], the authors proposed an algorithm that transforms acyclic CSDF graphs into constrained-deadline periodic tasks to achieve both minimum application latency and maximum throughput.

Away from methods for throughput and latency analysis, [Stuijk et al., 2006] presents a tool, inspired by Task Graphs For Free (TGFF) [Dick et al., 1998], called SDF For Free (SDF^3). SDF^3 is a tool that implements an SDF graph generation algorithm that constructs graphs that are connected, consistent, and deadlock-free, with support for analysing and visualising these graphs and calculating their throughput. Also, it can take dataflow applications as an input in the form of eXtensible Markup Language (XML) files and perform analysis and conversion to HSDF graphs.

The dataflow computational model can be used to analyse and derive different parameters that define a dataflow application. Examples of these parameters are throughput and latency. Moreover, it can be used to derive real-time parameters, e.g. offsets, deadlines and periods, as presented in [Ali et al., 2015, Bamakhrama and Stefanov, 2011, Bekooij et al., 2005, Hausmans et al., 2013, Liu et al., 2014, Saifullah et al., 2011]. These works are the main concern of this thesis and detailed in Section 2.2. Some of these analysis algorithms operate directly on SDF graphs, while many others require transformation to Homogeneous Synchronous Dataflow (HSDF) graphs prior to the analysis, using conversion algorithms proposed for such kind of transformations, i.e. [Lee and Messerschmitt, 1987a, Sriram and Bhattacharyya, 2000]. This transformation can lead to an exponential increase in the size of the original SDF graph, which significantly increases the run-time of the analysis algorithm.

To avoid the increase in graph size problem, dataflow graph reduction techniques are needed to decrease the size of HSDF graphs, and hence speed-up the analysis run-time. The following is a quick review to the state of the art related to reduction techniques for dataflow graphs.

In [Geilen, 2009], the authors propose a SDF graph reduction technique based on Max-Plus algebra. It transforms an SDF graph into a smaller HSDF graph with equivalent maximal throughput and latency, which is faster to analyse. Each actor in the smaller HSDF graph may comprise of single or multiple firings of different SDF actors. Due to this reason, the output HSDF graph of this technique hides the actual execution behaviour of the original SDF graph, because a single firing of an SDF actor can exist in multiple actors of the output HSDF graph. This means that a single firing in the SDF graph is executed multiple times in the output HSDF graph, which complicates extracting timing parameters and finding a feasible schedule. In contrast, we propose a reduction algorithm that generates a reduced-size HSDF graph called *slack-based merging*, as detailed in Chapter 4. This reduction algorithm speeds up, at relaxed throughput and latency constraints, the processes of extracting timing parameters and finding a feasible mapping and schedule for the application, as the experimental results show in Chapters 5 and 6, respectively. This is due to the generated reduced-size graph have a small number of tasks compared to the original HSDF graph. Also, the generated graph represents the actual execution behaviour of the original graph, avoiding the problem with the approach in [Geilen, 2009]. It also ensures that the throughput and latency constraints are met, although with a possibility of having a lower maximum throughput compared to the original graph. However, this is not a problem, because the main goal for real-time systems is satisfying timing constraints.

2.2 Timing Parameter Extraction

There is a trend towards embedded systems allowing mixed application models with timing constraints (dataflow and traditional real-time tasks) to run on the same multi-core platform. Therefore, a unified model is needed to represent dataflow with timing constraints and traditional real-time applications. This section reviews techniques for extracting timing parameters (*unified model*) of task graphs to enable applying real-time schedulers and analysis techniques.

In [Bamakhrama and Stefanov, 2011, 2012, Liu et al., 2014], the authors provide an analytical framework for computing timing parameters for actors of acyclic Cyclo-Static Dataflow (CSDF) applications with a single input. The actors are considered as implicit-deadline and constrained-deadline periodic tasks in [Bamakhrama and Stefanov, 2011] and [Bamakhrama and Stefanov, 2012, Liu et al., 2014], respectively. In contrast, this work is more general and can deal with any HSDF graph (CSDF can be converted to an HSDF), single/multiple input, and actors are modelled as arbitrary-deadline tasks. Modelling the application actors as arbitrary-deadline tasks allows capturing overlapping iterations, a main characteristic of dataflow applications that increases the throughput.

Another solution is presented in [Lipari and Bini, 2011]. The authors presented a deadline assignment approach called ORDER for dependent tasks composing real-time pipeline applications executing on a multi-core system. The proposed approach was considering the problem of scheduling a pipeline such that the end-to-end deadline is met and the amount of required resource capacity was minimal. Contrarily, this work considers the general problem of deadline assignment for dependent tasks comprising real-time application graphs, such as DAG and Directed Cyclic Graphs (DCG), which are not supported by [Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997, Lipari and Bini, 2011].

In [Saifullah et al., 2011], the authors also address the problem of scheduling periodic DAG tasks, each consisting of subtasks. They extract their timing parameters, i.e., individual deadlines, and scheduled using global Earliest Deadline First (EDF) and partitioned deadline monotonic scheduling. Another approach presented in [Qamhieh et al., 2013] extracts timing parameters for subtasks in a DAG task based on computing the interference between each subtask and the higher-priority subtasks of all DAG tasks running on the system. In contrast, we consider a more general problem where applications are represented as DCG and the extraction of the timing parameters is independent of the scheduling algorithm being used.

Another technique is presented in [Spuri and Stankovic, 1994]. The authors propose an exact characterization of EDF-like schedulers that can be used to correctly schedule dependent tasks, and show how preemptive algorithms, even those that deal with shared resources, can be easily extended to deal with dependencies. This was done by modifying deadlines in a consistent manner so that a run-time algorithm, such as EDF, could be used without violating the dependencies. Also, [Chetto et al., 1990] propose a similar approach by modifying the timing parameters of the tasks. This parameter modification is not only for the deadline of the tasks, but also include modification of the task start time. However, both works consider task parameters as already defined, which is not the case in our problem. Moreover, they are only concerned with uniprocessor platforms.

Also in [Moreira et al., 2007], the authors present a method to calculate individual deadlines of HSDF actors. The method is based on an Integer Linear Programming (ILP) optimization problem that finds the amount of slack for each actor that makes it able to extend its execution without violating the HSDF throughput and timing constraints. However, their proposed method is restricted to strongly connected HSDF graphs and the actor's offsets (release times) are calculated based on the static-order schedule of the application. In contrast, this work is neither restricted to strongly connected graphs nor does the offset calculation require static-order scheduling.

In [Hausmans et al., 2013], the authors propose a temporal analysis for dataflow applications modelled as cyclic HSDF graphs under a non-starvation-free scheduler i.e. Static-Priority Preemptive scheduler (SPP). To apply the analysis they extract timing properties like jitter (difference between best-case and worst-case offsets), periods, and execution times, but not deadlines, since SPP schedulers depend on periods not deadlines. The calculated jitter is based on the interference from the set of high-priority tasks with the task being analysed running on the same platform. This means that the timing parameters calculated are dependent on the set of applications running on the platform. Contrarily, this work is independent of the scheduler being used and other applications running on the same platform, since our proposed algorithm transforms the HSDF actors into a set of independent tasks that enables any bin-packing heuristic to be applied for mapping them on the platform.

In [Bouakaz et al., 2012], the authors present a new dataflow computational model that is a superset of SDF/CSDF application graphs called Affine Dataflow (ADF). The ADF is a time-triggered dataflow model that explicitly represents each firing of each actor in a complete iteration of the graph as a so-called clock tick. These clock ticks are related to each other using firing relations called affine relations. These relations maintain precedence constraints between different firings of actors in the graph, since it ensures the correct execution order of different clock ticks. Based on this framework, they present an algorithm that computes affine schedules for these clock ticks, which enables applying real-time scheduling algorithms, e.g. earliest-deadline first or rate-monotonic. However, the use of clock tick representation and affine relations to represent the firing behaviour of actors does not speed up the process of finding a feasible schedule, because it indirectly transforms the ADF to an HSDF graph (using the clock tick representation) to be able to find a feasible schedule. In addition, the presented algorithm does not support end-to-end latency constraints, since it assumes an implicit-deadline task model. In contrast, this thesis work supports end-to-end latency constraints and uses the arbitrary-deadline task model, which adds more generality to the work.

2.3 Efficient Mapping

The problem of task mapping in dataflow applications and task graphs has been the subject of quite some previous research.

In [Ramamritham, 1995], the author discusses a static algorithm for allocating and scheduling components of periodic tasks (SDF graphs) that consist of subtasks (actors) with precedence constrains across sites in distributed systems and multi-processor systems. This algorithm consists of two parts; the first part decides whether a group of communicating subtasks of a task should be assigned to the same site as a cluster, while the second part allocates the clusters of subtasks to the sites in a system (or cores of multiprocessor) based on the ability to find a feasible schedule for the subtasks as well as the communication between them. Compared to the work we propose in this thesis, the approach in [Ramamritham, 1995] tries to find a feasible static schedule for tasks inside the cluster. In contrast, we are aiming to use existing real-time scheduling methods, i.e. EDF, inside the clusters. Furthermore, this thesis takes into account the communication cost while satisfying the timing constraints.

In [Peng and Shin, 1989], the authors propose a similar approach to [Ramamritham, 1995]. They propose an optimal solution for the allocation of periodic tasks onto a heterogeneous distributed real-time system using a Branch and Bound (BB) algorithm. The periodic tasks are modelled as a graph, which describes computation and communication modules as well as the precedence constraints among them. However, they do not allow subtasks (nodes) of a task (graph) to execute on different sites (cores) and they use BB search for finding a feasible schedule, while in [Ramamritham, 1995] it is a heuristics-directed search.

The work presented in [Liu et al., 2007] proposes a task-allocation model for multi-core processors. Applications are represented as Task Interaction Graphs (TIG), where an iteration-based heuristic tries to allocate the graph's nodes based on a set of rules that includes: reducing communication overhead, reducing context switching and maintaining load balancing among cores. Evaluation results show that the algorithm can find near-optimal solutions in reasonable time compared to genetic algorithms when the number of threads increases (it can find solutions in much less time than Genetic Algorithm (GA) and Ant Colony Optimization (ACO) [Li et al., 2003]). Also [Ennals et al., 2005, Evans and Kessler, 1992, Lo, 1988, Ma et al., 1982] address the problem of tasks allocation to multi-processors, taking into account the sizes of the tasks, the communication between them and load balancing. However, these works does not take into account the timing constraints required by real-time applications, which is the main focus of the work presented in this thesis.

Stuijk et. al [Stuijk et al., 2007] presented a resource allocation strategy that can allocate multiple SDF graphs onto a heterogeneous multi-core platform with throughput guarantees to each individual application. The proposed method can deal with multi-rate graphs and cyclic dependencies without conversion to HSDF graphs. The allocation strategy consists of three main steps: 1) an actor binding, where every actor from the SDF graph is assigned to a core on the multi-core platform to achieve the application throughput constraint. This is done by considering first the actors whose execution times have large impact on the application throughput. Then, 2) a static order schedule for each core containing actors of the SDF graph is done. Finally, 3) time slices are allocated for cores based on a binary search algorithm which guarantees satisfying the throughput constraint. The experiments show that this enables a balanced resource allocation of time-constrained applications bound to a heterogeneous multi-core platform. Despite this allocation strategy being similar to the mapping technique in this work, explained in Chapter 6, in the sense of giving priority to allocation of actors whose execution have a large impact on application throughput (in our solution priority is given to actors in the sensitive path of the application), the use of static scheduling may not be able to satisfy the timing constraints of traditional real-time applications in the addressed research problem. Furthermore, the proposed mapping algorithm is restricted to SDF graphs only and cannot be applied on other types of dataflow graphs, e.g. CSDF.

In [Bamakhrama and Stefanov, 2011], the authors provide an approach where actors (nodes) of streaming applications are considered as implicit-deadline periodic tasks. They provide results of tests on real streaming applications from the SDF^3 Benchmark [Stuijk et al., 2006], and also use PEDF as the scheduling algorithm for periodic tasks. They use the FF algorithm for the allocation of nodes on the cores, and show that in more than 80% of the cases the throughput resulting from the approach is equal to the maximum achievable throughput.

In summary, the mapping problem has been tackled in several works either from a high-performance point-of-view ignoring timing constraints [Ennals et al., 2005, Evans and Kessler, 1992, Liu et al., 2007, Lo, 1988, Ma et al., 1982] or, applying FF taking into account timing constraints [Bamakhrama and Stefanov, 2011]. Applying the first approaches on the allocation of real-time applications will not guarantee satisfying its timing constraints, while applying the second approach will likely result in over-dimensioned systems.

Chapter 3

Background

In this chapter, we present relevant background information and mathematical formulation that is essential for understanding the computational model, the system model and the proposed solution. The presented background consists of three main sections: **1)** real-time systems, **2)** the dataflow computational model and **3)** multi-/many-core platforms. For real-time systems, detailed in Section 3.1, we discuss basic concepts and definitions, followed by multi-core scheduling algorithms, and feasibility tests. For the dataflow computational model, detailed in Section 3.2, we formalize the Synchronous Dataflow (SDF) and the Homogeneous Synchronous Dataflow (HSDF) models. In multi-/many-core platforms, detailed in Section 3.3, we give a quick overview on multi-/many-core platform architectures. After this detailed background, we present our system model in Section 3.4.

3.1 Real-time Systems

A real-time system is one in which the correctness of the computations not only depends on their logical correctness, but also on the time at which the result is produced. In other words, a late answer is a wrong answer. As we mentioned before, a real-time system runs several real-time processes called tasks. A real-time task τ_i generates periodic instances, called Jobs J_i. A real-time task τ_i is defined by several parameters. A job J_i inherits the same parameters of the task τ_i that generates it. These parameters are the period of execution T_i, the WCET C_i, the arrival time (offset) a_i, the start time S_i, the finishing time F_i and the deadline D_i, as illustrated in Figure 3.1. The period T_i determines the rate of execution of a task τ_i, which specifies the frequency of jobs J_i generation. The WCET C_i is the time necessary for the processor to execute a job J_i of a task τ_i without interruption. The arrival time (offset) a_i is the time at which a job J_i of a task τ_i becomes ready for execution, relative to its period T_i. The start time S_i is the time at which a job J_i of a task τ_i starts its execution. The finishing time F_i is the time at which a job J_i of a task τ_i finishes its execution. The deadline D_i is the time before which a job J_i of a task τ_i should be completed, relative to a_i, to avoid damage to the system or degradation in its performance according to its real-time system category classification. In this thesis, we refer to D_i as the relative deadline. The $\overline{D}_i$ is

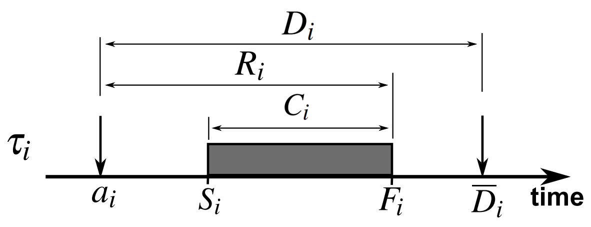


Figure 3.1: Real-time task parameters.

called the absolute deadline, which represents the absolute value of a deadline of a job J_i. In this research, our system model defines the real-time task τ_i by (a_i, C_i, T_i, D_i) parameters, neglecting S_i and F_i since they are not significant to our model.

A given set of jobs J_i must be ordered for the jobs to be executed such that the deadline constraints are satisfied. The execution of a job J_i may or may not be interrupted (preemptive or non-preemptive scheduling) by other jobs. Over the set of jobs, there is a precedence relation, in case of dependent tasks, which constrains the order of execution. The platform on which the jobs are to be executed is characterized by the amounts of resources available [Buttazzo, 2004, Joseph, 1996, Krishna, 1996, Stankovic and Ramamritham, 1989]. A real-time scheduling algorithm must achieve a main goal which is meeting the timing constraints of the system [Joseph, 1996, Krishna, 1996]. There are also other goals that a real-time scheduling algorithm should achieve, however, they are not a primary driver for the algorithm. Example of these side goals are:

1. Attaining a high degree of utilization.

2. Preventing simultaneous access to shared resources and devices.

3. Reducing the cost of context switches caused by preemption.

4. Reducing the communication cost in real-time distributed and multi-/many-core systems.

Basically, the scheduling problem is to determine a schedule for the execution of the jobs so that they are all completed before their deadline [Buttazzo, 2004, Joseph, 1996, Krishna, 1996, Stankovic and Ramamritham, 1989]. Given a set of real-time tasks, the appropriate scheduling approach should be designed based on the properties and category of the tasks, previously discussed in Section 1.1.1. In this work, we are considering *hard real-time task sets*.

The *response time* R_i of the job J_i is the difference between the time the job finishes executing that invocation F_i and the time it arrived a_i, which is the time it takes the job to complete its execution, as shown in Figure 3.1. A *critical instant* of a task, under a given scheduling algorithm, is a release that yields the longest possible response time of that task for the given task set. A schedule is said to be valid iff all deadlines of all tasks are met. The processor is said to be fully utilized, under a given scheduling algorithm and task set, if the algorithm produces a valid schedule for the given task set, but an increase in the execution time of any task in the task set would yield an overflow. A scheduling algorithm is considered *optimal* if it produces a valid schedule for every task set that is schedulable.

Scheduling can be classified according to the type of the platform that tasks runs on, which are uniprocessor or multi-core. Uniprocessor scheduling may be considered as priority driven in the sense that the task with the highest priority that has execution remaining should be scheduled. In that regard, there are two main types of priority-based scheduling algorithms:

1. **Fixed priorities**, where static priorities are assigned to tasks. These priorities are inherited by the instances of the tasks (jobs). The priority of a job remains static throughout the execution time. There are various fixed-priority assignment algorithms, e.g. Rate Monotonic (RM) [Lehoczky et al., 1989, Liu and Layland, 1973], and Deadline Monotonic (DM) [Leung and Whitehead, 1982]. Usually, the priority is assigned based on certain properties of a task. In case of the DM priority assignment algorithm, the task with the shortest deadline is assigned the highest priority. Similarly, in the RM priority assignment algorithm, the task with smallest period is assigned the highest priority.

2. **Dynamic priorities**, where priorities are calculated and assigned to tasks during the runtime of the system. A task can carry more than one priority during its execution, because priorities are assigned to jobs rather than their tasks. It means that different jobs of the same task may execute on a processor with different priorities. There are many scheduling algorithms that falls in this category , e.g. Earliest Deadline First (EDF) [Baruah et al., 1990, 1993, Leung and Merrill, 1980, Liu and Layland, 1973], and Modified Least Laxity First (MLLF) [Oh and Yang, 1998]. The priority of a job in this class of algorithms is usually assigned based on the fixed property of a job. For example, in case of EDF, the absolute deadline of a job is the fixed property that does not change throughout its active time.

3.1.1 Multi-core Scheduling

Multi-core scheduling can be classified into two categories: partitioned and global scheduling [Davis and Burns, 2011]. Partitioned scheduling statically assigns each task to a single processor, where uniprocessor scheduling algorithms can be applied afterwards to schedule tasks, e.g. Partitioned Earliest Deadline First (PEDF) [López et al., 2004]. In contrast, global scheduling allows tasks to migrate across cores of a multi-core platform and algorithms that simultaneously schedule on all the processors are used, e.g. Global Earliest Deadline First (GEDF) [Baruah and Baker, 2008a,b]. Many partitioning algorithms and their analysis [Baruah and Fisher, 2006, Fisher et al., 2006, Oh and Baker, 1998], and global scheduling algorithms and their analysis [Andersson et al., 2001, Baruah et al., 1996, Davis and Burns, 2011], have been proposed. In this thesis, we use a partitioned scheduling technique called *Partitioned Earliest Deadline First (PEDF)*.

3.1.2 Feasibility Tests

Real-time scheduling is the theoretical basis of real-time systems engineering. Feasibility tests can be sufficient or exact (necessary and sufficient). Sufficient tests are usually efficient but they are not powerful; many schedulable task sets are not judged to be schedulable. The simplest sufficient tests for real-time systems are utilization-based and they have polynomial complexity. However, they are not suitable for all types of task sets. In the following sections, we give an overview of two feasibility tests for EDF scheduling algorithm that are used in this work. They are the *demand bound function* and *Quick convergence Processor-demand Analysis*.

3.1.2.1 The Demand Bound Function

The demand bound function (dbf) [Baruah et al., 1990] represents the computational requirement for the system resources of a set of tasks τ. It is mainly used as a feasibility test to check the schedulability of τ within a certain interval by checking its demand against the available computational resources. If the demand exceeds the available computational resources, τ is not schedulable in this specific interval and vice versa. The dbf is the summation of computation time of all the instances of a set of tasks having their release and deadline within a certain interval $[t_0, t_1]$. The dbf calculation differs according to the scheduling algorithm and the task model used. In case of the asynchronous ($a_i \geq 0$) constrained-deadline task model ($D_i \leq T_i$) under an EDF scheduler, the dbf is defined as follows [Baruah et al., 1990]:

$$\text{dbf}(t_0, t_1) = \sum_{\forall \tau_i \in \tau} \max \left\{ 0, \left(\left\lfloor \frac{t_1 - D_i}{T_i} \right\rfloor - \left\lceil \frac{t_0 - a_i}{T_i} \right\rceil \right) \right\} \cdot C_i \tag{3.1}$$

However, to check that τ is schedulable at any point in time, an exact, necessary and sufficient feasibility test is to calculate the demand of τ over the hyperperiod interval H of all tasks' periods, because it forms the cycle over which the system repeats its behaviour. The hyperperiod interval H is denoted by Leung and Merrill in [Leung and Merrill, 1980] as :

$$H = [0, 2 \cdot \text{lcm}_{\forall \tau_i \in \tau}\{T_i\} + \max_{\forall \tau_i \in \tau}\{a_i\}] \tag{3.2}$$

where, $t_0 = 0$ and $t_1 = 2 \cdot \text{lcm}_{\forall \tau_i \in \tau}\{T_i\} + \max_{\forall \tau_i \in \tau}\{a_i\}$. Therefore, by substitution of t_0 in Equation (3.1) the dbf becomes as follows :

$$\text{dbf}(0, t_1) = \sum_{\forall \tau_i \in \tau} \max \left\{ 0, \left(\left\lfloor \frac{t_1 - D_i}{T_i} \right\rfloor - \left\lceil \frac{-\, a_i}{T_i} \right\rceil \right) \right\} \cdot C_i \tag{3.3}$$

3.1.2.2 Quick Convergence Processor-Demand Analysis

Quick convergence Processor-demand Analysis (QPA) [Zhang and Burns, 2009a,b] is a necessary and sufficient feasibility test for the schedulability of synchronous arbitrary-deadline model task sets scheduled using EDF. This means that any task τ_i arrives at time zero ($a_i = 0$) and its relative deadline D_i could be larger than its period T_i. The QPA builds on the traditional processor demand analysis (dbf), previously detailed in Section 3.1.2.1. However, it provides fast and simple schedulability test, because QPA has a tight interval $[t_0, t_1]$ compared to dbf. This decreases the number of absolute deadlines that need to be checked in the interval $[t_0, t_1]$, and hence reduces the

Algorithm 1: Quick convergence Processor-demand Analysis (QPA) [Zhang and Burns, 2009a].

```
τ: Task set.
h(t): Processor demand.
1  begin
2      $t \leftarrow \max_{\forall \tau_i \in \tau} \{\overline{D}_i | \overline{D}_i < t_1\}$
3      while $(h(t) \leq t) \wedge (h(t) > \min_{\forall \tau_i \in \tau} \{D_i\})$ do
4          if $(h(t) < t)$ then
5              $t \leftarrow h(t)$
6          else
7              $t \leftarrow \max_{\forall \tau_i \in \tau} \{\overline{D}_i | \overline{D}_i < t\}$
8          end
9      end
10     if $(h(t) \leq \min_{\forall \tau_i \in \tau} \{D_i\})$ then
11         – The task set is schedulable.
12     else
13         – The task set is not schedulable.
14     end
15 end
```

calculation effort exponentially in most situations.

The QPA checking interval starts by $t_0 = 0$ and ends by t_1, which is the minimum value of the upper bound for the schedulability test t_a and the synchronous busy period of a processor t_b. Considering that the upper bound t_a is not well defined (divide by 0) when the utilization of the task set U is equal to 1, let t_1 be defined as follows [Zhang and Burns, 2009a]:

$$t_1 = \begin{cases} \min(t_a, t_b) & U < 1 \\ t_b & U = 1 \end{cases} \tag{3.4}$$

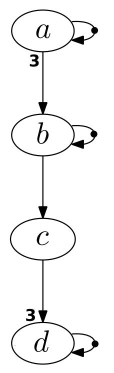
The upper bound for the schedulability test t_a is defined as follows [Zhang and Burns, 2009a]:

$$t_a = \max\left\{ D_1, \ldots, D_n, \frac{\sum_{i=1}^{n} (T_i - D_i) \cdot U_i}{1 - U} \right\} \tag{3.5}$$

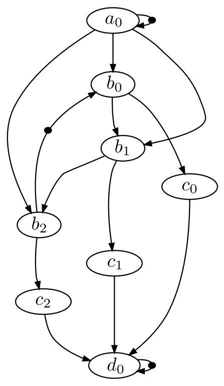
The synchronous busy period of a processor t_b is the period in which all tasks are released simultaneously at the beginning of the processor busy period at their maximum rate, and ended by the first processor idle period (the length of such a period can be zero). The length of the synchronous busy period t_b can be computed by the following process [Ripoll et al., 1996, Spuri, 1996]:

$$w^0 = \sum_{i=1}^{n} C_i \tag{3.6}$$

$$w^{m+1} = \sum_{i=1}^{n} \left\lceil \frac{w^m}{T_i} \right\rceil \cdot C_i \tag{3.7}$$



(a) SDF graph.



(b) HSDF graph.

Figure 3.2: An SDF graph and its HSDF representation.

where the recurrence stops when $w^{m+1} = w^m$, and then $t_b = w^{m+1}$.

The QPA is an iterative algorithm that starts with a value of t close to t_1, and then, iterates back through a simple expression toward 0. The value of this t sequence converges for an unschedulable system to $\min_{\forall\tau_i\in\tau}\{D_i\}$, and converges for a schedulable system to 0. A general task set is schedulable iff $U \leq 1$ and the result of the iterative Algorithm 1 is $h(t) \leq \min_{\forall\tau_i\in\tau}\{D_i\}$, where $h(t)$ is the processor demand defined as follows:

$$h(t) = \sum_{\forall\tau_i\in\tau} \max\left\{0, \left(1 + \left\lfloor\frac{t - D_i}{T_i}\right\rfloor\right)\right\} \cdot C_i \tag{3.8}$$

3.2 Dataflow Computational Model

Dataflow is a natural paradigm for describing DSP and streaming applications for concurrent implementation on parallel hardware. Dataflow programs are directed graphs where each node represents a function and each edge represents a signal path with a dependency. In this section, we give a quick overview of Synchronous Dataflow [Lee and Messerschmitt, 1987a] and Homogeneous Synchronous Dataflow models of computation, which are widely used in modelling and analysis of streaming applications.

3.2.1 Synchronous Dataflow

The Synchronous Dataflow (SDF) [Lee and Messerschmitt, 1987a] model of computation is widely used in modeling and analyzing streaming and concurrent multimedia applications [Bhattacharyya et al., 1999, Sriram and Bhattacharyya, 2000]. Its use has been increasingly considered for designing applications for multi-/many-core platforms [Poplavko et al., 2003]. Synchronous Dataflow (SDF) is a special case of dataflow; an actor is considered synchronous if the number of input tokens that are consumed on each input (consumption rate) and the number of output tokens that are produced on each output (production rate) can be specified a priori. An SDF application is a set of synchronous nodes connected to each other with channels, where the same behaviour repeats in each actor every time it is fired. These channels can have initial tokens. Every initial token represents a delay between the token produced and the token consumed at the other end of the channel. Tokens are always consumed in a First In First Out (FIFO) order.

From this definition, any SDF application can be formally represented by a Directed Cyclic Graph (DCG) $G = \langle V, E, d \rangle$, where V is the set of nodes, E is the edges connecting them and d is the set of delays (initial tokens) on the edges of the graph. Each node in this graph is an actor v_i and each edge is a communication channel. Figure 3.2(a) shows an example of an SDF graph that represents a streaming application. It consists of four actors (*nodes*) (v_a, v_b, v_c, v_d) connected to each other by channels (*edges*). Each actor's production and consumption rate is written next to its ports. However, in case not indicated it is equal to 1. For example, actor v_c has input and output ports with production and consumption rates of (1, 1), respectively. Initial tokens are indicated on the channel by a black dot and a number indicating the amount of initial tokens, as shown in Figure 3.2(a).

An SDF graph G can be described by a topology matrix Γ, where the element Γ_{ij} is defined as the number of tokens produced on the i^{th} channel (*edge*) by the j^{th} actor (*node*) [Lee and Messerschmitt, 1987b]. There is one row in this matrix for each channel in the graph, with a positive element for the actor that produces tokens on the channel and a negative element for the actor that consumes. All the other elements in the row are zero. Equation (3.9) shows the topology matrix Γ of the SDF graph in Figure 3.2(a).

$$\Gamma = \begin{bmatrix} 3 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -3 \end{bmatrix} \tag{3.9}$$

An SDF graph has two main properties, they are *liveness* and *consistency*. For an SDF graph to be *live*, all its actors must be firing indefinitely. If its actors have a maximal execution (firing) of finite length, the SDF graph has a *deadlock*. For an SDF graph to be *consistent*, a shortest non-empty sequence of actor firings should exist called a *repetition vector* $\vec{q}$. The *repetition vector* $\vec{q}$ must satisfy the balance equations

$$\Gamma.\vec{q} = \vec{0} \tag{3.10}$$

, where each element $\vec{q}_j$ of the repetition vector specifies the number of firings of the j^{th} actor. Applying Equation (3.10) on the example of Figure 3.2(a), the repetition vector $\vec{q}$ will be:

$$\vec{q} = \begin{bmatrix} 1 & 3 & 3 & 1 \end{bmatrix}^T \tag{3.11}$$

When each actor is fired the number of times specified by $\vec{q}$, the distribution of tokens on all channels return to their initial state. This is referred to as a *complete cycle* or *graph iteration*. Each actor $v_i \in V$ has a computation time denoted by C_i. The j^{th} firing of an SDF actor v_i in V is denoted by v_{i_j} and executes for C_i time units.

Every SDF application has a throughput requirement and a latency constraint that must be satisfied for correct execution of the application. The throughput requirement ζ is a performance measure that determines the minimum output data rate of the application (iterations per time unit). In contrast, the latency requirement $\mathcal{D}$ is an end-to-end timing constraint that defines the latest possible time a complete graph iteration of G could finish its execution relative to the iteration start time. In this work, the end-to-end deadline constraint $\mathcal{D}$ value must be greater than or equal to the execution time of the critical path (CP) in G, defined as follows:

$$\mathcal{D} \geq \sum_{\forall v_{i_j} \in CP} C_i \tag{3.12}$$

Intuitively, the CP is the longest path of firings v_{i_j}, in terms of execution time C_i, from the input to the output of G.

3.2.2 Homogenous Synchronous Dataflow

Homogeneous Synchronous Dataflow (HSDF) [Lee and Messerschmitt, 1987a] is a special case of SDF graphs in which all production and consumption rates associated with actor ports are equal to one. Therefore, when each actor is fired once, the distribution of tokens on all channels return to their initial state completing a graph iteration. Applying this definition, the repetition vector for an HSDF graph is one for all actors. Just like SDF, any HSDF application can be formally represented by a Directed Cyclic Graph (DCG) $G_h = \langle V_h, E_h, d \rangle$, where V_h is the set of nodes, E_h is the edges connecting them and d is the set of delays (initial tokens) on the edges of the graph.

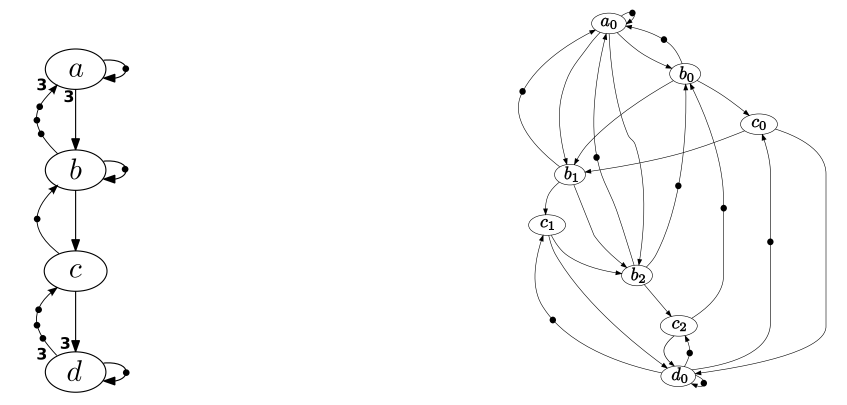
Similarly to SDF, an HSDF application has a throughput requirement and a latency constraint that must be satisfied for correct execution of the application. The throughput requirement ζ is a performance measure that determines the minimum output data rate of the application (iterations per time unit). In contrast, the latency requirement $\mathcal{D}$ is an end-to-end timing constraint that defines the latest possible time a complete graph iteration of G could finish its execution, as defined in Equation (3.12). The end-to-end timing constraint $\mathcal{D}$ is a deadline between the firings of the input and output actor(s) in the same iteration. The input and output actor(s) of an HSDF graph may have multiple route(s) between them, each referred to as a *time-constrained path* P. Fundamentally, the requirement $\mathcal{D}$ must be greater than or equal to the sum of execution times C_i of all actors on the critical path (CP) for the application to be schedulable. Formally, a time-constrained path P is defined as follows:

$$P = \{\langle v_x, \ldots, v_y \rangle : v \subseteq V_h\} \tag{3.13}$$

where, v_x refers to the input actor, v_y refers to the output actor, and its end-to-end latency constraint

$$\mathcal{D} \geq \sum_{\substack{i=x, \\ \forall v_i \in CP}}^{y} C_i \tag{3.14}$$

If P is cyclic, it terminates in the last node before reaching an already visited node. This means, in case of cyclic path, v_x refers to the first visited actor and v_y refers to the last visited actor before reaching an already visited one. For example, in the HSDF graph shown in Figure 3.2(b)



(a) SDF graph (b) HSDF graph

Figure 3.3: An SDF graph and its HSDF representation with finite-size buffers.

$(v_{a_0}, v_{b_0}, v_{c_0}, v_{d_0})$ is not cyclic, because it starts at actor v_{a_0} and ends at actor v_{d_0}, while $(v_{b_0}, v_{b_1}, v_{b_2})$ is cyclic because it terminates at actor v_{b_2} before repeating itself again. Each time-constrained path P starts at one of the input actors and ends at one of the output actors and its latency constraint is $\mathcal{D}$. For example, assume the HSDF application in Figure 3.2(b) has an end-to-end latency constraint $\mathcal{D}$. Then, all time-constrained paths must start with actor v_{a_0} and end with actor v_{d_0}, unless P is cyclic.

3.2.3 Buffer Modelling in Dataflow Graphs

In theory, SDF channels have infinite buffer sizes. However, in practice SDF channel buffer sizes must be finite. Finite buffer sizes for channels can be modelled by adding back-edges carrying a number of initial tokens. These initial tokens on each back-edge represent the buffer size (in tokens) available to the corresponding channel. Figure 3.3(a) shows the example application from Figure 3.2(a), considering finite buffer sizes. As we can see, the channels (e_{ab}, e_{bc}, e_{cd}) have buffer sizes of $(3, 1, 3)$ tokens, respectively. These buffer sizes are modelled as back-edges (e_{ba}, e_{cb}, e_{dc}) carrying initial tokens equivalent to the corresponding channel buffer size, as shown in Figure 3.3(a). Modelling buffers in an SDF graph affects its execution behaviour, because it adds extra dependencies between firings of different actors, limiting the set of possible firing sequences of the graph. Figure 3.3(b) shows an HSDF graph representation of the SDF graph shown in Figure 3.3(a). As we can see, firing v_{b_1} is dependent on the three firings v_{a_0}, v_{b_0} and v_{c_0}. However, in the infinite buffer case shown in Figure 3.2(b) the same firing v_{b_1} is only dependent on firings v_{a_0} and v_{b_0}, which gives the application the freedom to fire v_{b_1} and v_{c_0} in parallel.

3.3 Multi-/Many-Core Platforms

Multi-core platforms increasingly provide higher performance by increasing the number of cores in a chip, as a result of the consequences of Moore's Law and power dissipation. This widespread trend, usually referred as the "the multi-core revolution", is now even more challenging, as chips

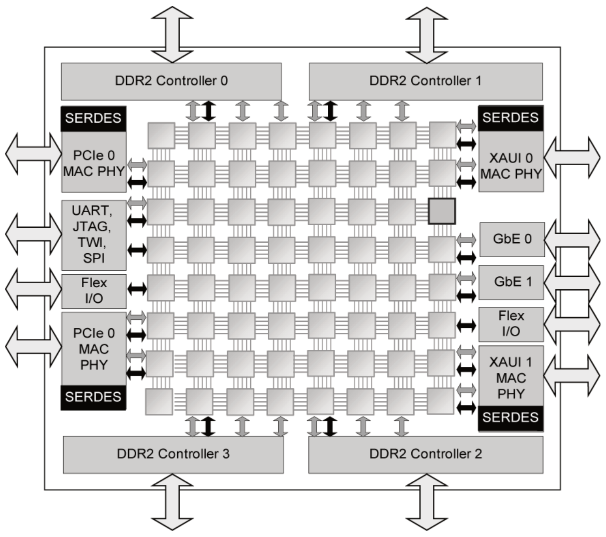


Figure 3.4: TILE64™ block diagram [Bell et al., 2008].

start to become many-core, that is multi-core chips with an even higher number of cores (tens to hundreds), interconnected by Networks-on-Chip (NoC). Examples of this trend include the Tilera Tile CPUs [Wentzlaff et al., 2007] (TILE64™ features 64 cores), Intel's Single-Chip Cloud Computer (SCC) [Mattson et al., 2010] (an experimental processor with 48 cores), Intel Many Integrated Core (MIC) [Seiler et al., 2008] (Xeon Phi features 60 cores), STMicroelectronics P2012 [Benini et al., 2012] (prototypes are available with 69 cores), Kalray's Multi Purpose Processing Array (MPPA) [de Dinechin et al., 2013] (up to 1024 cores – current version is 256 cores) or the Adapteva Epiphany with up to 4096 cores (available now with 1024 cores) [Ada].

These many-core architectures allow both to concentrate multiple applications into the same processor, maximizing the hardware utilization, and reducing cost, size, weight, and power requirements, and to improve application performance by exploiting parallelism at the application level.

This thesis considers multi-core platforms with identical cores (Homogeneous architecture), such as TILE64™ [Wentzlaff et al., 2007]. The processor model Π incorporates a number of identical cores π_n interconnected by a 2D-mesh IN, $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$. Each core is a full-featured processor that includes a non-blocking switch that connects the tile to the 2D mesh IN. The IN uses X-Y routing algorithm accompanied by wormhole switching and TDM arbitration for transferring data and managing traffic between different cores π_n. The speed of transferring data on the IN is determined by the link capacity $\mathcal{L}$ of the IN, which is measured in bits per second (bps). It is defined as in [Nikolić et al., 2013, Shi and Burns, 2008]:

$$\mathcal{L} = \frac{f}{l_{sw} + l_t} \tag{3.15}$$

where f represents the flit size in bits, l_{sw} and l_t represent the switch latency and transfer latency of one flit in seconds, respectively. An application A_i running on the platform Π can reserve a dedicated bandwidth on the IN to assure a required performance called reservation bandwidth $\mathcal{R}_i$, which is a fraction of $\mathcal{L}$. According to this specification, the time required for a packet p of an application A_i traversing the IN from source to destination $C_{i,p}$ is defined as:

$$C_{i,p} = C_{i,p}^{iso} + I_i^{TDM} \tag{3.16}$$

where $C_{i,p}^{iso}$ is the isolation time that represents the time required by the packet to reach its destination without suffering interference, and I_i^{TDM} is the interference caused by the TDM arbiter. The isolation time $C_{i,p}^{iso}$, also known in the literature as basic network latency, is equal to the delay of the first flit (*header*) to reach the destination router, augmented by the processing delay of all remaining flits (*payload*) at the destination router.

The isolation time $C_{i,p}^{iso}$ of packet p from an application A_i is defined as:

$$C_{i,p}^{iso} = \frac{C_{i,p}^{iso.\mathcal{L}}}{\mathcal{R}_i} \tag{3.17}$$

where $C_{i,p}^{iso.\mathcal{L}}$ is the packet isolation time assuming full link capacity $\mathcal{L}$. The $C_{i,p}^{iso.\mathcal{L}}$ is defined as in [Nikolić et al., 2013, Shi and Burns, 2008]:

$$C_{i,p}^{iso.\mathcal{L}} = \underbrace{h_p \cdot (l_{sw} + l_t)}_{\text{header}} + \underbrace{\left\lceil \frac{p_j}{f} \right\rceil \cdot l_t}_{\text{payload}} \tag{3.18}$$

where h_p represents the number of hops of the packet p, and p_j represents the packet size in bits. The first flit (*header*) establishes the path, which means it experiences the switch latency l_{sw} of the routers and the transfer latency l_t of the communication links on its path to destination, as demonstrated in the first term of Equation (3.18). However, the rest (*payload*) follows the header in a pipelined manner, i.e. when the first flit progresses from one router to the next, the rest of the flits follow, each separated by the transfer latency. This means, the payload only experiences the transfer latency l_t of the communication links, since the path has been already established by the header. As we are using X-Y routing algorithm for directing traffic on the IN, the number of hops h_p of a packet p is defined as:

$$h_p = |x_1 - x_2| + |y_1 - y_2| \tag{3.19}$$

where (x_1,y_1) and (x_2,y_2) are the locations of the source and destination cores in the platform Π, respectively. Substituting Equation (3.15) in Equation (3.18) results in:

$$C_{i,p}^{iso.\mathcal{L}} = \underbrace{h_p \cdot \frac{f}{\mathcal{L}}}_{\text{header}} + \underbrace{\left\lceil \frac{p_j}{f} \right\rceil \cdot \left(\frac{f}{\mathcal{L}} - l_{sw} \right)}_{\text{payload}} \tag{3.20}$$

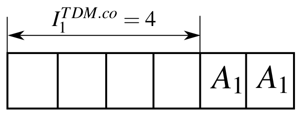


Figure 3.5: A TDM frame with frame size $\mathcal{F}$ of 6 where 2 allocated slots $\varkappa_1$ to application A_1 for continous slot assignment policy [Akesson et al., 2015].

By substituting Equation (3.20) in Equation (3.17)

$$C_{i,p}^{iso} = \underbrace{h_p \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i}}_{\text{header}} + \underbrace{\left\lceil \frac{p_j}{f} \right\rceil \cdot \left(\frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{l_{sw}}{\mathcal{R}_i} \right)}_{\text{payload}} \tag{3.21}$$

Before defining the equation that computes the TDM interference, we have to understand the mechanism of a TDM arbiter. A TDM arbiter operates by periodically repeating a schedule, referred to as a frame, that determines which application(s) that may be injected into the IN at a particular time to as a frame. The frame comprises a number of slots $\mathcal{F}$, each corresponding to a single IN access with bounded execution time of $l_{sw} + l_t$. Every application is allocated a number of slots $\varkappa_i$ in the frame at design time. The percentage of bandwidth allocated to an application A_i (reservation bandwidth $\mathcal{R}_i$) is determined by the number of allocated slots $\varkappa_i$ in the frame and is computed according to Equation (3.22), defined in [Akesson et al., 2015].

$$\mathcal{R}_i = \frac{\varkappa_i}{\mathcal{F}} \tag{3.22}$$

The TDM interference I_{TDM}, on the other hand, depends on the slot assignment policy that determines how the allocated slots are distributed in the frame. A commonly used slot assignment policy is to use a *continuous allocation* [Foroutan et al., 2013, Gomony et al., 2013, Goossens et al., 2013b,c, Vink et al., 2008], where slots allocated to an application appear consecutively in the frame, as shown in Figure 3.5. For this policy, the TDM interference of an application (in slots) can simply be computed according to Equation (3.23), as defined in [Akesson et al., 2015].

$$I_i^{TDM.co} = \mathcal{F} - \varkappa_i \tag{3.23}$$

For example, the TDM interference of an application A_1 (in slots) that has been assigned two slots ($\varkappa_1 = 2$) in a TDM frame of size six ($\mathcal{F} = 6$) using continuous slot assignment policy is equal to four ($I_i^{TDM.co} = 4$), as shown in Figure 3.5. The advantage of the continuous slot assignment policy is that it is simple to understand and implement, and that both the interference and the bandwidth are straight-forward to compute. By substitution Equation (3.22) in Equation (3.23), $I_i^{TDM.co}$ is equal to:

$$I_i^{TDM.co} = \frac{\varkappa_i}{\mathcal{R}_i} - \varkappa_i \tag{3.24}$$

In this thesis, we adopt the continuous slot assignment policy. Therefore, I_i^{TDM} (in seconds) is defined as:

$$I_i^{TDM} = \left(\frac{\varkappa_i}{\mathcal{R}_i} - \varkappa_i\right) \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i} \tag{3.25}$$

where $\frac{f}{\mathcal{L} \cdot \mathcal{R}_i}$ represents the duration of a single interfering slot in seconds. Equation (3.25) assumes that TDM slots are atomic, which means that the worst-case arrival of a packet p is just after its own slot has finished. However, the real worst-case arrival of a packet p is one clock cycle after its own slot has started, because TDM slots are not atomic, while clock cycles are atomic though. This means the packet p has missed the start of its own slot and it will either be empty, or used by the packets of another application depending on, whether or not, the resource is work-conserving. In this case, the I_i^{TDM} (in seconds) is defined as:

$$I_i^{TDM} = \left(\frac{\varkappa_i}{\mathcal{R}_i} - \varkappa_i\right) \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i} + \left(\frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{1}{IN_{freq}}\right) = \left(\frac{\varkappa_i}{\mathcal{R}_i} - \varkappa_i + 1\right) \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{1}{\mathcal{G}} \tag{3.26}$$

where $\mathcal{G}$ is the IN frequency and $\frac{1}{\mathcal{G}}$ is the duration of one cycle in seconds. By substituting Equations (3.21) and (3.26) in Equation (3.16), the WCET of a packet is defined as:

$$C_p = \underbrace{h_p \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i}}_{\text{header}} + \underbrace{\left\lceil \frac{p_p}{f} \right\rceil \cdot \left(\frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{l_{sw}}{\mathcal{R}_i}\right)}_{\text{payload}} + \underbrace{\left(\frac{\varkappa_i}{\mathcal{R}_i} - \varkappa_i + 1\right) \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{1}{\mathcal{G}}}_{\text{interference}} \tag{3.27}$$

Equation (3.27) shows that execution time of a packet comprises three terms. The first term is the time spent by the packet's header (a single flit) to traverse the IN. The second term is the time taken by the packet's payload to traverse the IN, following the header's established path in a pipelined manner. The third term is the interference suffered by the packet during traversing the IN.

3.4 System Model

Formally, we consider a system $\Psi = \langle \Pi, A \rangle$ based on a homogeneous symmetrical multi-core platform Π of size $n \times n$. Each core π_n is a full-featured processor that includes a non-blocking switch that connects the tile to the 2D-mesh IN. The IN uses X-Y routing algorithm accompanied by wormhole switching and TDM arbitration for transferring data and managing traffic between different cores π_n. The speed of transferring data on the IN is determined by the link capacity $\mathcal{L}$ of the IN, which is measured in bits per second (bps). The platform Π runs the set of periodic applications A that comprises independent real-time tasks and equivalent HSDF representation of SDF applications. Any SDF graph G can be converted to an equivalent HSDF graph G_h by using a conversion algorithm, such as the one presented in [Sriram and Bhattacharyya, 2000]. Each HSDF graph G_h running on the platform Π has a dedicated percentage of the link capacity $\mathcal{L}$ called reservation bandwidth $\mathcal{R}$. This reserved bandwidth ($\mathcal{L} \cdot \mathcal{R}_i$) per application A_i guarantees a dedicated link capacity for the tokens exchanged by the HSDF graph G_h across the IN, preventing racing between applications on the medium. The actors of the G_h represents the firings of the actors of the SDF graph G. Therefore, the set of actors of the HSDF graph V_h represents the firings of the set of actors of the SDF graph V and the number of initial tokens d for both of them is exactly the same. In this model, we assume that all the SDF applications in A have periodic input sources. Therefore, each actor v_i in the HSDF graph G_h can be considered a periodic task. All G_h actors can be scheduled on Π using traditional real-time schedulers.

A periodic task $\tau_i \in V$ is represented by the 4-tuple $\tau_i = (a_i, C_i, T_i, D_i)$, where a_i is the relative offset that specifies the start instant of an actor, C_i is the worst-case execution time, T_i is the relative period and D_i is the relative deadline of the task. The *utilization* of task τ_i is denoted by U_i and is defined as $U_i = C_i/T_i$, where $U_i \in (0,1]$. Additionally, the *density* of task τ_i is denoted by ρ_i and is defined as $\rho_i = C_i/D_i$, where $\rho_i \in (0,1]$. All tasks are modelled as arbitrary-deadline tasks.

In this model, we assume that all actors computation time C_i are equal to the Worst Case Execution Time (WCET), which can be determined using methods and tools detailed in [Wilhelm et al., 2008]. Therefore, each firing v_{i_j} of an actor v_i in any SDF application can be considered a periodic task with an execution time C_i equal to WCET. The choice of WCET is safe, because the dataflow model of computation is monotonic, which means faster execution of actors does not result in a worse performance.

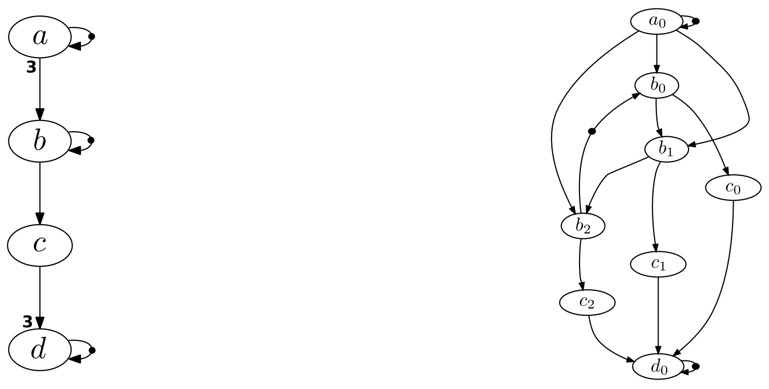
Chapter 4

Reducing Complexity of Dataflow Graphs

As explained Chapter 1, future real-time embedded systems integrate mixed application models with timing constraints on the same multi-core platform. Extraction of timing parameters (offsets, deadlines, periods) from these applications enables the use of real-time scheduling and analysis techniques, allowing to provide guarantees on satisfying timing constraints. However, existing extraction techniques require the transformation of the dataflow application from highly expressive dataflow computational models, e.g., Synchronous Dataflow (SDF) and Cyclo-Static Dataflow (CSDF) to Homogeneous Synchronous Dataflow (HSDF). This transformation can lead to an exponential increase in the size of the application graph that significantly increases the run-time of the analysis [Geilen, 2009].

In this chapter, we address this problem by proposing an offline heuristic algorithm called *slack-based merging* [Ali et al., 2017]. The algorithm is a novel graph reduction technique that helps speeding up the process of timing parameter extraction and finding a feasible real-time schedule, thereby reducing the overall design time of the real-time system, as we later show in Chapters 5 and 6. It uses two main concepts: **a)** the difference between the timing constraints of the SDF graph and the WCET of its firings (*slack*) to merge firings and generate a reduced-size HSDF graph, and **b)** the novel concept of merging called a *safe merge*, which is a merge operation that we prove cannot cause a live HSDF graph to deadlock.

We begin our journey through this chapter by defining parameters and concepts that help in understanding the slack-based merging algorithm in Section 4.1. Then, we explain the novel safe merge concept in Section 4.2. After this essential overview, we present the slack-based merging algorithm in Section 4.3, followed by the experimental evaluation of the proposed algorithm in Section 4.4. Finally, we conclude the chapter with a summary in Section 4.5.



(a) SDF graph (b) HSDF graph

Figure 4.1: An SDF graph and its HSDF representation.

4.1 Definitions

In this section, we define parameters and concepts essential to the specification of our algorithm. They are: **1)** the *set of predecessor firings* $\Omega(v_{i_j})$, **2)** the *set of successor firings* $\Phi(v_{i_j})$, **3)** the *earliest start time of a firing* ϑ_{i_j}, **4)** the *latest finish time of a firing* θ_{i_j}, **5)** *the topologically ordered set of actors* $\hat{V}$ and **6)** the concept of *dependent/independent firings*.

First, the set of predecessor firings $\Omega(v_{i_j})$, is defined as follows:

Definition 4.1 (***Set of predecessor firings*** $\Omega(v_{i_j})$)**.** *In an SDF application G, a set of predecessor firings* $\Omega(v_{i_j})$ *defines the collection of firings that must execute to enable firing* v_{i_j}*.* $\Omega(v_{i_j})$ *represents the set of precedence constraints that must be satisfied before firing* v_{i_j}*.*

Second, the set of successor firings $\Phi(v_{i_j})$, is defined as follows:

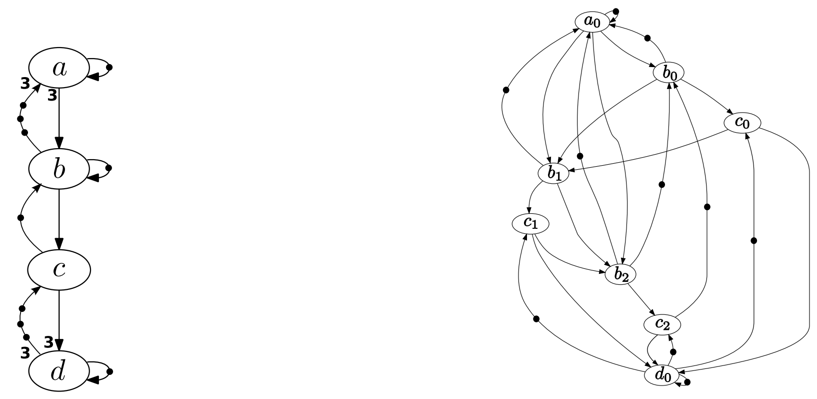
Definition 4.2 (***Set of successor firings*** $\Phi(v_{i_j})$)**.** *In an SDF application G, a set of successor firings* $\Phi(v_{i_j})$ *defines the collection of firings that cannot execute before* v_{i_j}*.* $\Phi(v_{i_j})$ *represents the set of firings dependent on firing* v_{i_j}*.*

Third, the earliest start time of a firing defines the earliest possible time instance a firing v_{i_j} can start its execution. It is defined as follows:

Definition 4.3 (***Earliest start time of a firing***)**.** *In an SDF application G, the earliest start time of the* j^{th} *firing* v_{i_j} *of an actor* v_i *occurs once all of its input ports have the required input tokens. The required input tokens are available when the latest firing in the set of predecessor firings* $\Omega(v_{i_j})$ *occur. Therefore, the earliest start time* ϑ_{i_j} *of a firing* v_{i_j} *is expressed as follows:*

$$\vartheta_{i_j} = \begin{cases} 0 & \textit{if } \Omega(v_{i_j}) = \varnothing \\ \max_{\forall v_{l_k} \in \Omega(v_{i_j})} (\vartheta_{l_k} + C_l) & \textit{if } \Omega(v_{i_j}) \neq \varnothing \end{cases} \tag{4.1}$$

where C_l *is the WCET of actor* v_l *and* $\varnothing$ *is the empty set.*



(a) SDF graph (b) HSDF graph

Figure 4.2: An SDF graph and its HSDF representation with finite-size buffers.

Fourth, the latest finish time of a firing parameter defines the latest possible time instance a firing v_{i_j} can finish its execution. It is defined as follows:

Definition 4.4 (***Latest finish time of a firing***)**.** *The latest finish time of the j^{th} firing v_{i_j} of an actor v_i in an SDF graph G defines the latest possible time it finishes its execution such that the latency constraint $\mathcal{D}$ of the graph G is satisfied. Therefore, the latest finish time θ_{i_j} of a firing v_{i_j} is expressed as follows:*

$$\theta_{i_j} = \begin{cases} \mathcal{D} & \textit{if } \Phi(v_{i_j}) = \varnothing \\ \min_{\forall v_{l_k} \in \Phi(v_{i_j})} (\theta_{l_k} - C_l) & \textit{if } \Phi(v_{i_j}) \neq \varnothing \end{cases} \tag{4.2}$$

Fifth, a topologically ordered set of actors defines the order in which firings are selected for a merge. It is defined as follows:

Definition 4.5 (***Topologically ordered set of actors***)**.** *The topologically ordered set of actors $\hat{V}$ is a set in which the actor set V is sorted in a breadth-first traversal sequence, where the input actors (parents) are in the beginning of the set followed by their successor actors (children). In case a group of actors are on the same level in the graph, they are listed in $\hat{V}$ in arbitrary order. The only order considered in $\hat{V}$ is parents followed by children. In case of cyclic graphs, all back edges with initial tokens are ignored.*

For example, in case of the graph shown in Figure 4.1(a), the topological ordered set of actors $\hat{V}$ is (v_a, v_b, v_c, v_d).

Last, the *dependent / independent firings* is a term that describes the connectivity relation between two firings, which helps in deciding whether a merge is safe or not. It is defined as follows:

Definition 4.6 (***Dependent / independent firings***)**.** *Two firings are* dependent *iff there is a sequence of edges (not a single edge) connecting them carrying zero initial tokens. Otherwise, they are* independent *firings.*

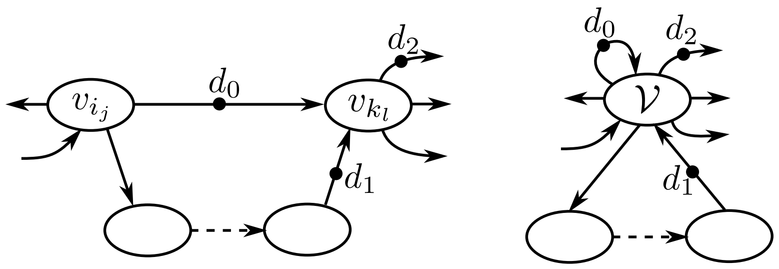


Figure 4.3: A *safe merge* operation of two *independent* firings (v_{i_j}, v_{k_l}) into a new cluster $\mathcal{V}$.

For example, the firings v_{b_0} and v_{b_1} of actor v_b in the cases with infinite and finite buffers shown in Figures 4.1(b) and 4.2(b), respectively. In case of infinite buffers, these firings are independent, since there is no path between them other than the direct edge (e_{b_0,b_1}), as shown in Figure 4.1(b). However, in case of finite buffers, they are considered dependent firings due to the existence of a path between the firings v_{b_0} and v_{b_1} that consists of the firings ($v_{b_0}, v_{c_0}, v_{b_1}$) connected by the sequence of edges (e_{b_0,c_0}, e_{c_0,b_1}) that have zero initial tokens, as shown in Figure 4.2(b).

4.2 Safe Merge

In this section, we present the concept of safe merge, which is a cornerstone of the slack-based merging algorithm presented in Section 4.3. First, we begin by defining a safe merge operation and its function in Section 4.2.1. Then, we discuss the safety of such operation and its effect on the liveness of HSDF graphs in Section 4.2.2.

4.2.1 Definition and Function

The safe merge concept is a novel idea for merging HSDF graphs. It is basically a merging operation of any two firings that is defined as follows:

Definition 4.7 (***Safe merge***)**.** *A safe merge operation is an act of combining two independent firings (v_{i_j}, v_{k_l}) creating a new cluster $\mathcal{V}$ with an execution time equal to the sum of execution time of both firings. The new cluster $\mathcal{V}$ has the union of input/output ports and channels of both firings except the ports and channels carrying zero initial tokens between both firings (v_{i_j}, v_{k_l}). A safe merge operation keeps all the initial tokens in the graph distributed on the same edges without change.*

Figure 4.3 shows a merging operation between two independent firings (v_{i_j}, v_{k_l}) into a new cluster $\mathcal{V}$. The two firings are *independent* according to the Definition 4.6, because the only path connecting them (other than the direct edge that carries the initial token d_0) consists of a sequence of edges that carry the initial token d_1. As we can see, the safe merge operation kept the distribution of the initial tokens (d_0, d_1, d_2) the same after the merge.

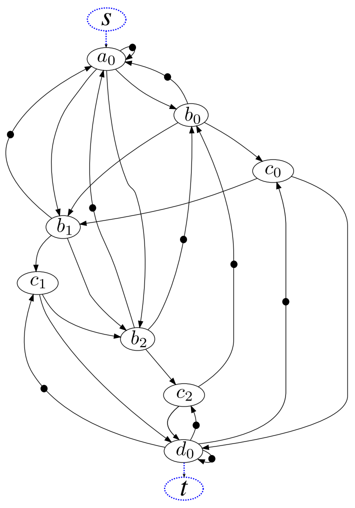


Figure 4.4: HSDF graph after adding s and t.

4.2.2 A Safe Merge is Deadlock-Free

Applying safe merge operations on the graph ensures that the resulting graph is deadlock free. However, before going into the proof details of this statement, we provide necessary preliminaries (definitions and theorems) that helps in understanding and constructing our proof.

Assume that $G_h = \langle V_h, E_h, d \rangle$ is a consistent and live (Section 3.2.1) HSDF graph, where V_h is the set of firings of the SDF actors, E_h is the set of edges connecting them and d is the set of initial tokens. Also, assume all the inputs/outputs of G_h are connected to dummy nodes source s and sink t, respectively. Accordingly, in case of the HSDF graph G_h shown in Figure 4.2(b), the dummy nodes s and t will be connected to v_{a_0} and v_{d_0}, respectively, as shown in Figure 4.4. First, we would like to define some terms:

Definition 4.8 (***End-to-end path***)**.** *An end-to-end path is a path P that consists of distinctive firings that traverses the graph from source s to sink t. It is defined as follows:*

$$P = \langle s, v_{i_j}, \dots, t \rangle \tag{4.3}$$

Definition 4.9 (***Path cover for a DAG***)**.** *Given a Directed Acyclic Graph (DAG), a path cover $\mathscr{P}$ is a set of end-to-end paths such that every firing in the DAG belongs to at least one end-to-end path $P \in \mathscr{P}$.*

Definition 4.10 (***Minimal feedback edge set***)**.** *Given a DCG, a minimal feedback edge set is the minimum set of edges which, when removed from the DCG, leave a DAG. In other words, it is a set containing one back-edge of every cycle in the DCG.*

Definition 4.11 (***Strongly Connected DCG***)**.** *A DCG is strongly connected iff there exists a directed path between each pair of firings.*

Definition 4.12 (***Strongly Connected Component***)**.** *A Strongly Connected Component (SCC) is any strongly connected DCG or a subgraph of it that is strongly connected.*

Definition 4.13 (***Consistency in dataflow graphs*** [Lee, 1991])**.** *A dataflow graph is consistent iff on each edge, in the long run, the same number of tokens are consumed as produced.*

From Definition 4.9, every DAG can be represented as a set of end-to-end paths, referred to as the path cover $\mathscr{P}$. From Definition 4.10, every DCG consists of a DAG and a set of back-edges that creates the cycles. Therefore, from Definitions 4.9 and 4.10, a DCG can be defined as follows:

$$G_h = \langle \mathscr{P}, \mathscr{O}, d \rangle \tag{4.4}$$

where $\mathscr{P}$ is the path cover that represents the DAG component in G_h and $\mathscr{O}$ is the set of cycles in G_h.

An essential theory regarding the liveness of an HSDF graph that has been proved and presented in [Ghamarian et al., 2006] (Theorem 24) states the following :

Theorem 4.1. *An HSDF graph is live and bounded iff it is consistent and all its SCCs are deadlock-free.*

Theorem 4.1 along with Equation (4.4) construct the base for proving our theory that states:

Theorem 4.2. *A safe merge operation on a consistent and live HSDF graph results in a new consistent and live HSDF graph.*

Proof. Let us assume that G'_m is the output graph after applying a single *safe merge* operation on G_h. It is defined as follows:

$$G'_m = \langle \mathscr{P}'_m, \mathscr{O}'_m, d \rangle \tag{4.5}$$

This single safe merge operation results in a consistent graph G'_m by Definition 4.7 and 4.13, because all G'_m ports have production/consumption rates equal to one and its initial tokens distribution is the same as G_h.

The single safe merge operation creates a new path cover and cycle sets, $\mathscr{P}'_m$ and $\mathscr{O}'_m$, respectively. The new path cover $\mathscr{P}'_m$ does not affect the liveness of G'_m. This is due to its elements (end-to-end paths) by Definitions 4.8 and 4.12 are not SCC. Therefore, according to Theorem 4.1 liveness is not affected.

Contrary to $\mathscr{P}'_m$, the cycles set $\mathscr{O}'_m$ consists of elements that are SCC by Definition 4.12. This means that the elements of the $\mathscr{O}'_m$ impact the liveness of G'_m. We proceed by distinguishing **two mutually exclusive and jointly exhaustive cases** for the cycles in $\mathscr{O}'_m$:

Case 1 The subset of cycles that **does not share** the two merged firings. This subset belongs to the original graph G_h before the merge. Also, the safe merge does not affect the distribution of the initial tokens in the graph by Definition 4.7. This means that every edge that carries initial tokens in G_h remains as it is in the graph after the merge G'_m. Therefore, this subset is live because no change occurred on its elements.

Case 2 The subset of cycles that **share** the two merged firings (newly created cluster $\mathcal{V}$). This subset is live as well, because all the firings, as well as $\mathcal{V}$, in G'_m have ports with production/consumption rates equal to one. Also, from Definition 4.7 a safe merge is only applied to independent firings. This means that a cycle can only be created iff there is a path, between the two firings to be merged, and at least one of its edges carries at least one initial token. This means that the newly created cycles have at least a single token on the back edge that keep them live. Therefore, a safe merge does not create a deadlock in G'_m.

Therefore, G'_m is consistent and live. □

From the proof of Theorem 4.2, applying several safe merge operations on G_h results in a consistent and live graph G_m.

4.3 Slack-Based Merging Algorithm

In this section, we present the slack-based merging algorithm intended to reduce the size of an HSDF graph with timing constraints. In the following sections, we introduce the merging strategy of our algorithm (Section 4.3.1), as well as the conditions for guaranteeing a valid merge (Section 4.3.2). Finally, we present the slack-based merging algorithm (Section 4.3.3) followed by its complexity analysis (Section 4.3.4) and an example illustrating how it works (Section 4.3.5).

4.3.1 Merging Strategy

The proposed algorithm combines two ideas: **1)** slack-based merging and **2)** merging firings of the same actor. Before introducing the complete algorithm, we will first discuss the idea of slack-based merging. For this purpose, we formalize the definition of slack.

Definition 4.14 (***Slack***)**.** *The* slack *of a firing j of actor i, v_{i_j}, is the difference between its latest finish time θ_{i_j} and its earliest start time ϑ_{i_j} minus its computation time C_i. It is defined as follows:*

$$\sigma_{i_j} = \theta_{i_j} - \vartheta_{i_j} - C_i \tag{4.6}$$

For example, consider two firings v_{i_j} and v_{i_l} of an actor v_i. If v_{i_j} has σ_{i_j} greater than or equal to the computation time of v_{i_l} ($\sigma_{i_j} \geq C_i$) and the reverse ($\sigma_{i_l} \geq C_i$), the algorithm can merge both firings together in one cluster. This strategy allows having a reduced-size graph without elongating the critical path (CP), defined in Section 3.2.2, larger than $\mathcal{D}$, satisfying the graphs end-to-end latency constraint. However, this is not the only condition to have a valid merge. Section 4.3.2 lists all the conditions in details.

The second strategy aims to merge the firings v_{i_j} of the same actor v_i together in the minimum number of clusters. This helps in generating a reduced-size graph that is suitable for mapping on a message-passing multi-core architectures, because the firings v_{i_j} of the same actor v_i will be mapped on the minimum number of cores. This results in a smaller memory footprint on the platform and less communication overhead.

However, safe merge operation may cause timing constraints to be violated. Therefore, the slack-based merging algorithm has an additional method to ensure that timing constraints are satisfied called a *valid merge*, which is detailed in Section 4.3.2.

4.3.2 Valid Merge

In this section, we present the concept of a *valid merge* that is used by the *slack-based merging* algorithm (Section 4.3.3) to decide whether to accept or reject a merging operation. It is defined as follows:

Definition 4.15 (***Valid merge***)**.** *A valid merge is a safe merge operation between two firings v_{i_j} and v_{i_l} of the same actor $v_i \in G$, resulting in a new graph G_m that satisfies the following two constraints:*

(1) the throughput constraint ζ such that,

$$\zeta_m \geq \zeta \tag{4.7}$$

(2) the end-to-end latency constraint $\mathcal{D}$ such that,

$$\mathcal{D} \geq \sum_{\forall v_{i_j} \in CP \in G_m} C_i \tag{4.8}$$

To satisfy the throughput constraint, G_m must fulfil two conditions:

(a) G_m must be live, i.e. deadlock-free, defined as follows:

$$\zeta_m \neq 0 \tag{4.9}$$

(b) the execution time of each cycle $\mathcal{C}_k \in G_m$ and each merged cluster $\mathcal{V}_o \in G_m$ must not exceed the period constraint $T_,$ which is equal to the inverse of the throughput constraint ζ, $T = 1/\zeta$. This is defined as follows:

$$(\forall \mathcal{C}_k \in G_m) \wedge (\forall \mathcal{V} \in G_m),\ T \geq \sum_{\forall v_{i_j} \in \mathcal{C}_k} C_i,\ T \geq \sum_{\forall v_{i_j} \in \mathcal{V}_o} C_i \tag{4.10}$$

The first condition is satisfied by the *safe merge* operation (Theorem 4.2). It ensures that the merge operation does not create a cycle without an initial token in the generated graph G_m (a deadlock situation). Therefore, we implemented a function that searches for a path between the two firings about to be merged, other than the direct edge connecting them. The function searches for a path that consists of firings connected by edges carrying zero initial tokens (dependent firings). If such a path is found, then the merge is not valid, because the merging process will create an extra illegal cycle that does not have an initial token and leads to deadlock in the application graph. Otherwise, the graph G_m is live. Consider as an example the scenarios in which we would like to merge the firings v_{b_1} and v_{b_2} of actor v_b in the cases with infinite and finite buffers shown in Figures 4.1(b) and 4.2(b), respectively. In case of infinite buffers, merging the firings v_{b_1} and v_{b_2} satisfies the first condition (independent firings), since there is no path between them other than the direct edge (e_{b_1,b_2}), as shown in Figure 4.1(b). Contrarily, in case of finite buffers, this merge does not satisfy the first condition (dependent firings), because it will create an illegal cycle without an initial token. This is due to the existence of a path between the firings v_{b_1} and v_{b_2} that consists of the firings ($v_{b_1}, v_{c_1}, v_{b_2}$) connected by the edges (e_{b_1,c_1}, e_{c_1,b_2}) that have zero initial tokens, as shown in Figure 4.2(b). In this case, the merge between (v_{b_1}, v_{b_2}) into a single cluster $\mathcal{V}_{b_1,b_2}$ creates an illegal cycle without an initial token between the cluster $\mathcal{V}_{b_1,b_2}$ and the firing v_{c_1}, which would result in deadlock.

The second condition is ensured by implementing a function that checks that both the execution time of each cycle $\mathcal{C}_k$ and each merged cluster $\mathcal{V}_o$ (in case of $\mathcal{V}_o$ does not have self-cycles) is not exceeding the application period constraint T. The algorithm identifies all cycles in the application graph and saves them in a lookup table. Each entry in the lookup table contains the cycle and its total execution time. When merging any actor involved in a cycle, the cycle is updated by replacing the actors with the new cluster and calculating the new execution time of the cycle. If the execution time of the cycle exceeds the period of the application the merge is not valid. Otherwise, the merge is approved. In case of merged clusters, the algorithm checks the execution time of every merged cluster and guarantees that it does not exceed the application period.

The slack-based merging algorithm merges as long as each firing v_{i_j} of every actor $v_i \in G$ has non-negative slack ($\sigma_{i_j} \geq 0$). This means that the execution time of the critical path of the application cannot exceed the application end-to-end latency constraint $\mathcal{D}$. This guarantees that the second constraint is satisfied.

4.3.3 The Algorithm

The slack-based merging algorithm, shown in Algorithm 2, aims to generate a simpler, smaller size graph G_m that reduces the run-time of its analysis. The proposed algorithm starts by calculating the earliest start time ϑ_{i_j} and the latest finish time θ_{i_j} for each firing v_{i_j} in the SDF graph G using Equations (4.1) and (4.2), respectively. Then, it computes the slack σ_{i_j} for each firing using Equation (4.6). If all the firings v_{i_j} in G have slack σ_{i_j} greater than or equal to zero ($\forall v_{i_j} \in G, \sigma_{i_j} \geq 0$), a merging operation can possibly be applied. Otherwise, the merging algorithm terminates. When all firings have non-negative slack, the algorithm needs to determine which firings to merge. An optimal algorithm would try all possible combinations of firings from the same actor, for each actor, although this approach does not scale to applications of realistic complexity. Instead, our heuristic algorithm picks the actors v_i in sequence from the topologically ordered set $\hat{V}$ to begin merging different firings. This particular way of selection of firings to be merged is not formally proven to be better than others, but we have experimentally determined that it works rather well. For each actor v_i, the algorithm tries each possible combination of two firings (v_{i_j}, v_{i_l}) for merging, such that $\sigma_{i_j} \geq C_i$ and $\sigma_{i_l} \geq C_i$, and generates a new graph G_m. After merging them, the algorithm checks the validity of the merging operation of (v_{i_j}, v_{i_l}) using the *valid_merge()*

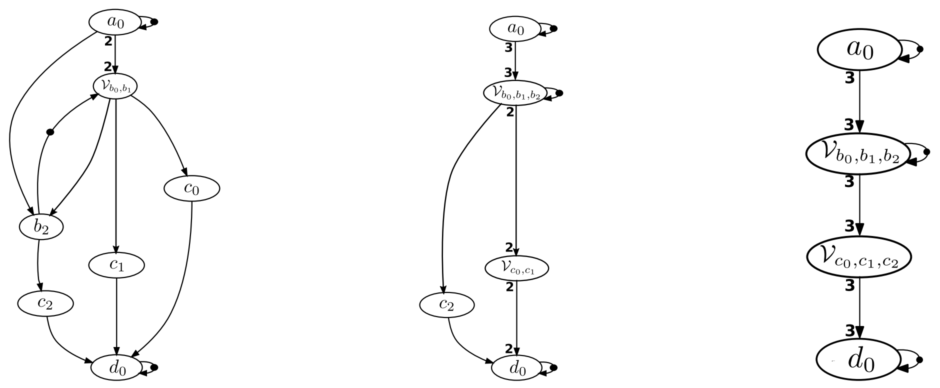
Algorithm 2: Slack-based merging

```
Input:
$G$: SDF application graph, $G = \langle V, E, d \rangle$.
Output:
$G_m$: merged HSDF application graph.
Variables:
$n$: number of actors in $G$.
$V$: set of SDF actors, $V = \{v_1, v_2, \dots, v_n\}$.
$\hat{V}$: breadth-first topologically ordered set of actors.
$\vec{q}$: repetition vector for $G$, $\vec{q} = \{q_1, q_2, \dots, q_n\}$, where $q_i$ is the corresponding number of
    firings of $v_i$.
$v_{i_j}$: is the $j^{th}$ firing of actor $v_i$, where $\{j : j \in \mathbb{Z}, j \in [1, q_i]\}$.
$G_h$: HSDF graph representation of $G$, where $G_h = \langle V_h, E_h, d \rangle$ and $v_{i_j} \in V_h$.

1  begin
2      Convert $G$ to $G_h$.
3      Calculate $\vartheta_{i_j}$, $\{\vartheta_{i_j} : \forall v_{i_j} \in G, Equation\ (4.1)\}$.
4      Calculate $\theta_{i_j}$, $\{\theta_{i_j} : \forall v_{i_j} \in G, Equation\ (4.2)\}$.
5      $\{\sigma_{i_j} : \forall v_{i_j} \in G, \sigma_{i_j} = \theta_{i_j} - \vartheta_{i_j} - C_i\}$.
6      $G_m = G_h$.
7      if $(\forall v_{i_j} \in G_m, \sigma_{i_j} \geq 0)$ then
8          foreach $v_i$ in $\hat{V}$ do
9              $\{v_{i_j},\ v_{i_l} : j \neq l, \sigma_{i_j} \geq C_i\ and\ \sigma_{i_l} \geq C_i\}$.
10             if $(valid\_merge(v_{i_j},\ v_{i_l}))$ then
11                 merge $v_{i_j}$ and $v_{i_l}$ in $G_m$.
12                 Calculate $\vartheta_{i_j}$, $\{\vartheta_{i_j} : \forall v_{i_j} \in G_m, Equation\ (4.1)\}$.
13                 Calculate $\theta_{i_j}$, $\{\theta_{i_j} : \forall v_{i_j} \in G_m, Equation\ (4.2)\}$.
14                 $\{\sigma_{i_j} : \forall v_{i_j} \in G_m, \sigma_{i_j} = \theta_{i_j} - \vartheta_{i_j} - C_i\}$.
15                 if $(\forall v_{i_j} \in G_m, \sigma_{i_j} \geq 0)$ then
16                     $G_h = G_m$
17                 else
18                     $G_m = G_h$
19                 end
20             else
                   // No Merge
21             end
22         end
23     else
           // Stop Merge
24     end
25 end
```

function previously explained in Section 4.3.2. If all the conditions of a valid merge are satisfied, the merge operation is valid. Otherwise, the algorithm will undo the last merging operation and pick up two new candidate firings for merging.

When the merge operation is considered a valid merge, the algorithm recalculates the earliest



(a) Merging of v_{b_0} and v_{b_1} (b) Merging of v_{c_0} and v_{c_1} (c) Final merged graph G_m

Figure 4.5: Example of slack-based merging.

start time ϑ_{i_j}, the latest finish time θ_{i_j} and the slack σ_{i_j} for each firing v_{i_j} in the new output merged graph G_m. If the slack of all firings in G_m are greater than or equal to zero ($\forall v_{i_j} \in G_m, \sigma_{i_j} \geq 0$), the merge operation of (v_{i_j}, v_{i_l}) is approved and the algorithm continues to try merging different firings. Otherwise, the algorithm will undo the last merging operation and move forward by picking up two new firings for merging. The algorithm iterates until no possible merges can be done. Reaching that stage, it generates a new small size compact HSDF graph G_m that reduces the analysis time, as later shown experientially in Section 5.7.

4.3.4 Complexity Analysis

In this section, we provide a complexity analysis for the slack-based merging algorithm, previously presented in Algorithm 2. The algorithm starts by calculating earliest start time ϑ_{i_j} and latest finish time θ_{i_j} of all firings, each having a complexity of $O(|V_h|+|E_h|)$, since they are based on a Breadth First Search (BFS) [Lynch, 1996]. Then, it continues with the calculation of the slack σ_{i_j}, which has a complexity of $O(|V_h|)$. The next part of the algorithm is a loop (**foreach** statement) that runs $|V_h|$ times (in the worst case) and contains earliest start time ϑ_{i_j}, latest finish time θ_{i_j} and slack σ_{i_j} calculations, with the previously stated complexities. Therefore, the complexity of the loop is equivalent to $O(|V_h| \cdot ((|V_h|+|E_h|)+(|V_h|+|E_h|)+(|V_h|))) = O(3|V_h|^2+2|V_h||E_h|)$. Hence, the final complexity of the *slack-based merging* algorithm is $O(|V_h|^2+|V_h||E_h|)$, which is polynomial and depends on both $|V_h|$ and $|E_h|$.

4.3.5 Example

In this section, we present an example that illustrates how to apply the *slack-based merging* algorithm on an SDF/HSDF graph, shown in Figure 4.1, until reaching the reduced-size HSDF graph G_m, shown in Figure 4.5(c). Here, we demonstrate the algorithm for a single iteration for brevity, as it is a repeated process that takes several iterations to reach the final output graph G_m. The following paragraphs explains this in detail.

Consider the SDF graph and its HSDF representation shown in Figure 4.1. Let us assume all

Table 4.1: SDF^3 benchmark applications.

Application	**Number of actors**	**Number of channels**	
		Infinite Buffer	**Finite buffer**
h263decoder	1190	2378	4160
h263encoder	201	399	785
modem	48	109	170
samplerate	612	1633	2654
satellite	4515	11619	18723
mp3playback	10000	32237	32237

the execution times of all actors are equal to 1, the throughput requirement is $\zeta = 1/3$, and the end-to-end latency constraint is $\mathcal{D} = 8$. The period T of this graph is equal to 3 and the total execution time of its CP $(v_{a_0}, v_{b_0}, v_{b_1}, v_{b_2}, v_{c_2}, v_{d_0})$ is equal to 6. Calculating $\langle \vartheta_{i_j}, \theta_{i_j}, \sigma_{i_j} \rangle$ for every firing v_{i_j} in the graph results in $v_{a_0} = \langle 0,3,2 \rangle$, $v_{b_0} = \langle 1,4,2 \rangle$, $v_{b_1} = \langle 2,5,2 \rangle$, $v_{b_2} = \langle 3,6,2 \rangle$, $v_{c_0} = \langle 2,7,4 \rangle$, $v_{c_1} = \langle 3,7,3 \rangle$, $v_{c_2} = \langle 4,7,2 \rangle$, $v_{d_0} = \langle 5,8,2 \rangle$. As we see, every firing v_{i_j} has non-negative slack σ_{i_j}, which allows going forward with the merging process. From Figure 4.1(a), we can get the topologically ordered set $\hat{V} = \{v_a, v_b, v_c, v_d\}$. The algorithm will skip actor v_a and move on to actor v_b, because v_a consists of a single firing v_{a_0}. It picks up the two firings (v_{b_0}, v_{b_1}), because they have non-negative slack that satisfy the two conditions $\sigma_{b_0} \geq C_b$ and $\sigma_{b_1} \geq C_b$. Then, it merges them into a single cluster $\mathcal{V}_{b_0,b_1}$ with execution time $C_{b_0,b_1} = 2$, as shown in Figure 4.5(a). This merging operation is a valid merge, because it satisfies the throughput ζ and the end-to-end latency $\mathcal{D}$ constraints defined by Equations (4.7) and (4.8), respectively. The throughput constraint ζ is satisfied, because the total execution time of the maximum cycle in the graph $(\mathcal{V}_{b_0,b_1}, v_{b_2})$ is equal to 3, which means that ζ_m of the resulting graph, shown in Figure 4.5(a), did not change ($\zeta_m = 1/3$). Also, the end-to-end latency $\mathcal{D}$ constraint is satisfied, because the total execution time of the CP of the resulting graph did not change (equal to 6). Then, the algorithm recalculates $\langle \vartheta_{i_j}, \theta_{i_j}, \sigma_{i_j} \rangle$ for every firing v_{i_j} and repeats the process again. Figure 4.5(b) shows the output of an intermediate step of the merging algorithm, while Figure 4.5(c) shows the final output HSDF graph G_m of the merging algorithm.

The final output HSDF graph G_m consists of four actors $(v_{a_0}, \mathcal{V}_{b_0,b_1,b_2}, \mathcal{V}_{c_0,c_1,c_2}, v_{d_0})$ with execution times $(1,3,3,1)$, respectively. Its throughput ζ_m is equal to $1/3$, while the total execution time of its CP $(v_{a_0}, \mathcal{V}_{b_0,b_1,b_2}, \mathcal{V}_{c_0,c_1,c_2}, v_{d_0})$ is equal to 8. Therefore, G_m satisfies the throughput ζ and the end-to-end latency $\mathcal{D}$ constraints of the original SDF/HSDF graph. As we see, G_m has a single path $(v_{a_0}, \mathcal{V}_{b_0,b_1,b_2}, \mathcal{V}_{c_0,c_1,c_2}, v_{d_0})$ compared to the original HSDF graph, shown in Figure 4.1(b). This speeds up the timing parameter extraction process since it depends on the number of paths exists in the graph. We later demonstrate this experimentally in Section 5.7.

4.4 Experiments

In this section, we evaluate the slack-based merging algorithm using SDF applications from the SDF^3 benchmarks [Stuijk et al., 2006]. Table 4.1 shows the size of these benchmark applications

Table 4.2: Run-time (seconds) of the algorithm.

Application	**Run-time (sec)**		
	Infinite Buffer Sizes	**Minimum Buffer Sizes**	
		ζ_{max}	ζ
h263decoder	264	495	11824
h263encoder	0.55	8.9	11.13
modem	0.215	0.47	0.65
samplerate	38	51	53
satellite	14390	20917	26334
mp3playback	5 (days)	∞	∞

after transforming them into HSDF graphs. The main goal is to evaluate its run-time with SDF graphs of different sizes, but also to show the impact of different buffer sizes on the performance of the slack-based merging algorithm. To illustrate its impact, we consider three values of buffer sizes. First, is the infinite buffer sizes, assuming the availability of infinite resources. Second and third, are the minimum buffer sizes, but at two different execution throughputs of the SDF graph, which are the maximum throughput ζ_{max} and the throughput constraint ζ of the application. The latency constraint $\mathcal{D}$ for the input applications is set to the inverse of their throughput constraint, $\mathcal{D} = 1/\zeta$. This choice is made to provide enough slack for the applications while we study the effect of changing other parameters, i.e., throughput and buffer sizes, as shown in the experiment.

Tables 4.2 and 4.3 show the summary of the results. In most cases, the algorithm succeeds in generating a reduced-size graph in reasonable time. However, for some cases, e.g. *mp3playback*, the run-time varies from seconds to days depending on the complexity of the graph. This result is in-line with our expectations because the original graph is huge and consists of 10000 firings. The algorithm achieves large reduction rates of the original HSDF graph, as shown in Table 4.3, ranging from 50% in case of *mp3playback* up to 99.7% (approximately) in case of *h263decoder*, in case of infinite buffers. In case of finite buffers, the reduction rates are less compared to infinite case. It ranges from 35.4% up to 94.5% (approximately) depending on the buffer sizes and the throughput constraint. Also, we notice that the slack-based merging algorithm's run-time and output graph size have an inverse relation with the buffer size of the application. The reason is that small buffer sizes add extra dependencies in the graph that prevent further merging and makes the algorithm spend more time exploring every combination of actors that could be merged. The ∞ and N/A entries imply that the merging algorithm spend unreasonable time ($>$ 1 week) without generating any output.

From these results, we can conclude that the slack-based merging algorithm typically succeeds in achieving large reduction rates in the size of the output graphs. This result reflects positively on the timing parameter extraction (TPE) algorithm, as shown in the experiments of Chapter 5.

Table 4.3: Number of actors before and after merging.

Application	**Number of actors**			
	Before Merge	**After Merge**		
		Infinite Buffer Sizes (red. %)	**Minimum Buffer Sizes**	
			ζ_{max} **(red. %)**	ζ **(red. %)**
h263decoder	1190	4 (99.7%)	71 (94.0%)	300 (74.8%)
h263encoder	201	5 (97.5%)	11 (94.5%)	181 (10.0%)
modem	48	16 (66.7%)	31 (35.4%)	31 (35.4%)
samplerate	612	6 (99.0%)	127 (79.2%)	263 (57.0%)
satellite	4515	22 (99.5%)	988 (78.1%)	1972 (56.3%)
mp3playback	10000	5000 (50.0%)	N/A	N/A

4.5 Summary

In this chapter, we presented a new heuristic reduction algorithm for synchronous dataflow graphs called slack-based merging. The proposed algorithm generates reduced-size HSDF graphs that satisfy the throughput and latency constraints of the original application graph. This helps in speeding up the process of timing parameter extraction and finding a feasible real-time schedule, thereby reducing the overall design time of the real-time system, as we later show experimentally in the next Chapters 5 and 6. The slack-based merging algorithm uses two main concepts: **a)** the difference between the WCET of the SDF graph's firings and its timing constraints (slack) to merge firings together and generate a reduced-size HSDF graph, and **b)** the novel concept of merging called safe merge, which is a merge operation that we prove cannot cause a live HSDF graph to deadlock. Experimental results with real application models from the SDF^3 benchmark show that the reduced graph: **1)** respects the throughput and latency constraints of the original application graph and **2)** when the throughput constraint is relaxed with respect to the maximal throughput of the graph, the merging algorithm is able to achieve a larger reduction in graph size.

Chapter 5

Timing Parameter Extraction

The previous chapter presented the first stage of our solution to integrate mixed application models with timing constraints coexisting on the same multi-core platform. That stage was a graph reduction technique called *slack-based merging* that aims to reduce the complexity of dataflow applications by generating reduced-size HSDF graphs, avoiding possible large HSDF graphs generated from traditional conversion methods [Sriram and Bhattacharyya, 2000]. In this chapter, we present the second stage of our solution called Timing Parameters Extraction (TPE) [Ali et al., 2015]. This algorithm extracts the timing parameters (offsets, deadlines and periods) of cyclic Homogeneous Synchronous Dataflow (HSDF) graphs with periodic sources (the output of the first stage, Chapter 4) transforming them into real-time periodic tasks. This creates a unified model for all applications running on the multi-core platform, where traditional real-time analysis and scheduling techniques can be applied assuring real-time guarantees for the complete system.

This chapter starts by explaining key concepts, definitions and techniques that pave the way for understanding the methodology of timing parameter extraction of HSDF graphs. First, we present existing *deadline assignment strategies for pipelines* (Section 5.2), which we extend to be applicable on Directed Cyclic Graphs (DCG). Second, we define the *path sensitivity* (Section 5.3) concept that determines the order in which TPE traverses paths in the graph to extract timing parameters. Last, we propose a methodology for *deriving latency constraints* (Section 5.4) in case of its absence from the graph timing properties to help in the TPE process. After this essential overview, we explain in detail the TPE algorithm in Section 5.5. Then, we prove its correctness in Section 5.6, followed by the experimental evaluation of the TPE algorithm in Section 5.7. Finally, we conclude the chapter with a summary in Section 5.8.

5.1 Preliminaries

The TPE algorithm, presented in this chapter, works with HSDF graphs that have a single or multiple latency constraints. This shows the ability of the TPE algorithm to deal with a more complex model than the one proposed in this thesis (Section 3.4), which states a single end-to-end latency constraint $\mathcal{D}$ for each HSDF graph. The multiple latency constraints are defined as actor-to-actor deadlines (maximum timing constraints) between firings in the same iteration of any two actors, v_x and v_y, that have a single or multiple route(s) between them in the HSDF graph. In contrast, $\mathcal{D}$ is a latency constraint between an input and output actor only. Due to this, this chapter refers to latency constraints with the symbol $\mathcal{D}_{xy}$, where x and y are indices that refer to the actors in the HSDF graph that are governed by the $\mathcal{D}_{xy}$ constraint. Therefore, a *time-constrained path* P is defined as any route between two actors v_x and v_y that has a latency constraint $\mathcal{D}_{xy}$. Fundamentally, P and $\mathcal{D}_{xy}$ are defined by the same Equations (3.13) and (3.14), but with two significant differences. First, actors v_x and v_y refer to any two actors in the HSDF graph (not only to input and output actors). Second, the end-to-end latency constraint $\mathcal{D}$ is substituted by the actor-to-actor latency constraint $\mathcal{D}_{xy}$ in Equation (3.14). Therefore, for the TPE algorithm Equations (3.13) and (3.14) will be defined as:

$$P = \{\langle v_x, \ldots, v_y \rangle : v \subseteq V_h\} \tag{5.1}$$

where, v_x and v_y refer to two actors in the HSDF graph defining the beginning and the end of the path P, respectively. The latency constraint of P is defined as:

$$\mathcal{D}_{xy} \geq \sum_{\substack{i=x,\\ \forall v_i \in CP}}^{y} C_i \tag{5.2}$$

In conclusion, the ability to allow representing multiple latency constraints that may be required by some applications, extends the generality of the TPE algorithm, allowing it to be applied on wider range of applications represented as HSDF graphs.

5.2 Deadline Assignment Strategies for Pipelines

The problem of assigning individual deadlines to dependent tasks of a *pipeline application* A_p, represented by the graph $G_p = \langle V_p, E_p \rangle$, distributed on multiple processors using its end-to-end deadline has been addressed in previous research [Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997, Lipari and Bini, 2011]. The pipeline application consists of a set of tasks (actors) V_p that execute in sequence. The application has a latency constraint $\mathcal{D}_{xy}$ that represents the end-to-end deadline of A_p, where v_x and v_y is the start and end task of A_p, respectively. Therefore, the pipeline application graph G_p contains a single time-constrained path P with a latency constraint $\mathcal{D}_{xy}$. The proposed TPE algorithm supports two well-known deadline assignment methods for pipelines, referred to as NORM and PURE. These are detailed next.

5.2.1 The NORM Method

The NORM method [Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997] is an assignment strategy to divide the end-to-end deadline $\mathcal{D}_{xy}$ of a pipeline proportionally to the computation time of its tasks. Therefore, the individual deadline of a task in a pipeline D_i is computed as follows:

$$D_i = \frac{C_i}{\sum_{\forall v_j \in P} C_j} \cdot \mathcal{D}_{xy} \tag{5.3}$$

From Equation (5.3), the NORM method assigns individual deadlines D_i to tasks with the same end-to-end deadline $\mathcal{D}_{xy}$, such that all tasks have equal densities ρ_i.

$$\rho_i = \frac{C_i}{D_i} = \frac{\sum_{\forall v_j \in P} C_j}{\mathcal{D}_{xy}} \tag{5.4}$$

5.2.2 The PURE Method

The PURE method [Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997] is a different deadline assignment strategy based on the distribution of the laxity ε equally among all tasks of the pipeline, such that each task have slack δ. The laxity ε on the time-constrained path P is defined as follows:

$$\varepsilon = \mathcal{D}_{xy} - \sum_{\forall v_j \in P} C_j \tag{5.5}$$

Then, the slack δ of the tasks is equal to:

$$\delta = \frac{\varepsilon}{|V_p|} \tag{5.6}$$

where $|V_p|$ is the number of tasks in the pipeline. Therefore, the individual deadline of a task in a pipeline D_i is computed as follows:

$$D_i = C_i + \delta \tag{5.7}$$

Therefore,

$$D_i = C_i + \frac{\mathcal{D}_{xy} - \sum_{\forall v_j \in P} C_j}{|V_p|} \tag{5.8}$$

From Equation (5.7), the PURE method assigns individual deadlines D_i, such that tasks have relative densities ρ_i. This means, a task with high C_i have high ρ_i relative to a task with small C_i.

$$\rho_i = \frac{C_i}{D_i} = \frac{C_i}{C_i + \delta} \tag{5.9}$$

5.3 Path Sensitivity

In this section, we define a key concept in our algorithm called *path sensitivity*, that enables supporting general HSDF graphs, as opposed to being limited to pipelines. Dealing with actors in general graphs implies that an actor can be present on multiple time-constrained paths of the graph. The path sensitivity parameter helps in addressing this problem by determining the order in which to consider the time-constrained paths when extracting the timing parameters.

Definition 5.1 (***Path sensitivity*** γ**).** *Path sensitivity is a measure of the criticality of a time-constrained path with respect to density. It is calculated as follows:*

$$\gamma = \sum_{\forall v_j \in P} \frac{C_j}{\mathcal{D}_{xy}} \tag{5.10}$$

The density is the measure of how tight the latency constraint $\mathcal{D}_{xy}$ is for a time-constrained path P compared to its execution time. γ is in the range $(0, 1]$ (because of the relation in Equation (3.14)), where higher values indicate higher sensitivity. In case of NORM, substituting Equation (5.10) in Equation (5.3) gives:

$$D_i = \frac{C_i}{\gamma} \tag{5.11}$$

by solving for γ and substituting Equation (5.4) in Equation (5.11)

$$\rho_i = \gamma \tag{5.12}$$

This means that all tasks τ_i on the same time-constrained path P have densities ρ_i equal to the path sensitivity γ.

In case of PURE, substituting Equation (5.10) in Equation (5.8) gives:

$$D_i = C_i + \delta = C_i + \frac{(1-\gamma)\cdot\mathcal{D}_{xy}}{|P|} \tag{5.13}$$

by dividing Equation (5.13) by D_i, then substituting by Equation (5.4) and solving for ρ_i

$$\rho_i = 1 - \frac{\delta}{D_i} = 1 - \frac{(1-\gamma)\cdot\mathcal{D}_{xy}}{|P|\cdot D_i} \tag{5.14}$$

From Equations (5.11), (5.12), (5.13) and (5.14), we can draw two conclusions. First, *there is an inverse relation between the path sensitivity γ and the task relative deadline D_i for both NORM and PURE*. This conclusion is obvious from Equation (5.11). In case of Equation (5.13), since $0 < \gamma \leq 1$, an increase in the value of γ decreases the value of D_i and vice versa, confirming the inverse relation. Second, *when the sensitivity γ of a time-constrained path increases, the value of its task densities ρ_i increases too*. This is confirmed from Equations (5.12) and (5.14) and the first conclusion.

5.4 Deriving Latency Constraints

In this section, we present two techniques for deriving latency constraints for HSDF graphs. First, we derive latency constraints for cyclic paths. We then derive end-to-end latency constraints in case it is not specified by the application.

5.4.1 Deriving Constraints for Cyclic Paths

HSDF applications can have several cycles in its graph. Each cycle requires a latency constraint that satisfies the throughput requirement ζ_i of the application. A quick choice for a cycle latency constraint $\mathcal{D}_{xy}^{cycle}$ value is the period of the application A_i. However, such a choice of latency constraint ignores the number of tokens d involved in the cycle and limits possible pipeline parallelism in the application. Therefore, the latency constraint of a cyclic time-constrained path $\mathcal{D}_{xy}^{cycle}$ must take into account the number of tokens involved in this cycle d_{cycle} such that the application throughput ζ_i is not violated. The latency constraint for a cyclic time-constrained path is defined as follows [Moreira et al., 2007]:

$$\mathcal{D}_{xy}^{cycle} = C_{cycle} + (\frac{1}{\zeta_i} - \frac{C_{cycle}}{d_{cycle}}) \cdot d_{cycle} = \frac{d_{cycle}}{\zeta_i} \tag{5.15}$$

where C_{cycle} is the summation of execution times of the actors involved in the cycle. The latency constraint of a cycle tells us how much the execution of the actors on the cycle as a whole can be extended while still guaranteeing the desired application throughput ζ_i.

5.4.2 Deriving End-to-End Latency Constraint

Our proposed algorithm requires an end-to-end latency constraint for each HSDF application to satisfy the precedence constraints and the throughput requirement. In case of an HSDF application without a specified end-to-end latency constraint $\mathcal{D}_{xy}$, we derive it as follows:

$$\mathcal{D}_{xy} = \max\{T_i, \beta \cdot \sum_{\forall v_i \in CP} C_i\} \tag{5.16}$$

As we can notice $\mathcal{D}_{xy}$ is set to the maximum of two values. The first, the application period T_i, which is extracted from the inverse of its throughput requirement ζ_i, $T_i = 1/\zeta_i$. The second, is the sum of the C_i of actors in the critical path (CP) of the application multiplied by a constant β, where the CP of an application is defined as its longest execution path from input to output, as defined in Section 3.2.2.

The β constant has a value that ranges $[1, \infty)$. Selecting $\beta = 1$ results in unnecessarily tight actor deadline values and increases the total density of the application that makes it more critical and hard to schedule with other applications, since the actors in the application CP have $\rho_i = 1$. On the other hand, selecting higher values of β relaxes the criticality of the application and eases its schedulability with other applications. A good value for β that we use in this thesis is when the sensitivity of the CP of the application γ_{CP} is equal to the maximum sensitivity of all the cycles γ_{cycle} in the application,

$$\max_{\forall cycle \in G}\{\gamma_{cycle}\} = \gamma_{CP} = \sum_{\forall v_j \in CP} \frac{C_j}{\mathcal{D}_{xy}} = \frac{\sum_{\forall v_j \in CP} C_j}{\beta \cdot \sum_{\forall v_j \in CP} C_j} \tag{5.17}$$

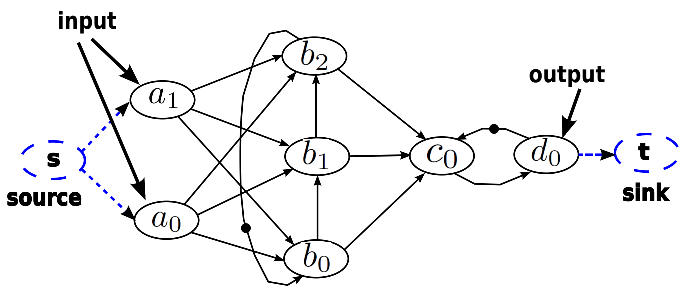


Figure 5.1: HSDF graph after adding source s and sink t.

At this value of β, the individual deadlines of actors participating in cycles and time-constrained paths governed by the derived end-to-end latency constraint in the application graph can be extended to the maximum possible limit (latency constraint computed in Equation (5.15)), while still satisfying the throughput requirement ζ. Therefore, solving for β in Equation (5.17) defines it as:

$$\beta = \frac{1}{\max_{\forall cycle \in G}\{\gamma_{cycle}\}} \tag{5.18}$$

The derived end-to-end latency constraint $\mathcal{D}_{xy}$, shown in Equation (5.16), is considered a lower bound using β computed in Equation (5.18). Choosing a larger value will not affect the throughput requirement of the application, but it increases the schedulability of the application. However, it also delays the first output by a latency equal to the chosen value of the end-to-end latency constraint $\mathcal{D}_{xy}$. It is up to the system designer to choose a different larger value of β than Equation (5.18) if it suits the system.

5.5 Timing Parameters Extraction Algorithm

The algorithm presented in this section is intended for extracting the timing parameters (a_i, C_i, T_i, D_i) of HSDF applications with periodic sources. It is divided into two phases. The first phase finds all time-constrained paths in the graph, while second phase extracts the timing parameters of individual actors. The following sections explain these two phases in detail.

5.5.1 First phase: Finding All Time-Constrained Paths

In this phase, we calculate all time-constrained paths for a given HSDF in non-increasing order of sensitivities. A time-constrained path in an HSDF can be between any two actors that have a latency constraint. The first phase of the algorithm is divided into the following two stages, creation of source and sink actors and path enumeration. These two stages are detailed next.

5.5.1.1 Creation of Source and Sink Actors

This technique have been used before in Section 4.2.2 to define the end-to-end path (Definition 4.8). Here, we use it again to easily traverse the graph G. First, we search the graph G to find all input (output) actors. Actors associated with the input (output) data stream are specified as

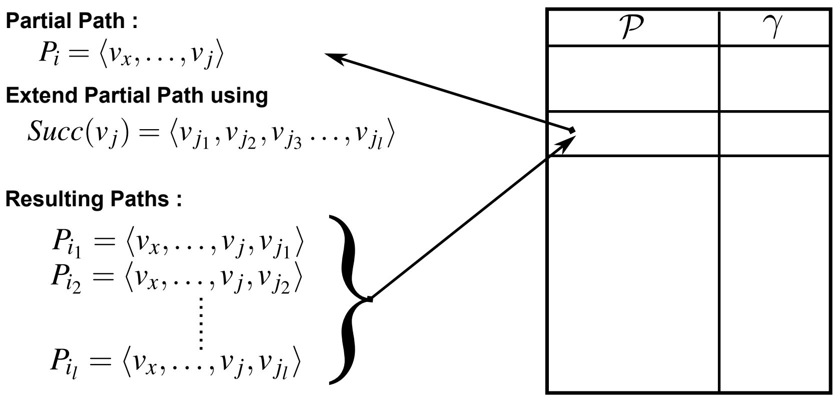

Partial Path :
$P_i = \langle v_x, \ldots, v_j \rangle$
Extend Partial Path using
$Succ(v_j) = \langle v_{j_1}, v_{j_2}, v_{j_3} \ldots, v_{j_l} \rangle$
Resulting Paths :
$P_{i_1} = \langle v_x, \ldots, v_j, v_{j_1} \rangle$
$P_{i_2} = \langle v_x, \ldots, v_j, v_{j_2} \rangle$
$P_{i_l} = \langle v_x, \ldots, v_j, v_{j_l} \rangle$
$\mathcal{P}$
γ

Figure 5.2: Enumeration of time-constrained paths.

the starting-actors (ending-actors), respectively. A dummy source s (sink t) actor that has a zero execution time is inserted at the beginning (end) of the graph G, as shown in Figure 5.1. These two actors (s, t) are connected with dummy links to starting and ending actors, respectively. Adding these dummy actors with their edges converts the graph into a canonical form, since all the paths that traverse the graph from the input to the output of the graph have a uniform form that starts with s and ends with t. This is helpful when traversing multi-input/multi-output graphs.

5.5.1.2 Path enumeration

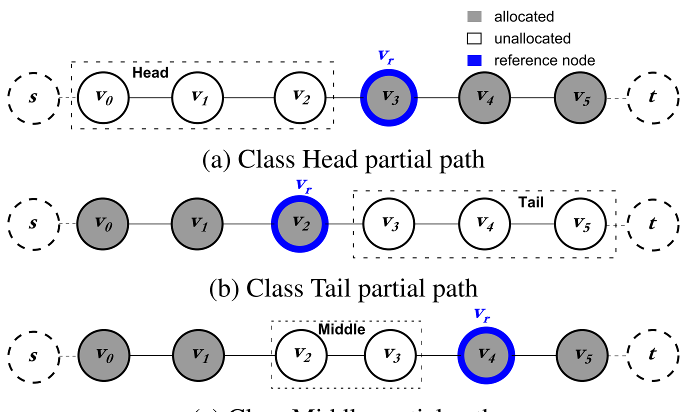
This is an iterative process where all time-constrained paths between source s and sink t actors in the HSDF are generated. In case of having latency constraints between two specific actors, the path enumeration phase generates all time-constrained paths between these two actors in addition to the ones generated from s to t. The set of all time-constrained paths between actors with latency constraints is called $\mathcal{P}$, which is arranged in non-increasing order of path sensitivities γ. It is defined as follows:

$$\mathcal{P} = \{\langle P_i, \gamma_i \rangle : V, \gamma_{i-1} \geq \gamma_i, \gamma \in (0,1]\} \tag{5.19}$$

The process starts by initializing $\mathcal{P}$ with a few *partial paths*. In this case, these initial partial paths are all single hop paths generated by combining the start actors with the elements in their list of successor actors. The list of successor actors is a set of child actors that are one hop away from their parent. For example, in Figure 5.1, the list of successor actors for source actor s is $Succ(v_s) = (a_0, a_1)$. The starting actors can be the source actor s or any actor that starts a set of time-constrained paths v_x with a specific latency constraint $\mathcal{D}_{xy}$. The list of successor actors $Succ(v_x)$ is defined as follows:

$$Succ(v_x) = (v_{x_1}, v_{x_2}, v_{x_3}, \ldots, v_{x_l}) \tag{5.20}$$

where l is the number of actors in $Succ(v_x)$. Then, the process picks up a partial path $P_i = \langle v_x, \ldots, v_j \rangle$ from $\mathcal{P}$, where v_j is not equal to the end actor v_y, and extends it to a full path (Equation (3.13)), as shown in Figure 5.2. The extension process starts by getting the $Succ(v_j) =$



(a) Class Head partial path

(b) Class Tail partial path

(c) Class Middle partial path

Figure 5.3: Partial path classes for offsets setting

$(v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_l})$. Then, it extends the partial path P_i to its l possible extended paths, $P_{i_1} = \langle v_x, \ldots, v_j, v_{j_1} \rangle, P_{i_2} = \langle v_x, \ldots, v_j, v_{j_2} \rangle, \ldots, P_{i_l} = \langle v_x, \ldots, v_j, v_{j_l} \rangle$. It then removes P_i and inserts its l possible continuations in $\mathcal{P}$ in non-increasing order of sensitivity. The path enumeration process continues until all partial paths in $\mathcal{P}$ are extended to full time-constrained paths.

5.5.2 Second phase: Extracting Timing Parameters

The second phase, shown in Algorithm 3, repeats for each application in the application set A. It picks a time-constrained path P_i in order of sensitivity from $\mathcal{P}$. The selected path P_i is checked whether or not it has actors v_j with assigned deadlines D_j. If P_i has no actors with assigned deadlines ($\forall v_j \in P_i$), the algorithm assigns individual deadlines D_j for the actors v_j using *dead_assign()* function that implements either NORM or PURE (Equations (5.3) or (5.8), respectively), using the corresponding latency constraint $\mathcal{D}^i_{xy}$.

On the other hand, if P_i has a set of actors with assigned deadlines X_i (shared actors v_k with any previously processed time-constrained paths), the algorithm assigns individual deadlines D_j to the unassigned actors v_j using either NORM or PURE based on the corresponding latency constraint, which is the difference between $\mathcal{D}^i_{xy}$ and the sum of individual deadlines D_k already assigned to actors, $(\mathcal{D}^i_{xy} - \sum_{\forall v_k \in X_i} D_k)$. In all cases, the period of the actor T_j is derived from the throughput constraint ζ_A of the application. It is defined as follows:

$$T_j = 1/\zeta_A \tag{5.21}$$

This follows naturally for an HSDF graph, since each actor executes only once per iteration by definition.

Once the application A_i actors relative deadline are determined, the offset of the actors a_j are calculated in a similar fashion. Algorithm 3 generates a new set $\hat{\mathcal{P}} \subseteq \mathcal{P}$ containing time-constrained paths that include s and t actors only. $\hat{\mathcal{P}}$ is arranged in a non-increasing order of $\mathcal{D}_{xy}$. If two paths have the same $\mathcal{D}_{xy}$, they are ordered in a non-increasing order of γ. The algorithm picks a time-constrained path P_i from $\hat{\mathcal{P}}$. If the path has no actors with assigned offsets, it assigns

Algorithm 3: Extracting timing parameters of HSDF

P_i: A full time-constrained path in $\mathcal{P}$ set.
$\mathcal{D}^i_{xy}$: deadline constraint between actor v_x and actor v_y on a full time-constrained path P_i.
$\mathcal{P}$: totally ordered set of all time-constrained paths of an application ordered according to γ, $\mathcal{P} = \{P_i : \gamma_{i-1} \geq \gamma_i\}$.
$\hat{\mathcal{P}}$: totally ordered set of time-constrained paths from s to t of an application ordered according to $\mathcal{D}_{xy}$, $\hat{\mathcal{P}} \subseteq \mathcal{P}$,
$\hat{\mathcal{P}} = \{P_i : v_x = s, v_y = t, [\mathcal{D}^{i-1}_{xy} > \mathcal{D}^i_{xy} \ or \ \{\mathcal{D}^{i-1}_{xy} = \mathcal{D}^i_{xy}, \gamma_{i-1} \geq \gamma_i\}]\}$.
P^H_i: set of higher sensitivity time-constrained paths than P_i, $P^H_i = \{\langle P_1, \dots, P_{i-1}\rangle : \gamma_{i-1} \geq \gamma_i\}$
X_i: set of shared actors between P_i with higher sensitivity time-constrained paths set P^H_i, $X_i = \{v_k : v_k \in P_i, v_k \in P_j \in P^H_i\}$
p_i: partial path in time-constrained path P_i.

```
begin
    // Actor deadline assignment
    foreach $P_i$ in $\mathcal{P}$ do
        if ($\forall v_j \in P_i, D_j = \varnothing$) then
            foreach $v_j$ in $P_i$ do
                $D_j = dead\_assign(\mathcal{D}^i_{xy})$; // NORM/PURE
            end
        else // $X_i \subseteq P_i$
            foreach $v_j$ in $P_i - X_i$ do
                $D_j = dead\_assign(\mathcal{D}^i_{xy} - \sum_{\forall v_k \in X_i} D_k)$; // NORM/PURE
            end
        end
    end
    // Actor offset assignment
    foreach $P_i$ in $\hat{\mathcal{P}}$ do
        if ($\forall v_j \in P_i, a_j = \varnothing$) then
            $a_0 = 0$;
            foreach $v_j$ in $P_i$, $j = 1..sizeof(P_i)$ do
                $a_j = a_{j-1} + D_{j-1}$;
                $T_j = 1/\zeta_{A_i}$
            end
        else
            Determine all $p_i \in P_i$ with $a_j = \varnothing$.
            Determine reference actor $v_r$.
            foreach $p_i$ in $P_i$ do
                if ($p_i$ is Head or Middle) then
                    foreach $v_j$ in $p_i$ do
                        $v_r = v_{j+1}$;
                        $a_j = a_r - D_j$;
                        $T_j = 1/\zeta_A$;
                    end
                else
                    foreach $v_j$ in $p_i$ do
                        $v_r = v_{j-1}$;
                        $a_j = a_r + D_r$;
                        $T_j = 1/\zeta_A$;
                    end
                end
            end
        end
    end
    // Validation check
    foreach $P_i$ in $\mathcal{P}$ do
        if (($\sum_{\forall v_j \in P_i} D_j \leq \mathcal{D}^i_{xy}$) & ($a_y + D_y - a_x \leq \mathcal{D}^i_{xy}$) ) then
            Algorithm Succeeds;
        else
            Algorithm Fails;
        end
    end
end
```

offsets a_j for the actors v_j on the path in the direction from s to t as follows:

$$a_j = a_{j-1} + D_{j-1} \tag{5.22}$$

If time-constrained path P_i has a set of actors with assigned offsets (actors assigned in previously processed paths), the algorithm traverses P_i in search for partial path segments p_i of actors with unassigned offsets. Once they are listed, the algorithm determines the reference actors v_r and classify them into one of three types: *Head*, *Middle* or *Tail*, as shown in Figure 6.3. This information is used to calculate the offsets a_j, as shown in Algorithm 3. If the partial path p_i is of type *Head* or *Middle*, the reference actor v_r is always on the right hand side of p_i, as shown in Figures 6.3(a) and 6.3(c), and the offsets of p_i actors are assigned using the following equation:

$$a_j = a_r - D_j \tag{5.23}$$

After assigning the offset of the actor v_j, the reference actor v_r advances its position to the already offset assigned actor, preparing for the offset assignment of the next actor in the partial path p_i, as shown in Algorithm 3. Offset assignment of *Head* and *Middle* in this way instead of traversing the path from s to t assigning offsets using Equation (5.22), enables larger offset values to be assigned to actors delaying their execution allow satisfying wider range of latency constraints, as we show in Section 5.5.4.

If the partial path p_i is type *Tail*, the reference actor v_r is always on the left hand side of p_i, as shown in Figure 6.3(b), and the offsets of p_i actors are assigned using the following equation:

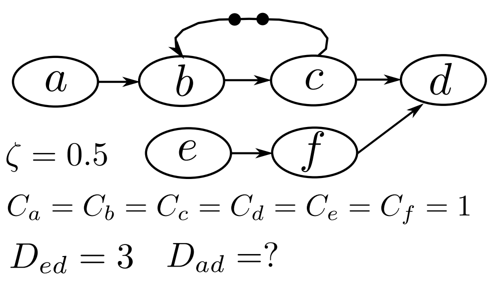
$$a_j = a_r + D_r \tag{5.24}$$

The reference actor v_r advances in the same way mentioned previously. After assigning deadline and offsets for the application actors, the algorithm checks the application for the validity of the assigned values and that they do not violate the latency constraints specified.

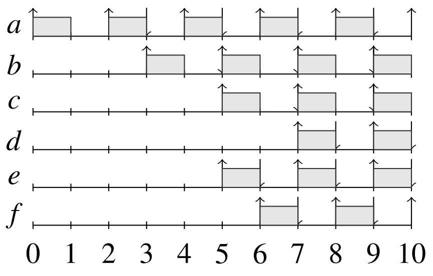
Finally, we can conclude that Algorithm 3 preserves relative deadline values D_j computed from high-sensitivity time-constrained paths. This is clear from determining the actors with unassigned deadlines in P_i, and their corresponding latency constraint ($\mathcal{D}^i_{xy} - \sum_{\forall v_k \in X_i} D_k$), leaving the preassigned set of actors X_i untouched. In case of using deadline-based schedulers, this property makes actors in high-sensitivity time-constrained paths have a higher priority compared to actors in low-sensitivity time-constrained paths, since they have tighter deadlines (as concluded from Equations (5.11) and (5.13)).

5.5.3 Complexity Analysis

In this section, we provide a complexity analysis for the TPE algorithm, previously presented in Section 5.5. The TPE algorithm consists of two phases. The first phase, detailed in Section 5.5.1, is concerned with finding the set of all time-constrained paths $\mathcal{P}$, which have a complexity of $O(|V_h| + |E_h|)$, since it is based on a Breadth First Search (BFS) [Lynch, 1996]. The



(a) HSDF application.



(b) HSDF timing diagram.

	a_i	C_i	T_i	D_i
a	0	1	2	3
b	3	1	2	2
c	5	1	2	2
d	7	1	2	1
e	5	1	2	1
f	6	1	2	1

(c) Actors' timing parameters.

Figure 5.4: HSDF example.

second phase, detailed in Section 5.5.2, is concerned with extraction of timing parameters, as shown in Algorithm 3. It is composed of three main parts: **1)** actor deadline assignment, **2)** actor offset assignment and **3)** validation check. Each part is represented by a loop (**foreach** statement) that runs $|\mathcal{P}|$ times (in the worst case). The actor deadline and offset assignment parts contains inside loops that run $|V_h|$ times. Consequently, the complexity of the second phase is equivalent to $O((|\mathcal{P}| \cdot |V_h|) + (|\mathcal{P}| \cdot |V_h|) + |\mathcal{P}|) = O(2|\mathcal{P}||V_h| + |\mathcal{P}|)$. In conclusion, the total complexity of the TPE algorithm is the sum of its two phases $O(|V_h| + |E_h| + (|\mathcal{P}| \cdot |V_h|) + (|\mathcal{P}| \cdot |V_h|) + |\mathcal{P}|) = O(2|\mathcal{P}||V_h| + |\mathcal{P}| + |V_h| + |E_h|)$. Hence, the final complexity of the TPE algorithm is $O(|\mathcal{P}| + |\mathcal{P}||V_h| + |V_h| + |E_h|)$, which is polynomial and depends on $|\mathcal{P}|$, $|V_h|$ and $|E_h|$.

5.5.4 Example

In this section, we present an example, illustrated in Figure 5.4, to demonstrate our proposed algorithm step-by-step. The following paragraphs explains this in detail.

Figure 5.4(a) shows an HSDF graph application comprising six actors (a,b,c,d,e,f) with execution times of all actors equal to 1, throughput requirement $\zeta = 0.5$, and two end-to-end latency constraints, one is specified $\mathcal{D}_{ed} = 3$, while the other $\mathcal{D}_{ad}$ is not. The example HSDF graph is not trivial, as it features multiple input actors a and e, a cycle, and multiple initial tokens. Applying the first phase of our proposed algorithm results in three time-constrained paths. The first time-constrained path is $P_1 = \langle e,f,d \rangle$ with an end-to-end latency constraint $\mathcal{D}^1_{ed} = 3$ and sensitivity $\gamma_1 = 1$. The second time-constrained path is $P_2 = \langle b,c \rangle$, which represents a cycle in the graph with a latency constraint $\mathcal{D}^2_{bc} = 4$ calculated by substituting with $C_{cycle} = C_b + C_c = 2$, $\zeta = 0.5$ and number of tokens in the cycle $d = 2$ in Equation (5.15). The sensitivity of P_2 is hence $\gamma_2 = 0.5$ (Equation (5.10)). The third time-constrained path is $P_3 = \langle a,b,c,d \rangle$ with a latency constraint $\mathcal{D}^3_{ad}$ equal to the second end-to-end deadline, which is not specified by the application. Therefore, we calculate $\mathcal{D}^3_{ad}$ using Equation (5.16) ($\beta = 1/\gamma_2 = 2$, Equation (5.18)) that results in $\mathcal{D}^3_{ad} = 8$ and its sensitivity is $\gamma_3 = 0.5$. Therefore, the set of all possible time-constrained paths is $\mathcal{P} = \{\langle P_1, \gamma_1\rangle, \langle P_2, \gamma_2\rangle \langle P_3, \gamma_3\rangle\} = \{\langle (e, f, d), 1\rangle, \langle (b, c), 0.5\rangle, \langle (a, b, c, d), 0.5\rangle\}$.

The second phase of the proposed algorithm picks up P_1 and assigns individual deadlines to actors (e, f, d) equal to ($D_e = 1, D_f = 1, D_d = 1,$), respectively. Picking up the next time-constrained path P_2 for deadline assignment results in ($D_b = 2, D_c = 2$). Finally, picking up the last time-constrained path P_3 for deadline assignment results in ($D_a = 3$). The individual deadline values calculated are the same for both NORM and PURE.

For offset assignment, the algorithm creates the set of time-constrained paths that goes from source s to sink t, ordered according to the constraint $[\mathcal{D}^{i-1}_{xy} > \mathcal{D}^{i}_{xy}\ or\ \{\mathcal{D}^{i-1}_{xy} = \mathcal{D}^{i}_{xy}, \gamma_{i-1} \geq \gamma_i\}]$, $\hat{\mathcal{P}} = \{\langle P_3, \mathcal{D}^3_{ad}\rangle, \langle P_1, \mathcal{D}^1_{ed}\rangle\}$. First, it picks the time-constrained path with the longest end-to-end delay P_3 for offset assignment. Since none of its actors have assigned offsets, the actor offsets are ($a_a = 0, a_b = 3, a_c = 5, a_d = 7$). Then, it picks P_1 where one of its actors d has already assigned offset a_d equal to 7. It discovers a single partial path of type *Head* in P_1 which is $p_1 = (e, f)$. The reference actor for p_1 is actor d. Therefore, the offsets of actors e and f are ($a_e = 5, a_f = 6$), respectively. As noted, actor e is triggered at time ($a_e = 5$) even though its input data is available from time instance zero to satisfy the latency constraint ($\mathcal{D}_{ed} = 3$) of the application. For the periods, ($T_a = T_b = T_c = T_d = T_e = T_f = 1/\zeta = 2$). Therefore, the extracted timing parameters (a_i, C_i, T_i, D_i) for the graph actors $\{a, b, c, d, e, f\}$ are $\{(0, 1, 2, 3), (3, 1, 2, 2), (5, 1, 2, 2), (7, 1, 2, 1), (5, 1, 2, 1), (6, 1, 2, 1)\}$, respectively. These extracted parameters, shown in Figure 5.4(c), preserve the precedence, throughput and latency constraints of the HSDF application, indicated in the timing diagram in Figure 5.4(b). The timing diagram also shows that multiple iterations of the graph execute in parallel assuming at least three processors are available.

5.6 Validation of the TPE algorithm

This section validates the proposed algorithm by proving that it assigns individual deadlines for actors of any application graph such that it respects all its latency constraints. First, we start by the following property driven from the inverse relationship between path sensitivity γ and actor relative deadline D_v (concluded from Equations (5.11) and (5.13)):

Property 5.1. *If there are two time-constrained paths P_i and P_j, where $\gamma_i > \gamma_j$ and there is a shared actor v between them. The deadline value D^i_v computed for actor v on P_i is less than the value D^j_v computed for the same actor on P_j, $D^i_v < D^j_v$.*

Another important property of the deadline assignment strategies NORM and PURE, derived from Equations (5.3) and (5.8) is:

Property 5.2. *A time-constrained path P with a latency constraint $\mathcal{D}_{xy}$, whose actors v_j are assigned individual deadlines D_j, using NORM or PURE, has the following property:*

$$\mathcal{D}_{xy} = \sum_{\forall v_j \in P} D_j \tag{5.25}$$

From Property 5.2, it follows that applying Algorithm 3 on any time-constrained path P, whose actors has no assigned deadlines, results in a time-constrained path that satisfies its latency constraints. This is for the simple case where the actors in P has no assigned deadlines. However, when P shares some actors with higher sensitivity time-constrained paths the situation gets more complex. Lemma 5.1 proves the correctness of this case.

Lemma 5.1. *If a time-constrained path P_i with a latency constraint $\mathcal{D}^i_{xy}$, has a set of actors X_i shared with higher sensitivity time-constrained paths $P^H_i = \langle P_1, \ldots, P_{i-1} \rangle$ in an application graph G, Algorithm 3 assures that the sum of individual deadlines D_j of actors in P_i is equal to $\mathcal{D}^i_{xy} = \sum_{\forall v_j \in P_i} D_j$.*

Proof. Let us assume a time-constrained path $P'_i = P_i$, except that all its actors v'_j have $D'_j = \varnothing$ (empty element). Assigning individual deadlines D'_j to the actors of time-constrained path P'_i using either NORM or PURE (Equations (5.3) and (5.8)) and its latency constraint $\mathcal{D}^i_{xy}$ under the system model constraint specified in Equation (3.14) then

$$\forall v'_j \in P'_i,\ \ D'_j \geq C_j,\ \ \mathcal{D}^i_{xy} = \sum_{\forall v'_j \in P'_i} D'_j \tag{5.26}$$

The set of shared actors X_i in P_i has a sum of individual deadlines equal to κ.

$$\kappa = \sum_{\forall v_j \in X_i} D_j,\ \ \forall v_j \in X_i, D_j \geq C_j \tag{5.27}$$

Here, κ represents the value calculated from the higher sensitivity time-constrained paths P^H_i. Let us assume κ' represents the value calculated for the same set of actors X_i on time-constrained path P'_i. Then, from Property 5.1:

$$\kappa < \kappa' \tag{5.28}$$

And,

$$\mathcal{D}^i_{xy} - \kappa > \mathcal{D}^i_{xy} - \kappa' \tag{5.29}$$

Again, let us assume that the sum of computation time of actors in X_i is c.

$$c = \sum_{\forall v_j \in X_i} C_j \tag{5.30}$$

Then, from Equation (5.27)

$$\kappa \geq c \tag{5.31}$$

And, since the summation of individual deadlines of actors in P_i' such that $v_j' \in P_i' - X_i$ is

$$\sum_{v_j' \in P_i' - X_i} D_j' = \mathcal{D}_{xy}^i - \kappa' \tag{5.32}$$

Therefore, from Equations (5.26) and (5.29) and the system model constraint specified in Equation (3.14)

$$\mathcal{D}_{xy}^i - \kappa > \mathcal{D}_{xy}^i - \kappa' \geq \sum_{\forall v_j} C_j - c \tag{5.33}$$

The intuitive reason behind Equation (5.33) is that the sum of deadlines of unshared actors is greater than the sum of their execution times, according to our system model. Also, it can be written as

$$\mathcal{D}_{xy}^i - \sum_{\forall v_j \in X_i} D_j > \mathcal{D}_{xy}^i - \sum_{\forall v_j' \in X_i} D_j' \geq \sum_{\forall v_j} C_j - c \tag{5.34}$$

According to Equations (5.31) and (5.33), $\mathcal{D}_{xy}^i - \kappa$ and κ follows the system model constraint specified in Equation (3.14). Then, applying NORM or PURE (Equations (5.3) and (5.8)) using the corresponding latency constraint $\mathcal{D}_{xy}^i - \kappa$, the sum of individual deadlines of all the actors in P_i is

$$\sum_{\forall v_j \in P_i - X_i} D_j + \sum_{\forall v_j \in X_i} D_j = \mathcal{D}_{xy}^i - \kappa + \kappa = \mathcal{D}_{xy}^i \tag{5.35}$$

Therefore, Algorithm 3 assures that $\mathcal{D}_{xy}^i = \sum_{\forall v_j \in P_i} D_j$ even when actors are shared across time-constrained paths. $\square$

After proving that in case of a time-constrained path P sharing some actors with higher sensitivity time-constrained paths, the proposed algorithm assures that P satisfies its latency constraints. Here comes the main proof through Theorem 5.1 that states the validity of the proposed approach and assures that any type of application graph (DAG or DCG) satisfies its latency constraints.

Theorem 5.1. *Consider an HSDF DCG $G = \langle V, E, d \rangle$ with multiple latency constraints $\mathcal{D}_{xy}^i$. Assuming that G is represented by a set of all possible time-constrained paths $\mathcal{P}$ ordered by non-increasing order of sensitivity γ, Algorithm 3 assures that the actors of G are assigned individual deadlines that makes any $P \in \mathcal{P}$ not exceed its specified latency constraint.*

Proof. For any time-constrained path P_i there are two cases:
Case 1: P_i has no actors with assigned deadlines,

$$\forall v_j \in P_i, D_j = \varnothing \tag{5.36}$$

Therefore, Algorithm 3 applies either NORM or PURE stated by Equations (5.11) or (5.13) under the system model constraint $\mathcal{D}_{xy} \geq \sum_{\forall v_j \in P} C_j$. Therefore, from Property 5.2:

$$\sum_{\forall v_j \in P_i} D_j = \mathcal{D}_{xy}^i \tag{5.37}$$

and, P_i does not exceed its specified latency constraint $\mathcal{D}^i_{xy}$.
Case 2: P_i has a set of shared actors X_i with a set of high-sensitivity time-constrained paths P^H_i,

$$\forall v_k \in X_i, D_k \neq \varnothing \tag{5.38}$$

Therefore, Algorithm 3 determines the set of unassigned actors and their corresponding latency constraint ($\mathcal{D}^i_{xy} - \sum_{\forall v_k \in X_i} D_{v_k}$). Since P_i has a set of shared actors X_i with a set of high-sensitivity time-constrained paths P^H_i, Lemma 5.1 assures that the sum of individual deadlines D_j of actors in P_i is equal to $\mathcal{D}^i_{xy} = \sum_{\forall v_j \in P_i} D_j$.

Therefore, Algorithm 3 assures that the assigned deadlines of all actors in G are such that all latency constraints are satisfied. □

Finally, we would like to show that in the special case of pipeline application graphs, the proposed algorithm behaves identically to [Di Natale and Stankovic, 1994, Kao and Garcia-Molina, 1997, Lipari and Bini, 2011] and gives the same results. This is proved in Corollary 5.1.

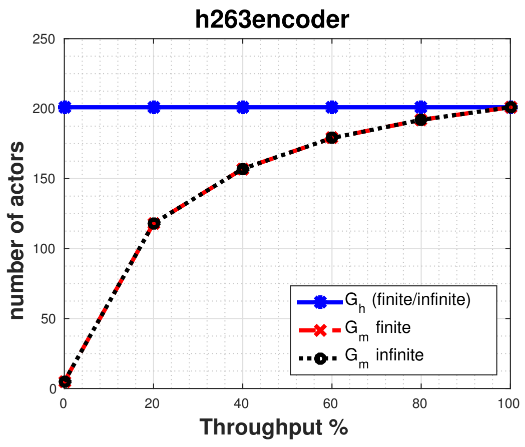
Corollary 5.1. *In case of pipeline application graph $G = \langle V, E, d \rangle$, where G is a multiple actor graph with each actor having a single input/output connected in sequence, applying the proposed algorithm will lead to exactly the same results as previous deadline assignment work for pipelines.*

Proof. Let us assume that we have a pipeline application graph $G = \langle V, E, d \rangle$, where each actor has a single input/output connected in sequence. Applying the first phase of the algorithm (finding all possible time-constrained paths) on G results in a list $\mathcal{P}$ with a single time-constrained path $P = \langle s, v_1, v_2, \ldots, v_z, t \rangle$, where z is number of actors in G. Since it is a single time-constrained path graph and its actors have no assigned deadlines, it will be covered by the first case (1) in Theorem 5.1. Therefore, applying the proposed algorithm will lead to exactly the same results as previous deadline assignment work for pipelines, Equations (5.3) and (5.8) will be applied in this case. □

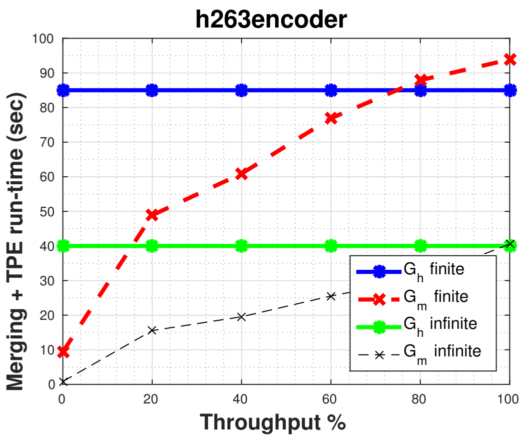
Corollary 5.1 is an important finding, since it shows that *our proposed algorithm is more general and deals with any types of application graphs without any particular drawbacks.*

5.7 Experiments

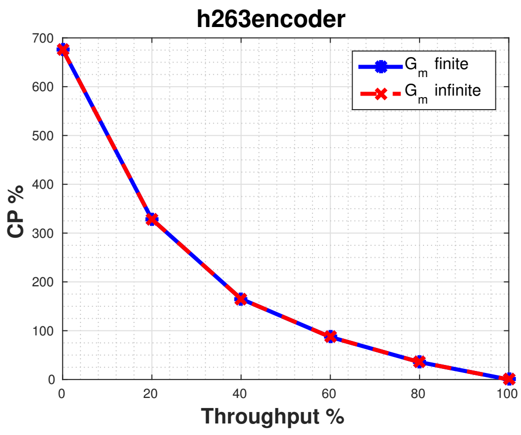
The Timing Parameter Extraction (TPE) [Ali et al., 2015] is proposed for HSDF applications, enabling them to be scheduled and analysed using traditional real-time analysis techniques. This means that TPE requires conversion from an SDF graph to an HSDF graph, which may result in large graphs and hence long run-times of the algorithm. However, in Chapter 4 we introduced a graph reduction technique called slack-based merging, which addresses this problem by generating a reduced-size HSDF graph that maintains the throughput and latency constraints of the original application graph. In this experiment, we evaluate the run-time of the TPE algorithm with HSDF



(a) Results in terms of number of actors



(b) Results in terms of run-time



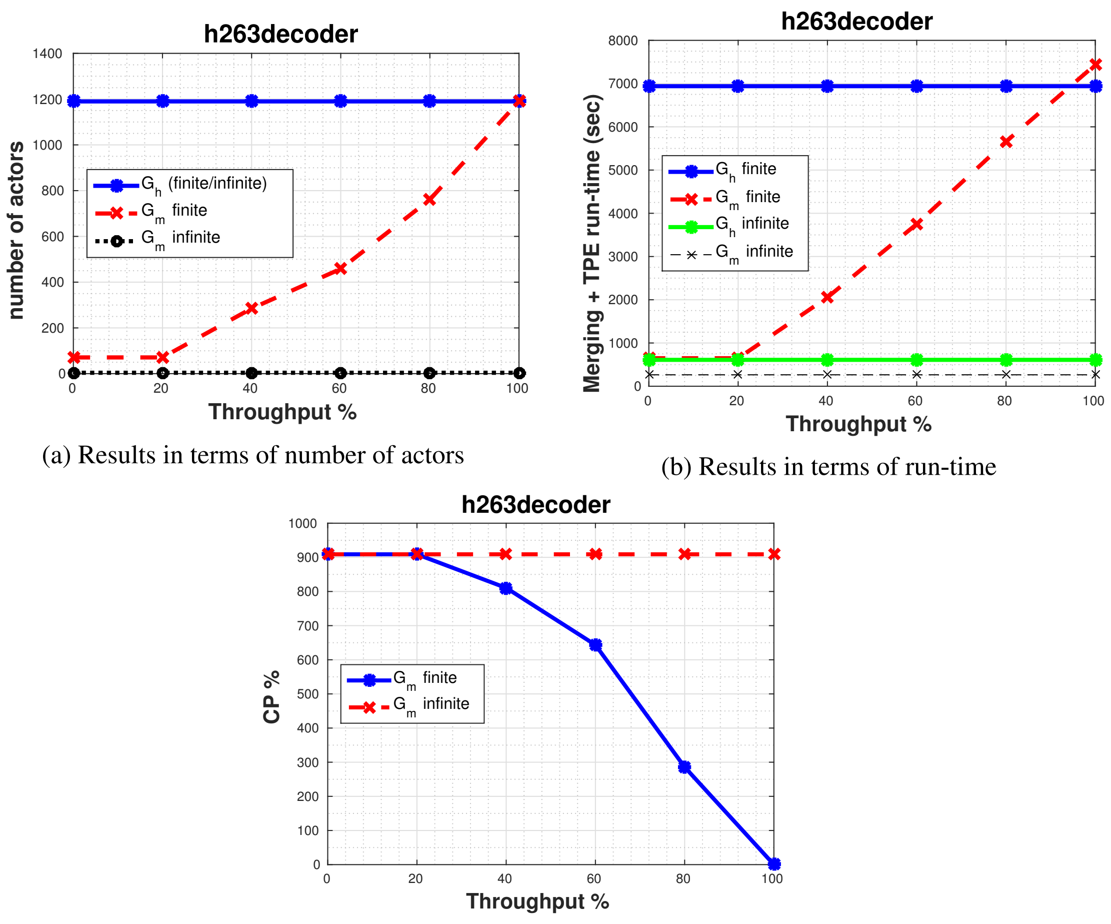
(c) The percentage of change in the CP execution time of G_m compared to G_h

Figure 5.5: *h263encoder* results.

graphs obtained using the classical conversion algorithm from [Sriram and Bhattacharyya, 2000] (G_h) and the slack-based merging algorithm (G_m) presented in Chapter 4. This experiment will show that spending this extra time running the slack-based merging algorithm to generate a graph G_m typically results in a reduction in the run-time of the TPE algorithm, thereby reducing the overall run-time of the complete process.

5.7.1 Experimental Setup

In this experiment, we have the same settings as previously used in Section 4.4. We change the throughput requirement of the tested applications from the given throughput constraint (denoted by 0%) to the maximum throughput (denoted by 100%) in a step-wise fashion in increments of 20%. The latency constraint $\mathcal{D}_{xy}$ of each application is set to the inverse of the throughput constraint of the application, $\mathcal{D}_{xy} = 1/\zeta$. At each throughput step, we apply our merging algorithm on G to generate a reduced-size HSDF graph G_m. Then, both types of graphs (G_h and G_m) are provided as inputs to the TPE algorithm to compare and record their run-time.



(a) Results in terms of number of actors

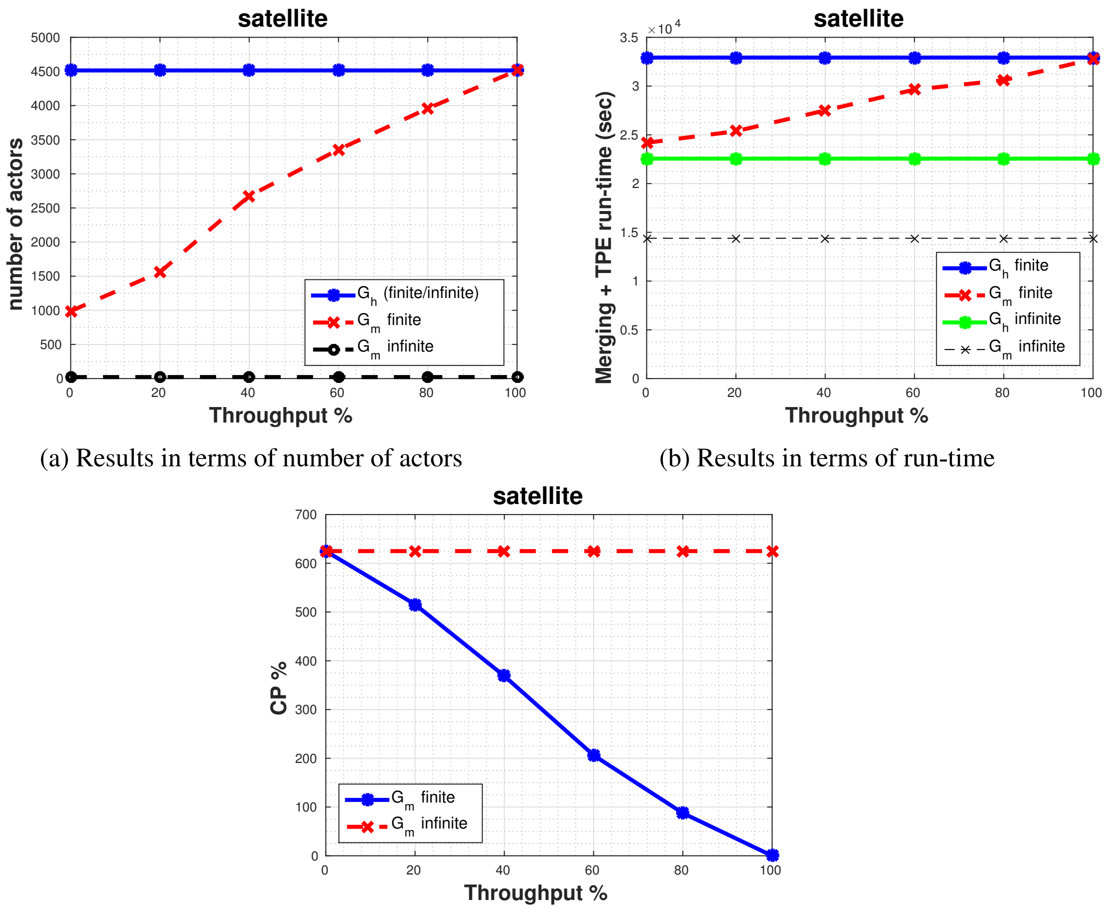
(b) Results in terms of run-time

(c) The percentage of change in the CP execution time of G_m compared to G_h

Figure 5.6: *h263decoder* results.

5.7.2 Experimental Results

The experiment is on applications with two types of buffer sizes, infinite buffers and minimum buffers for maximum throughput (finite buffers). In case of applications with infinite buffers, the results show that the proposed algorithm succeeds in generating a reduced-size compact graph G_m at the maximum throughput (100%) in most cases, as shown in Figure 5.6(a), 5.7(a) and 5.8(a). This is reflected in the large reduction in the run-time of slack-based merging added to the TPE algorithm, that ranges from 39% to 95%, compared to the run-time of the TPE algorithm on the original G_h graphs, as shown in Figure 5.6(b), 5.7(b) and 5.8(b). Also, the results show that having a reduced-size graph G_m at the maximum throughput is not always possible in case of infinite buffers. The *h263encoder* application results, shown in Figure 5.5, illustrates that there are cases where the ability to generate a reduced-size graph decreases with increasing the application throughput (see Figure 5.5(a)). This is natural, because a higher throughput requirement restricts the ability to merge parallel firings, which results in larger output graphs. This is reflected in the increase in the total run-time of slack-based merging and TPE algorithm following the increase in throughput constraint due to the increase in the G_m graph size, as shown in Figure 5.5(b).



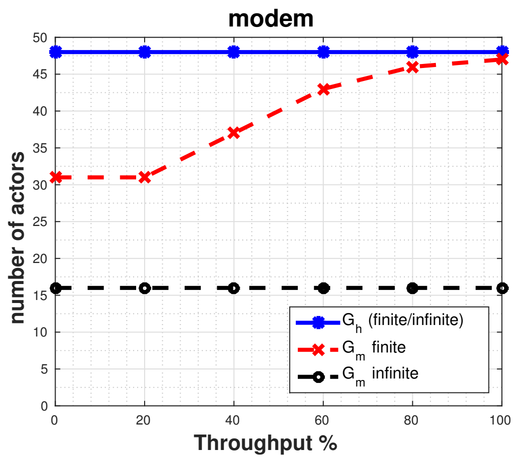
(a) Results in terms of number of actors

(b) Results in terms of run-time

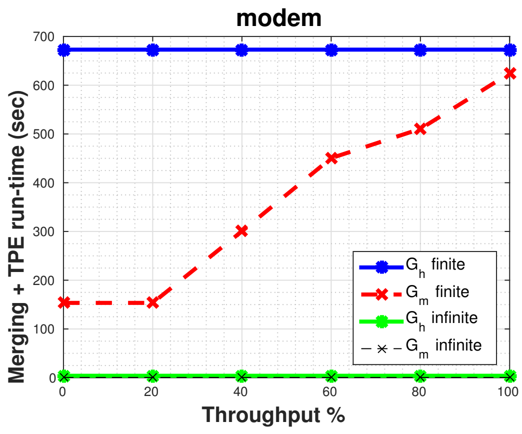
(c) The percentage of change in the CP execution time of G_m compared to G_h

Figure 5.7: *satellite* results.

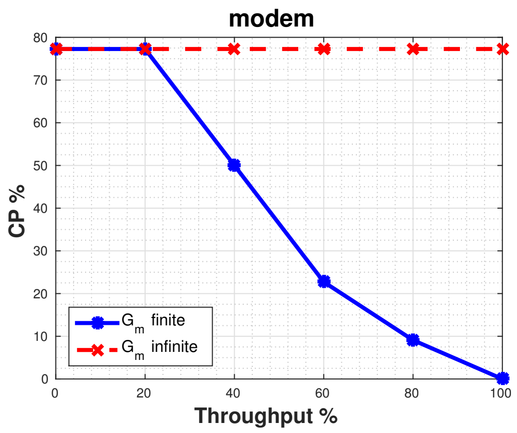
In case of applications with minimum buffers for maximum throughput (finite buffers case), the results show that when the throughput constraint is relaxed with respect to the maximum throughput of the application, the proposed algorithm is able to achieve larger reduction in the application graph size, as shown in Figures 5.5(a), 5.6(a), 5.7(a) and 5.8(a). This significantly reduces the total run-time of slack-based merging and the TPE algorithm, within a range from 27% to 92%, at relaxed throughput constraints. This effect gradually decreases when approaching the maximum throughput of the graph, as shown in Figures 5.5(b), 5.6(b), 5.7(b) and 5.8(b). Moreover, in some finite buffer cases, i.e. *h263encoder* and *h263decoder*, when approaching the maximum throughput the total run-time of slack-based merging and the TPE algorithm exceeds the run-time of applying TPE directly on G_h, as shown in Figures 5.5(b) and 5.6(b). This is due to the increase in the throughput constraint that decreases the ability of merging parallel firings, as in the infinite buffer case. Also, the minimum buffers introduce more dependencies in the graph compared to the infinite buffer case, which reduces the ability to achieve a large reduction in the graph size. For the *mp3playback*, the output graph G_m takes indefinitely long time for extracting its timing parameters that we terminated the experiment after two weeks without reaching any result. This is due to the size of the output graph G_m is still huge (5000 actors), although it has



(a) Results in terms of number of actors



(b) Results in terms of run-time



(c) The percentage of change in the CP execution time of G_m compared to G_h

Figure 5.8: *modem* results.

been reduced to 50% of its size.

Figures 5.5(c), 5.6(c), 5.7(c) and 5.8(c) show a decrease in the percentage of the total execution time of the CP of the applications (0% means execution time of CP is equal to the CP of G_h) with the increase of the throughput constraint for a fixed end-to-end latency constraint $\mathcal{D}$. This means that the remaining slack (after generating the reduced-size graph G_m) increases along with the increase in the throughput constraint. The interpretation of this phenomena is, when the throughput constraint increases a merging decision could be rejected despite of the availability of enough slack, because it could result in a violation of the throughput constraint by increasing the period of the application. This conforms with the previous results which states that the increase in the throughput constraint limits the ability of merging parallel firings.

From these results, we can conclude that our merging algorithm typically succeeds in generating reduced-size graphs, in particular for applications that do not need to execute at maximum throughput, which helps in speeding up the derivation of the timing parameters.

5.8 Summary

In this chapter, we presented a new algorithm for extracting the real-time properties of dataflow applications with timing constraints called Timing Parameter Extraction (TPE). The algorithm can be applied on dataflow applications modelled as HSDF graphs with periodic sources. The main novelty is that the HSDF graphs can be cyclic or acyclic and the graph actors are modelled as arbitrary-deadline real-time tasks. In addition, it enables applying traditional real-time schedulers and analysis techniques on HSDF dataflow graphs. Moreover, it provides a method to assign individual deadlines for real-time dataflow actors and support for two deadline assignment techniques (NORM/PURE) that are widely used in the literature. Through the chapter, we demonstrate the functionality and the validity of the proposed algorithm using an example and proofs. Furthermore, we showed the positive effect of the reduction technique for synchronous dataflow SDF graphs called slack-based merging, explained in Chapter 4, on the run-time of the TPE. The experiments shows that the generated reduced-size HSDF graphs typically enable faster extraction of timing parameters compared to using the original larger HSDF graphs.

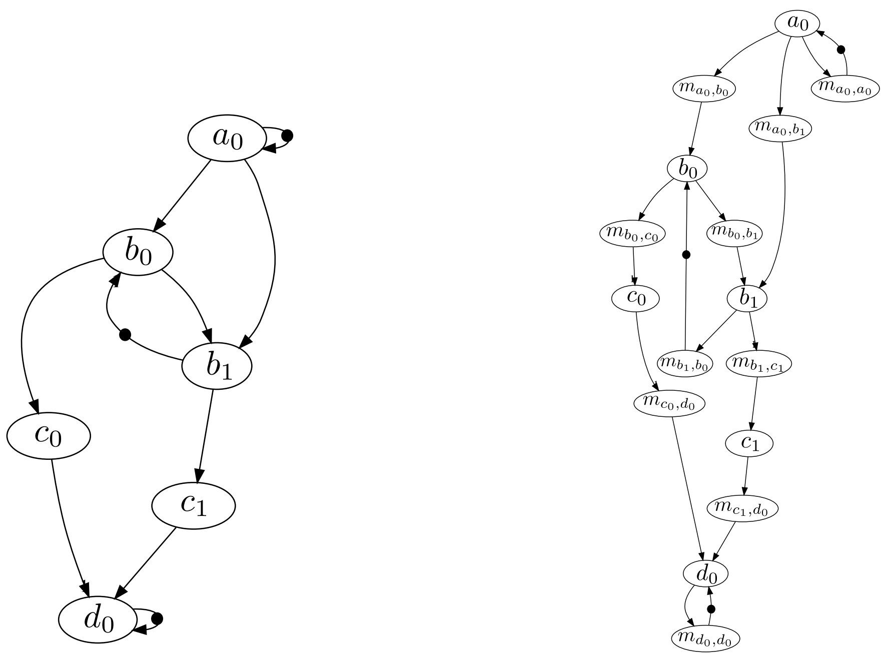
Chapter 6

Communication-Aware Mapping

The preceding chapters (Chapters 4 and 5) presented a detailed overview of how to represent dataflow applications with timing constraints as periodic independent arbitrary-deadline real-time tasks, enabling the usage of traditional real-time scheduling and analysis techniques. The proposed solution starts by proposing a graph reduction technique called slack-based merging algorithm, demonstrated in Chapter 4, which aims to reduce the complexity of dataflow applications by generating reduced-size HSDF graphs possibly avoiding large HSDF graphs generated from traditional conversion methods [Sriram and Bhattacharyya, 2000]. These reduced-size HSDF graphs are used as an input to the next stage called Timing Parameters Extraction (TPE) [Ali et al., 2015], where we extract the timing parameters (offsets, deadlines and periods) of the actors, transforming them into real-time periodic tasks (Chapter 5). This creates a unified model for all applications running on the multi-core platform, where traditional real-time analysis and scheduling techniques can be applied assuring real-time guarantees for the complete system.

Now, we reached the final stage towards a *complete approach* for combining mixed application models on multi-core real-time systems, which is application mapping. Such systems require efficient techniques to map applications to cores, while satisfying their timing constraints, to avoid over-dimensioned systems. In this chapter, we introduce an efficient mapping algorithm called *communication-aware mapping*. This algorithm aims to provide an efficient solution to improve utilization of the platform resources. The effectiveness of the approach is demonstrated through experiments that show the improvement in utilizing the multi-core platform resources compared to the well-known First Fit (FF) bin-packing heuristic. Also, the proposed algorithm takes into account the communication cost caused by message transfer between communicating tasks, which is essential for dataflow applications in the mixed model. However, the mapping algorithm ignores the communication modelling for independent real-time tasks, because they are not communicating. This work is based on the heuristic algorithm for the mapping of real-time streaming applications modelled as dataflow graphs on 2D mesh multi-core processors called Critical-Path-First (CPF) [Ali et al., 2013].

This chapter begins with defining a methodology for modelling communication cost in dataflow applications, detailed in Section 6.1. Next, we introduce the core selection methodology,



(a) HSDF graph (b) HSDF graph with message actors G_{com}

Figure 6.1: Initial modelling of communication.

which is a method used by *communication-aware mapping* algorithm for selecting the next core for mapping in Section 6.2. Then, we present the algorithm itself in detail in Section 6.3, followed by a discussion of its limitations and complexity analysis in Sections 6.4 and 6.5, respectively. We experimentally evaluate our proposed approach in Section 6.6. Finally, we conclude the chapter with a summary in Section 6.7.

6.1 Modelling Communication Cost

Dataflow applications are data-driven networks of actors where there is data transfer (communication) occurring between actors during application execution. This communication is significant, since it plays an important role in determining when actors can fire. Also, it impacts the overall utilization of the resources and the end-to-end response time. Therefore, this communication should be modelled in a way that ensures correct execution of dataflow applications, satisfying their timing constraints. In this work, we model the communication in a two step process. The first step is *initial modelling*, where we transform all the messages in the HSDF graph to actors, as shown in Figure 6.1. We refer to these actors as *message actors*. Figure 6.1(a) depicts a HSDF graph, where actors communicate with each other by sending a single message (token) on each channel. These messages have been transformed into message actors to model the communication cost, as shown in Figure 6.1(b). For example, the message actor $v_{m_{a_0,b_0}}$ represents the message transferred from actor v_{a_0} to actor v_{b_0}. No transformation of initial tokens is required. The reason is that initial tokens represent messages that are ready in the input buffers of their destination actors, waiting to be consumed. This means their execution time is equal to zero and they do not affect the communication model.

The message actors have a WCET equal to the time required to traverse the IN of the platform from the source to destination. Therefore, according to the platform described in Section 3.4, which states a *homogeneous* multi-core platform with a *2D Mesh* IN topology using *X-Y routing*, *wormhole switching* and *TDM arbitration*, the WCET of a message actor is defined by Equation (3.27), where p refers to the message size (bits) in this work. However, Equation (3.27) shows that the WCET of a message actor depends on the number of hops h a message traverses on the IN, which cannot be computed a priori, because the mapping of the application graphs is not known yet. To overcome this problem, we initially assume that each message have its source and destination located on the two furthest cores on the platform. This means that each message has to traverse the maximum number of hops $\hat{h}$ on the IN of the platform. By evaluating Equation (3.19), the maximum number of hops $\hat{h}$ on a multi-core platform Π of size $n \times n$ is equal to:

$$\hat{h} = (n-1) + (n-1) = 2n-2 \tag{6.1}$$

By substituting $\hat{h}$ instead of h in Equation (3.27), the initial value of the WCET of a message actor $\hat{C}_{i,p}$ is defined as:

$$\hat{C}_{i,p} = \underbrace{\hat{h} \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i}}_{\text{message header}} + \underbrace{\left\lceil \frac{p_j}{f} \right\rceil \cdot \left(\frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{l_{sw}}{\mathcal{R}_i} \right)}_{\text{message payload}} + \underbrace{\left(\frac{\varkappa_i}{\mathcal{R}_i} - \varkappa_i + 1 \right) \cdot \frac{f}{\mathcal{L} \cdot \mathcal{R}_i} - \frac{1}{\mathcal{G}}}_{\text{interference}} \tag{6.2}$$

where f is the flit size in bits, l_{sw} represents the switch latency in seconds, $\mathcal{G}$ is the IN frequency in hertz (Hz), $\mathcal{L}$ is the multi-core link capacity in bits per second (bps), and $\mathcal{R}_i$ is the reserved bandwidth of $\mathcal{L}$ in percentage (%), dedicated to a specific dataflow application A_i.

The second step comes after mapping the application on the platform Π, where we update the WCET of the message actors. At this point, we know exactly which messages flow on the IN of the platform Π and which are not. Also, we can precisely determine the number of hops h each message take to reach its destination. Based on this information, the WCET of message actors that flow internally in cores are set to zero, since they reach their destination instantly once they are generated. However, the WCET of message actors that flow on the IN are updated according to the actual number of hops h they traverse. This is achieved by replacing maximum number of hops $\hat{h}$ in Equation (6.2) with the actual number of hops h, which results in an equation identical to Equation (3.27).

Equation (6.2) shows that the execution time of a message actor comprises three terms. The first term is the time spent by the message header (a single flit) to traverse the IN. The second term is the time taken by the message payload to traverse the IN. The third term is the interference suffered by the message actors during traversing the IN. From this, we can deduce two conclusions. First, *the reservation bandwidth $\mathcal{R}$ has a great impact on both the WCET of message actors and the response time of a dataflow application, due to the inverse relation between $\mathcal{R}$ and WCET in* *all three terms in Equation* (6.2). A larger value of reservation bandwidth $\mathcal{R}$ decreases the WCET of message actors and improves the response time of the dataflow application, which enhances its schedulability. However, in case of limited communication resources, it may also reduce the schedulability of multiple communication flows in the network, as the flows on some communication links may exceed the link capacity $\mathcal{L}$. By changing the reservation bandwidth $\mathcal{R}$, the system designer can understand the impact of communication on the mapping of the applications, which is evaluated later in Section 6.6.2. Second, we claim that *despite the pessimism in the assumption of maximum number of hops $\hat{h}$ as an initial value for the WCET of message actors, its impact is insignificant*. The maximum number of hops $\hat{h}$ affects only the time spent by the message header, which is the first term of the equation. The message header is a single flit compared to the rest of the message (payload). This means that our assumption has an insignificant effect on the WCET of the message actors. Especially, when the message payload size is in terms of hundreds of flits. For example, the *h263decoder* application has messages with a payload of 304128 Bytes. Assuming a multi-core platform Π with the configuration mentioned in Table 6.1 (Section 6.6.1), the $\hat{C}_{i,p}$ of a message actor is equal to 1×10^{-3} seconds, approximately. Although the maximum number of hops $\hat{h}$ is equal to 14, the value of the execution time of the message payload (9.3×10^{-4} seconds) added to the interference (5×10^{-6} seconds) is 1336% greater than the execution time of the message header (7×10^{-7} seconds). This means, the impact of the maximum number of hops $\hat{h}$ on the WCET of message actors is insignificant. However, there are two cases where the impact of the execution time of the message header can be significant. Either if the maximum number of hops $\hat{h}$ is in the order of 10^4 hop, or the message payload size is very small, e.g. 1 or 2 flits. Both these cases are highly unlikely in the work considered in this thesis. Moreover, *the second step of updating the value of the WCET of message actor decreases more the effect of maximum number of hops $\hat{h}$ initial assumption*, as explained later in Section 6.3.

The modelling of the communication cost is presented in the complete approach, shown in Algorithm 4. Following Algorithm 4, it starts by generating a reduced-size graph G_m using the slack-based merging algorithm, detailed in Chapter 4. Then comes the first stage of modelling the communication cost by applying a graph transformation for the merged graph G_m, called *channel convert*. This transformation process converts all channels of G_m into actors with WCET equal to $\hat{C}_{i,p}$. If initial tokens exist, they are added on the edge created between the message actor and the destination actor. For example, the channel e_{b_1,b_0} connecting actors v_{b_1}, v_{b_0} is carrying an initial token, as shown in Figure 6.1(a). After transforming it into a message actor $v_{m_{b_1,b_0}}$ the initial token is added on the new edge $e_{m_{b_1,b_0},b_0}$ created between the message actor $v_{m_{b_1,b_0}}$ and the actor v_{b_0}, as shown in Figure 6.1(b). This transformation helps the timing parameter extraction (TPE) algorithm, detailed in Chapter 5, to derive timing parameters (*offset*, *period*, *deadline*) also for messages such that they can transfer safely from the source to the destination in a synchronized manner with the execution of the actors in the graph. After extracting the timing parameters, the main topic of this chapter called *communication-aware mapping* starts, which involves mapping the tasks on the platform and updating their timing parameters according to the final placement of the application graph on the platform Π. The communication-aware mapping is dicussed in detail

Algorithm 4: Complete approach for integrating mixed application models on the same platform Π

```
Input:
G: SDF application graph, $G = \langle V, E, d \rangle$.
Output:
Ψ: The complete system that consists of a homogeneous symmetrical multi-core platform
   $\Pi$ and the mapped application set $A$, $\Psi = \langle \Pi, A \rangle$.
Variables:
$G_m$: merged HSDF application graph.
$G_{com}$: HSDF graph with channels modelled as actors.
$\mathcal{P}$: totally ordered set of all time-constrained paths of an application ordered according to
   $\gamma$, $\mathcal{P} = \{P_i : \gamma_{i-1} \geq \gamma_i\}$.
Π: Homogeneous symmetrical multi-core platform, $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$
1 begin
2     foreach $A_i$ in $A$ do
3         $G_m = slack\text{-}based\ merging(G)$ ........................// Chapter 4
4         $G_{com} = channel\ convert(G_m)$ ........................// Section 6.1
5         $\mathcal{P} = TPE(G_{com})$ ........................// Chapter 5
6         $\Psi = communication\text{-}aware\ mapping(G_{com}, \mathcal{P}, \Pi)$ ......// Chapter 6
7     end
8 end
```

in Section 6.3.

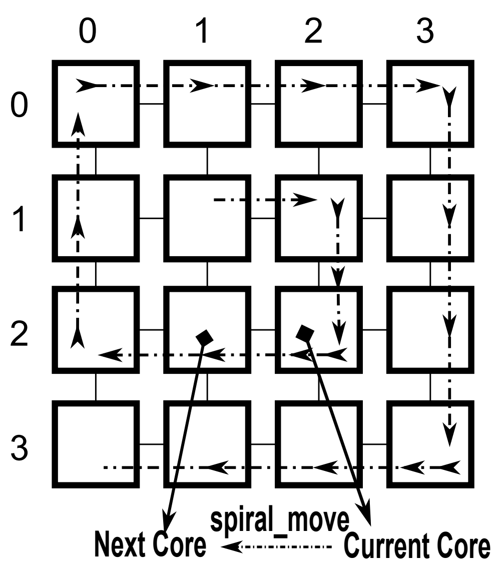
6.2 Core Selection Methodology

In this section, we present the *core selection* methodology, which helps the communication-aware mapping algorithm select a new core for allocating actors. This selection is based on a concept called *independent / dependent path* that classifies the time-constrained paths of a graph G into two types based on whether or not all its actors are unallocated. The independent / dependent path concept is defined as follows:

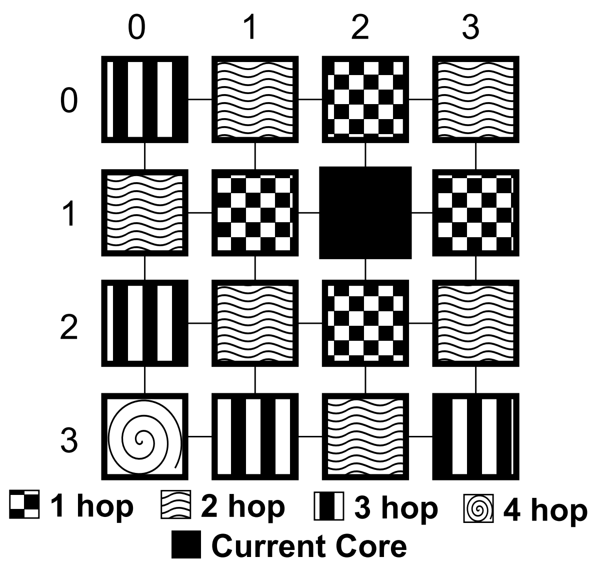
Definition 6.1 (***Independent / Dependent Path***)**.** *A path $P_{A_i} = \langle v_0, v_1, v_2, \ldots, v_j \rangle$ of a certain application A_i is said to be* ***independent*** *iff all its actors are unallocated. If at least one of P_{A_i} actors is already allocated, the path is considered* ***dependent****.*

The core selection is the process where a new core is selected for assigning actors. It is composed of two different methods, which are called *spiral_move* and *find_nearest_core*, as shown in Figure 6.2. Depending on the type of path to be allocated, *independent* or *dependent*, one of these methods is applied, respectively.

For *independent* paths, the selection is performed by *spiral_move*. As shown in Figure 6.2(a), every time the *spiral_move* function is called it returns the next core in the spiral path. The *spiral_move* function is called when the current core fails the feasibility test (Section 3.1.2). The spiral path for core selection is initialized only once at the beginning of the allocation process



(a) *spiral_move*



(b) *find_nearest_core*

Figure 6.2: Core selection methodology

using one of the middle cores in the platform Π, and advances to the next core each time the feasibility test (Section 3.1.2) fails. In case of the platform dimensions are even (n is even), the middle cores are the four cores at the middle of the 2D-Mesh IN. Otherwise (n is odd), the middle core is uniquely defined. For example, the middle cores in the platform Π shown in Figure 6.2 are $(1,1)$, $(2,1)$, $(1,2)$ and $(2,2)$, where core $(1,1)$ is selected to initialize the spiral path.

For *dependent* paths, as they are partially allocated, allocation of child (unallocated) actors is done *as near as possible to their parent (allocated) actors to reduce communication cost*. The function *find_nearest_core* starts searching for a suitable core (a core that passes the *feasibility test* explained in Section 3.1.2) one hop away from the reference core (defined in Section 6.3.2), where the first core that passes the feasibility test is selected. If not possible, it searches for a suitable core two hops away, and so on, until finding a possible core. The search criteria starts by finding the nearest core in this order: North, South, East and West. In each of these directions, starting from two hops distance from the reference core, there are several cores that can be selected. For example, Figure 6.2(b) shows in the South direction there are three cores that are two hops away from the reference core. The *find_nearest_core* function arbitrarily chooses a core among them and returns it for allocation. Figure 6.2(b) shows the searching regions, classified according to the distance from the reference core.

6.3 Communication-Aware Mapping

The communication-aware mapping algorithm is a heuristic for allocating mixed application models on a 2D-mesh multi-core platform. These mixed application models comprise dataflow applications with timing constraints and real-time independent tasks. It aims to maximize the usage

```
Algorithm 5: Communication-aware mapping
  Input:
  $G_{com}$: HSDF graph with channels modelled as actors.
  $\mathcal{P}$: totally ordered set of all time-constrained paths of an application ordered according to
      $\gamma$, $\mathcal{P} = \{P_i : \gamma_{i-1} \geq \gamma_i\}$.
  Output:
  $\Psi$: The full system that consists of Homogeneous symmetrical multi-core platform $\Pi$ and
      the mapped application set $A$, $\Psi = \langle \Pi, A \rangle$.
  Variables:
  $\Pi$: Homogeneous symmetrical multi-core platform, $\Pi = \{\pi_1, \pi_2, \dots, \pi_n\}$
1 begin
2     $\Pi = SPF(G_{com}, \mathcal{P})$
3     $G_{com} = EZM(G_{com})$
4     $\mathcal{P} = TPE(G_{com})$
5 end
```

of system resources while taking the communication cost of dataflow applications into consideration. In case of dataflow applications, the algorithm uses the time-constrained paths and periodic task set information output of the TPE algorithm to allocate the application graph on the platform. For independent real-time tasks, the algorithm deals with them as graphs with a single node. The communication-aware mapping algorithm is based on two main criteria:

1. Allocating time-constrained paths in decreasing order of sensitivity.

2. Exploiting parallelism in the application by allocating parallel time-constrained paths P on different cores.

The first criteria allows the algorithm to map the time-constrained paths that have the highest impact on the schedulability of the application first, which allows maximizing the usage of the available resources. Also, it gives the mapping algorithm a tendency to order the allocation of tasks from heaviest to lightest density, which has been shown to provide a better solution than the well-known FF [Hoffman, 1999]. The second criteria potentiates parallelism, which improves the performance of the allocated applications and allow mapping more applications, as demonstrated later in Section 6.6.

This algorithm is inspired by the heuristic dataflow graph mapping algorithm called Critical-Path-First (CPF) [Ali et al., 2013]. However, the communication-aware mapping uses the path density as parameter for path sensitivity, as stated in Definition 5.1, while in case of CPF the execution time of a path determines the path sensitivity. The communication-aware mapping is more general then the CPF algorithm. This is because CPF ignores the communication cost contrary to the communication-aware mapping. In the following sections, we present a description of the general functionality of the communication-aware mapping algorithm in Section 6.3.1. Section 6.3.2, then provides a detailed explanation of the SPF mapping heuristic, which is a main building block in the proposed algorithm.

6.3.1 General Functionality

The communication-aware mapping consists of three stages, as shown in Algorithm 5. The first stage is the mapping heuristic called Sensitive-Path-First (SPF). The SPF algorithm is responsible for allocating the application actors (not the message actors) of the G_{com} graph on the multi-core platform Π, such that the system is schedulable. This is assured through using the Partitioned Earliest Deadline First (PEDF) as scheduler and the Quick convergence Processor-demand Analysis (QPA) (Section 3.1.2.2) as feasibility test to decide whether or not to map an actor to a specific core. The following Section 6.3.2 explains the SPF mapping algorithm in detail.

The second stage is eliminating message actors with zero computation time from the G_{com} graph, which we refer to as $EZM(G_{com})$ in Algorithm 5. This stage searches G_{com} for message actors whose source and destination actors have been mapped to the same core to eliminate them from the G_{com} graph. This is because these messages are produced at their destination and never use the IN of the platform. For example, the actors v_{b_0} and v_{c_0} in the G_{com} shown in Figure 6.1(b) have been mapped on the same core, so applying $EZM(G_{com})$ stage eliminates the message actor $v_{m_{b_0,c_0}}$.

The third stage is the TPE algorithm that plays the role of updating the timing parameters for the actors and the message actors in the graph according to the placement of the actors on the platform Π. This update process is necessary because the initial values of timing parameters are calculated based on two pessimistic assumptions, which are:

1. Each message have to cross the maximum number of hops $\hat{h}$ on the platform Π.

2. All message actors flow on the IN of the platform Π.

This makes the initial values of the timing parameters pessimistic compared to the actual reality. The new computed timing parameters relax the individual deadlines of the mapped actors by recalculating them based on how many messages of the application graph use the IN and the actual number of hops h that a message traverse on the IN. This means that the density of the mapped actors decreases, allowing more new applications to be allocated, whether they are dataflow graphs or independent real-time tasks, which helps increasing the utilization of the platform resources.

However, recalculating timing parameters raises a question about the schedulability of the system, since the task's timing parameters have been changed. Although it is a valid question, the system is still schedulable. After mapping the application graph using SPF, two mutual exclusive and jointly exhaustive cases can happen. They are:

1. G_{com} is mapped such that every two communicating actors in the graph are located on two different cores with a distance equal to the maximum number of hops $\hat{h}$. In this case, all the message actors are flowing on the IN of the platform Π. This means $EZM(G_{com})$ will not eliminate any of the message actors. Also, all message actors traverse the maximum number of hops $\hat{h}$ on the IN. This means that their WCET is still the same and the TPE stage will not update any of the timing parameters of the graph actors or message actors. Therefore, the system is schedulable.

2. G_{com} is mapped such that every two communicating actors in the graph are **not** located on two different cores with a distance equal to the maximum number of hops $\hat{h}$. This means either, some of the message actors have been eliminated in the $EZM(G_{com})$ stage and the rest is traversing a number of hops h less than or equal to the maximum $\hat{h}$, or all of the messages are traversing a number of hops h less than or equal to the maximum $\hat{h}$ (excluding the first case). In both cases, the TPE algorithm will find more latency slack, resulting from the eliminated message actors and the current mapping pattern, to distribute on G_{com} graph actors. This means that the relative deadline D_i of the mapped actors will increase, which will not affect the schedulability of the system. Also, the offsets a_i of the mapped actors will change, but the schedulability will not be affected. This is because of the QPA feasibility test being offset agnostic, as shown in Algorithm 1 and Equation (3.8). This means QPA assumes that all actors start simultaneously at time instant zero ($a_i = 0$), which is a pessimistic assumption.

Therefore, updating the timing parameters helps in increasing the utilization of the platform resources without negatively affecting the schedulability of the system.

6.3.2 Sensitive-Path-First Algorithm

Sensitive-Path-First (SPF) is a heuristic algorithm that allocates mixed application models with timing constraints, after unifying them, on a 2D-mesh multi-core platform. The main criteria of the SPF algorithm is to allocate time-constrained paths P that have the highest sensitivity γ first. It is also able to exploit parallelism in the application by allocating parallel time-constrained paths P on different cores. These criteria allow maximizing the usage of the available resources and potentiates parallelism, which helps increasing the number of mapped applications and improve their performance.

The proposed approach, shown in Algorithm 6, picks a path P_i in order of sensitivity γ from $\mathcal{P}$. The selected path P_i is checked whether it is independent or dependent. P_i is always independent by definition if it is the *most sensitive path* in the graph.

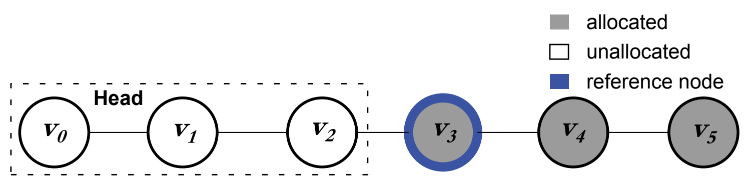
If path P_i is independent, the algorithm allocates its actors $\langle v_0, v_1, v_2, \dots, v_j \rangle$ onto the multi-core processor $\Pi = \{\pi_1, \pi_2, \dots, \pi_n\}$. For each actor v_j, the allocation process checks the feasibility test for the current core. If the test is true, it assigns the actor v_j to the current core. Otherwise, the next core is selected using *spiral_move* and the process is repeated again.

On the other hand, if path P_i is dependent, the algorithm searches its partial paths P_i^p (unallocated path sections) and classifies them into three classes: Head, Middle and Tail, similar to the offset assignment mechanism mentioned in Section 5.5.2. Figure 6.3 shows the three classes of partial paths. For each partial path P_i^p, the algorithm determines a reference allocated actor (parent) and uses its core as a reference core in the process of selecting the nearest possible core. This reference actor (parent) is determined according to the P_i^p class. In case of P_i^p being a Head, the reference actor is the successor of the last actor in the partial path, as shown in Figure 6.3(a). In case of a Tail, the reference actor is the last allocated actor before the partial path, as shown

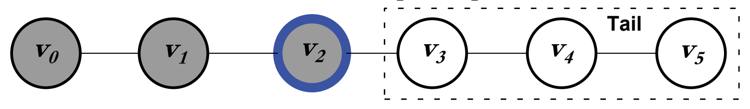
Algorithm 6: Sensitive-Path-First (SPF)

```
$\mathcal{P}$: totally ordered set of all time-constrained paths of an application $A_i$ ordered according
    to $\gamma$, $\mathcal{P} = \{P_i : \gamma_{i-1} \geq \gamma_i\}$.
$P_i$: A full time-constrained path in $\mathcal{P}$.
$P_i^p$: Partial path of full path $P_i$.
$LP_i^p$: List of partial paths in $P_i$.
begin
    $n = spiral\_move()$;
    foreach $P_i$ in $\mathcal{P}$ do
        if $P_i$ is Independent then
            foreach $v_j$ in $P_i$ do
                while (all cores are not tested) and ($v_j$ not allocated) do
                    if feasibility test then
                        allocate $v_j$ on core $\pi_n$.
                    else
                        $n = spiral\_move()$;
                    end
                end
                if $v_j$ not allocated then
                    unallocate $\forall v_j \in A_i$ from $\Pi$.
                end
            end
        else // Dependent Path Case
            search for possible $P_i^p$ in $P_i$.
            classify found $P_i^p$ & add them to $LP_i^p$.
            foreach $P_i^p$ in $LP_i^p$ do
                if Head or Tail then
                    find the reference actor (Parent).
                    allocate using find_nearest_core.
                else if Middle then
                    calculate mid-point (core).
                    allocate using find_nearest_core.
                end
                if ($v_j$ in $P_i^p$) not allocated then
                    unallocate $\forall v_j \in G$ from $\Pi$.
                end
            end
        end
    end
end
```

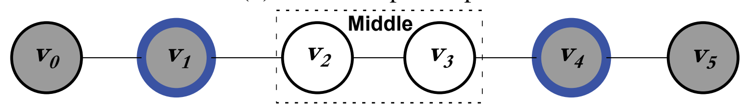
in Figure 6.3(b). In the case of a Middle, the reference core is selected differently. The class middle partial path is surrounded by two allocated actors (parents), as shown in Figure 6.3(c). The reference core is thus selected by computing the middle core between the parents. If the number of cores between the parents is even, an arbitrary core is selected from the two middle cores in



(a) Class Head partial path



(b) Class Tail partial path



(c) Class Middle partial path

Figure 6.3: Partial path classification used by SPF heuristic.

between the parents. The location of the computed reference core is given to *find_nearest_core* as an input to find the possible nearest core to allocate the child actors.

The SPF approach uses two different techniques, *spiral_move* and *find_nearest_core*, for allocating independent and dependent paths, respectively. This is because independent paths can be allocated on any set of cores that have enough capacity to accommodate the path. However, unallocated parts (children) of a dependent path need to be allocated near to their parents to decrease the communication cost between child and parent actors. The partial path classification discovers potential parallelism in the application, since, by definition, the full path (containing the partial path) shares some of its actors with another allocated path. This feature is an advantage and this knowledge allows to allocate these parallel sections on different cores (if possible), thus, enhancing the performance and reduce the end-to-end worst-case response time of the application graph. If the heuristic fails in the allocation of any path P_i, the heuristic unallocates all previously allocated actors of the graph.

6.4 Limitations

The communication-aware mapping algorithm has a communication model that is used to account for the communication cost of dataflow applications running on the platform Π. This communication model guides the mapping algorithm, aiming to increase utilization of the full system. It comprises two phases that begin with modelling all the messages exchanged between actors as message actors, as demonstrated in Section 6.1. Then after mapping the dataflow application on the platform Π, the update phase recalculates the timing parameters of all actors and message actors reflecting the current mapping decisions, as explained previously in Section 6.3.1. This model successfully allows to show the impact of communication on the schedulability of the system in terms of number of allocated applications, as experimentally demonstrated in Section 6.6. However, it does not guarantee the feasibility of the communication on the IN of the platform Π. This means that it does not guarantee that the sum of total reservations on each link on the IN is less than or equal to 100%. Another limitation that affects the feasibility of the communication is coming from reserving a dedicated bandwidth per application and not per communication flow. This means that the WCET of the communication flows (messages actors) can be more than the one computed, which can affect the communication feasibility. This is because of the possibility of multiple communication flows of the same application using the same link of the IN at the same time. This means that the dedicated bandwidth for this specific application is shared between these communication flows on this link, which increases the WCET of these flows, and hence affecting the communication feasibility. Performing schedulability analysis at the network level during mapping would be possible, but would be dependent on the routing strategy used, and would increase the complexity and overhead of the mapping. For simplicity, the approach considers that the feasibility at the network is performed in a final step, after the applications are mapped. Improvements to this approach are relevant future work and are briefly presented in Chapter 7. Note that the reservation bandwidth $\mathcal{R}$ parameter allows the designer to understand the impact of communication in the allocation of applications. When a particular mapping is found not to be schedulable, this understanding can guide the selection of a new $\mathcal{R}$.

The communication-aware mapping does not assure that firings of the same SDF actor are mapped on the same core. This affects the correct execution of the HSDF application. Therefore, we assume that shared states between firings of the same SDF actor are always communicated through self edge channels.

6.5 Complexity Analysis

In this section, we provide a complexity analysis for the communication-aware mapping algorithm (Section 6.5.1) and the complete approach (Section 6.5.2), previously presented in Algorithms 5 and 4, respectively. The complexity analysis is done assuming a single application graph as an input.

6.5.1 Communication-Aware Mapping

The communication-aware mapping algorithm is composed of two sub-algorithms. They are the SPF and the TPE. The SPF consists of a two-level nested loop (**foreach** statement) that runs $|\mathcal{P}| \cdot |V_h|$ times. Consequently, the complexity of SPF is equivalent to $O(|\mathcal{P}||V_h|)$. The complexity of TPE is equivalent to $O(|\mathcal{P}|+|\mathcal{P}||V_h|+|V_h|+|E_h|)$ according to the calculations in Section 5.5.3. Therefore, the total complexity is the sum of the complexity of the two algorithms SPF and TPE $O(|\mathcal{P}|+|\mathcal{P}||V_h|+|V_h|+|E_h|+|\mathcal{P}||V_h|) = O(|\mathcal{P}|+2|\mathcal{P}||V_h|+|V_h|+|E_h|)$. Hence, the final complexity of the communication-aware mapping algorithm is $O(|\mathcal{P}|+|\mathcal{P}||V_h|+|V_h|+|E_h|)$, which is polynomial and depends on $|\mathcal{P}|$, $|V_h|$ and $|E_h|$.

6.5.2 Complete Approach

Now, we are ready to compute the complexity of the complete approach, shown in Algorithm 4. It is composed of four sub-algorithms. They are the slack-based merging, the channel convert, the TPE and the communication-aware mapping. First, the slack-based merging has a complexity of $O(|V_h|^2 + |V_h||E_h|)$, as detailed in Section 4.3.4. Second, the channel convert has a complexity of $O(|E_h|)$, since it traces every edge in the HSDF graph and converts it into an actor, as shown in Figure 6.1. Third, the TPE has a complexity of $O(|\mathcal{P}| + |\mathcal{P}||V_h| + |V_h| + |E_h|)$, as detailed in Section 5.5.3. Fourth, the communication-aware mapping has a complexity of $O(|\mathcal{P}| + |\mathcal{P}||V_h| + |V_h| + |E_h|)$, as explained previously. Therefore, the final complexity of the complete approach is equivalent to $O(|V_h|^2 + |V_h||E_h| + |\mathcal{P}| + |\mathcal{P}||V_h| + |V_h| + |E_h|)$, which is still polynomial and depends on $|\mathcal{P}|$, $|V_h|$ and $|E_h|$.

6.6 Experiments

Finally, we reached the evaluation section of this chapter. Through the previous ones, we evaluated the primal stages of the complete approach step by step. In Chapter 4, we evaluated the slack-based merging algorithm and showed that it generates reduced-size HSDF graphs that satisfy the throughput and latency constraints of the original application graph. Then, we followed it by evaluating the TPE algorithm in Chapter 5, where we showed it typically extracts timing parameters faster using these reduced-size HSDF graphs compared to using the original larger graphs.

In this section, we evaluate the full system solution using three experiments that test different algorithms of its structure. The first experiment, detailed in Section 6.6.2, evaluates the communication modelling methodology of the communication-aware mapping algorithm through the testing of the communication cost and its effect on the schedulability of the system. The second experiment, presented in Section 6.6.3, evaluates the SPF mapping heuristic of the communication-aware mapping algorithm by comparing it against the well-known FF bin-packing heuristic. The reason for choosing FF is it has been shown to behave as well as other bin-packing algorithms, and outperform them in some cases, in terms of achieved throughput [Guo and Bhuyan, 2006, Hoffman, 1999]. The final experiment, presented in Section 6.6.4, evaluates the complete approach and shows the trade-off between using original and merged dataflow graphs in terms of number of allocated applications and the overall run-time of the complete approach. This evaluation assess the suitability of the proposed approach for different types of applications.

6.6.1 General Experimental Setup

The set of input applications comprises SDF^3 benchmark applications [Stuijk et al., 2006]. The SDF^3 benchmark applications are classified into two types: high ($u > 0.5$) and low ($u \leq 0.5$) total utilization, as shown in Table 6.2. From these two types, each experiment uses different weights for the random generator to create five sets, of 500 applications each, with a range of Low/High:

Table 6.1: General configuration of the experimental setup.

Platform size $n \times n$	8×8
Maximum number of hops $\hat{h}$	14
Router switch latency l_{sw}	1 cycle
flit size f	16 Byte
Feasibility test	QPA
Link Capacity $\mathcal{L}$	256 Gbps
IN frequency $\mathcal{G}$	2 GHz
Number of allocated Slots $\varkappa_i$	1
Reservation Bandwidth $\mathcal{R}_i$	$\mathcal{R}$

Table 6.2: SDF^3 benchmark applications.

Applications	Utilization (Low/High)
h263decoder	0.76 (High)
h263encoder	1.2 (High)
modem	0.9 (High)
samplerate	0.37 (Low)
satellite	0.6 (High)
MP3 decoder (granule level)	0.41 (Low)
MP3 decoder (block level)	0.41 (Low)

90% Low - 10% High, 60% Low - 40% High, 40% Low - 60% High, 20% Low - 80% High, 10% Low - 90% High. Each experiment runs the complete approach, shown in Algorithm 4, on these five input data sets trying to allocate as many applications as possible on the multi-core platform Π using this approach. To ensure the schedulability of the system, the Quick convergence Processor-demand Analysis (QPA) is used to guarantee the feasibility of the mapped applications. The Π is an 8×8 2D-mesh homogeneous multi-core with a NoC of link capacity $\mathcal{L}$ equal to 256 Gbps. Each application A_i has a single allocated slot in a TDM frame, and the reservation bandwidth per application $\mathcal{R}_i$ equal to $\mathcal{R}$. Table 6.1 summarizes the general configuration of the experimental setup.

6.6.2 Evaluation of the Communication Cost

In real-time multi-core platforms that run dependent tasks, communication plays a big role in the schedulability of the system. In our system, this role can be noticed in Equation (3.27) that shows the inverse relation between the IN link capacity $\mathcal{L}$ and the reserved bandwidth $\mathcal{R}$ on one side, and the WCET of messages $C_{i,p}$ on the other side. The link capacity $\mathcal{L}$ and the reserved bandwidth $\mathcal{R}_i$ represent the communication resources available to an application. When the communication resources increase the resource utilization decreases allowing the IN to handle more traffic, and vice versa. This experiment aims to demonstrate the effect of availability of communication resources on the schedulability of the system in terms of number of allocated applications. To show this, we run Algorithm 4 on the input data sets, mentioned previously in Section 6.6.1, using three values

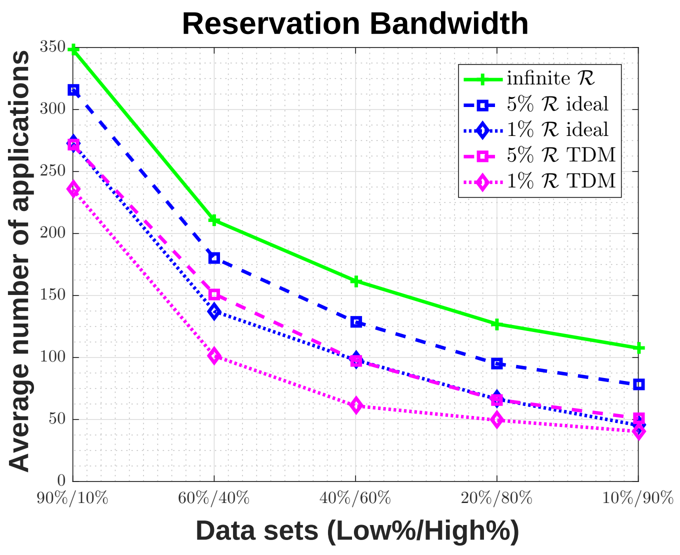


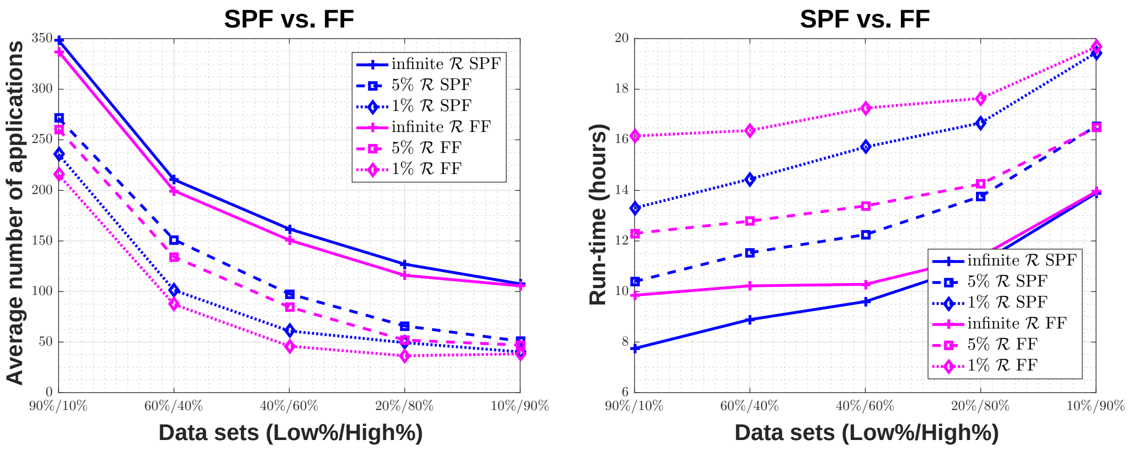
Figure 6.4: Effect of reservation bandwidth $\mathcal{R}$.

of the reservation bandwidth $\mathcal{R}$. These values considered are: infinity, 5% and 1%.

The infinity value of reservation bandwidth $\mathcal{R}$ represents a system that does not model communication costs in any way where messages reach their destination immediately once they are produced. Therefore, in such system the WCET of message actors is equal to zero and its communication is always feasible. This allows to set an upper bound on the number of allocated applications, no matter which type of arbiter and arbiter configuration is used. On the contrary, the 5% and 1% values of reservation bandwidth $\mathcal{R}$ represent a system with limited communication resources. As we mentioned previously in Section 6.4, the communication-aware mapping does not guarantee the communication feasibility of the system. This means the experimental results may be optimistic, but demonstrates the impact of communication cost on the schedulability of the system. To decrease the margin of optimism in our results, we give away bandwidth at a fine granularity to assure that each communication link can handle messages from a lot of applications before it becomes infeasible. For example, a reservation bandwidth of 1% allows 100 applications to use a communication link safely, since they have one allocated slot each.

In this thesis, we propose a platform with a TDM arbiter for the IN to guarantee dedicated bandwidth to mapped applications and to provide traffic isolation. Using a TDM arbiter results in interference, as the reserved bandwidth $\mathcal{R}$ is not available immediately once requested by the application. This interference shows up as the term I_{TDM} in Equation (3.16), which evolves into Equation (3.27) used in our experiments. However, we would like to investigate the boundaries of our complete approach in case of using different types of arbiters for particular reservation bandwidth $\mathcal{R}$ values (5% and 1%). Therefore, we run the same experiment assuming an ideal arbiter, which means WCET of message actors $C_{i,p}$ is equal to the isolation time $C_{i,p}^{iso}$. This experiment gives an upper bound on the number of allocated application by our complete approach using any type of arbiters at specific reservation bandwidth $\mathcal{R}$ values.

Figure 6.4 shows the summary of the results in case of infinite communication resources (in-



(a) Results in terms of average number of mapped applications.

(b) Results in terms of run-time.

Figure 6.5: Evaluation of the mapping heuristic.

finite $\mathcal{R}$), TDM and an ideal arbiter. The results show that the case of infinite communication resources (infinite $\mathcal{R}$) upper bound any type of arbiter for any reservation bandwidth $\mathcal{R}$ value in terms of average number of mapped applications. As expected, it upper bounds the ideal arbiter that is considered as an optimistic upper bound for the TDM arbiter. *The infinite $\mathcal{R}$ exceeds by an average of* 26% *and* 76% *more mapped applications over* 5% *and* 1% $\mathcal{R}$ *values for the ideal arbiter, respectively. This shows the importance of considering communication cost and its effect on the number of schedulable applications on the system.*

Another optimistic upper bound is the ideal arbiter to the actual TDM arbiter for any reservation bandwidth $\mathcal{R}$ value, as shown in Figure 6.4. The ideal arbiter allocates an average of 31% and 28% more applications over the actual TDM arbiter in 5% and 1% $\mathcal{R}$, respectively. This shows the effect of the TDM interference I_{TDM} on the number of schedulable applications on the system. Also, the results illustrate the direct relation between the reservation bandwidth $\mathcal{R}$ and the number of allocated applications on the platform Π. As we notice from Figure 6.4, when $\mathcal{R}$ is equal to infinity, the maximum number of allocated applications is achieved in all input data sets, no matter which type of arbiter is used. However, the number of allocated applications reduces following the decrease of $\mathcal{R}$. In addition, we notice that the number of mapped application decreases as the percentage of high utilization application increases in the input data sets. This is due to high utilization applications consume a lot of the platform resources preventing our algorithm from mapping more applications.

Finally, we can conclude that *the communication cost has a significant impact on the schedulability of the system. When the communication resources (reservation bandwidth $\mathcal{R}$) increase the number of mapped applications increase, and vice versa.*

6.6.3 Evaluation of the Mapping Heuristic

The proposed complete approach is modular and easily allows using different bin-packing heuristics. In this experiment, we compare two different mapping heuristics, Sensitive-Path-First (SPF) and First Fit (FF). The choice of FF for comparison with SPF heuristic is because FF surpasses other bin-packing algorithms in terms of achieved throughput [Guo and Bhuyan, 2006]. This experiment uses the same input data sets and settings as illustrated in the previous experiment in Section 6.6.2, except it assumes TDM arbitration and runs for both SPF and FF heuristics.

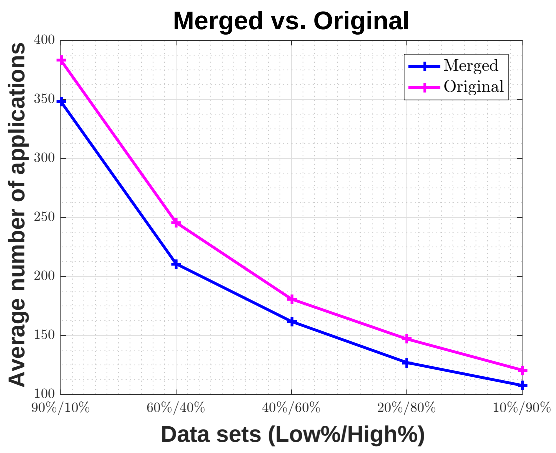
The experimental results are demonstrated in Figure 6.5. In terms of number of allocated applications, Figure 6.5(a) shows that the SPF heuristic dominates FF, succeeding efficiently in utilizing the computational resources through the allocation of more applications in all input data sets and for different reservation bandwidth $\mathcal{R}$ values. In case of infinite reservation bandwidth $\mathcal{R}$, which represents an upper bound on number of mapped applications using any arbiter type, the achieved gain using SPF ranges from 2% to 10% (approximately) with an average gain of 6%. In case of 5% reservation bandwidth $\mathcal{R}$, the achieved gain ranges from 4% to 24% (approximately) with an average gain of 12%. In case of 1% reservation bandwidth $\mathcal{R}$, the achieved gain ranges from 3% to 28% (approximately) with an average gain of 15%. This is due to the selective nature of the SPF heuristic that enables the allocation of actors in the most sensitive paths first that have higher impact on application schedulability, previously discussed in Section 6.3.2. Also, SPF actively encourages mapping independent and partial (Head, Tail, Middle) paths on different cores, which enables parallelism to be exploited in each application. In addition, Figure 6.5(a) illustrates the effect of the communication resources on the number of allocated applications, whatever bin-packing heuristic is used (SPF or FF). The presented results conforms with the conclusions of the previous experiment detailed in Section 6.6.2.

In terms of run-time, the SPF heuristic outperforms FF. As noticed in Figure 6.5(b), the SPF heuristic achieves a lower run-time for most data sets and reservation bandwidth $\mathcal{R}$ values. The overall achieved gain, in terms of run-time, ranges from 1% to 22% (approximately) with an average of 9%. This occurs because SPF has a tendency to order the tasks in decreasing order of density while mapping, which enables the heuristic to find a feasible core quicker than FF. This tendency is coming from the nature of SPF to map higher sensitive paths first. The higher sensitive paths comprise tasks with high densities that have great impact on the schedulability of the system. Therefore, mapping highly sensitive paths first means mapping tasks in decreasing order of density. The results show that there is no added complexity from using SPF compared to FF. Also, the results show that the run-time of both heuristics converge for high utilization data sets. The reason behind this is that both heuristics struggle similarly to map applications, because high utilization applications consume a lot of resources leaving no space for mapping others. This struggle is illustrated in the rise of run-time with the increase of high utilization applications in the input data sets.

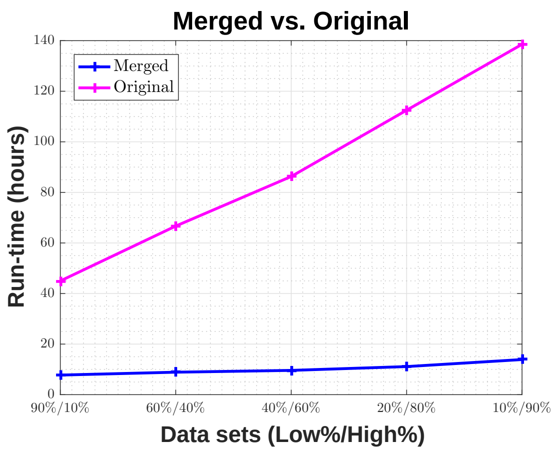
Based on these results, we conclude that *SPF outperforms the well-known FF both in terms of number of mapped applications and run-time.*

6.6.4 Evaluation of Slack-based Merging

In this experiment, we evaluate an important part of the proposed complete approach, which is the slack-based merging. The evaluation illustrates the trade-off between using merged and original



(a) Results in terms of average number of mapped applications.



(b) Results in terms of run-time.

Figure 6.6: Mapping results for merged and original HSDF graphs.

HSDF graphs in terms of number of mapped applications and run-time of the complete approach. This experiment uses the same five input data sets and settings described in Section 6.6.1. The mapping heuristic is SPF, and the reservation bandwidth $\mathcal{R}$ is set to infinity. The choice of setting the value of $\mathcal{R}$ to infinity is because it has the shortest run-time, as shown in Figure 6.5(b). Any choice of a different value of $\mathcal{R}$ definitely will lead to new results in terms of absolute value. However, these new results will have the same trend of the existing results and conclusions will be the same. Moreover, the new experiments will take unreasonable longer run-time.

The experimental results are summarized in Figure 6.6. In terms of number of mapped applications, the original HSDF graphs (complete approach without slack-based merging) enabled the complete approach to map approximately 12% (approximately) more applications than the merged ones in all input data sets. This is because the original HSDF graphs contain fine-grained parallelism that the SPF heuristic exploits to efficiently use the platform resources. However, the merged HSDF graphs lose such fine-grained parallelism in the slack-based merging process, decreasing the ability of the SPF heuristic to map applications.

As expected, the complete approach, using merged HSDF graphs achieves lower run-time in all input data sets compared to the original ones. The reduction in run-time ranges from 82% to 90%. This trade-off between the number of mapped applications and the run-time, clearly, goes into the favour of using slack-based merging and merged HSDF graphs, since it speeds up the overall design time of the system. However, this does not mean that the slack-based merging is the best in all cases. The experimental results of Chapter 5 have shown that dataflow applications with high throughput fail in the merging process and generates a new graph with almost the same size as the original one. This means that the slack-based merging will not help in reducing the overall time of the system design process. Even more, it will slow it down by adding the run-time of the merging process as an overhead.

In general, *the proposed complete approach succeeds in decreasing the overall design time of the system significantly, especially at relaxed throughput constraints.*

6.7 Summary

This chapter presented the final stage of the complete approach called *communication-aware mapping*. It is a heuristic algorithm for mapping mixed application models, dataflow graphs with timing constraints and independent real-time tasks, taking into account the communication cost of dataflow graphs. The platform considered in this work is 2D-Mesh homogeneous multi-core processors operated using X-Y routing, wormhole switching and TDM arbitration. The communication-aware mapping algorithm comprises three heuristics. They are: **1)** Sensitive-Path-First (SPF), **2)** eliminate messages with zero execution time (*EZM*) and **3)** Timing Parameter Extraction (TPE). SPF is responsible for mapping mixed application models in the communication-aware mapping algorithm, guaranteeing the schedulability of the system. It is based on the heuristic algorithm for the mapping of real-time streaming applications called Critical-Path-First (CPF) [Ali et al., 2013]. The SPF main criteria is to allocate time-constrained paths P that have the highest sensitivity γ (density) first. It is also able to exploit parallelism in the application by allocating parallel time-constrained paths P on different cores. These criteria allow maximizing the usage of the available resources and potentiates parallelism, and hence helps increasing the number of mapped applications and improve their performance. Before the communication-aware mapping begins execution, all messages in a application are initially modelled as real-time tasks. This initial modelling is updated using the two heuristics *EZM* and TPE to reflect the actual estimate of communication cost after mapping the application on the platform. The experimental evaluation reveals a direct relation between the number of allocated applications and the availability of communication resources, which demonstrates the importance of considering communication cost. The experiments shows that ignoring communication cost allows mapping up to 76% more applications (infinite case), which gives a wrong perception of the ability to map applications with timing constraints. These extra applications can be mapped, but they would not actually meet their timing constraints, which is a dangerous situation in real-time systems. Also, it shows that the SPF mapping heuristic surpasses the well-known FF bin-packing heuristic in terms of number of allocated applications and run-time that reaches up to a maximum of 28% and 22%, respectively. Moreover, it shows that the slack-based merging has a great impact on the run-time of the complete approach achieving a reduction in the overall system design time that ranges from 82% to 90%.

Chapter 7

Conclusion and Future Directions

In this chapter, we conclude this work by briefly discussing the research question of integrating mixed application models with timing constraints (dataflow and real-time applications) on the same multi-core platform and recapping the proposed solution. We discuss our contributions stating their pros and cons in Section 7.1. Then, we provide possible extensions of our work in Section 7.2.

7.1 Conclusions

We are surrounded by embedded systems that help us in various daily life activities. Initially, embedded systems were designed to perform a dedicated function within a larger system. However, the increase in the application demands and the advancement in processor architectures allowed them to perform multiple functions from different computing domains simultaneously. For example, autonomous driving systems enable cars to navigate without human input, while providing infotainment to the passengers. Both autonomous navigation and infotainment are functions from two different computing domains. Hence, there is a growing trend of embedded systems running mixed application models on their processing platform. In this thesis, we are concerned with embedded systems running mixed application models with timing constraints. The considered mixed applications models are dataflow applications with timing constraints (latency and throughput) and traditional real-time applications represented as independent periodic tasks. Such embedded systems running mixed application models require real-time guarantees that assure satisfying timing constraints.

We proposed an approach, formulated in Algorithm 4, which transforms SDF graphs into periodic arbitrary-deadline tasks, to enable applying real-time scheduling and analysis techniques that guarantee that the timing constraints of the applications are satisfied when they are mapped on the multi-core platform. The proposed approach comprises three main contributions, *Slack-Based Merging*, *Timing Parameter Extraction (TPE)* and *Communication-Aware Mapping*. In the following sections, we recap on our main contributions, discussing their advantages and disadvantages.

7.1.1 Slack-Based Merging

Slack-based merging is an algorithm for addressing the problem that SDF graphs may grow exponentially when converted to an HSDF graph. It is based on two main concepts. First is the concept of *slack*, which is the difference between the WCET of the SDF graph's firings and their relative deadlines. Second is the novel concept called *safe merge*, which is a merge operation that we prove cannot cause a live HSDF graph to deadlock. The algorithm generates reduced-size HSDF graphs that satisfy the throughput and latency constraints of the original application graph. The experimental results of Chapter 4 show that the proposed algorithm achieves large reduction rates of the original HSDF graph, in terms of number of actors, that reaches up to 99.7% in some applications. This result reflects positively on the run-time of the complete approach, achieving a reduction in the overall system design time that ranges from 82% to 90%, as demonstrated in the experimental evaluation in Chapter 6. This does not mean that the slack-based merging is always a good solution for reducing the complexity of HSDF dataflow applications. One of the drawbacks of this algorithm is a reduction of fine-grained parallelism in the application, which is a main benefit of the dataflow computational model. This reduction decreases the maximum throughput a dataflow application is able to reach, although never below the throughput constraint. Another drawback the experimental results of Chapter 5 have shown is that the merging algorithm may be ineffective and generates a new graph with almost the same size as the original for dataflow applications with high throughput requirements. In this case, slack-based merging will not reduce the overall time of the system design process. In fact, it will slow it down by adding the run-time of the merging process as an overhead.

7.1.2 Timing Parameter Extraction

Timing Parameter Extraction (TPE) is an algorithm for converting HSDF graphs with multiple timing constraints (throughput constraint and multiple latency constraints), represented as a Directed Cyclic Graphs (DCG), into arbitrary-deadline real-time tasks defined with *offsets*, *periods*, *deadlines* as timing parameters. This enables applying well known real-time schedulers and analysis techniques on HSDF dataflow graphs. The proposed algorithm provides a method to assign individual deadlines for real-time dataflow actors and support for two deadline assignment techniques (NORM/PURE) that are widely used in the literature. In addition, it allows capturing overlapping iterations, which is a main characteristic of the execution of dataflow applications, by modelling actors as tasks with arbitrary-deadlines. However, the TPE algorithm has a downside related to the first phase of the algorithm that finds all possible time-constrained paths (Section 5.5.1). This phase of the TPE algorithm is a very computationally expensive process, especially when the HSDF graph is large. The experimental results of Chapter 5 shows that applying TPE on the *satellite* large size HSDF graph with 4515 actors takes 3.2×10^4 seconds approximately. When the HSDF graph becomes larger, i.e. *mp3playback* with size of 10000 actors, the TPE takes indefinitely long time for extracting its timing parameters that we terminated the experiment after two weeks without reaching any result. Speeding up the TPE run-time was the main motivation for proposing the slack-based merging algorithm, which it achieved successfully with improvements of up to 92% and 95% for the cases with finite and infinite buffer, respectively.

7.1.3 Communication-Aware Mapping

Communication-aware mapping is an algorithm for mapping mixed application models (dataflow application and independent real-time tasks) with timing constraints taking into account the communication cost of dataflow applications. The proposed algorithm is able to exploit parallelism in the application by allocating parallel paths on different cores. The main criteria for the allocation is to allocate paths with higher impact on the schedulability of the application first. Also, it models the messages (tokens) exchanged in dataflow applications as real-time tasks and hence, accounts for the communication cost. The experimental evaluation (Chapter 6) demonstrated four key results that concern both the communication-aware mapping algorithm and the complete approach. They are:

1. the importance of the communication cost and its impact on the number of allocated applications and the schedulability of the system. The results show that ignoring communication cost, as frequently done in existing work, allows mapping up to 76% more applications, which gives a wrong perception of the ability to map applications with timing constraints. These extra applications can be mapped, but they would not actually meet their timing constraints, which is a dangerous situation in real-time systems.

2. the direct relation between the number of allocated applications and the availability of the communication resources. The experimental results show, when the reservation bandwidth is equal to infinity, the maximum number of allocated applications is achieved in all input data sets, no matter which type of arbiter is used. However, the number of allocated applications reduces following the decrease of the reservation bandwidth.

3. the effect of the TDM arbiter interference on the number of allocated applications on the platform, which shows that an ideal arbiter allocates an average of 31% and 28% more applications over the actual TDM arbiter in 5% and 1% reservation bandwidth, respectively. This result sets the boundary for the possibility of using any type of arbiter based on bandwidth reservations, since it quantifies how much better a different type of arbiter could maximally do.

4. the ability of the proposed algorithm, particularly its main mapping heuristic called Sensitive-Path-First (SPF), to efficiently use platform resources and speed up the mapping process compared to well known bin-packing heuristics like First-Fit (FF). The results show that SPF surpasses FF in terms of number of allocated applications and run-time that reaches up to a maximum of 28% and 22%, respectively. This shows that there is no added overhead when using the SPF heuristic. On the contrary, it saves time.

Although the communication modelling of communication-aware mapping algorithm successfully allows to show the impact of communication on the schedulability of the system in terms of number of allocated applications, as experimentally demonstrated in Chapter 6, this model is subject to the limitations discussed in Section 6.4 and proposed as future work in Section 7.2.

7.2 Future Work

In every research, there is room for improvement. In this section, we discuss possible future directions for improving and extending our work.

7.2.1 Timing Parameter Extraction (TPE)

The TPE algorithm transforms a HSDF graph with multiple latency constraints into independent arbitrary-deadline real-time tasks. One of the main phases of this transformation process is an algorithm that traverses the HSDF graph to find all time-constrained paths, as explained in Section 5.5.1. This phase has a run-time that grows rapidly with the increase in size of the HSDF graph. In this thesis, we have addressed this problem by introducing the reduction algorithm called slack-based merging that reduces the complexity of HSDF dataflow graphs, speeding up the run-time of the TPE algorithm, demonstrated in the experimental results in Chapters 4 and 5. However, the nature of the algorithm has a downside of reducing fine-grained parallelism, which is a main benefit of the dataflow computational model. Also, in some cases its run-time adds an overhead on the overall design time, as shown in the experimental results in Chapter 5. A future direction to address this problem is to propose an algorithm to find only the *necessary* time-constrained paths in the HSDF graph that are critical for correct execution that satisfies timing constraints. This is because many time-constrained paths share the same actors (dependent paths). Once the timing parameters of an actor is derived from a high sensitivity time-constrained path, it is not mandatory to check the same actor for a lower sensitivity time-constrained path. This will speed-up the run-time of both the TPE algorithm and the complete approach. A possible start is the work of [Geilen, 2009], where the author proposes an SDF graph reduction technique based on Max-Plus algebra that transforms an SDF graph into a smaller HSDF graph with equivalent maximal throughput and latency, which is faster to analyse. This smaller HSDF graph can be used to find the *necessary* time-constrained paths in the graph that are critical for correct execution that satisfies timing constraints.

7.2.2 Communication-Aware Mapping

The communication-aware mapping algorithm is based on a communication model that accounts for the communication cost and its effect on the schedulability of the system without guaranteeing the communication feasibility. This means that it does not guarantee that the sum of total reservations on each link on the interconnect IN is less than or equal to 100%, as discussed previously in Section 6.4. A future direction is to improve the communication model to check the feasibility of the communication while mapping tasks on the platform. This requires accounting for different message routing mechanisms, i.e. X-Y routing, on the IN of the platform. Another possible extension is to consider a real-time communication model that incorporate fixed-priority for scheduling messages on the IN, such as the communication models discussed in [Nikolić et al., 2013, Shi and Burns, 2008]. Such a communication model will provide real-time guarantees for the messages flowing on the IN, allowing communication feasibility and satisfying timing constraints for both communication and the system.

References

Adapteva Epiphany multi core architecture. URL http://www.adapteva.com.

ARM Ltd.Cortex™-A Series. URL http://www.arm.com/products/processors/cortex-a/cortex-a17-processor.php.

Benny Akesson, Anna Minaeva, Premysl Sucha, Andrew Nelson, and Zdenek Hanzalek. An efficient configuration methodology for time-division multiplexed single resources. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 161–171, April 2015. doi: 10.1109/RTAS.2015.7108439.

Hazem Ismail Ali, Luís Miguel Pinho, and Benny Akesson. Critical-path-first based allocation of real-time streaming applications on 2d mesh-type multi-cores. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 201–208, Aug 2013. doi: 10.1109/RTCSA.2013.6732220.

Hazem Ismail Ali, Benny Akesson, and Luís Miguel Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Proceedings of the 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '15, pages 701–710, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8491-6. doi: 10.1109/PDP.2015.57. URL http://dx.doi.org/10.1109/PDP.2015.57.

Hazem Ismail Ali, Sander Stuijk, Benny Akesson, and Luís Miguel Pinho. Reducing the complexity of dataflow graphs using slack-based merging. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):24:1–24:22, January 2017. ISSN 1084-4309. doi: 10.1145/2956232. URL http://doi.acm.org/10.1145/2956232.

Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202, Dec 2001. doi: 10.1109/REAL.2001.990610.

Mohamed Bamakhrama and Todor Stefanov. Hard-real-time Scheduling of Data-dependent Tasks in Embedded Streaming Applications. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0714-7. doi: 10.1145/2038642.2038672. URL http://doi.acm.org/10.1145/2038642.2038672.

Mohamed Bamakhrama and Todor Stefanov. Managing Latency in Embedded Streaming Applications Under Hard-real-time Scheduling. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 83–92, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1426-8. doi: 10.1145/2380445.2380464. URL http://doi.acm.org/10.1145/2380445.2380464.

Arnab Banerjee, Pascal T. Wolkotte, Robert D. Mullins, Simon W. Moore, and Gerard J. M. Smit. An Energy and Performance Exploration of Network-on-Chip Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):319–329, March 2009. ISSN 1063-8210. doi: 10.1109/TVLSI.2008.2011232.

Sanjoy Baruah and Theodore P. Baker. Global EDF Schedulability Analysis of Arbitrary Sporadic Task Systems. In *2008 Euromicro Conference on Real-Time Systems*, pages 3–12, July 2008a. doi: 10.1109/ECRTS.2008.27.

Sanjoy Baruah and Theodore P. Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008b. ISSN 1573-1383. doi: 10.1007/s11241-007-9047-9. URL `http://dx.doi.org/10.1007/s11241-007-9047-9`.

Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 55(7):918–923, July 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.113.

Sanjoy Baruah and Joël Goossens. Scheduling real-time tasks: Algorithms and complexity. *Handbook of scheduling: Algorithms, models, and performance analysis*, 3, 2004.

Sanjoy Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, Nov 1990. ISSN 1573-1383. doi: 10.1007/BF01995675. URL `http://dx.doi.org/10.1007/BF01995675`.

Sanjoy Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90360-6. URL `http://www.sciencedirect.com/science/article/pii/0304397593903606`.

Sanjoy Baruah, Neil K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. ISSN 1432-0541. doi: 10.1007/BF01940883. URL `http://dx.doi.org/10.1007/BF01940883`.

Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, Jan-David Mol, Sander Stuijk, Valentin Gheorghita, and Jef van Meerbergen. Dataflow Analysis for Real-Time Embedded Multiprocessor System Design. In Peter Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research Book Series*, pages 81–108. Springer Netherlands, 2005. ISBN 978-1-4020-3453-4. URL `http://dx.doi.org/10.1007/1-4020-3454-7_4`.

Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, Feb 2008. doi: 10.1109/ISSCC.2008.4523070.

Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 983–987, March 2012. doi: 10.1109/DATE.2012.6176639.

Shuvra Shikhar Bhattacharyya, Praveen K. Murthy, and Edward Ashford Lee. Synthesis of Embedded Software from Synchronous Dataflow Specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):151–166, 1999. ISSN 0922-5773. doi: 10.1023/A:1008052406396. URL http://dx.doi.org/10.1023/A%3A1008052406396.

Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995. doi: 10.1109/ICASSP.1995.479579.

Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Qnoc: Qos architecture and design process for network on chip. *J. Syst. Archit.*, 50(2-3):105–128, February 2004. ISSN 1383-7621. doi: 10.1016/j.sysarc.2003.07.004. URL http://dx.doi.org/10.1016/j.sysarc.2003.07.004.

Adnan Bouakaz, Jean-Pierre Talpin, and Jan Vitek. Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications. In *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design*, ACSD '12, pages 183–192, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4709-1. doi: 10.1109/ACSD.2012.16. URL http://dx.doi.org/10.1109/ACSD.2012.16.

Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004. ISBN 0387231374.

John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 247–258, july 2007. doi: 10.1109/ECRTS.2007.81.

Donald D. Chamberlin. The "Single-assignment" Approach to Parallel Processing. In *Proceedings of the November 16-18, 1971, Fall Joint Computer Conference*, AFIPS '71 (Fall), pages 263–269, New York, NY, USA, 1971. ACM. doi: 10.1145/1479064.1479114. URL http://doi.acm.org/10.1145/1479064.1479114.

Houssine Chetto, Maryline Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Syst.*, 2(3):181–194, September 1990. ISSN 0922-6443. doi: 10.1007/BF00365326. URL http://dx.doi.org/10.1007/BF00365326.

Jack B. Copeland. *Colossus: The Secrets of Bletchley Park's Codebreaking Computers*. Oxford University Press, first edition edition, feb 2006.

Daily Autonomous Car News. WHEN WILL SELF DRIVING CARS BE READY - BOSCH SYSTEMS CONTROL PRESIDENT ANSWER, dec 2015. URL http://www.autonomous-car.com/2015/12/when-will-self-driving-cars-be-ready.html.

Morteza Damavandpeyma, Sander Stuijk, Marc Geilen, Twan Basten, and Henk Corporaal. Parametric throughput analysis of scenario-aware dataflow graphs. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 219–226, Sept 2012. doi: 10.1109/ICCD.2012.6378644.

Ali Dasdan and Rajesh K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, Oct 1998. ISSN 0278-0070. doi: 10.1109/43.728912.

Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978814. URL http://doi.acm.org/10.1145/1978802.1978814.

Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013. doi: 10.1109/HPEC.2013.6670342.

Marco Di Natale and John A. Stankovic. Dynamic end-to-end guarantees in distributed real time systems. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 216–227, 1994.

Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*, pages 97–101, mar 1998. doi: 10.1109/HSC.1998.666245.

Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43:25–59, 2009. ISSN 0922-6443. doi: 10.1007/s11241-009-9073-x. URL http://dx.doi.org/10.1007/s11241-009-9073-x.

Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 264–265, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278545. URL http://doi.acm.org/10.1145/1278480.1278545.

Glenn A. Elliott, Namhoon AKim, Jeremy P. Erickson, Cong Liu, and James H. Andersony. Minimizing response times of automotive dataflows on multicore. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2014. doi: 10.1109/RTCSA.2014.6910527.

Robert Ennals, Richard Sharp, and Alan Mycroft. Task Partitioning for Multi-core Network Processors. In *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, pages 76–90, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25411-0, 978-3-540-25411-9. doi: 10.1007/978-3-540-31985-6_6. URL http://dx.doi.org/10.1007/978-3-540-31985-6_6.

Gerald Estrin and Rein Turn. Automatic Assignment of Computations in a Variable Structure Computer System. *IEEE Transactions on Electronic Computers*, EC-12(6):755–773, Dec 1963. ISSN 0367-7508. doi: 10.1109/PGEC.1963.263559.

John D. Evans and Robert R. Kessler. A Communication-Ordered Task Graph Allocation Algorithm. Technical report, IEEE Transactions on Parallel and Distributed Systems, 1992.

Nathan Fisher, Sanjoy Baruah, and Theodore P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 10 pp.–127, 2006. doi: 10.1109/ECRTS.2006.30.

Sahar Foroutan, Benny Akesson, Kees Goossens, and Frédéric Petrot. A general framework for average-case performance analysis of shared resources. In *2013 Euromicro Conference on Digital System Design*, pages 78–85, Sept 2013. doi: 10.1109/DSD.2013.116.

Stefan K. Gehrig and Fridtjof J. Stein. Dead reckoning and cartography using stereo vision for an autonomous car. In *Intelligent Robots and Systems, 1999. IROS '99. Proceedings. 1999 IEEE/RSJ International Conference on*, volume 3, pages 1507–1512 vol.3, 1999. doi: 10.1109/IROS.1999.811692.

Marc Geilen. Reduction techniques for synchronous dataflow graphs. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 911–916, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3. doi: 10.1145/1629911.1630146. URL http://doi.acm.org/10.1145/1629911.1630146.

Amir Hossein Ghamarian, Marc Geilen, Twan Basten, Bart D. Theelen, Mohammad Reza Mousavi, and Sander Stuijk. Liveness and Boundedness of Synchronous Data Flow Graphs. In *2006 Formal Methods in Computer Aided Design*, pages 68–75, Nov 2006. doi: 10.1109/FMCAD.2006.20.

Amir Hossein Ghamarian, Sander Stuijk, Twan Basten, Marc Geilen, and Bart D. Theelen. Latency minimization for synchronous data flow graphs. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 189–196, Aug 2007. doi: 10.1109/DSD.2007.4341468.

Amir Hossein Ghamarian, Marc Geilen, Twan Basten, and Sander Stuijk. Parametric Throughput Analysis of Synchronous Data Flow Graphs. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 116–121, March 2008. doi: 10.1109/DATE.2008.4484672.

Manil Dev Gomony, Benny Akesson, and Kees Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1307–1312, San Jose, CA, USA, 2013. EDA Consortium. ISBN 978-1-4503-2153-2. URL http://dl.acm.org/citation.cfm?id=2485288.2485602.

Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 22(5):414–421, Sept 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.99.

Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, Andrew Nelson, and Shubhendu Sinha. Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow. *SIGBED Rev.*, 10(3):23–34, October 2013a. ISSN 1551-3688. doi: 10.1145/2544350.2544353. URL http://doi.acm.org/10.1145/2544350.2544353.

Sven Goossens, Benny Akesson, and Kees Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 525–530, March 2013b. doi: 10.7873/DATE.2013.118.

Sven Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Sept 2013c. doi: 10.1109/CODES-ISSS.2013.6658989.

Micheal Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.41.

Jiani Guo and Laxmi Narayan Bhuyan. Load balancing in a cluster-based web server for multimedia applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(11):1321–1334, November 2006. ISSN 1045-9219. doi: 10.1109/TPDS.2006.159. URL http://dx.doi.org/10.1109/TPDS.2006.159.

Joost P. H. M. Hausmans, Maarten H. Wiggers, Stefan J. Geuns, and Marco Bekooij. Dataflow Analysis for Multiprocessor Systems with Non-starvation-free Schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, M-SCOPES '13, pages 13–22, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2142-6. doi: 10.1145/2463596.2463603. URL http://doi.acm.org/10.1145/2463596.2463603.

Steve Heath. 1 - What is an embedded system? In Steve Heath, editor, *Embedded Systems Design (Second Edition)*, pages 1–14. Newnes, Oxford, second edition edition, 2002. ISBN 978-0-7506-5546-0. doi: 10.1016/B978-075065546-0/50002-5. URL http://www.sciencedirect.com/science/article/pii/B9780750655460500025.

Paul Hoffman. *The Man Who Loved Only Numbers: The Story of Paul Erdos and the Search for Mathematical Truth*. Biography-science. Hyperion Books, 1999.

Jingcao Hu and Radu Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10688–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2. URL http://dl.acm.org/citation.cfm?id=789083.1022804.

Damir Isović and Gerhard Fohler. Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints. In *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*, RTSS'10, pages 207–216, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0900-2. URL http://dl.acm.org/citation.cfm?id=1890629.1890656.

Manuel Jiménez, Rogelio Palomera, and Isidoro Couvertier. *Introduction to Embedded Systems : Using Microcontrollers and the MSP430*. Springer-Verlag New York, 1 edition, 2014. doi: 10.1007/978-1-4614-3143-5.

Mathai Joseph. *Real-time systems - specification, verification and analysis*. 1996.

Ben Kao and Hector Garcia-Molina. Deadline assignment in a distributed soft real-time system. *Parallel and Distributed Systems, IEEE Transactions on*, 8(12):1268–1274, 1997.

Richard M. Karp and Raymond E. Miller. Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing. *SIAM Journal on Applied Mathematics*, 14(6):pp. 1390–1411, 1966. ISSN 00361399. URL http://www.jstor.org/stable/2946247.

Kenya Tech News. Top Ten Smartphones In Kenya Quarter 3 - July To September 2015, Sep 2015. URL http://www.kachwanya.com/2015/10/06/top-ten-smartphones-in-kenya-quarter-3-july-to-september-2015/.

Minsoo Kim, Joonho Song, Dohyung Kim, and Shihwa Lee. H.264 decoder on embedded dual core with dynamically load-balanced functional paritioning. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3749–3752, Sept 2010. doi: 10.1109/ICIP.2010.5653439.

C. Mani Krishna. *Real-Time Systems*. McGraw-Hill Higher Education, 1st edition, 1996. ISBN 0070570434.

Edward Ashford Lee. Consistency in dataflow graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(2):223–235, Apr 1991. ISSN 1045-9219. doi: 10.1109/71.89067.

Edward Ashford Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987a. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987b. ISSN 0018-9340. doi: 10.1109/TC.1987.5009446. URL http://dx.doi.org/10.1109/TC.1987.5009446.

John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171, Dec 1989. doi: 10.1109/REAL.1989.63567.

Hennadiy Leontyev and James H. Anderson. A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 191–200, july 2008. doi: 10.1109/ECRTS.2008.22.

Joseph Y.-T. Leung and M. L. Merrill. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11:115–118, 1980.

Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

Min Li, Hui Wang, and Ping Li. Tasks mapping in multi-core based system: hybrid ACO GA approach. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 1, pages 335–340 Vol.1, Oct 2003. doi: 10.1109/ICASIC.2003.1277556.

Giuseppe Lipari and Enrico Bini. On the Problem of Allocating Multicore Resources to Real-Time Task Pipelines. *4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS '11)*, November 2011.

C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973.

Di Liu, Jelena Spasic, Jiali Teddy Zhai, Todor Stefanov, and Gang Chen. Resource optimization for CSDF-modeled streaming applications with latency constraints. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014. doi: 10.7873/DATE.2014.201.

Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN 9780130996510. URL https://books.google.pt/books?id=855QAAAAMAAJ.

Jian Liu, Li-Rong Zheng, and Hannu Tenhunen. Interconnect Intellectual Property for Network-on-chip (NoC). *J. Syst. Archit.*, 50(2-3):65–79, February 2004. ISSN 1383-7621. doi: 10.1016/j.sysarc.2003.07.003. URL `http://dx.doi.org/10.1016/j.sysarc.2003.07.003`.

Yi Liu, Xin Zhang, He Li, and Depei Qian. Allocating Tasks in Multi-core Processor based Parallel System. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, pages 748–753, Sept 2007. doi: 10.1109/NPC.2007.26.

Virginia Mary Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comput.*, 37(11):1384–1397, November 1988. ISSN 0018-9340. doi: 10.1109/12.8704. URL `http://dx.doi.org/10.1109/12.8704`.

José María López, José Luis Díaz, and Daniel F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Syst.*, 28(1):39–68, October 2004. ISSN 0922-6443. doi: 10.1023/B:TIME.0000033378.56741.14. URL `http://dx.doi.org/10.1023/B:TIME.0000033378.56741.14`.

Angelo Kuti Lusala and Jean-Didier Legat. Combining sdm-based circuit switching with packet switching in a NoC for real-time applications. In *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, pages 2505–2508, May 2011. doi: 10.1109/ISCAS.2011.5938113.

Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

Perng-Yi Richard Ma, Edward Y. S. Lee, and Masahiro Tsuchiya. A task allocation model for distributed computing systems. *IEEE Trans. Comput.*, 31(1):41–47, January 1982. ISSN 0018-9340. doi: 10.1109/TC.1982.1675884. URL `http://dx.doi.org/10.1109/TC.1982.1675884`.

Paul Marchal, Diederik Verkest, Adelina Shickova, Francky Catthoor, Frédéric Robert, and Anthony Leroy. Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs. In *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, pages 81–86, Sept 2005. doi: 10.1145/1084834.1084858.

Timothy G. Mattson, R.F. Van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Gregory Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer's View. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, nov. 2010. doi: 10.1109/SC.2010.53.

Mehdi Modarressi, Hamid Sarbazi-Azad, and Mohammad Arjomand. A hybrid packet-circuit switched on-chip network based on SDM. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 566–569, April 2009. doi: 10.1109/DATE.2009.5090728.

Orlando Moreira, Frederico Valente, and Marco Bekooij. Scheduling Multiple Independent Hard-real-time Jobs on a Heterogeneous Multiprocessor. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1. doi: 10.1145/1289927.1289941. URL `http://doi.acm.org/10.1145/1289927.1289941`.

Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, February 1993. ISSN 0018-9162. doi: 10.1109/2.191995. URL http://dx.doi.org/10.1109/2.191995.

Borislav Nikolić, Hazem Ismail Ali, Stefan Markus Petters, and Luís Miguel Pinho. Are virtual channels the bottleneck of priority-aware wormhole-switched noc-based many-cores? In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 13–22, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2058-0. doi: 10.1145/2516821.2516845. URL http://doi.acm.org/10.1145/2516821.2516845.

Dong-Ik Oh and Theodore P. Baker. Utilization Bounds for N-Processor Rate MonotoneScheduling with Static Processor Assignment. *Real-Time Syst.*, 15(2):183–192, September 1998. ISSN 0922-6443. doi: 10.1023/A:1008098013753. URL https://doi.org/10.1023/A:1008098013753.

Sung-Heun Oh and Seung-Min Yang. A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, RTCSA '98, pages 31–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-9209-X. URL http://dl.acm.org/citation.cfm?id=600376.828687.

Victor Pankratius, Ali Jannesari, and Walter F. Tichy. Parallelizing Bzip2: A Case Study in Multicore Software Engineering. *Software, IEEE*, 26(6):70–77, Nov 2009. ISSN 0740-7459. doi: 10.1109/MS.2009.183.

Dar-Tzen Peng and Kang G. Shin. Static allocation of periodic tasks with precedence constraints in distributed real-time systems. In *Distributed Computing Systems, 1989., 9th International Conference on*, pages 190–198, jun 1989. doi: 10.1109/ICDCS.1989.37947.

Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-chip. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 63–72, New York, NY, USA, 2003. ACM. ISBN 1-58113-676-5. doi: 10.1145/951710.951721. URL http://doi.acm.org/10.1145/951710.951721.

Mahendra PratapSingh and Manoj Kumar Jain. Evolution of Processor Architecture in Mobile Phones. *International Journal of Computer Applications*, 90(4):34–39, March 2014. doi: 10.5120/15564-4339.

Peter Puschner and Alan Burns. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2):115–128, 2000. ISSN 1573-1383. doi: 10.1023/A:1008119029962. URL http://dx.doi.org/10.1023/A:1008119029962.

Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 287–296, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2058-0. doi: 10.1145/2516821.2516836. URL http://doi.acm.org/10.1145/2516821.2516836.

Qualcomm. Qualcomm® Snapdragon™ 820 processor, July 2016. https://www.qualcomm.com/products/snapdragon/processors/820.

Krithi Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, Apr 1995. ISSN 1045-9219. doi: 10.1109/71.372795.

Ismael Ripoll, Alfons Crespo, and Aloysius K. Mok. Improvement in Feasibility Testing for Real-time Tasks. *Real-Time Syst.*, 11(1):19–39, July 1996. ISSN 0922-6443. doi: 10.1007/BF00365519. URL http://dx.doi.org/10.1007/BF00365519.

Jorge E Rodrigues. A GRAPH MODEL FOR PARALLEL COMPUTATIONS. Technical report, Cambridge, MA, USA, 1969.

Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. pages 217–226, Nov 2011. ISSN 1052-8725. doi: 10.1109/RTSS.2011.27.

Hrishikesh Salunkhe, Orlando Moreira, and Kees van Berkel. Mode-controlled dataflow based modeling amp; analysis of a 4g-lte receiver. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014. doi: 10.7873/DATE.2014.225.

Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. An Analysis of On-chip Interconnection Networks for Large-scale Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 7(1):4:1–4:28, May 2010. ISSN 1544-3566. doi: 10.1145/1756065.1736069. URL http://doi.acm.org/10.1145/1756065.1736069.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360617. URL http://doi.acm.org/10.1145/1360612.1360617.

Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 161–170, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3098-7. URL http://dl.acm.org/citation.cfm?id=1397757.1397996.

Michael W. Shields. *Karp and Miller Computation Graphs*, pages 75–92. Springer London, London, 1997. ISBN 978-1-4471-0933-4. doi: 10.1007/978-1-4471-0933-4_8. URL http://dx.doi.org/10.1007/978-1-4471-0933-4_8.

Firew Siyoum, Marc Geilen, Orlando Moreira, Rick Nas, and Henk Corporaal. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *2011 International Symposium on System on Chip (SoC)*, pages 14–21, Oct 2011. doi: 10.1109/ISSOC.2011.6089222.

Brinkley Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, Pittsburgh, PA, USA, 1990. AAI9107570.

Marco Spuri. Analysis of Deadline Scheduled Real-Time Systems. Technical report, 1996.

Marco Spuri and John A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *Computers, IEEE Transactions on*, 43(12):1407–1412, dec 1994. ISSN 0018-9340. doi: 10.1109/12.338100.

Sundararajan Sriram and Shuvra Shikhar Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

Sundararajan Sriram and Edward Ashford Lee. Determining the Order of Processor Transactions in StaticallyScheduled Multiprocessors. *J. VLSI Signal Process. Syst.*, 15(3):207–220, March 1997. ISSN 0922-5773. doi: 10.1023/A:1007956226232. URL http://dx.doi.org/10.1023/A:1007956226232.

John A. Stankovic and Krithi Ramamritham, editors. *Tutorial: hard real-time systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

Sander Stuijk, Marc Geilen, and Twan Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 276–278, june 2006. doi: 10.1109/ACSD.2006.23.

Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. In *2007 44th ACM/IEEE Design Automation Conference*, pages 777–782, June 2007.

Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002. ISBN 0130661023.

TESLA. TESLA Motors, July 2016. https://www.tesla.com/.

Bart D. Theelen, Marc Geilen, Twan Basten, Jeroen P. M. Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, pages 185–194, July 2006. doi: 10.1109/MEMCOD.2006.1695924.

András Vajda. *Multi-core and Many-core Processor Architectures*, pages 9–43. Springer US, Boston, MA, 2011. ISBN 978-1-4419-9739-5. doi: 10.1007/978-1-4419-9739-5_2. URL http://dx.doi.org/10.1007/978-1-4419-9739-5_2.

Jelte Peter Vink, Kees van Berkel, and Pieter van der Wolf. Performance analysis of soc architectures based on latency-rate servers. In *2008 Design, Automation and Test in Europe*, pages 200–205, March 2008. doi: 10.1109/DATE.2008.4484686.

Yu Wang, Kai Zhou, Zhonghai Lu, and Huazhong Yang. Dynamic TDM virtual circuit implementation for NoC. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 1533–1536, Nov 2008. doi: 10.1109/APCCAS.2008.4746325.

David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.4378780.

Maarten H. Wiggers, Marco Bekooij, and Gerard J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *2007 44th ACM/IEEE Design Automation Conference*, pages 658–663, June 2007.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank

Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL http://doi.acm.org/10.1145/1347375.1347389.

Fengxiang Zhang and Alan Burns. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, Sept 2009a. ISSN 0018-9340. doi: 10.1109/TC.2009.58.

Fengxiang Zhang and Alan Burns. Improvement to quick processor-demand analysis for edf-scheduled real-time systems. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 76–86, July 2009b. doi: 10.1109/ECRTS.2009.20.

Linlin Zhang, Virginie Fresse, Mohammed A. S. Khalid, Dominique Houzet, and Anne-Claire Legrand. Evaluation and Design Space Exploration of a Time-Division Multiplexed NoC on FPGA for Image Analysis Applications. *CoRR*, abs/1002.1881, 2010. URL http://arxiv.org/abs/1002.1881.

Gouqing Zhou and Jun Wu. Unmanned Aerial Vehicle (UAV) data flow processing for natural disaster response. *ASPRS 2006*, May 2006.

Haitao Zhu, Steve Goddard, and Matthew B. Dwyer. Response Time Analysis of Hierarchical Scheduling: The Synchronized Deferrable Servers Approach. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 239–248, Nov 2011. doi: 10.1109/RTSS.2011.29.