Agile Heterogeneous Computing: Variable Platform Heterogeneous Computing and its Design Flow

Avishek Chakraborty

Agile Heterogeneous Computing: Variable Platform Heterogeneous Computing and its Design Flow

by

Avishek Chakraborty
Bachelor of Computer Science (Hons 1)

A thesis submitted for the degree of
Doctor of Philosophy
February 2020

University of
South Australia

Computational Learning Systems Laboratory
School of Information Technology & Mathematical Sciences
Division of Information Technology, Engineering and the Environment

Abstract

Agile heterogeneous embedded computing refers to the consideration of platform architectures as a first class design variable in deploying applications on heterogeneous platforms, which constitute one or more of the following compute engines: FPGA, GPU and CPU. Design flows for agile heterogeneous computing are in the early phases of development. Few state of the art design flows that consider variable platform architectures have the following drawbacks: (1) restricted template-based representations, which cannot express all possible architectural topologies, (2) limited support for concurrent tasks on all three compute engines and (3) design space exploration approaches indirectly consider variable platform architectures, because they are derived from fixed platform design flows, where different platform architectures are iteratively evaluated without closely considering the application's computational needs.

This research explores solutions to these three major issues that have yet to be addressed in agile heterogeneous computing design flows. The solutions to these issues are encapsulated into a new design flow called *agile heterogeneous computing flow* (AhcFlow) that can support both design space exploration and deployment. In order to realise AhcFlow, a new representation to express variable platform architectures called *parameterised platform graph* (PPG) is conceived, a new intermediate data structure called *augmented synchronous dataflow* (ArcSDF) is created and a new design space exploration algorithm called *agile mapping and scheduling algorithm* (AMS) is developed. PPG is a constraint based representation, which unlike template-based representations has higher expressive capabilities to represent the different topologies of agile heterogeneous computing. ArcSDF is a new dataflow based intermediate data structure, which is built on the well-known *synchronous dataflow graphs* (SDF) to express the architectural decisions together with the application. ArcSDF extends the capabilities of the dataflow paradigm to incorporate computation resource analysis and capital cost analysis. Due to these extra analysis capabilities, ArcSDF is used within design space exploration. The design space exploration algorithm is AMS that considers the variable platform architecture in a fully integrated way together with mapping and scheduling decisions.

A prototype of the new design flow (AhcFlow) has been created and

shown to be valid both for an exhaustive number of synthetic test cases and for a large real life embedded multi-object visual tracking application. The new analysis capabilities of ArcSDF have been validated through its integration within the design flow prototype. The prototype also consists of a deployment module, which is used to validate the design space exploration predictions with the actual deployment results. The results from the real life tracking application show that design space exploration estimates closely match the deployment results. The new mapping and scheduling algorithm, that also makes architecture topology decisions, has shown to be competitive with published results of applications implemented manually to hand crafted architectures.

Contents

List of Figures

List of Tables

List of abbreviations

$AhcFlow$	Agile heterogeneous computing flow
AMS	Agile mapping and scheduling
AND	Average normalised difference
$ArcSDF$	Architecture augmented synchronous dataflow
CE	Compute engine
ce_{comm}	Communication factor
ce_{global}	Global factor
ce_{local}	Local factor
CZ	Compute zone
DAG	Dataflow acyclic graph
$DCER$	Dynamic compute engine rank
EFT	Earliest finish time
EH	Exploration history
EST	Earliest start time
ETS	Earliest time slot
$FASG$	Final ArcSDF generation
$GPAU$	Global platform architecture update
$HEFT$	Heterogeneous earliest-finish-time
$HSDF$	Homogeneous synchrohous dataflow

$LPAE$	Local platform architecture expansion
MT	Mapping threshold
PI	Platform architecture instance
PPG	Parameterised platform graph
RE	Resource edge
RF	Resource factor
$rHEFT$	Resouorce-based heterogeneous earliest-finish-time
rpv	Resource performance value
SDF	Synchronous dataflow
SEF	Shape estimating filter
SLR	Schedule length ratio
slr	Schedule length ratio
$Y\text{-}chart$	Y-chart design methodology

Declaration

I declare that:

This thesis presents work carried out by myself and does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; to the best of my knowledge it does not contain any materials previously published or written by another person except where due reference is made in the text; and all substantive contributions by others to the work presented, including jointly authored publications, are clearly acknowledged.

Avishek Chakraborty
11^{th} May 2020

Acknowledgements

I would like to thank my supervisor, Dr David Kearney for his wealth of knowledge, encouragement and support. I consider myself lucky to have him as my supervisor straight out of undergraduate school, especially while trying to get accustomed to a new country. David has given me a lot of freedom to choose my research topic, while providing essential direction and guidance. His multifaceted research outlook closely aligns with the direction of modern computing platform architectures and this has had a great influence on my thesis work and research interests. I would also like to thank David for his contribution to my conference travel expenses, where I was able to engage with fellow researchers and exchange ideas.

Next, I would like to thank my associate supervisor, Dr Sebastien Wong, for being a steady source of support and encouragement. He introduced me to an ongoing multi-object visual tracking project, which eventually provided me with a real-world application for the case-study of my thesis. This further helped me with the lab equipment for my research work. I am also thankful to Sebastien for creating the opportunity to work on a publication with Dr Victor Stamatescu. It was a great learning experience. Although Victor was not directly involved with my thesis, he constantly encouraged me, and I am grateful to have had him around the lab. I would further like to acknowledge Sebastien and Victor for their suggestions to improving the thesis document.

During the initial phase of my PhD, Dr Stewart Von Itzstein, Dr Grant Wigley and Dr Ivan Lee were enormous sources of help. I am thankful to Stewart for the opportunities to teach various computer science courses at the University. Although teaching the courses did not contribute to my research work, they were valuable experiences and allowed me to maintain proficiency on different computer languages. Grant provided valuable guidance with lab equipment, travel arrangements and advice to efficiently conduct research. I am also grateful for all the help and support I have received from the Reconfigurable Computing Laboratory, which has now grown as Computational Learning Systems Laboratory. My research was financially supported by University of South Australia President's Scholarship, school of ITMS, later on by University of South Australia Postgraduate Award and Defence Science and Technology Group (previously DSTO), which I gratefully acknowledge.

I would like to thank all my laboratory peers for their company and maintaining the research environment which has obliquely influenced my thesis. This gives me the opportunity to thank Dr Anthony Milton, Dr Kelly Foreman, Dr Adam Gatt, Wanjung Zhai and Samya Bagchi. They were always present to listen to my research ideas. I consider myself fortunate to have been surrounded by them. I am thankful to Anthony for taking time out to read my thesis.

This PhD journey is close to 5 years of full-time work and fuelled by blood, sweat and tears, would not have been possible without the constant support of my friends and family. I am grateful to Akash, Karan, Arindam, Daniel, Melissa, William, Shaurya, Krishna, Yogesh and Annanya for the fun weekends and always ensuring that there is a positive source of distraction. I am also thankful to Dr Santanu Roy, Dr Anuttam Patra and Dr Debdeep Banerjee for having inspired me to embark on this journey of research.

My special thanks go to my brother, Bubun who kept reminding me the importance of having fun and also for maintaining a constant accommodation space for me in Adelaide. There are many others, but not possible to mention all of them here, so I end this with the following dedication.

To my grandfather, Santi Gopal Das

To my mother, Sonali

To my father, Asoke

Chapter 1

Introduction

Contents

1.1 Motivation

Parallel hardware is now the mainstream for high performance computing. This has been driven by power dissipation limits on clock rates in single core microprocessors. A side effect of the parallel revolution is the fragmentation of the architectures of compute engines that implement concurrency. In the mainstream there are now three major architectures - shared memory multicore (CPU), graphics processing units (GPU), and field programming gate arrays (FPGA). While each has been created in response to a particular need, two or more are often used in combination to exploit the different types of parallelisms within an application [1]. The motivation to use different compute architectures for better performance and resource utilisation dates back to the 1970s [2] when a combination of different processing architectures started to be used for compute intensive applications. In the present time an influence of this motivation has given birth to the field of modern day heterogeneous computing, where applications are spread over two or more compute engine types. In particular there are many applications that use three compute engines - FPGA, GPU, and CPU [3–5]. Such heterogeneous platforms often have the flexibility to change their configuration and the quantity of the compute engines easily. This flexibility and readiness to change the overall platform architecture is referred in this research work as agile heterogeneous computing.

The implementation of algorithms on agile heterogeneous platforms is much more complex than on multicore CPUs alone. The design space is much greater, there are often multiple languages and tools to contend with and the prediction of performance depends on both the nature of the compute engines and how they are configured into a platform. It may not be clear to the application developer on which compute engine each component should be implemented and there is a combinatorial explosion in the choices for implementation of components on multiple compute engines. The diversity of languages and tools means that compilation of an algorithm from a single language source is problematic from a performance point of view. More importantly to evaluate the design space, each component of an algorithm will require compute engine specific implementations because of the differences in the nature of the architecture, the number of compute resources available on a specific instance of the architecture, and the memory availability of the overall architecture. On an FPGA there is a further a wide choice in the actual architecture employed. The influence of the platform inter-connection topology is likely to be significant. The communication channels between CPU and GPU may slow a well performing GPU kernel down. Performance will be influenced by the call pattern of the kernels in the algorithm. Hence, the often-used single acceleration library approach used in CPUs does not scale up to an agile heterogeneous environment.

The above challenges can be summarised into three broad problems: finding the best compute engine for each component of the application, finding the best communication sequence among other components and finding the right mix of compute

engines to constitute the platform. These three problems can be characterised as design space exploration. In addition, there are challenges associated with the implementation of the various components of the application based on the decisions made during design space exploration, which is a deployment problem. Design space exploration and deployment are usually handled by a design flow.

Design flows for fixed architecture heterogeneous platforms are well established. These design flows can be adapted to variable architecture agile heterogeneous applications by placing the fixed architecture design flow, as a black box, inside an iteration whereby the architecture is changed at each step. However this approach means that there is no intimate interaction between the algorithms used in fixed architecture design flow (which effectively remain a black box) and changes in the architecture. Opportunities for simultaneous adjustment of mapping and scheduling with architectural changes may be lost. In the new design flow proposed in this work these opportunities are explored. The parallel computing resources of the heterogeneous platform and within the compute engines that make up the platform can form the basis of resource aware versions of mapping and scheduling algorithms previously developed for fixed architecture design flows. In this new concept of a design flow for agile heterogeneous computing; a fixed platform definition is no longer relevant as an input. Instead a new generalised conception of an agile platform definition is required. The new design flow will also require a new integrated representation of the application, architecture and the mapping and scheduling decisions. This thesis explores the feasibility and means of creating all the required elements of a new design flow for agile heterogeneous computing.

1.2 Thesis structure

The thesis is organised as follows. Chapter 2 presents the current research results relevant to concerning design flows for agile heterogeneous computing. In Chapter 3, the research questions and a methodology for answering them are presented. Chapter 4 presents an overview of *agile heterogeneous computing flow* (AhcFlow), which is the proposed new design flow for agile heterogeneous computing. In Chapter 5, the representations of inputs, called the *Parameterised Platform Graph* (PPG), and the internal data structures, called *Architecture augmented Synchronous Dataflow* (ArcSDF), for the new design flow are described. In Chapter 6, a new design space exploration algorithm, called the *Agile Mapping and Scheduling* (AMS) algorithm is presented and evaluated. In Chapter 7, the design flow is evaluated with a real life visual tracking algorithm (CACTuS) and the details of the deployment are explained. The final chapter concludes this thesis and proposes future work.

1.3 Research contributions

The primary contributions of this dissertation are summarised as follows:

- A new design flow called heterogeneous computing flow (AhcFlow) is created that closely combines architectural exploration with the mapping and scheduling decisions. This is achieved by first generalising the Y-chart design methodology [6] (section 2.3.1) to enable the use of platform constraints rather than a fixed platform architecture. Secondly, an intermediate data-structure (ArcSDF) that can express agile design decisions is developed. Finally, a design space exploration strategy based on platform constraints is created.

  The research contributions associated with the new design flow are:

  - Exploration of platform architectures at a the same time as mapping and scheduling decisions of the application-algorithm.

  - Separation of concerns between design space exploration and deployment by using the new intermediate data-structure (ArcSDF) that can express design decisions and can be analysed for performance and capital cost prior to deployment.

  - Automation for the creation of a runtime on the designated platform architecture for deployment.

- A novel way to represent platform constraints at a higher abstraction, called Parameterised platform graph (PPG). PPG consists of two tiers. PPG tier1 represents the platform constraints, whereas PPG tier2 expresses an initial valid platform architecture instance. PPG tier1 constraints the design space exploration algorithm, which starts with the PPG tier2 instance for the selection of the final platform architecture to be used for deployment.

  The research contributions associated with PPG include:

  - A high-level representation that incorporates the necessary details for design space exploration of applications on agile heterogeneous platform architecture.

  - The expression of the platform constraints and the representation of instances of platform architectures possible within those constraints.

- A new data-structure called architecture augmented synchronous dataflow (ArcSDF) is developed to represent design decisions. The research contributions related with ArcSDF are associated with four new constructs.

- A compute zone which groups actors for sequential execution and also expresses the resource consumption of the group.

- Interfaces which are the ports through which compute zones communicate.

- Resource edges represents the resource dependencies between compute zones.

- Control actors which represent the common design patterns whereby the execution of an actor on one compute engine is managed from another compute engine. For example the launch and execution of a GPU kernel is initiated from a code running on the CPU.

Finally in addition to throughput analysis ArcSDF enables the evaluation of maximum resource usage, time slots for scheduling a new actor and optimisation of the number of compute zones used.

- The agile mapping and scheduling (AMS) algorithm. AMS is a combination of a resource aware HEFT Topcuoglu et. al. 1999 [7] Heterogeneous Earliest-Finish-Time (HEFT) and the Sakellariou et. at 2007 [8] Gain/Loss algorithm. Research contributions associated with AMS include:

  - A new version of HEFT that enables concurrent execution of actors on a compute engine and supports consideration of resources.

  - A further enhancement of HEFT that allows specialised connectivity topologies including restricted connectivity between compute engines.

  - Adaptation of the Gain/Loss algorithm to use capital cost metrics.

  - The incorporation of the Gain/Loss algorithm with capital cost metric into a final enhanced version of HEFT.

Chapter 2

Literature review

Contents

2.1 Introduction

In this chapter, the current research literature in relation to the challenges of design flows for agile heterogeneous computing is examined. The literature selected for the survey is based on its likely impact on the design flow itself, the application representation, and algorithms for the simultaneous optimization of architectural topology with mapping and scheduling.

This chapter is organised as follows. Firstly, the landscape of heterogeneous computing as it exists today is considered. Then, in the next section the published design flows for heterogeneous computing are explored. Then, the representations of architectural decisions that are made during design space exploration are reviewed. Finally, the mapping and scheduling algorithms that are relevant to heterogeneous computing are surveyed. In this section, proposed algorithms dealing with agile architectures are also reviewed. The outcome of this survey shows that there are gaps in the research literature relating to design flows, representations and heuristics for architecture, scheduling and mapping exploration.

2.2 Agile Heterogeneous computing

Heterogeneous computing has evolved from its early days of a few slightly dissimilar microprocessors to its present day where three disparate compute engines (FPGA, GPU and CPU) are in use. Due to these recent developments, there are new challenges in designing applications. In this section, notable and recent literature on heterogeneous computing are reviewed with the aim of highlighting the new challenges. This is achieved in two steps. At first, a historic perspective is presented to show how the field of heterogeneous computing has evolved to include these compute engines. In this part, the challenges associated with classical heterogeneous computing are shown and how design flows are used to mitigate them are highlighted. Then, in the second part, present day heterogeneous computing with different compute engines is considered. It is shown that compute engines can be included or excluded to form the platform architecture. There are also decisions to be made as to how compute engines are connected to one another. Therefore, along with the usual challenges of heterogeneous computing, a designer needs to consider variable compute platform architecture. Other challenges of managing simultaneous execution of multiple tasks on a compute engine and handling different tool-sets for deployment, are also discussed.

2.2.1 Historic perspective

The classical era of heterogeneous computing reflects the current motivation behind using more than one type of processors. A very early reference to heterogeneous computing was by Liu and Yang at 1974 [2]. In this paper the term "heterogeneous" refers to processors with different speed being combined together. Heterogeneous computing was studied as a theoretical discipline and the concepts were simulated on the existing computers that used one or more similar microprocessors. The physical existence of heterogeneous computing can be traced from the early days of supercomputers. Menascé and Almeida 1990 [9] states that the reason for heterogeneity in supercomputing was use of different types of processors to boost performance in a cost efficient way through dedicated processors for certain parts of the application.

Ercegovac 1988 [10] was an early advocate for evaluating different heterogeneous architectures to achieve cost effective performance. Menascé and Almeida [9] developed further a cost performance analysis of computers available during the early 1990s such as the NCUBE [11] and the connection machines [12].

Menascé and Almeida [9] treats the elements of their ideal model of heterogeneous computing as CPUs with different throughput. The idea of heterogeneity was further expanded to include algorithms that run partially on SIMD and MIMD processors by Watson et al. 1993 [13]. The authors further promoted a taxonomy called EM3 which separated the concepts of individual machine architectures and the way these architectures were connected. However, at this time the heterogeneity of individual machine architectures was limited to variation of what would now be single core and multicore CPUs. Although the potential benefit of heterogeneous computing was appreciated, heterogeneous computing was not a commercial success due to the constant progress of Moore's Law. This started changing from the mid 2000, when alternate sources of acceleration started to emerge in response to power responses of a single core. In this way the field of heterogeneous computing was re-imagined.

Heterogeneous computing also evolved with the advent of FPGAs [14]. However, the initial motivation of FPGAs was not performance, rather they were used as a mechanism to prototype ASICs. But with the demand for more parallel computation, FPGAs were used for performance acceleration.

Around mid-1990s graphics cards appeared in volumes for personal computers [15, 16]. It was not long before there were attempts to use the graphics commands supported by these card to do general purpose computing [17, 18], which later was introduced under the term general purpose graphics processing unit computing - GPGPU [19].

A major factor in the rapid development of multicore CPUs, GPUs and FPGAs to become the primary constituents for heterogeneous computing as compared to

CPUs was power [20]. It is well known that by 2003 the clock rates on CPUs had plateaued [21] and the need to consider massively parallel platforms to avoid power issues had become the primary driver for heterogeneous computing engines.

Although the compute engines FPGA, GPU and multicore CPU originated for different applications, there is a growing trend to use them together and the reason for this is the same that started heterogeneous computing, which is utilising different compute elements effectively for maximising performance [2, 9]. Heterogeneous computing platforms that are constituted of multicore CPU, GPU, and FPGA are reviewed in the next subsection.

2.2.2 Agile heterogeneous computing challenges

In this subsection, the challenges of designing applications for heterogeneous platforms that incorporates the three compute engines; CPU, GPU and FPGA are reviewed. These include the selection of the platform architecture, the management of design complexity using a design flow, mapping and scheduling of tasks with the compute engine resources, and the diverse software and hardware skills that might be required for the implementation.

Silva. et. al. 2013 [22] experiments with dissimilar platform architectures to study the performance variations of a pedestrian recognition application. The application algorithm is based on a version of histogram oriented gradient (HOG) detector. These platform architectures consists of all three types of compute engines. The FPGA and GPU are connected through standard PCIe slots with the CPU. Four different platform architectures are used. These variations are formed by using two different models of GPU and by changing the number of FPGAs. Although the primary motive of measuring performances on dissimilar platform architectures is to show the benefits of heterogeneous computing with distinct compute engines, the authors have implicitly revealed the importance of selecting a suitable platform architecture for a specific application algorithm.

Similar experimental studies [23, 24] though with limited number of compute engines, one of each kind, are used to find the best platform architecture for an electro-physical and n-body simulation application algorithms. Again, the prime finding of these research articles are performance gains obtained through the collaboration of three distinct compute engines, but they also shed light on the rigorous experimentation necessary to find a suitable platform architecture. In both these studies, a single application algorithm is targeted, which enables it to be manually profiled and partitioned and mapped on the available compute engines.

Current commodity platform architectures feature communication links such as PCIe slots, where readily available compute engines can be added and removed. However, these links can also be considered as the bottleneck [25] that restricts

complete utilisation of heterogeneous platforms. Every time two compute engines that are not a CPU needs to pass data, it often takes place through CPU. These bottlenecks can be tackled through direct communication links between compute engines [26, 27] that bypass the CPU, such as NVLink [28] for some GPU models from NVIDIA. Accordingly many more options for the platform architecture topology will be feasible in the future. Thus platform architecture will increasing become a design variable which can be optimized.

Quite recently, Chung. et. al. 2018, [29] have proposed servers with flexible resource swapping options called composable systems. They provide a guided way to create platform architectures based on the requirements of an application. They have mentioned the necessity of selecting appropriate platform architecture, but it has been left to the designer.

Given these observations it is clear that the design process for agile heterogeneous computing is likely to be complex. The way similar complexities have been managed in the past is through a design flow [30–33]. The challenge is to come up with a design flow that allows the platform architecture to become a design variable and not just a static platform architecture given at the start of the design process.

Another characteristic of agile heterogeneous computing is that several actors [1](components of an application) can be mapped to a single compute engine (like a GPU). A scheduling decision needs to be made as to when to run actors concurrently. This decision will interact with resource allocation on the GPU. The reason for this is that present day GPUs and FPGAs are constituted of large quantities of compute resources, which can be exploited by simultaneously executing more than one task. Simultaneous execution of tasks has led to the improvement of performance and optimised resource utilisation [34–36]. Cruz. et. al. 2017 [37] have experimentally demonstrated that the order in which simultaneous actors are submitted to a GPU will affect resource usage and performance. The order of actor submission essentially enforces a schedule that restricts certain concurrent executions, while permitting other simultaneous executions. The schedule for agile heterogeneous computing, thus, must take into account of resource consumption for compute engines that allow concurrent actors.

The expansion of the design space for agile heterogeneous computing is self-evident; variable platform architecture and scheduling decisions based on resource consumption. In section 2.5, it is shown that efficient mapping and scheduling algorithms for a fixed architecture exist. Algorithms will need to be adapted to find mapping and scheduling decisions while considering the resource consumption of individual

[1]Carl Hewitt in 1977 [38] used the term "actor" for autonomous agents, which was later used by Gul Agha in 1990 [39] to describe concurrent components of an application. More recently, actors are used by Lee et. al. 2004 [40] to describe components of a parallel application as a graph-based representation, where actors are nodes that communicates by passing messages through channels (edges). It will be shown later (in section 2.4.1) that such dataflow representations are widely used for design flows due to their analytical strengths.

actors.

A final challenge of heterogeneous computing is that a diverse skill set is required to implement software and hardware components for all the different compute engines and interfaces between them. Increasingly, it can be assumed that components written by experts will come to dominate the design process. In this thesis, the approach to agile heterogeneous computing assumes that such components are readily available together with performance data of their executions on the various compute engines.

In summary agile heterogeneous computing raises new challenges for designers. In the rest of the literature review, progress that has been made so far to address these challenges will be explored.

2.2.3 Conclusion

Agile heterogeneous computing was introduced in this section. At first an historic perspective of heterogeneous computing was presented, where it was shown how FPGA, GPU and CPU started to be used together in a platform architecture. Then, previous research literatures highlighting the challenges of using these computing engines together and the variability of the platform architecture due to compute engine type and the connection topology were described. The necessity for a design flow that can automate some stages of the design process was reflected while reviewing literatures on agile heterogeneous computing.

2.3 Design flows for agile heterogeneous computing

The previous section highlighted the necessity of a design flow to mitigate the unique challenges of designing applications for agile heterogeneous computing. These challenges are; (1) forming a heterogeneous compute platform architecture, (2) schedule simultaneous execution of multiple actors on compute engines, and (3) manage complex tool-flows to target disparate compute engines. In this section, prominent and recent design flows proposed for agile heterogeneous computing are reviewed. It is shown that although there are a large number of design flows, they do not cater to all the aforementioned challenges. The reason being, previous design flows consider either fixed or a restrictive format of platform architectures. Furthermore, simultaneous execution of multiple tasks on GPUs and in combination with FPGAs is a relatively new concept, thus it is not fully supported in most design flows. During this review of different design flows, it is shown that along with the overall design methodology, the representations that form the foundation of a design flow and the algorithms used for the design space exploration might not be sufficient for agile heterogeneous computing.

This section is organised into two parts. The first part introduces the Y-chart [41] approach to fixed platform design flows where algorithms and platform architectures are separated. The second part reviews variable platform design flows that allows platform architectures to be *first class* [2] design variables.

2.3.1 Fixed platform design flows

A major innovation in design flows, that is relevant to the challenges in the selection of a platform architecture in agile heterogeneous computing, is the development of the so called *Y-chart* methodology [42, 43] for the construction of design flows. In this methodology, two separate high-level representations, one for the application-algorithm and the other for the target platform architecture is required. The application representation can be a dataflow model where nodes are computation and the channels are communications. Whereas, the platform architecture representation, expresses the communication and computation services offered by the platform, such as CPU, GPU and the connections between them. These two representations are then used to find the mapping and scheduling decisions. These decisions are finally used to deploy the application on the chosen platform architecture. The Y-chart methodology is illustrated in Figure 2.1, where the light bulbs depict the iterations to improve the design space exploration for improved performance by either modifying the mapping and scheduling decisions, or by changing the platform architecture outside the design space exploration stage.

The concepts underpinning the Y-chart design methodology were independently proposed in 1997 by Kienhuis et. al. [42] and Balarin et. al. [43]. A refined exposition that actually coined the term Y-chart was later presented by Kienhuis. et. al. 2002 [41]. Y-chart approach encompass a number of innovations. The most significant is the separation of application-algorithm and (fixed) platform architecture all the way through the design flow. The Y-chart approach implies that during design space exploration, a mapping must be created between the application-algorithm and the platform architecture components, which will drive the final implementation. The Y-chart paper proposed a number of refinements of the architectural representation as the design flow proceeds from specification to implementation. However, tools that are based on Y-chart, such as SDF3 [44] and PeaCE [45] are constructed on a single level of abstraction in the architecture representation. So refinement is not a key concept in Y-charts rather it is the separation of concerns between algorithm and architecture. It is this key idea which is likely to be most helpful in developing a design flow for agile heterogeneous computing.

Another possible separation of concerns (in addition to that suggested by the Y-chart papers) is between design space exploration and deployment. The *Algorithm*

[2] The term *first class* is used to emphasise that platform architecture is a design variable that always exists as a variable during the design process.

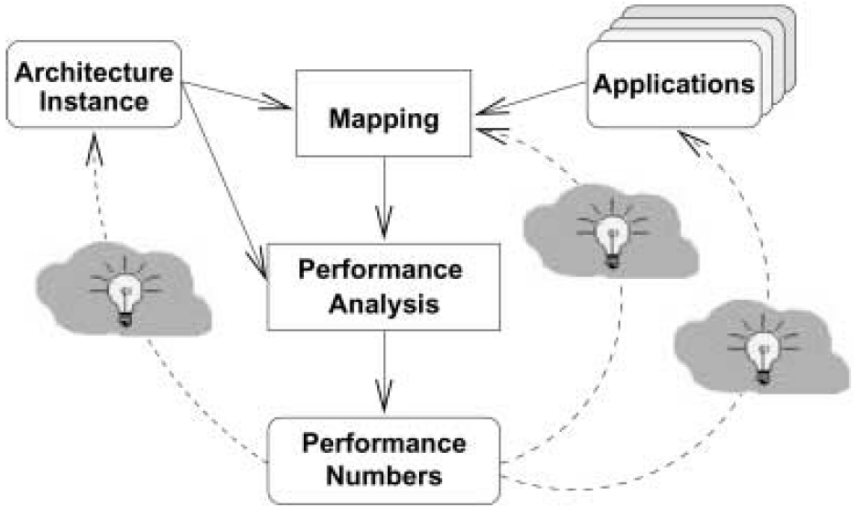


Figure 2.1: The Y-chart methodology. This figure is reproduced from [41]. The light bulbs indicates the possible iterations to improve performance by changing the application-algorithm set, mapping and scheduling decisions or by modifying the platform architecture. These modifications can only take place after the completion of the design space exploration stage.

Architecture Adequation (AAA) methodology [46] proposes this. It is the basis of a more recent design flow called *Preesm* [30]. However, both of these methodologies are for fixed platform architectures. In AAA approach, the results of design space exploration are expressed in a format that is independent of any deployment technology and the deployment step is separate. This format is an intermediate representation (IR) that contains the mapping and scheduling, and routing decisions. The IR contains all the necessary details for deployment. It can be used for performance prediction and for further analysis. Furthermore, it exhibits separation of concerns between the design space exploration and deployment. Since this representation is used for deployment, it can be analysed further. From the standpoint of agile heterogeneous computing the analysis of such an IR to know if it is deployable is desirable. Platform architecture is variable, it is important to know before hand of all the resources that are available for deployment. Preesm enables the use of independently developed pre-engineered components for deployment which mitigates the need for designers to have skills on all the compute engines used in the platform.

Thus, a well designed intermediate representation can foster separation of concerns between design space exploration and deployment. Ideally, it would support further analysis that was not possible just with the application-algorithm representation.

Since, for fixed platform architectures, modern machine learning libraries like, Torch [47], TensorFlow [48], MatConvNet [49] and MXNet [50] have started to support multiple GPUs [51, 52] and more recently FPGAs [53, 54], these libraries

are also reviewed. These libraries are used by machine learning application developers to target a fixed heterogeneous platform. However, they are limited in their abilities to analyse a platform architecture to find if the platform architecture is suitable for an application-algorithm. Due to this limitation the machine learning libraries cannot be used as a design flow for agile heterogeneous computing. This limitation exists because these machine learning libraries were developed to provide an abstraction layer to program machine learning application-algorithms as a dataflow acyclic graph (DAG) and then implement the DAG on the target heterogeneous platform. In principle, during the design phase, the platform architecture is assumed that it cannot be changed. It is important note that although these libraries are not suitable as a design flow for agile heterogeneous platforms, they show the usage of DAGs for implementation in heterogeneous platform architectures.

2.3.2 Variable platform

SESAME [55] is a design flow that is constructed on the concepts of Y-chart methodology [41] with limited ability to handle platform architecture as a design variable. Like any other Y-chart based design flow, it has two separate high-level representations. One for the application algorithm and the other for the platform architecture. The applications are represented as high-level Kahn Process Networks (KPN) [56], whereas the platform architecture is expressed as a high-level graph based representation.

It is important to recollect here that KPN is the most general dataflow representation. For this reason, it is difficult to analyse because the first-in-first-out (FIFO) channels that connects dataflow graph's nodes are not bounded. Another limitation of KPN is that its FIFOs can introduce deadlocks. A KPN cannot be scheduled without transformation. It needs to be converted to one of the more restrictive dataflow models, such as synchronous dataflow (SDF) [57] or cyclo-static synchronous dataflow CSDF [58] or some other type of decidable dataflow representations [59] that can be scheduled during compile-time [60]. However, conversion from a general dataflow instance such as KPN to its equivalent restrictive form is not always feasible. A work around in this situation is a runtime scheduler proposed by Parks in 1995 [61]. This scheduler ensures that the deadlocks are prevented during runtime. However, a runtime scheduler can introduce execution overheads [59].

In Sesame, the KPN is thus first transformed to a mapping layer. After the mapping layer, the KPN is now deadlock free and has finite FIFO buffers. Now in this mapping layer, each KPN processor is mapped to a virtual processor. Virtual processors are mapped to a single core of a CPU. The mapping is one-to-one, such that each KPN process is mapped to one unique virtual processor. The mapping layer is comprised of virtual processes linked by bounded FIFOs, onto which the

KPN FIFOs are mapped.

The most interesting aspect of SESAME is its ability to treat the architectures as a design variable in an design space exploration based on evolutionary algorithms. A meta platform which is a type of template for possible platform architectures is the starting point for the optimization algorithm. This template features a common bus to which processors and RAM can be added. The evolutionary optimization algorithms used, mutate the template architecture by removing and adding processors and memory (RAM) components. The evolutionary algorithm has no knowledge of the internal mapping and scheduling. In this, they are treated as a black box.

After mapping, the performance of an instance of the design is derived from simulation. This is possible because the constrained architectural choices allow the usage of off-the-shelf simulators. These estimates are used to evolve the platform architecture for improvement using a cost function. There is no allowance for a change in the processor type. As discussed previously in section 2.2, for heterogeneous computing, changing of processor type may be a desirable part of design space exploration.

The other main area where variable platform architectures have been considered is in the general domain of work-flow scheduling in the cloud by [62]. The various research works in this area are not strictly a complete design flow but they do allow for a restricted form of variable architecture as described here. There are several algorithms, such as GA-ETI by Casas. et. al. 2016 [63], Gain/Loss approach by Sakellariou. et. al. 2007, [8], MOHEFT by Durillo. et. al. 2012 [64] and EGA-TS by Akbari. et. al. 2017 [65] . All are proposed with the goal of reducing the rental cost of cloud resources without compromising performance. In these bodies of work, the platform architecture variability is restricted to selecting the number of virtual resources. Interconnections and their associated bandwidth are bundled into the performance of each virtual resource. It is assumed that there is potential all-to-all connectivity available between virtual resources. The designer cannot determine the way various physical components of a cloud platform are connected and it is usually assumed that bandwidth between virtual resources is equal. Therefore, it can be said that in the cloud, the platform architectures are primarily fixed with the option to choose the required amount of virtual resources. Heterogeneity is modelled in a very simplistic way such that virtual resources can be scaled but the actual nature of the underlying execution engines are abstracted away. The details of the assumptions and the scheduling and mapping algorithms used in these cloud work-flow schedulers is explored in more detail in section 2.5.

2.3.3 Conclusion

There are design flows both in the context of embedded and cloud applications that consider platform architecture as a variable, which was suggested in section 2.2.2 for

agile heterogeneous computing. SESAME maintains a separation of architecture and, mapping and scheduling. Cloud papers have a more integrated approach but assume that all processors are connected to a common bus and assume all compute engines are capable of communication with each other without details being considered.

2.4 Representation of application algorithm, architectural decisions and platform architecture

In the previous section, the design flows for agile heterogeneous computing that appear in literature were reviewed. In this section, the literature on representations for application-algorithms and platform architectures are discussed.

Representation is important because it is unlikely that a design flow can escape from the assumptions implied by its internal representation. Agile architectures will need unique representations for expression of variability in platform architecture and in the expression of mapping and scheduling decisions for such variable platforms. Finally, a separation of concerns between design space exploration and deployment makes the design process more manageable, such an approach was discussed in the previous section. Separation of concerns are dependant on finding suitable representations. Therefore, in this section, ways of representing parallel applications, platform architectures and, mapping and scheduling decisions are reviewed.

This section is divided into three parts. In the first part, the representations of parallel application algorithms using dataflow models are reviewed. Then, in the second part, approaches of representing architectural decisions are visited, which is followed by the third part that reviews the platform architecture representations.

2.4.1 Representing application algorithms

Whilst representations that have been used in other domains include petri nets [66] and statecharts [67]; the dominant representation of algorithms in cloud workflows and embedded applications is dataflow. This can be seen from the large number of design flows, reviewed in the previous section, that uses dataflow to represent the application algorithms. However, certain design flows, such as COMPLEX [68] and MOPCOM [69] are not based on dataflow representations, rather they uses a form of statecharts extended as a UML profile, which is called as MARTE/UML. Nevertheless, they have triggered research work for the support of dataflow models with MARTE/UML [68, 70, 71]. There are two ways in which this is supported. Firstly, the application-algorithms are expressed initially as a statechart instance,

which is then converted to its native MARTE/UML [72] format. Secondly, a transformation from MARTE/UML to different kinds of dataflow models are also supported, so that analysability for performance estimation and resource usage are enhanced [68, 70]. The widespread use of dataflow models can be attributed to their analysis strength and also, to their natural ability of expressing parallelism visually in the form of graphs. Therefore, in this subsection, notable and recent dataflow models are reviewed.

A dataflow representation is composed of actors (nodes) and channels (edges). The actors denote computation and the channels symbolise token (data) transfers between them. Actors consume tokens from their input channels and release the resulting tokens at the output channels. An early dataflow representation was proposed by Kahn 1974 [56], which is a network of actors connect through unbounded FIFOs. It is commonly known as Kahn Process Network (KPN). Due to its expressivity of parallelism, it has been used in the representations of course-grained parallel applications [42]. Another important characteristic of a KPN is the order in which the actors are executed do not affect the application. Only the input tokens determines the final outcome, not scheduling of the actors. Dataflow models that exhibit this characteristic are known as deterministic [60]. Representing applications through a deterministic model removes the burden from the designer regarding its functional consistency during deployment. Although KPN are excellent at expressing parallelism, they require special care during implementation. This is due to the fact that they have channels that are unbounded. A runtime scheduler is necessary to ensure that the channels do not exceed memory limits [61]. Such schedulers add extra runtime overheads and restrict compile-time analysis [73].

With rising complexities of parallel signal processing applications, the demand for a representation approach with powerful analysis and optimisation capabilities grew. Synchronous dataflow (SDF) proposed by Lee. et. al. 1987 [57] was the first dataflow model that was widely accepted to represent application-algorithms for various design flows [44, 74]. It is based on the concept of actor firing (execution) with pre-defined rates (number of tokens needed for firing). Actors can fire when they have a pre-defined number of tokens at their input channels. Similarly, there is a predefined number of tokens that are released at the output channels after firing. In Figure 2.2(a) an example SDF graph of three actors A, B and C is illustrated with their data token consumption and release rates. The small black circle is the initial token, which is necessary to determine the starting actor. Due to the predefined actor rates, an SDF representation can be scheduled during compilation for a platform composed of single or multiple processors [60]. A schedule is a periodic order of actors that will be repeated infinitely (lifetime of the application). For the example SDF graph in Figure 2.2(a) the sequential schedule is $<A, B, B, C>$. The number of times an actor fires within a schedule is called as the firing rate, such as the firing rate of actor A is 1 and for actor B is 2. The order in which the actors fire determines the buffer size of the channels, which can be optimised to reduce the buffer sizes [60, 75].

In order to schedule an SDF graph, a topology matrix Γ which represents the tokens consumed and released by the actors on the channels (shown id Figure 2.2(b)) is required. A topology matrix is created by assigning the actors to the column and channels to the row and each entry represents the tokens given or taken to/from the channel. For actor A and channel ab the entry is 2, as when A fires it gives the channel 2 tokens. Whereas, for actor C and channel bc, the entry is -2, as B takes 2 tokens from the channel. A zero implies that the actor neither gives or takes tokens from the channel.This topology matrix is next used to determine the consistency of the graph and also used for scheduling.

A condition needs to be first fulfilled to check whether the SDF graph is consistent. This condition is $rank(\Gamma) = S-1$, where $rank(\Gamma)$ is the rank of the matrix and S is the number of actors(nodes) of the SDF graph. Then, a non-trivial repetition vector r, where $r.\Gamma = 0$ is calculated to create the sequential schedule. The minimum repetition vector for the example SDF graph is: $A : 1$, $B :2$ and $C : 1$. Now a sequential schedule can be created based on the repetition vector and the starting actor, which is determined by the initial token. The sequential schedule is the first step to construct a parallel schedule on multiple processors. The second step is to create an equivalent dataflow acyclic (DAG) graph of the SDF graph. Figure 2.2(c) shows the equivalent DAG of the example SDF. Then, the third step is to map the actors on the respective processors and then find the order (schedule) in which the actors must fire. Lee et. al have shown that this task of mapping and scheduling on each processor is NP complete. After it is decided where the actors will be mapped, partial DAGs for each processors are constructed. Each of these partial DAGs comprises of a multi-processor schedule. The compile time multi-processor schedule can be used to estimate throughput [60,76]. These analysis capabilities led to the wide usage of SDF model. However, SDF has limited expressive capabilities that prevents the representation of certain traits of an application-algorithm [77]. Newer models [58, 77, 78] are proposed to cater for these needs. Most of these new models rely on the SDF semantics, as they are converted back to a SDF form during analysis.

Since SDF requires to be expanded to its homogeneous form and then to a DAG for mapping and scheduling decisions, there is a chance of exponential increase in the number of DAG actors [60]. An approach to directly analyse SDF graphs was proposed by [76]. However, as this approach does not expand the SDF graph, complete task and data parallelisms cannot be achieved [79]. Partial expansion of the SDF graph can be a probable solution to prevent the graph from becoming very large and also revealing some of the data parallelisms [79, 80]. Expression of partially expanded graph has gained little attention.

Depending on which actor is expanded for data parallelism, which actor is to be mapped on a given processor and their order of firing, an SDF graph can be implemented in a large number of different ways. Although in a deterministic dataflow model like SDF, the scheduling or the mapping decisions will not affect the functional outcome [81], they will impact performance and resource usage [60]. This

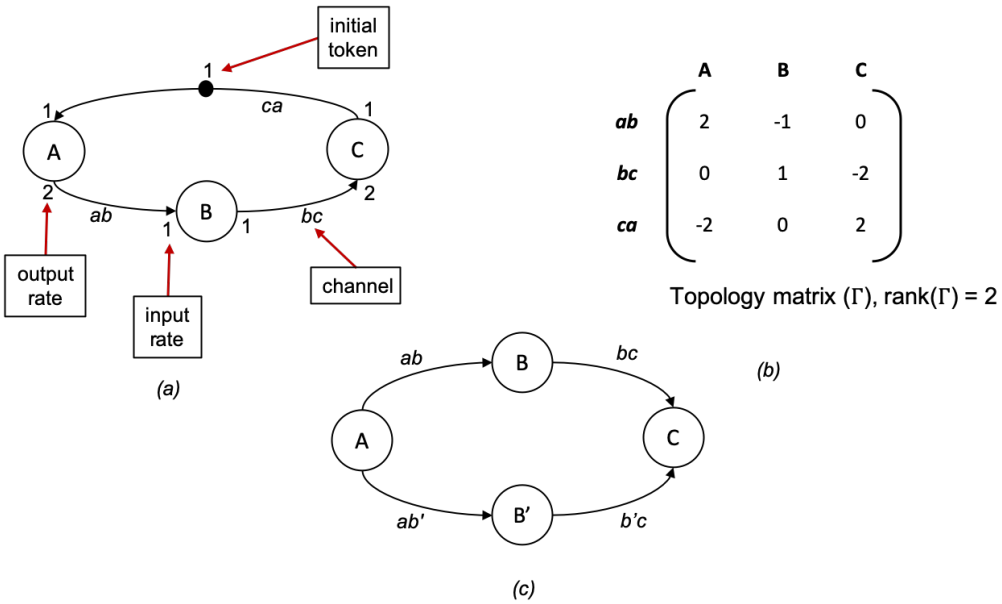


Figure 2.2: (a) an SDF graph, (b) the graph's topology matrix with its rank which is $3-1$ thus can be scheduled and (c)an equivalent dataflow acyclic graph(DAG).

has led to an increased interest in considering alternative platform architectures at the design stage. For this reason, representations of these design decisions are gaining prominence. They act as an intermediate representation to transition from the design space exploration stage to the deployment stage. This is the subject of the next subsection.

2.4.2 Intermediate representation

Dataflow models that can also represent design decisions are reviewed here. A new taxonomy of representations based on expressing mapping and scheduling decisions is presented in Figure 2.3. Following this taxonomy, the three options of *separate graph*, *SDF compatible* and *new semantics* are now discussed.

Separate graph

In this approach the design decisions are represented through an entirely separate graph, which is not extended from the application-algorithm. This approach of representation adds extra responsibilities to the design flow in managing another model apart from the already existing representations of application-algorithm and the platform architecture. During analysis and also for deployment, the application-algorithm representation needs to be used along with this model. This is the approach taken by the data flow schedule graph (DSG) [82] and the implementation model in *Algorithm Architecture Adequation* (AAA) [46] design methodology.

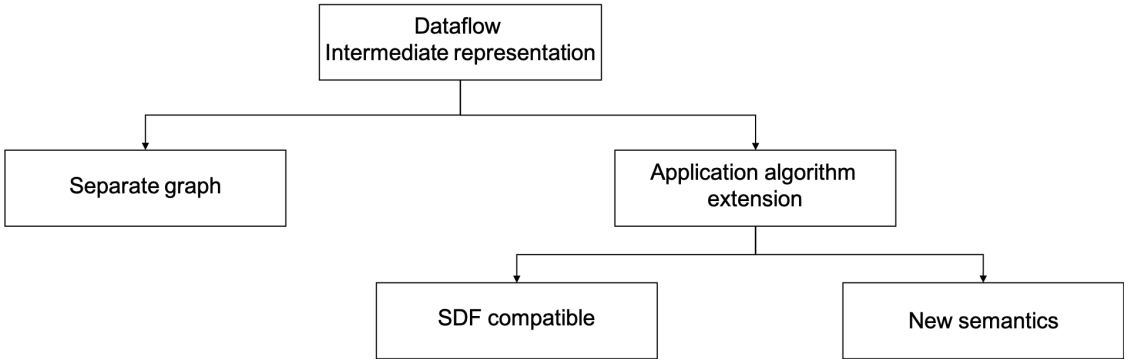


Figure 2.3: This diagram shows a taxonomy of dataflow models expressing design decisions. The taxonomy is based on the different approaches that are developed to expressing the extra information of design decisions. There are two major approaches; separate graph models that do not extend the application algorithm and extending the application algorithm representation with the design decisions. The later can be classified further into two groups. The first group are the representations that just uses SDF semantics. The second group are the dataflow models with new semantics.

A DSG [82] is a separate dataflow representation with extra semantics to contain the mapping and scheduling decisions. These extra semantics consists of reference actors and schedule control actors. The reference actors shadow the actual actor of the corresponding dataflow representation, whereas the schedule control actors represent the loops and other control-flows. Sequential execution of actors, when they are mapped on the same processor is guaranteed through special tokens called *control flow tokens*. They are passed within reference actors to control the flow of execution. These special tokens are also duplicated and merged for concurrent executions when actors are mapped on different processors. Since resource consumption of actors are not explicitly expressed by DSG, expression of concurrent executions on compute engines like GPUs, where multiple actors can execute together (when there are sufficient resources) is limited. It can only be achieved by mapping parts of the actor on individual GPU processors, which may not be practical, as GPU architectures are generally abstracted by vendor specific tools. Another disadvantage of DSG is that any analysis and deployment would require the merger of the DSG instance with the application-algorithm representation. The paper does not address these issues directly.

In the *Algorithm Architecture Adequation* (AAA) methodology [46], the intermediate representation (IR), called the *implementation model* is formed by assigning actors (nodes) of the application graph to the chosen processors of the platform architecture representation. Extra actors are created to denote the additional task of communication that takes place between the processors. These communication tasks are mapped on the available physical links. If a communication is between processors that do not have direct physical links, then a route that travels between processors is used. This feature is a robust way to handle platform architectures where all the processors are not directly connected to one another. After the assignments of the actors to processors and their communication on the physical

links, their executions are scheduled. The schedule is expressed as partial orders for every processor and communication links. This partial order schedule is based on the assumption that a processor only executes one actor at a time. Although this successfully expresses all the decisions necessary for deployment on a platform of few processors, *implementation model* has severe limitations in the expression of simultaneous execution of multiple actors as would be required on a compute engine like an FPGA or a GPU.

This limitation is similar to DSG and can be explained further with the help of an example. Suppose two actors execute concurrently on a GPU. Since the assumption is one actor at a time can execute on a processor, this decision can be expressed by breaking down the actor into multiple smaller actors that just uses one single-core processor of the GPU. Then, creating a corresponding platform architecture, where the GPU is represented into its constituent singe-core processors. But the details of GPU architectures are often abstracted by the vendors, making this approach impractical. How this intermediate representation can be applied to FPGAs has not been studied. Furthermore, since consumption of compute resources are not explicitly expressed, analysis of resource consumptions are not possible. Another limitation of *implementation model* is that it is based on a dataflow representation that does not explicitly denote tokens, unlike SDF, thus impairing memory optimisations.

SDF compatible

Stuijk. et. al 2007 [44] extended the application algorithm SDF to bind-aware SDF that incorporates scheduling and mapping decisions, and expresses how resources are consumed at the start of actor execution and released after it finishes firing. Similar to DSG, resource consumption is limited to the consideration of one actor mapped on a processor at a time. Thus, cannot be directly used for compute engines like GPU and FPGA. Although this is insufficient for agile heterogeneous computing, bind-aware SDF can be used to analyse the trade-offs of resource requirements and performance [83], which can be an essential component for design space exploration. Bind-aware SDF is also tied to a restrictive form of platform architecture representation (reviewed in section 2.4.3). This restrictive platform architecture not only assumes that one actor executing at a time on a processor but also the processors are connected using a single bus. Due to the assumption of a simplistic connection topology, mapping of channels to multiple processor routes is difficult to express. Such situations are common in agile heterogeneous computing, as the tasks on GPU and FPGA are usually routed from the CPU.

The techniques used in bind-aware SDF [44] to express mapping and scheduling decisions and consumption of resources are based on the usage of annotations and extra edges on the original data flow graph. The actors and channels of the original SDF graph are annotated with the resource information; like the processors they

are mapped and memory consumption values. This is limited to the mapping of only one actor on a processor and sharing is not allowed. Extra edges are used to restrict possible concurrent execution of actors. These extra edges are not part of the application-algorithm and only enforce sequential execution.

Extra edges with annotations are also used to represent mapping and scheduling decisions in *synchronisation graphs* (SG) [84]. An SDF is converted to the more restrictive homogeneous-SDF (HSDF). In HSDF, the rates of all the actors are reduced to one. This conversion results in multiple duplications of actors and channels. The algorithm for this graph transformation is provided in [57, 60]. It is evident that as a consequence the size of the model can greatly expand; but the advantage is that it exposes all the implicit data parallelisms [74, 79, 80]. After this conversion, actors that can execute concurrently in the dataflow, but due to lack of available processors cannot execute in parallel, are added a special channel. This special channel, known as synchronisation channel, is an elegant way to use existing semantics of dataflow to express these design decisions. The partial orders for every processor is implied through these edges. However, such representations lack resource information. They also don't consider mapping of channels between processors that do not connect directly.

New semantics

In the next part of this subsection, representations that add new semantics to the algorithm data flow graph to represent mapping and scheduling decisions are reviewed. In some cases, these extra semantics are incompatible with the SDF model. In other cases, it is possible to propose a conversion between the graph with extra semantics and the SDF model. However, these conversions have not been pursued by the authors.

Decision state model (DSM) [85] and Partial ordered approach (POA) [86] are representations like afore-mentioned *synchronisation graphs* (SG) that can express mapping and scheduling decisions. Another commonality is that none of them explicitly express resource usage, they are based on the assumption of only a single actor executing on a compute engine and also rely on simple inter-processor connections. This implies that they cannot be used within a design flow for agile heterogeneous computing where FPGAs and GPUs are to be considered. However, as these representations extend dataflow semantics, they still have strong analytical abilities. These will now be considered in detail.

DSM [85] have extra actors and edges to represent mapping and scheduling decisions. Extra actors which are not compatible with standard SDF actors. They are added to avoid the expansion of the algorithm SDF to its equivalent HSDF. These extra actors enforce the assumption that all the firings of an actor are mapped on the same processor. Therefore, a possible design decision to expand an actor

for data parallelism and then mapping it onto two separate processors cannot be represented. The authors have shown that decision state models are more effective in analysing memory size versus throughput trade-off, as compared to SG. This can be attributed to the fact that in a decision state model, the SDF structure is preserved, thus the analysis algorithms can be directly applied to the original model.

Zebelein et. al. 2013 [86] proposed a dataflow representation based on partial ordered sets with extra semantics to express mapping and scheduling decisions. This representation is referred here as *partial order approach* (POA). It is proposed with the objective of minimising the runtime overheads of executing a dataflow application. This run-time overhead is due to the checks done by actors to know if there are enough tokens to fire and also if there are enough space to release the tokens. Some of these checks can be avoided from the mapping decisions, if an actor that has input and output channels on the same processor because the processor usually has a round-robin scheduler that fires the actors infinitely in a sequential order. This sequential order is the schedule for the processor. Therefore, an actor that has predecessors and successors on the same processor will have the dependencies cleared. The checks become essential when there are channels from other processors. POA splits the long schedule on a processor into partial orders removing the need to check tokens and space available for tokens on other processors. However, this approach introduces a centralized scheduler which is contrary to the concept of a model with independent actors. It must be noted that these overheads addressed by the partial order approach can now be handled quite quickly in software. In an application requiring FPGAs and GPUs practical overhead delays may be insignificant with respect to the communication delays amongst various compute engines. On the other hand, partial ordered approach shows the innovation of using composition of actors (actors inside a bigger actor) to assist in expressing mapping and scheduling decisions. However, the usage is limited to the sequential execution of actors on a single processor and without explicitly expressing resource usage, making it inapplicable to agile heterogeneous computing.

From this review, it is noted that the advantages of using a representation that is compatible with the SDF model used for the application-algorithm are powerful performance analysis, compile-time memory estimation and possibility to leverage the existing tool-sets for SDF representation. It is noted further that all representations considered here ignore resources that can be used to determine concurrent executions on a compute engine.

2.4.3 Representing heterogeneous platform architecture

Design flows that are based on Y-chart methodology [42] promotes the idea of high-level representations of platform architectures, so that design decisions relating to

platform architecture are made at a higher level of abstraction. In this subsection, representations that are relevant to the expression of platform architectures for agile heterogeneous computing are reviewed. This review aims to find how well the existing representations can handle variable platform architectures.

The platform architecture representation in *Algorithm Architecture Adequation* AAA methodology [46] is a graph that can only express a limited range of fixed platform architecture. Later, this representation is improved by Grandpierre et al. 2003 [87] to make it more general, so that a wide range of platform architectures can be expressed. The representation called an *architecture graph* consists of nodes that represents four kinds of finite state machines (FSM) and the edges correspond to the input and output of the state machines. The four types of FSMs sequentially manage the following: operator, communicator, bus, and memory. Operator is any kind of computation that can be a processor or an IP block. Communicator enables data transfers and does not involve operators. Bus is a physical link that communicators compete to get access to. In this representation, a platform architecture with switches cannot be expressed. This drawback is addressed in [88] by adding extra nodes to model switches. The architecture graph is further generalised into S-LAM [88], which is based on the idea that modern day processors are too complex to be represented on a high-level model. Rather than using clock rates, estimated or measured execution times are used for every (actor, operator type) pair. Although S-LAM is significantly generalised to handle a wide variety of platform architectures, it is limited to the expression of sequential actor firings and lacks the ability to represent architectures that can vary. The attributes of the platform architecture are fixed. Furthermore, resources requirements of an actors are not considered.

The design flow proposed by [44] extends that platform model initially described by Culler et al. 1997 [89] to create a platform architecture representation in the form of tiles, known as *multi-processor platform template*. Each tile consists of the a processing element, network interface, a memory assist and a local memory. The processing element represents a type of processor core. The network interface is responsible for data transfers to and from a tile, whereas the communication assist acts an arbitrator to access memory. A variable number of tiles can be connected through bus. Complex topologies using switches are difficult to be expressed by *multi-processor platform template*. Moreover, in regards to agile heterogeneous computing, this representation lacks the ability to express resources. Actor firings on a tile are expressed sequentially and any expression of variability in the platform architecture is not considered.

Linear *system-level architecture model* (LSLA) is proposed by Pelcat et al. 2018 [90], which is a high-level platform architecture model focusing on reproducibility and energy analysis. Reproducibility refers to capturing enough details of the platform architecture in the abstract model so that it can be physically assembled. The platform architecture represented is fixed. There is no notion of variability present in the representation. Also, it is unclear how concurrent executions of

multiple actors can be represented on a compute engine.

In order to cater for many kinds of platform architecture [91] extended the Y-chart methodology [42] to separately represent communication topologies. This separation of communication topologies gives more options to try in the design space exploration stage. However, during the design space exploration, only a fixed platform architecture is considered. The topologies are not created during the course of the design space exploration, rather they are fixed.

A rather restricted way of considering platform architectures as a variable parameter in the design space exploration can be seen in the later versions [92] of SESAME [55] design flow. It uses a platform architecture representation known as *meta-platform architecture model*. This representation consists of the following elements: processor, bus, memory and point-to-point links. These elements are used to create a *basic topology unit* (BTU). BTUs are connected together to form a platform architecture. This implies BTUs are used to form ta suitable platform architecture in the design space exploration stage. Although variability is represented through this meta-platform model, it is heavily restrictive. BTUs are defined manually and connections of BTUs other than using a straight bus is not explored in the paper. This is a drawback for agile heterogeneous computing, as connectivity topologies amongst various compute engines cannot be expressed through a simplified bus model.

2.4.4 Conclusion

Data flow graphs are good for representing parallel algorithms because they can naturally express parallelism and they are underpinned by a well-defined model. SDF compatible representations of mapping and scheduling decisions are elegant and preserve analysis. However, they do not consider resource consumption to express concurrent execution of actors on compute engines. Furthermore, platform representations reviewed are aligned to fixed architectures, or if variable are quite restrictive. The representation of GPU and FPGA compute engines is not well explored.

2.5 Mapping and scheduling algorithms

The review of design flows in section 2.3 showed that mapping and scheduling algorithms are key to the exploration of the design space. The decisions made by these algorithms are used further in deploying the application-algorithm. In this section, various algorithms that have been proposed for mapping and scheduling on a heterogeneous platform are reviewed with the aim of finding those that are suitable for agile heterogeneous computing. This review is conducted in three steps and for

each step, there is a dedicated subsection. In the first subsection, a distinction is made between static and dynamic mapping and scheduling algorithms. It is then shown how the challenges of agile heterogeneous computing are best addressed with static algorithms for mapping and scheduling. Then in the second subsection, a taxonomy is presented to simplify this review of the vast literature on static mapping and scheduling algorithms. It is noted that the major division between them are the so called heuristic and meta-heuristic algorithms. Among these two major groups, it is shown that list-based heuristic algorithms and evolutionary-based (also called genetic) meta-heuristic algorithms are widely used in design flows for heterogeneous computing. Therefore, the third and the fourth subsections focus on the review of evolutionary-based and list-based algorithms, respectively. It is shown that current evolutionary-based and list-based algorithms are limited in relation to the challenges of agile heterogeneous computing. These limitations are; insufficient consideration for variable platform architectures, simultaneous execution of actors on an FPGA or a GPU and economic objectives based on the rental cost model rather than capital costs as for embedded heterogeneous computing.

2.5.1 Static and dynamic algorithms

The mapping and scheduling problem can be divided into two forms; static and dynamic [93]. In static mapping and scheduling, the allocation of all the actors and the timings or order in which they will execute are determined during design space exploration (also known as compile-time schedule). Whereas, in dynamic mapping and scheduling, these decisions are taken at runtime. Since, mapping and scheduling are a NP complete problem [94], the runtime scheduler can consume a significant amount of resources [95] apart from running the application-algorithm.

Compile-time scheduling requires various details to be known in advance of the application-algorithm such as, the computation timings of the actors, communication delays and data dependencies [93].

Static mapping and scheduling approaches are the focus of this work and are reviewed in the next sections.

Since in agile heterogeneous computing the platform architectures are determined during design space exploration, the mapping and scheduling decisions must also be known, so that a predictable performance is achieved. It is a problem of static mapping and scheduling. Hence, static mapping and scheduling approaches are reviewed here.

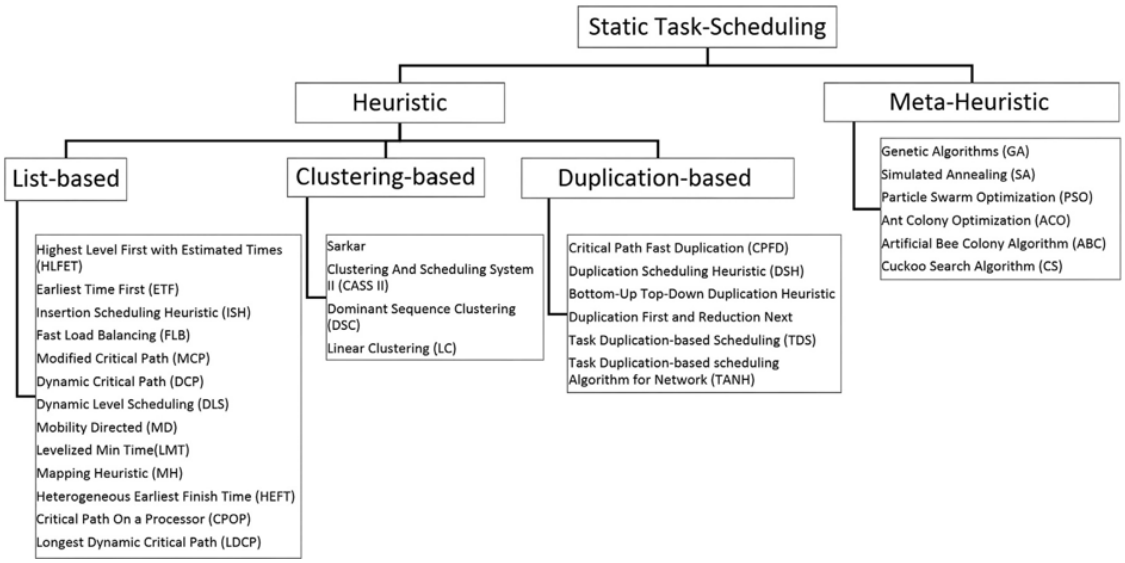


Figure 2.4: A taxonomy of mapping and scheduling algorithms from Akbari et al. 2017 [65].

2.5.2 Static algorithm's taxonomy

There exists a wide range of static mapping and scheduling approaches. A classification of these approaches by Akbari et al. 2017 [65] is shown in Figure 2.4. Note that in the literature it is common to refer to mapping and scheduling algorithms simply by the label of scheduling algorithms as is done in Figure 2.4.

The major distinction in the figure is between heuristic and meta-heuristic approaches. All scheduling algorithms use heuristics because in almost every case the problem is known to be NP complete [94]. Meta heuristic approaches are exemplified by evolutionary approaches such as genetic algorithms. These used generalized approaches that have not been developed with any specific application domain. For example a commonly used meta heuristic package for multi-objective optimization is *Strength Pareto Evolutionary Algorithm* (SPEA2) [96]. Heuristic approaches make use of more detailed information about the application-algorithm and the platform architecture. For example the very widely cited list based algorithm *Heterogeneous Earliest Finish First* (HEFT) [7] directly interacts with the application-algorithm graph. Within the heuristic group, list-based algorithms dominate. Other alternatives of clustering and duplication are not further considered in this review.

2.5.3 Evolutionary meta-heuristic algorithms

The key publication that discusses a meta heuristic approach to mapping and scheduling of tasks in heterogeneous computing is [55]. The authors advocate the

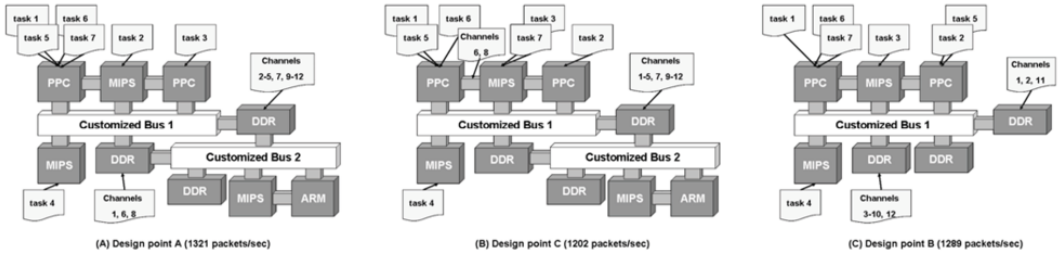


Figure 2.5: Design space example showing different design points, which are actually disparate platform architectures. This Figure is reproduced from [92].

Y-chart [42] design methodology which allows them to have variable platform architecture separate from the application-algorithm with a separate mapping step from one to the other. The authors use the SPEA2 [96] package in conjunction with accurate simulations to simultaneously explore the design space of power consumption, processing time and architecture cost. Resource consumption of actors is considered for mapping to find a suitable platform architecture but the application-algorithm is not taken into consideration. The mapping and scheduling decisions are taken separate to the platform architecture selection.

In a follow up paper [92] the approach is refined and several suitable designs are produced as shown in Figure 2.5. It is noted that very similar performance is achieved by all the cases and that only two platform architectures result from the study. The heterogeneity of the examples is limited only to different microprocessor architectures and this refinement also considers the search of platform architecture separate to the application-algorithm.

2.5.4 List-based heuristic algorithms

List-based heuristics algorithms use the structure of the input application-algorithm directly, as they first prioritise the nodes and then consider each node separately for mapping and scheduling. These algorithms were initially built for homogeneous platform architectures [97–99]. In these early algorithms, the platform architecture was limited to a network of few homogeneous processors. Also, most of the early algorithms assumed that the communications overheads between the processors were negligible. Although these assumptions were limiting, they created the foundation for the later sophisticated algorithms. [100–102] progressed list-based algorithms to consider communication delays. However, heterogeneous computing was still not the prime focus.

This changed when Topcuoglu et al. 1999 [7] proposed Heterogeneous Earliest Finish First (HEFT) for platform architectures consisting of a set of heterogeneous processors. Other algorithms started to be proposed for heterogeneous platform architectures, such as PEFT [103] and CPOPS [102], but HEFT got wider accep-

tance in cloud computing [64, 104] and in embedded application [105–107]. The widespread adoption of HEFT can be attributed to its low complexity and its good mapping and scheduling performance (a lower makespan). Makespan is the time to complete the critical path of the graph representing the application [60].

Even though the results of mapping and scheduling decisions of HEFT are competitive with other algorithms [104, 108], its major draw-back with respect to agile heterogeneous computing is that the platform is assumed to be of fixed architecture. The connectivity topology of this fixed architecture is considered to be fully connected, where every compute engine can be connected to every other. This is contrasting to agile heterogeneous computing, where all compute engines may not be capable of being directly connected and the architecture is a design variable. Furthermore, when multiple actors are assigned to compute engines they are restricted to sequential execution on that engine. This does not capture possible parallel execution of actors on compute engines like GPUs and FPGAs. Since HEFT is widely adopted, many enhancements surrounding HEFT have been proposed. These enhancements have either extended the algorithm to consider additional objectives such as the economic platform architecture cost [8, 64], or have focused on the improvement of performance [104, 109].

In order to more deeply examine the enhanced versions of HEFT, the original HEFT algorithm is described in detail using an example and then the enhancements are reviewed.

Heterogeneous earliest finish time (HEFT) algorithm

INPUT:
a. the application DAG G,
b. set of processor P,
c. actor execution time matrix EX,
d. communication delay matrix COMM,
e. upward ranking algorithm $rank_{up}$ and
f. earliest finish time (EFT) allocation algorithm

ALGORITHM:
1. Calculate $rank_{up}$ for all nodes (actors) by traversing G upward, starting from the exit node.
2. Sort the nodes in a list L by non-increasing order of $rank_{up}$ values.
3. Initialise empty schedule S
4. **while** there are unscheduled nodes in L **do**
5. Select the first task n_i in L and remove it.
6. **for** each compute engine ce_k of the platform architecture **do**
7. Calculate $eft_{i,k}$, $est_{i,k} \leftarrow EFT(n_i, p_k, EX, S, G)$
8. **end for**
9. Assign n_i to p_j that minimises eft value of n_i.
10. Update S to record $est_{i,j}$ and $eft_{i,j}$
11. **end while**
12. **return** S

Upward rank calculation

$rank_{up}$ is calculated as upward rank starting from the exit node. The $rank_{up}$ of node n_i can be recursively written as,

$$rank_{up(n_i)} = \overline{ex_i} + \max_{n_i \in succ(n_i)} \left(\overline{c_{i,j}} + rank_{up}(n_j) \right)$$

,where $\overline{ex_i}$ is the weight of each actor is the average execution time on all the processors; it is defined as, $\overline{ex_i} = \sum_j^P ex_{i,j} / |P|$. The *max* block represents all the succeeding nodes of actor n_i. $\overline{c_{i,j}}$ is the average communication delay connecting actors n_i and n_j.

Figure 2.6: Pseudocode for the HEFT algorithm [7]. The algorithm to calculate earliest finish time (EFT) is shown in Figure 2.7.

The HEFT algorithm

The HEFT algorithm first ranks the actors (nodes) of the application DAG on the basis of their average computation time on various compute engines (processors) that constitutes the platform architecture. Then, on the basis of the rank, actors are selected one by one to be mapped and scheduled on a compute engine. The compute engine is selected on the basis of the earliest finish time (EFT) of the selected actor. EFT on all the compute engines of the platform is calculated and the one with minimum EFT is selected. The pseudocode of the HEFT algorithm is shown in Figures 2.6 and 2.7.

In order to elaborately explain the pseudocode of HEFT, an example from [104] is used. The application DAG of this example is shown in Figure 2.8(a). The platform architecture is shown in Figure 2.8(b). It comprises two distinct compute engines connected through a communication link *L1*. The execution timings of the

Earliest finish time (EFT) and earliest start time (EST) calculation

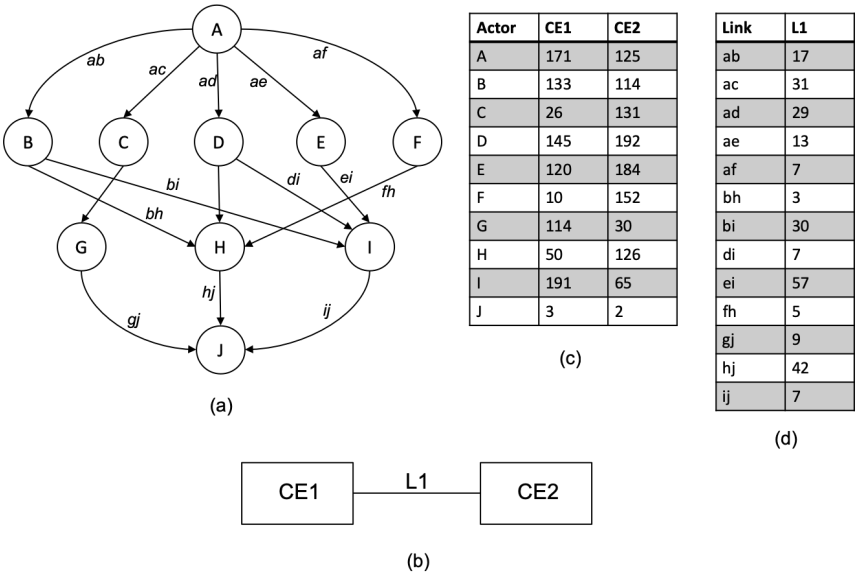
```
INPUT:
   a. the node n_i to be allocated,
   b. the processor p_k for allocation,
   c. actor execution time matrix EXE,
   d. communication delay matrix COMM,
   e. the application DAG G, and
   f. the ongoing schedule S
ALGORITHM:
   1.  Search S to find the earliest time slot (et) available on p_k to
       execute n_i for the duration exe_{i,k}.
   2.  Find all preceding nodes pre_i of n_i from G
   3.  for all preceding actors a_k in pre_i do
   4.      From S, retrieve processor m where a_k is mapped and scheduled.
   5.      Calculate completion time ct_k ← exe_{k,m} + comm_{k,i}.
   6.  end for
   7.  From all the predecessor's ct find the maximum, which is the start
       time (st) of n_i.
   8.  est ← max(st, et)
   9.  eft ← est + exe_{i,m}
   10. return eft, est
```

Figure 2.7: The pseudocode to calculate earliest finish time (EFT) for the HEFT algorithm [7], which is shown in Figure 2.6.

actors on each compute engine are shown in Figure 2.8(c). From this table, the average execution time is calculated: which is the sum of the total execution time of an actor on the different compute engines divided by the number of compute engines. This value will be used later to rank the actors, which is shown in table 2.1. Similarly the communication delay times are in Figure 2.8(d). HEFT can handle a heterogeneous set of communication links in which the delay depends on which compute engines are connected.

The ranking of these actors are based on a heuristic that estimates every actor's importance in determining the objective of the mapping and scheduling algorithm. The objective is usually performance but other objectives have also been proposed in the more recent works. This are discussed in the next part. In the original HEFT, the so called upward rank ($rank_{up}$) values of actors are calculated by summing the average execution times and communication delays whilst traversing the graph upwards from the last actor to the actor in question. When there are more than one possible path up the graph, the path that gives the largest $rank_{up}$ value is chosen. Table 2.1 lists the actor rank values for this example. Because there is only one communication link in the example the average communication delay is replaced by the delay on this link. The rank of an actor n_i is denoted as $rank_{up}(n_i)$ (see Figure 2.6). The Gantt chart in Figure 2.9 illustrates the resulting mapping and scheduling decisions.

After the actors are ranked, a ready list is created. Initially, the ready list contains only the actors that don't have any predecessors. Predecessor actors are denoted as pre in the pseudocode in Figure 2.7. From this ready list, the actor with highest



Actor	CE1	CE2
A	171	125
B	133	114
C	26	131
D	145	192
E	120	184
F	10	152
G	114	30
H	50	126
I	191	65
J	3	2

Link	L1
ab	17
ac	31
ad	29
ae	13
af	7
bh	3
bi	30
di	7
ei	57
fh	5
gj	9
hj	42
ij	7

Figure 2.8: An example originally published in [104] to describe the HEFT algorithm. (a) The application DAG and the platform architecture to demonstrate the HEFT algorithm. (b) The computation timings of the actors and (c) their communication timings for the platform architecture.

Table 2.1: The mapping and scheduling decisions based on the HEFT algorithm for the example shown in figure 2.8.

Rank	Rank Value	Mapped CE	Earliest start time (EST)	Earliest finish time (EFT)
A	507.5	CE2	0	125
E	346.5	CE1	138	258
D	313.0	CE2	125	317
B	291.0	CE1	258	391
F	218.5	CE1	391	401
C	178.0	CE1	401	427
I	137.5	CE2	421	486
H	132.5	CE1	427	477
G	83.5	CE2	486	516
J	2.5	CE2	519	521

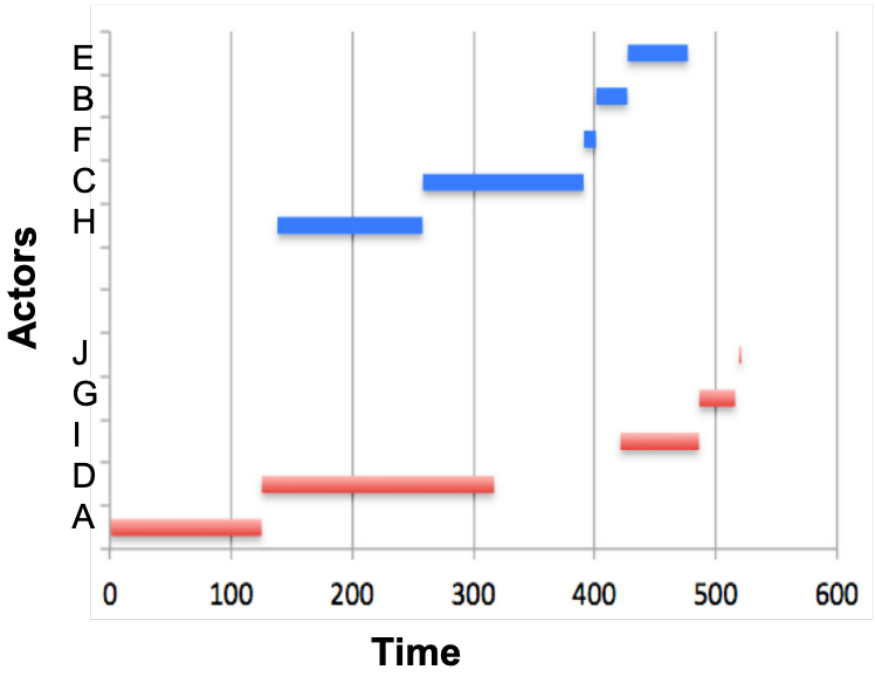


Figure 2.9: The mapping and scheduling decisions illustrated as a Gantt chart for the example previously shown in Figure 2.8 and in table 2.1.

($rank_{up}$) value is selected for mapping. This actor is then assigned to a compute engine based on the earliest finish time (EFT). The calculation EFT is the sum of the execution time of n_i on the compute engine p_k and its earliest start time (EST). The EST of n_i on p_k is the earliest time when n_i can execute on it. This is the maximum of when p_k becomes available for execution (start time), or the time when n_i receives the necessary data at its input channels for execution.

Start time (st) of a compute engine is the earliest time slot when it can start execution of a new actor. HEFT assumes that compute engines are single core processors, only one actor is allowed at a time. Amongst already mapped and scheduled actors, a time slot, if it is earliest can be selected for the new actor to be placed. The authors of HEFT call this an "insertion-based schedule". Since there is no pre-emption allowed in HEFT, the time slots must be minimum to the computation time of the actor on the compute engines.

In the example, the EFT and EST of the compute engines that are selected are listed in table 2.1. The resulting makespan after running HEFT with the mapping and scheduling decisions are illustrated using a Gantt chart in Figure 2.9.

This description of HEFT reveals two major limitations, which are the implications of assuming single core compute engines and all-to-all connected compute engines. The first assumption does not allow resources to be considered for concurrent actor execution. The second assumption prevents the possibilities of channels queuing for the same communication link. Both of these are limitations for agile heterogeneous computing, as concurrent actor execution on a compute engine and specialised connectivity topologies are important aspects of agile heterogeneous computing. The original HEFT assumes a fixed platform architecture. Therefore, the economic costs of different platform architectures are not considered. Some of these restrictions have been removed by various advancements around HEFT which are discussed in the next paragraph.

HEFT enhancements

Shetti et al. 2013 [109] proposed HEFT-NC (no crossover), an enhancement of HEFT for compute platforms with GPUs. The authors note that HEFT can become stuck in a local optimum when the compute engines have very different speeds for typical actors, as would be the case for a CPU and GPU pair. No crossover refers to the prevention of assignment of an actor to a compute engine just based on the EFT. Since the acceleration of an actor can be much higher on an FPGA or GPU than on a CPU, sometimes EFT based compute engine placement may not result to the optimal results. For this reason, the authors have proposed a crossover threshold in the compute engine selection heuristic. This threshold is compared against how much an actor can be accelerated on a GPU and its present EFT. Furthermore, the authors have modified the actor ranking heuristic to reflect that tasks that are large and that have a higher speed-up on a GPU are prioritized first. The combination of this ranking approach and the usage of no-crossover threshold showed reduction of makespan when compared to the original HEFT algorithm. However, HEFT-NC did not consider communication delays between CPU and GPU during the choice of compute engines. These communication delays due to PCIe links can be significant. Thus, they must be included in the design process. Alebrahim et al. 2017 [104] further extended HEFT-NC to include the communication link delays for platforms with GPU. Although the attributes of including an accelerator like a GPU is added in these HEFT extensions, the platform is static and concurrent actor execution is not considered.

Several variations of HEFT are proposed for cloud computing [8, 62, 64]. In cloud, the cost of renting virtual machines is an important consideration, these modifications have incorporated into the objective function this economic rental cost as well as performance. In the MOHEFT [64] extension, instead of assigning an actor to one compute engine, several other options are explored. These extra solutions are then examined in relation to performance rental cost trade-offs. The number of alternative solutions that will be explored for trade-offs is a hyper-parameter which has a lower bound that is defined at the beginning of the algorithm. The selected solution is based on the concept of crowding distance, which provides a measure of possible solutions that are closer to the one selected. The authors have shown that this algorithm preforms better than genetic (evolutionary) algorithms coded in SPEA2 [96]. However, it has a higher complexity as compared to the original HEFT algorithm. This is due to the introduction of a search and evaluation of multiple solutions.

A major drawback of MOHEFT is that the economic cost is based on a rental model, which is coherent with cloud pricing schemes but is not applicable to the embedded domain for agile heterogeneous computing where costs are capital costs and the platform architecture is the one off purchase of its components. The second drawback of MOHEFT is that the compute engines of the platform are assumed to be fully connected and the communication delays are assumed to be uniform. The

pricing of bandwidth in the MOHEFT cost model is also rental based where each byte transferred adds to the time that the compute engine will need to be rented. Bandwidth is thus not priced explicitly and the capital cost of communication links is not explicitly represented as a cost. In other respects MOHEFT is the same as HEFT in that parallel actor executions on a compute engine are not possible.

A later version with lower time complexity than MOHEFT with cloud cost models is proposed by Verma et al. 2015 [62]. In this algorithm, during the selection of compute engines, the one which has the lowest rental cost is simply selected. In this paper the authors have only compared the performance with original HEFT algorithms, whereas an evaluation of this method with other cloud enhanced versions is expected. This algorithm also has similar deficiencies as that of MOHEFT, which is a capital cost of the entire platform architecture is not considered and communication delays are uniform.

There are many other extensions to the HEFT algorithm that have been proposed for cloud applications. For a recent review see [110]. Of particular interest, in a well cited paper, Sakellariou et al. 2007 [8] proposed the Gain/Loss algorithm attempts to move from a high cost schedule to a low cost schedule (LOSS) or a low cost schedule to a higher cost schedule (GAIN) without unduly increasing the makespan using initial schedules created by HEFT. The cost is defined according to a rental cost model similar to that of the cloud and the platform architecture is assumed to be fully connected; because of these two factors, the Gain/Loss algorithm is not directly applicable for agile heterogeneous computing. However, this algorithm shows the avenues to adjust scheduling and mapping decisions by considering new compute engines or by discarding a previously used one. This provides the opportunity to change the platform architecture based on a pricing budget. This can be explained in detail by looking into the pseudocode of the Gain/Loss algorithm, shown in Figure 2.10.

In the Gain/Loss algorithm, the weight of an actor on a compute engine is used to decide whether the actor will be mapped and scheduled on the compute engine. This weight is calculated as the execution time decrease or increase for cost increase or decrease of the gain or loss versions of the algorithm, respectively. If a compute engine was not initially used (in the first HEFT schedule) but satisfies the present budget conditions, then the new compute engine will be used. The same is applicable to discard an initially used compute engine.

Gain/Loss algorithm

```
REQUIRED:
   h. application DAG G,
   i. set of compute engines CE,
   j. actor execution time matrix EXE,
   k. communication delay matrix COMM,
   l. like HEFT, a DAG mapping and scheduling algorithm H, and
   m. available budget, B
ALGORITHM: (two options: GAIN and LOSS)
   1.  if LOSS then
   2.     use H to generate the mapping and scheduling decisions MSD
   3.  else
   4.     generate MSD by mapping the actors onto the cheapest CE
   5.   end if
   6.  for each actor a_i in G do
   7.     for each compute engine ce_j in CE do
   8.          if according to MSD, a_i in mapped onto ce_j then
   9.             A[a_i][ce_j] ← 0
   10.         else
   11.            A[a_i][ce_j] ← EXE_i,j
   12.         end if
   13.    end for
   14. end for
   15. if LOSS then
   16.    condition (cost of schedule MSD > B)
   17. else
   18.    condition (cost of schedule <= B)
   19. end if
   20. while (condition and not all possible re-assignments have been
       tried) do
   21.    if LOSS then
   22.         find the smallest non-zero value from A, A[i][j]
   23.    else
   24.         find the biggest non-zero value from A, A[i][j]
   25.    end if
   26.    re-assign a_i to ce_j in MSD and calculate new cost of MSD
   27.    if (GAIN and cost of MSD > B) then
   28.         invalidate previous re-assignment of a_i to ce_j
   29.    end if
   30. end while
   31. if (cost of MSD > B) then
   32.    use the cheapest assignment for MSD
   33. end if
   34. return MSD
```

Figure 2.10: This Figure shows the pseudocode of the Gain/Loss algorithm [8].

2.5.5 Conclusion

The closest papers to the concept of agile heterogeneous are Pimentel et al. 2006 [55] and Sakellariou et al. 2007 [8]. However, they both have significant deficiencies for agile heterogeneous computing. The first paper [55] considers template-based platform architectures that have limited connectivity topology and the exploration is based on evolutionary algorithm that do not consider the application-algorithm directly. The second paper [8] is for rental cost optimisation, where the underlying mapping and scheduling algorithms do not consider concurrent actor executions on the same compute engine. The connectivity amongst the compute engines are all-to-all which is simplistic for agile heterogeneous computing. In both the algorithms, it is not clear how they would cope with resource consumption of actors on architectures that include components such as GPUs and FPGAs.

The [8, 62, 64] researchers have different assumptions in their architecture model namely zero bandwidth costs that allow an easy encoding for the evolutionary algorithm. Their cost model is also based on rental costs of compute engines rather than capital cost. Most other work on design flows do not envisage the architecture being a design variable. These works also do not take into account the resource consumption of actors that can allow simultaneous execution of multiple actors on a compute engine.

2.6 Conclusion

In this chapter, the literature related to the agile heterogeneous computing design flows were reviewed. The following gaps have been noted:

1. There is no agile design flow that has a close linking between the variable platform architecture and mapping, and scheduling of the application-algorithm. For example SESAME [55] treats the variable architecture quite separately from the mapping and scheduling decisions, whereas research emanating from cloud computing assumes a generalised all-to-all connection infrastructure and are based on rental costing metric.

2. There is no constraint based representation that allows for generalised variable architectures to be explored in conjunction with mapping and scheduling. Generalised platform representations have only been applied to fixed architectures. Variable platform definitions based on tiles and templates are not fully generalised.

3. Current SDF-based scheduling and mapping representations cannot express groups of actors together with their resource consumptions. They are also incapable of representing communication between compute engines, resource

dependences and control actors for the remote management of the actors on another compute engine. Separate graph representations have provision for many of the mapping and scheduling requirements for embedded systems but require separate throughput analysis as they are not based on SDF semantics. SDF compatible representations do not represent resources, thus not allowing the possibility of expressing how groups of actors execute concurrently on a compute engine.

4. Whilst mapping and scheduling has been applied to heterogeneous architectures, compute resources have not been widely included and capital cost metrics are not considered. Further, there has been no integration between HEFT typed algorithms and Gain/Loss algorithms. Evolutionary meta-heuristic approaches to variable architectures do consider resources but not at the same time as mapping and scheduling decisions. List based mapping and scheduling algorithms do not consider resources and specialised communication channels. Gain/Loss algorithm derived from cloud have very simplistic approach to resources, where only one actor is executed on compute engine at a time and all-to-all communication links are assumed to be always available. The cost model for resources in cloud model are based on a rental metric.

Chapter 3

Research methodology

Contents

3.1 Introduction

The previous chapter has revealed that there are significant gaps in the literature with respect to agile heterogeneous computing design flows. A summary of these gaps are:

- Current agile computing design flows do not treat the present platform architecture within the mapping and scheduling context.
- There is no generalised constraint based platform description that would serve as an input to an integrated agile heterogeneous computing design flow.
- Whilst SDF-based representations for mapping and scheduling decisions have been used to integrate applications and the platforms on which they execute, there are many details of embedded platforms that are not captured by current SDF representations.
- Finally, there has been no comprehensive incorporation of the requirements of agile heterogeneous computing into modern list based mapping and scheduling algorithms.

In this chapter, research questions aimed at filling these gaps that arise from the literature review are presented together with the methodology adopted to answer each question. The methodology is based on the general framework suggested by Crnkovic [111] and shown in table 3.1. The types of questions, strategy to answer the question and the method to validate the work used to answer each question are described in the next section.

Table 3.1: The categories of research questions, strategy/result and validation identified by suggested by Crnkovic [111].

Question	Strategy/Result	Validation
Feasibility Does X exist and what is it? Is it possible to do X at all?	Qualitative model Report interesting observations Generalise from examples Structure a problem area	Persuasion I thought hard about this, and I believe . . .
Characterisation What are the characteristics of X? What exactly do we mean by X? What are the varieties of X and how are they related?	Technique Invent new ways to do some tasks, including implementation techniques Develop ways to select from alternatives	Implementation Here is a prototype of a system that . .
Method/means How can we do X? What is a better way to do X? How can we automate doing X?	System Embody result in a system using the system both for insight and as a carrier of results	Evaluation Given these criteria, the object rates as . .
Generalisation Is X always true of Y? Given X, what will Y be?	Empirical model Develop empirical predictive models from observed data	Analysis Given the facts, here are the consequences
Selection How do I decide whether X or Y?	Analytic model Develop structural models that permit formal analysis	Experience Report on use in practise

3.2 Research questions

In this section, the research question addressed by this thesis are presented. The research questions are as follows:

1 **Is it feasible to create a design flow for agile heterogeneous computing where architectural exploration is closely integrated into mapping and scheduling decisions?**

According to the categories of the research questions described in table 3.1, this is a *feasibility* question. The strategy to answer this question is to develop a new *technique* by combining architectural exploration with mapping and scheduling decisions. The question will be validated by *implementing* and *evaluating* the new technique using a metric to measure the price of the platform architecture and the performance of the application-algorithm on the platform architecture. Capital cost is used to measure the price of the platform architecture. Makespan and throughput is used to measure the performance of the application-algorithm.

2 **How to incorporate in a design flow, for agile heterogeneous computing, a constraint based platform definition?**

This is a *method* question which will be answered by constructing a *system*

which embodies a constraint based architecture inputs and generated deployable implementations. The validation of this question will be through its successful *implementation* in a prototype.

3 **How to represent simultaneously the application, architecture, mapping and scheduling decisions, and deployment details of an agile heterogeneous computing solution?**

This is a *method* question which will be answered by embodying a data structure to represent the application, architecture, mapping and scheduling decisions and deployment details within a full design flow *system*. The representation will be considered validated once it has been successfully *implemented* in the prototype.

4 **How to best perform mapping and scheduling in the context for this new design flow for agile heterogeneous computing?**

This is a *method/means* type of question. It will be answered using the strategy *technique* of incorporating the new algorithms into a system. The *evaluation* of this technique will be performed by measuring metrics based on capital cost and makespan. Makespan is used instead of throughput [1] because the former is widely used in the mapping and scheduling literature.

5 **How can an agile heterogeneous design flow incorporate decisions concerning the sharing of the compute engine resources amongst multiple concurrent actors?**

This is a method/means type of question. It will be answered using the strategy technique for incorporating resource constraints into mapping and scheduling that inherently allow the concurrent allocation of actors to a compute engine. The evaluation of this technique will be performed in the same way as the evaluation of the previous research question; i.e measuring metrics based on capital cost and makespan.

The aforementioned research questions are summarised in Figure 3.1 along with the steps to categorise the research question, then selection of a strategy that will lead the question being answered and finally selection of a validation approach to verify the results obtained.

[1] Makespan and throughput are related as, $throughput_{min} = \frac{1}{makespan}$, where $throughput_{min}$ are the minimum possible throughput [60].

Research questions	Question category	Research Strategy	Validation approach
Q1. Is it feasible to create a design flow for agile heterogeneous computing where architectural exploration is closely integrated into mapping and scheduling decisions?	Feasibility	Technique	Implementation and evaluation
Q2. How to incorporate in a design flow, for agile heterogeneous computing, a constraint based platform definition?	Method / Means	System	Implementation
Q3. How to represent simultaneously the application, architecture, mapping and scheduling decisions, and deployment details of an agile heterogeneous computing solution?	Method / Means	System	Implementation
Q4. How to best perform mapping and scheduling in the context for this new design flow for agile heterogeneous computing?	Method / Means	Technique	Evaluation
Q5. How can an agile heterogeneous design flow incorporate decisions concerning the sharing of the compute engine resources amongst multiple concurrent actors?	Method / Means	Technique	Evaluation

Figure 3.1: The research questions along with their categories, research strategy and the validation approaches. This table is constructed on the basis of the research Crnkovic's [111] research methodology.

3.3 Research methodology

In the previous section, the research questions were defined and an overview of the methodologies to answer them were described. In this section, the research methodology to address the research questions are discussed in detail. This section is organised into five subsections where every research question is discussed separately.

3.3.1 A design flow for agile heterogeneous computing

The overarching research question of this thesis is:

1. **Is it feasible to create a design flow for agile heterogeneous computing where architectural exploration is closely integrated into mapping and scheduling decisions?**

As described in the previous section that this is a *feasibility* question which will be answered by creating a new technique and that will be validated using a combination of implementation and evaluation.

The steps in creating the *technique* comprises of first conceiving the new design flow by generalising the previous Y-chart design methodology [41], so that variable platforms can be included. The next step is to consider the mapping and scheduling

decisions in conjunction with the exploration of the platform architecture. The third step is to define the key components of the new design flow and the new elements required in the creation of this new design flow. The fourth step is to create the new elements for the realisation of the design flow. The final step is to create a prototype of the design flow system. The validation of the new design flow is done is two stages, where the design space exploration is first evaluated with synthetic and real application DAGs. Then in the the second stage, the complete design flow that includes automated deployment is evaluated with CACTuS [112] a published hand crafted multi-object tracking application.

This research question is primarily answered in Chapter 4 where the new design flow called *Agile heterogeneous computing flow* (AhcFlow) is conceived. Also, the necessity of the following new elements were described: a new agile platform representation called *Parameterised platform graph* (PPG), a new intermediate representation called *Architecture augmented synchronous dataflow* (ArcSDF), a design space exploration algorithm called Agile mapping and scheduling algorithm (AMS) and a deployment technique. The validation of the new design flow will be conducted in parts, in Chapters 6 and 7, after these new elements are created.

In order to answer this question, the variable platform architectures are limited to the ones that can be created by one or more of the following compute engines: FPGA, GPU and CPU. It is assumed that these compute engines are connected with each other using PCIe links and executions on GPU and FPGA are controlled from the CPU.

3.3.2 Constraint based platform representation

In order to realise the new design flow (AhcFlow), a new constraint based platform representation is necessary, which brings to the second research question:

2. **How to incorporate in a design flow, for agile heterogeneous computing, a constraint based platform definition?**

This question is introduced in section 3.2 where it is categorised as a *method* question, the strategy chosen to answer it is incorporating an agile platform representation within the design flow *system*. The validation technique is its successful *implementation*.

At first an agile platform representation called *Parameterised platform graph* (PPG) is created. As PPG was one of the new elements required for AhcFlow, its key requirements will already be defined while presenting AhcFlow. These requirements will be used to create PPG. Once PPG is created, it will be validated by using it to construct the prototype of the new design flow. PPG will be one of the foundations to create the design space exploration algorithm and the deployment technique. Both of these stages of the design flow will be tested with a large

number of synthetic DAG datasets, some real application DAGs and the CACTuS application.

This question is answered primarily in the first section of Chapter 5 where PPG is defined and its usage in the definition of ArcSDF is also described. PPG will be implemented and used for design space exploration on Chapter 6. In this chapter, synthetic datasets will be used for the evaluation of the design space exploration algorithm. Then, in Chapter 7 it will shown the PPG can be used for real platform architectures.

3.3.3 Intermediate data structure

Another key ingredient of AhcFlow is an intermediate data structure (ArcSDF) that can represent the application and all the decisions to deploy it on a platform architecture. Thus the third research question is:

3. **How to represent simultaneously the application, architecture, mapping and scheduling decisions, and deployment details of an agile heterogeneous computing solution?**

During the definition of the research questions, this question is categorised as a *method* question. The strategy chosen is integrating the intermediate data structure (ArcSDF) within the AhcFlow system and the validation is its successful *implementation*.

The first step to answering this research question will be to define the architectural decisions, which include the mapping and scheduling decisions and all other decisions that are necessary to deploy a dataflow graph (SDF) on a heterogeneous platform. The PPG representation will be used to represent agile platform architectures for the identification of the architectural decisions. Once the architectural decisions are defined, they will then be used to create a new dataflow representation that can express the application-algorithm together with the architectural decisions. This will be achieved by augmenting the original SDF graph to express the architectural decisions. The ArcSDF graph is verified mainly by incorporating it with the design flow prototype.

The ArcSDF graph will be defined in Chapter 5 where it will be used to conduct various other analysis that goes beyond throughput prediction. After the definition, in Chapter 6, ArcSDF and its analysis approaches will be used in the design space exploration algorithm. In this chapter, a large number of synthetic and real application DAGs will be used for the ArcSDF validation. In Chapter 7 ArcSDF will be used for deployment using real applications.

3.3.4 Agile mapping and scheduling algorithm

The algorithm to conduct design space exploration for the new AhcFlow design flow raises the following research question:

4. **How to best perform mapping and scheduling in the context for this new design flow for agile heterogeneous computing?**

In section 3.2 this research question is categorised as a *methods/means*, where the question will be answered by developing a new *technique* and this new technique will be validated using *evaluation* of metrics-based on makespan and capital cost.

Previous mapping and scheduling algorithms for variable platforms have considered the platform architecture selection separate from the mapping and scheduling decisions. However, the design space exploration algorithm in AhcFlow evolves the platform architecture with the mapping and scheduling decisions. This design space exploration algorithm called the agile mapping and scheduling (AMS) algorithm will be developed in three steps. In the first HEFT will be extended to incorporate resources so that it can allow concurrent execution of actors on the same compute engine. Then, in the second step, HEFT will be generalised to incorporate specialised connectivity topologies of agile heterogeneous computing. Finally, in the third step this advanced HEFT will be merged with Gain/Loss algorithm, so that platform architecture instances can be created within the mapping and scheduling algorithm. The Gain/Loss algorithm is also adopted for capital cost from the existing rental cost model.The AMS algorithm will be evaluated first for the advanced HEFT and next for the overall design space exploration algorithm. Advanced HEFT with modified internal heuristics to consider resources will be compared to a version of advanced HEFT with original internal heuristics. The overall AMS algorithm will be compared against random brute-force approach with two approaches of AMS (called expansion and reduction). In order to conduct these experiments, a prototype will be constructed which will also show the feasibility of the new design flow and the underlying representations. The prototype will support a framework to generate many synthetic datasets and also allow real application data.

The AMS algorithm will be constructed in Chapter 6 where all the stages of its development will be described. Then the prototype to conduct the evaluation experiments will be presented. Finally, the evaluation results for advanced HEFT and the overall AMS algorithm will be discussed. An exhaustive case study with a complex visual tracking application called CACTuS will be presented in Chapter 7, where the design space exploration will be compared with published hand crafted results. Furthermore, the predictions from AMS will be matched with the actual deployed results.

3.3.5 Sharing of compute engine resources

The compute engines (FPGA, GPU and multicore CPU) can be fully utilised if multiple actors can execute concurrently, which raises the following question:

5. **How can an agile heterogeneous design flow incorporate decisions concerning the sharing of the compute engine resources amongst multiple concurrent actors?**

Like the previous research question, this question is also categorised as a *methods/means*, answered by developing a new *technique* and validated using *evaluation* of metrics-based on makespan and capital cost.

It is noted that this question intersects all the previous research questions. Hence the technique will be developed across PPG, ArcSDF, AMS and the deployment technique. For PPG, the platform representation will be created to express resources present in a compute engine. ArcSDF will be created to have features that can represent resource consumption of actors on a compute engine. This ability will enable further analytical possibilities with the incorporation of resources. Since the amount of resources an actor requires determines other actors to be placed on the compute engine, in the AMS algorithm, resources will be introduced at the starting of the AMS construction. The deployment will also take the resources expressed by the ArcSDF into account. This technique will be evaluated along with the AMS algorithm by using synthetic and real application DAGs to compare the the benefits of the resource-based enhancements.

This question will be answered across Chapter 5, 6 and 7. In Chapter 5, this question will be addressed in the PPG and ArcSDF, whereas in Chapter 6 resource based heuristics will be developed for AMS and evaluation will be conducted in Chapter 6 and 7.

3.4 Conclusion

This chapter presented the research questions to fulfil the gaps identified in chapter 2. Then methodologies to answer the research questions and to evaluate the answers are described. It is also described how this thesis is organised to answer the research questions into next four chapters. In the rest of the chapters, the research methodologies will be the basis of answering the research questions.

Chapter 4

A design flow for agile heterogeneous computing

Contents

4.1 Introduction

In the previous chapter, the review of the literature on design flows for heterogeneous computing showed that whilst progress has been made on the separation of concerns between the algorithm and the platform architecture (using the Y-chart approach [6]), there has been less attention focused on platform architectures that can change. Of the papers that discuss variable architectures, [55] proposed a limited topology template architecture which is evolved using a genetic algorithm without considering the details of the platform architecture and the application-algorithm, whereas list-based mapping and scheduling algorithms [7, 62, 113] consider a simplified platform architecture that merely consists of the number and the "power" of processors to form the final deployment platform. Cloud mapping and scheduling algorithms, such as HEFT [7], however takes into account the details of the simplified platform architecture which may contribute to better outcomes with lower complexity, as compared to the evolutionary approach used by SESAME [55].

This chapter presents a new design flow called the *agile heterogeneous computing flow* (AhcFlow) that proposes to use the constraints of a platform architecture as a design flow input rather than a fixed platform architecture. The input "architecture" is expressed as a set of constraints to create the platform architecture. These constraints allow more platform architecture options to be considered as compared with the template-based approaches in [55]. Furthermore, these constraints enable a design space exploration approach that can generate platform architecture instances by taking into account the details of the application-algorithm and the platform constraints. The new design flow, AhcFlow can be viewed as a generalization of the Y-chart approach [6], where the fixed platform architecture is replaced with the a representation of the set of platform architecture constraints. The platform constraint representation, called the *parameterised platform graph* (PPG) is derived from the platform components desired by the designer. PPG is presented in the next chapter.

Another input to AhcFlow are the pre-engineered (implemented) actors of the dataflow model that represents the application-algorithm. It is assumed that all the actors have an implementation for the compute engines that the designer wants to consider. It is not necessary to have implementations for every actors on all the available compute engines; however if an implementation is missing for a particular actor on a specific compute engine, then this option will not be considered in the design space exploration. Due to the presence of these implementations at the start of the design flow, performance and capital cost data is available to use in the design space exploration stage.

Design space exploration can be viewed as the central organizing principle in the new design flow (AhcFlow). The design flow proceeds from the inputs of the application-algorithm dataflow model, platform constraints (PPG) and actor performance data, to the stage of design space exploration using an algorithm inspired

by HEFT. The new algorithm, called the *agile mapping and scheduling* (AMS) algorithm, performs platform architecture instance generation along with mapping and scheduling decisions, so that the platform architecture is a design variable consistent with the supplied constraints. To facilitate this approach, each instance of platform architecture is embedded in a single representation that includes the application-algorithm with the mapping and scheduling decisions. This representation is called *architecture augmented synchronous dataflow* (ArcSDF). It is discussed in more detail in the next chapter.

After the design space exploration is concluded, the output is preserved in a deployment neutral form, as the final ArcSDF representation. The final stage of the design flow is deployment that transforms the final ArcSDF to a runtime using the set of pre-engineered actors and the vendor specific tools of the compute engines. The runtime is targeted for the platform architecture instance corresponding to the final ArcSDF, which is essentially the platform architecture instance selected by the design space exploration (AMS) algorithm.

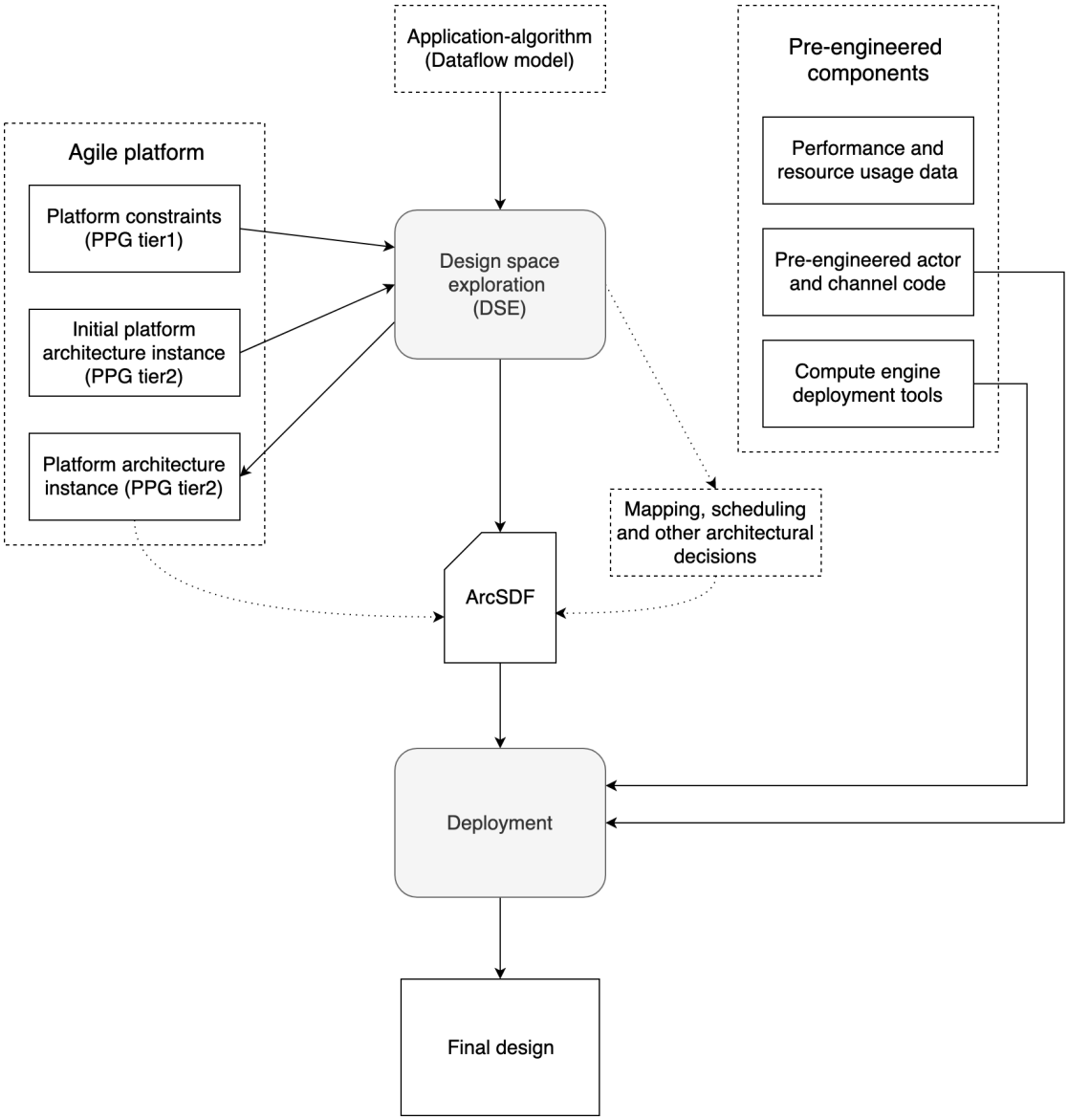


Figure 4.1: This figure shows AhcFlow, the design flow for agile heterogeneous computing.

A short description of the new design flow described in 4.1 is as follows.

The design flow begins with three manually created inputs. The first is the dataflow representation of the application-algorithm, the second is the representation of the platform architecture constraints (later referred to as the tier1 of the PPG model) and the third is a collection of pre-engineered actor's performance, capital cost and resource usage data. The pre-engineered actors have implementations that are used during deployment, whilst their other data is used for the design space exploration (DSE). Various kinds of application dataflow representations are allowed and these are converted to the format that the design space explorer can understand.

The AMS algorithm is used for the exploration of the design space. It searches for

the best mapping and scheduling decisions including a suitable platform architecture. The search starts with an initial instance of the platform architecture that is provided as input. This initial platform architecture instance is called tier2 of the PPG model. The designer can provide an initial platform instance as small or as large as they think fit. The initial platform instance provides a hint as to where the AMS algorithm commences its search. The first iteration of the AMS algorithm creates an initial ArcSDF based on the initial platform instance. This initial ArcSDF is then used to estimate performance and the capital cost. Depending on the estimated values, the initial platform architecture is either expanded or reduced by including or excluding compute engines, respectively. These steps are iterated several times before the final platform architecture instance is chosen. This architecture will then be used for deployment. The maximum number of iterations is provided by the designer but the algorithm can also stop if the theoretical best makespan of the application-algorithm is reached with a certain capital cost budget.

After the AMS algorithm stops, the design flow progresses to the deployment stage. The pre-engineered implementation for all actors and channels are stored inside the repository and the deployer reads the final ArcSDF to retrieve the required code. The actor code chosen depends on the compute engine selected by the AMS algorithm. The deployer then connects the actor code on each compute engine with generated channel code. Channel code may use pre-engineered libraries for communication between compute engines that is consistent with the constraints defined in tier1 of the PPG model. When this channel code is combined with actor code, it forms the basis of the runtime. Developers need to conform to an abstract actor class exposed by the application programmer's interface (API) when developing the pre-engineered actors. This allows the integration of the actor code to be automated by the deployer. It is assumed that channel code libraries implementing the actor interface API have been pre-engineered for each compute engine upon which an actor might be deployed.

The remainder of this chapter, which expands on the description of the AhcFlow, is organized as follows. In section 4.2, all the inputs to the design flow are described. At first the specifications of the new agile platform constraint representation (PPG) is presented. Then the application-algorithm is introduced as a dataflow graph. Next, a summary of the pre-engineered actors of the dataflow graph is presented. With these pre-engineered actors, their performance, capital cost and resource usage data are also included. In section 4.3, the specifications of the design space exploration part of the design flow is presented. In section 4.4, the requirements of the deployment part of the design flow are specified.

4.2 Inputs to the design flow

This section describes the inputs that the designer must provide to the new design flow (AhcFlow). These inputs are agile platform representation, dataflow representation of the application-algorithm, and pre-engineered actors and channels of the application-algorithm. Amongst all these inputs, the agile platform representation entails a new approach that allows variable platform architectures to be considered in the design flow and also provides the basis to express concurrent execution of multiple actors on the same compute engine. In this section, there are three subsection, where each of the inputs are described in a separate subsection.

4.2.1 Agile platform representation

The major innovation of the design flow is the introduction of a generalised definition of the platform architecture that allows for a wide range of platform architecture instances to be considered. The most significant parts of this new platform architecture definition is a set of constraints that defines the limits of the architectures that will be considered in design space exploration. Next the architecture definition has a set of capital costs associated with potential execution engines to be used in an instance of the architecture. Finally an initial platform architecture is contained in the definition that conforms to the previously mentioned constraints. The platform architecture that starts with the initial platform instance and is an outcome of the design space exploration; needs to conform the constraints of the number of and connections between the compute engines.

There are two types of constraints that can be included. The first type of constraint is limitations inherent when a particular compute engine is chosen. Formation of the platform architecture is viewed as the connection between various compute engines and the system memory. The review of agile heterogeneous computing in section 2.2 shows that there can be more than one type of connection between the compute engines. Furthermore, a compute engine has limitations on the kind and number of other compute engines it can be connected to. The size of the system memory in the platform architecture depends on the compute engines of type CPU used. These connection restrictions are also required within the platform model along with the available platform components.

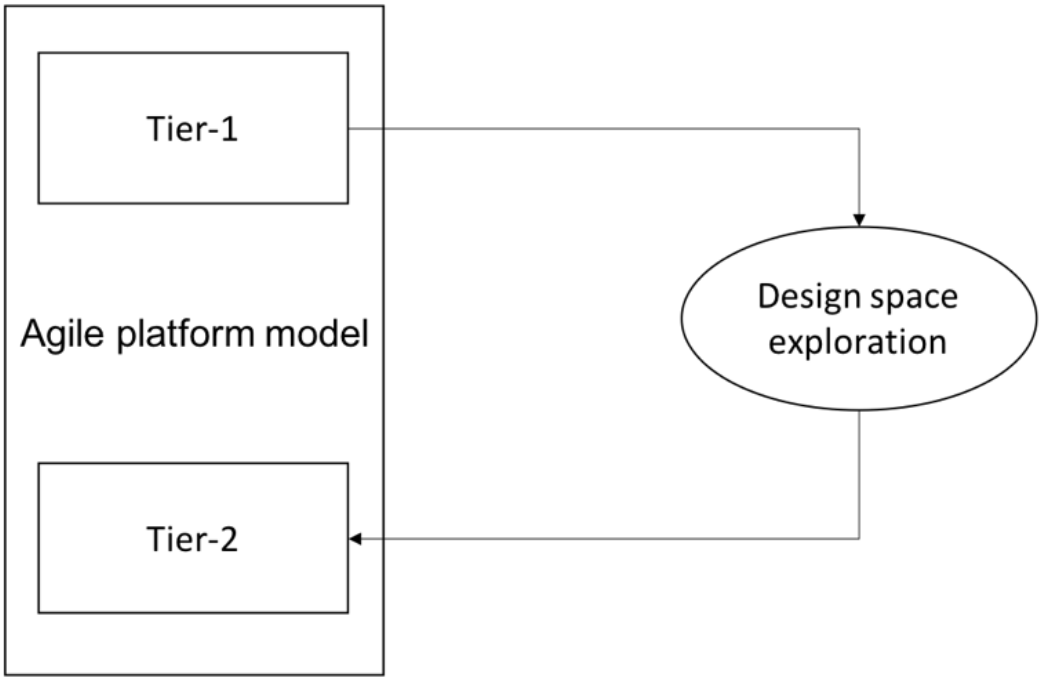


Figure 4.2: Agile platform representation with two tiers and their interaction with the design space exploration algorithm.

The second type of constraint is provided by the designer on the number of compute engines to be used; for example, a design should only use a maximum of two CPUs and 3 GPUs.

A unique feature of the agile platform representation and as a consequence, of the proposed design flow is that multiple concurrent actors can be mapped to the same compute engine. This feature recognises the capabilities of modern GPUs and FPGAs which are capable of executing multiple actors in parallel. A *compute zone* is a term that is used here to mean a part of a compute engine and its associated resources that can be allocated to an actor. The design flow also allows for platforms where compute zones can be dynamically created and destroyed. This feature is only practical for compute engines such as GPUs which can switch kernel software quickly as the algorithm is executed. Even on modern GPUs for this to be a practical consideration the time it takes to switch kernels must be comparable to the compute time of a particular kernel; otherwise a possibly slow switch time will come to dominate the performance on that platform architecture. However, there may be situations where the GPU kernels are not on the critical path for the algorithm as a whole in which case the switch time may be covered by other parallel operations on other compute platforms. In the case of FPGAs dynamic compute zones correspond to partial reconfiguration which on most modern FPGAs is considered as too slow[1]

[1]The minimum time to reconfigure a reconfigurable area of an FPGA that can only accommodate a double precision floating point adder is approximately $300\mu s$ [114]. This time significantly

to be incorporated into the main data path of the algorithm. The information about the capabilities of compute engines to support compute zones is contain in the same files as the platform constraints.

The concept of agile platform thus uses an abstract platform model. The constraints are denoted tier1. Tier1 reflects the available platform components, along with their limitations of connectivity and the system memory. This limitation control the way the platform components are connected to form the platform architecture. It can be said that the tier1 expresses the entire design space available. Based on it, the design space exploration algorithm finds a suitable platform architecture, which is the components and the way they are connected together. Tier2 of the agile platform model represents the initial platform architecture instance. The DSE algorithm creates and evaluates the possible multiple platform architectures constrained by a tier1. A starting point of the exploration, which is a tier2, is provided by the designer. This concept of agile platform as a two tiered model is shown in 4.2.

4.2.2 Application algorithm

The input of the application algorithm is the most straightforward aspect of the proposed design flow. The selection of the dataflow paradigm for algorithms is widely adopted to represent highly parallel embedded applications and from the literature review there seems to be no compelling competitor approach. The only choice seems to revolve around various forms of data flow representations. Synchronous dataflow (SDF)has the advantage of static analysis that allows buffer sizes to be determined statically and there is a well-founded theory that enables prediction of performance before the deployment [60, 76]. Another benefit of using SDF is that it can be converted to a dataflow acyclic graph (DAG). The conversion algorithm is described in [60]. This conversion allows the standard mapping and scheduling algorithms to be directly applied for design space exploration.

There are a number of different representations that have evolved from synchronous data flow, such as CSDF [58], PiSDF [115] and PiMM [78]. These dataflow representations, as previously mentioned in the literature review, can be statically analysed by converting to an equivalent SDF and finally to a DAG. The design flow (AhcFlow) has elected to use the SDF model [57] as the basis of representing application-algorithms. The reasons for the selection of the SDF model is that it is extensively used to express parallel applications and is widely supported in the research community with analysis algorithms. Due to these reasons, the SDF representation is also used as the basis of the intermediate data structure (ArcSDF). Since the design space exploration algorithm is based on ArcSDF and the primary input of the deployment module is ArcSDF, using other representations that can be converted to SDF does not require modifications to the overall design flow.

increases with the size of the reconfiguration area.

4.2.3 Pre-engineered components

There are two types of pre-engineered components those that implement actors and those that implement communication channels between actors executing on compute engines. The performance of a component on each compute engine, its resource usage on each compute engine and its pre-engineered code for deployment are inputs to the design flow.

The advantage of assuming pre-engineering of individual components of a large application is that it avoids the need for the design flow to consider the specific details of how best to implement an actor or communication channel when this is likely best done by a person with expertise in each case. For an instance, a GPU developer may not be the right candidate to build the same actor on FPGA. It is also possible to have different implementations of the same actor for the same compute engine. Different implementations are possible when either they are different from a theoretical perspective, such as a convolution actor can be implemented in spatial domain or in the frequency domain, or when the implementation is performed with a different architecture. This choice of selecting the right actor implementation is dependent on the platform architecture and the application-algorithm.

Insights from the literature review pertaining to the development challenges of heterogeneous computing also point the failure of technologies that claim automatic parallelisation of existing sequential code and the wider acceptance of component-based design flows. Whilst well-know from the software engineering domain [116], component based design has also become the basis of development with systems consisting of hardware and software parts [117, 118].

Multiple implementations of an actor or a channel can exists in this design flow for different compute engines. For some actors there may be no pre-engineered implementation on a particular compute engine or there may be several versions or no pre-engineered communication code between two particular compute engines. More than one implementation of a certain component for a similar compute engine are useful if each is optimised for different input token sizes. The design flow needs to have avenues to specify the implementation types. Since the actors and the channels are pre-engineered, the information related to their performance and the resources required is known statically. The components of an application-algorithm are built by experts. Information regarding resource consumption and execution times are thus pre-determined and available to be used in the design space exploration.

4.3 Design space exploration

In agile heterogeneous computing, the platform architecture is an additional design variable which must be optimised together with mapping and scheduling of the application-algorithm on it. Adding and removing different types of compute engines and their associated internal connections is as important architectural decision, as is changing which actors run on what compute engine and when they run. Design space exploration needs to find the optimal platform architecture and the set of architectural decisions required to implement the application-algorithm on it, so as to meet the performance and capital cost targets set by the designer.

Design flows that are based on dataflow models are reviewed in section 2.3. In these design flows, the platform architecture is represented as a fixed high-level model, which is then used for the design space exploration. The models that represent the application-algorithm and the platform architecture are separate. As also stated in the literature review; previous approaches to design space exploration when the architecture was a design variable were of two types. The first type had a very simplified choice of architectural decisions (such as just the number of CPUs connected to generic but ill-defined bus). The second type performs optimisation of the architecture using a generic optimisation package (typically an evolutionary algorithm) that did not make use of any specific detailed information available from the mapping and scheduling optimizations. In the latter case, the only information passed from the mapping and scheduling optimiser to the architecture optimiser was the performance (and possible cost) on the decisions with reference to current architecture. Thus the architecture optimiser could not gain any insights from the current mapping and scheduling algorithms as to where to look for the most promising changes to the platform architecture. In this proposed design flow, the platform architecture decisions are integrated with the mapping and scheduling decisions. This is enabled by the usage of an internal data structure (ArcSDF) that integrates these decisions. The proposed design space exploration algorithm is shown in Figure 4.3

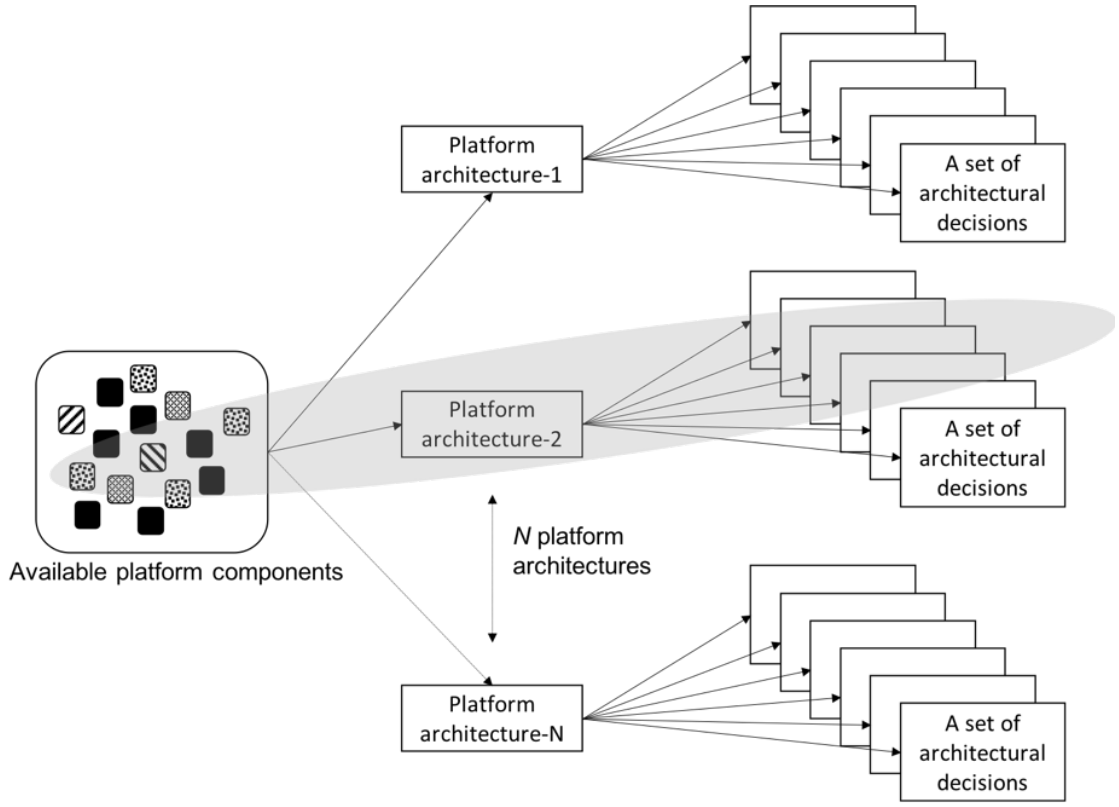


Figure 4.3: Joint design space exploration of platform architecture and corresponding architectural decisions. The shaded region shows the exploration without iterating all the possible platform architectures.

Furthermore, the design space exploration algorithms used in the previous design flows assumed that only one actor can be executed at a time on a compute engine. The reason being that compute engines were assumed to be simply a sequential processor. This however has changed with modern compute engines, where actors can be executed concurrently depending on the resource usage requirements of the actors mapped. It is shown in the literature review that previous mapping and scheduling algorithms that were used for design space exploration cannot be directly used and indirectly usage is cumbersome requiring extreme modelling effort, which defeats the purpose of using a design flow. In the proposed design flow, the resource usage of the actors are taken into account allowing more than one actor to concurrently execute on the same compute engine. This is achieved by enabling ArcSDF to express resource usage of actors and by extending a widely used list-based mapping and scheduling algorithm known as heterogeneous earliest first (HEFT) to incorporate resource usage. The extended version of HEFT, called resource-HEFT (rHEFT), is a part of the overall design space exploration algorithm called the agile mapping and scheduling (AMS) algorithm. Both of these algorithms are presented in chapter 6.

The AMS algorithm starts by using an initial instance of the platform architecture

provided by the designer that already conforms to the constraints set in agile platform input model. Combining the initial instance with the application-algorithm and the pre-engineered component library; a mapper and scheduler (which is part of the DSE algorithm) generates the first version of the ArcSDF representation of architectural decisions. The application-algorithm is converted to a dataflow acyclic graph (DAG) so that standard mapping and scheduling algorithms can be applied. Then using data structures already set up by the mapper and scheduler, the DSE algorithm makes a change to the initial platform architecture instance to move towards a more favourable performance or capital cost design point. The new design point results in a new platform architecture instance and then to its associated ArcSDF instance after a further iteration of mapping and scheduling algorithms. Thus at each iteration of the DSE algorithm a new platform instance is created. The mapping and scheduling part of the DSE algorithm updates the ArcSDF representation based on the new instance.

Note that at any stage after scheduling and mapping, the platform architecture instance could be recovered directly from the ArcSDF instance. The actual platform architecture instance is a convenience rather than an essential definition. At each iteration, estimates emerge for overall capital cost and performance. These estimates along with the decisions on platform architecture instance are based on internal data from the mapping and scheduling algorithms.

The design flow begins with two separate high-level representations, first one is for the constraints of the possible agile heterogeneous platforms and second one is for the application-algorithm. The design space exploration algorithm acts on these models to find the most suitable platform architecture and the set of suitable architectural decisions.

4.4 Deployment

The deployer receives the intermediate data structure containing all the information of the architecture, it must read the ArcSDF instance and retrieve necessary actor code from the repository. In order to connect the actors together, the channel information must also be read to create the required channels. During this code retrieval and creation of the connection code, the deployment technique must also create necessary threads that support multiple compute zones.

This design flow requires a new deployment approach because of the introduction of compute zones within compute engines such as GPUs and FPGAs. Previous research reported in [119] has results which need to be adapted for the presence of compute zones. In this proposed design flow the concurrent processor of [119] loosely correspond to compute zones. They don't correspond exactly because compute zones, unlike concurrent processors are based on the idea of a certain amount of compute engine resources, which are released after the actors within a compute

zone completes execution. Compute zones are defined in chapter 5 with ArcSDF. Following [119] the design flow allows for a set of actors to execute sequentially on a compute zone. Actors that must execute in parallel must of course be mapped to different compute zones.

In a classical SDF mapping and scheduling algorithm, where the compute engines do not have compute zones, the only way concurrent actors can be allocated is as if they are scheduled to run sequentially. In fact it has been shown [119] that the only schedule required is the sequential schedules of actors on the each of the processing cores that constitutes the compute engine. Along with the execution behaviour expressed through ArcSDF, this simplicity of the SDF schedule forms the basis of this deployment technique. The deployment technique must read schedule information from the ArcSDF instance and generate code that maintains this schedule. This is achieved through a runtime that orchestrates the actors on different compute engines. It is important that this runtime is lightweight so that actor performances are not affected. This lightweight runtime must also conduct the initial actor placement and the initial thread spawning. One of the threads of the CPU must be assigned to execute the runtime.

The final specification of the deployment technique is to validate the ArcSDF instance to ensure that the platform architecture has all the resources and communication links for its deployment. Since ArcSDF contains the architectural decisions, the compute engines required and the amount of resources needs for execution can be retrieved.

4.5 Conclusion

This chapter presented a new design flow called *Agile heterogeneous computing flow* (AhcFlow) that considers platform architectures as a design variable. This design flow is built by generalising the Y-chart design methodology [6] to consider constraints of a platform architecture instead of a fixed platform architecture. At first, the overall structure of the new design flow was presented. Then, its inputs and the major stages of design space exploration and deployment were described. It was identified that in order to realise AhcFlow, the following new elements are required:

1. A representation to express platform constraints called*Parameterised platform graph* (PPG) was conceived. The key requirement of PPG is the ability of express platform constraints and a platform architecture instance at high-level.

2. A data structure called *Architecture augmented synchronous dataflow graph* (ArcSDF) that can express the application-algorithm together with the architectural decisions is required for the design space exploration and to automate

the deployment.

3. A design space exploration algorithm called *Agile mapping and scheduling* (AMS) algorithm that will be able to explore the platform architecture closely with the mapping and scheduling decisions of the application-algorithm.

4. An automated deployment technique that can automatically generate a run-time from the results of the design space exploration algorithm.

These new elements of the AhcFlow are presented in the next three chapters. In chapter 5, *Parameterised platform graph* (PPG) and *Architecture augmented synchronous dataflow* (ArcSDF) graph are presented. In chapter 6, *Agile mapping and scheduling* (AMS) algorithm is presented along with its evaluation. In chapter 7, along with an overall evaluation of the new design flow, the deployment technique is described.

Chapter 5

Representing platform architecture and architectural decisions

Contents

5.1 Introduction

A design flow for agile heterogeneous computing was presented in the previous chapter. The new design flow calls for new representation of the agile platform and the intermediate data structure used in design space exploration. These representations are explained in this chapter. The agile platform is represented by the *parameterised platform graph* (PPG). The PPG has two tiers as described in the previous chapter, *tier1* is a set of constraints for the platform and *tier2* represents the initial instance of the platform architecture but is also updated as the design space exploration proceeds. The *architecture augmented synchronous dataflow graph* (ArcSDF) is the intermediate data structure that the design space exploration stage uses and which is passed to the deployment stage. The PPG is described first because the ArcSDF depends on it.

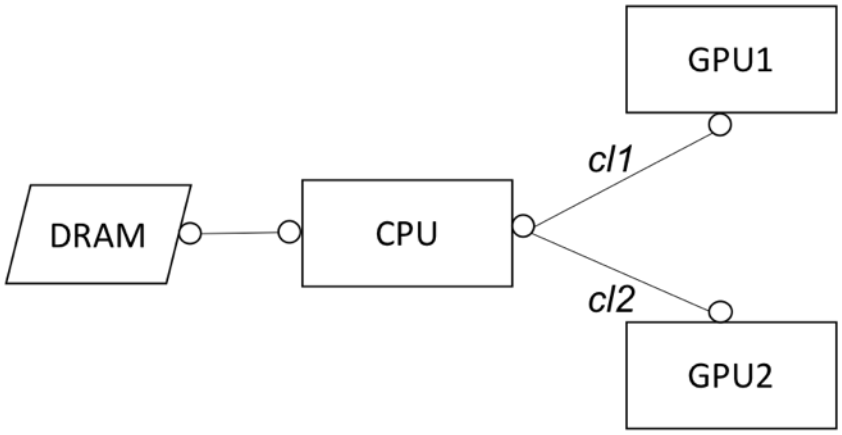


Figure 5.1: A simple platform architecture instance, where two GPUs are sharing the same dock. This shows possible contention between *cl1* and *cl2*. The communication links are the straight lines, the compute nodes are the rectangles, the parallelogram is the system memory node and the smaller circles are the docks.

5.2 PPG: Parameterised platform graph

In this section, a high-level model to represent the platforms for agile heterogeneous computing is presented. This representation underpins the principles of an agile platform for heterogeneous computing. The platform model consists of two tiers. The first tier (tier1) represents the available compute engines, their resources and the restrictions in connecting them together, which serves as one of the primary inputs for design space exploration. There are thus a multitude of platform architectures possible whilst conforming to the tier1 definition. A platform architecture instance conforming to tier1 is represented in tier2. The user supplies the first version of tier2 and the design space exploration updates this as it progresses. Prior to explaining tier1 and tier2 in detail an overview is provided.

5.2.1 Overview

Whilst conceptually tier1 provides the framework for the tier2 instance it is convenient to explain the structure of the tier2 instance first.

Tier2 is a graphical model of a platform architecture instance, where the platform architecture is shown as compute engines linked together by communication links. It is best represented as a graph with nodes representing the compute engines and the system memory and the edges representing communications pathways between them. An example of a tier2 graphical model is shown in Figure 5.1.

The building blocks, which are the available nodes and edges to create the graphs

are selected from tier1.

The available nodes and the edges in tier1 all have attached properties which are denoted as elements of nodes and edges. Examples of elements of nodes include the cost of a compute engine and or the characteristics of a compute engine such as number of cores if it is a CPU or the number of configurable logic blocks for an FPGA. The maximum usable resources of a node are identified by the element of the node. The maximum usable resources are important during design space exploration to estimate the simultaneous executions of multiple actors. For example a CPU may have 2 cores which limits its simultaneous execution to two actors. Examples of elements of edges include the set of nodes to which the edge can connect to or the bandwidth of this type of edge. Edges have types to account for the possibility of similar nodes being connected with edges that have different capabilities (elements). Tier1 has rules which can be applied to nodes or edges. Unlike elements, rules influence the way that the nodes and edges can be connected to one another. For example a CPU might have a PCI lane limitation rule. Associated with every rule is a parameter. So the CPU PCI lane rule might have a parameter of 4. This limits the number of PCI lanes attached to the CPU to 4. This rule and its associated parameter might limit the number of FPGAs that can connect to that CPU via PCI. The connections between nodes and edges are made through docks. Docks are effectively switches which if applied allow a node to exceed its rules for the number of connections. So In Figure 5.1 the CPU has a dock to which both GPU1 and GPU2 are connected. This implies that the input output capacity of the CPU at this point of connection is shared between the two GPUs.

CPU compute engines are nodes in their own right and require separate memory nodes to operate. GPU and FPGA nodes are assumed to be bundled with memory and do not need it as a separate node. Edges are high-level abstractions of the physical communication between two nodes. The data transfer rate between the nodes depend on the internal architecture or the edge. It is possible that the data communication rates of edges might be asymmetrical. An edge has a function, called *communication speed function* to represent its data transfer rate. Each edge has a list of nodes to which it can connect. Where there is hardware support, edges representing direct links between multiple GPUs, or direct connection between GPU and FPGA, can be modelled. An edge is by default full-duplex, however it can be identified as half-duplex.

In the next subsection, more detailed explanations of tier1 and tier2 are provided.

5.2.2 Tier 1

The tier1 contains all the available components for the platform architecture. These components are the compute nodes, the memory nodes and the communication links. Thus, tier1 is defined as 3-tuple $T_1 = (CN, CL, MN)$. The individual

elements of T_1 are defined as follows.

- *CN* is the set of available compute nodes, where each of them corresponds to a compute engine. Some of these compute engines, if not all, forms the platform architecture represented as tier2.
- *MN* is the set of memory nodes available for the system memory.
- *CL* is the set of communication links to connect the nodes with each other.

These sets are written as $CN = \{cn_1, cn_2, \ldots cn_j\}$, $CL = \{cl_1, cl_2, \ldots cl_k\}$ and $MN = \{mn_1, mn_2, \ldots mn_l\} \forall (j, k, l) \in Z^+$, where *cn*, *cl* and *mn* are the individual compute node, communication link and memory node, respectively.

Edges (Communication links)

A communication link *cl* represents the connection between two nodes. Every communication link connects two distinct nodes and has a function to calculate the rate of data transfer. The pair of nodes and the communication speed function are denoted as *ns* and *cs*, respectively. Since the communication links between a CPU and FPGA/GPU are restricted by the number of PCIe lanes, a communication link contains the number of PCIe lanes it requires. This number is denoted by *pl*. It is used to calculate the number of communication links a compute node of type CPU can support. However, for the direct communication links between the compute nodes of type FPGA/GPU, this element may not be necessary. Thus, it is an optional element. These direct communication links might need the PCIe infrastructure. For example the direct FPGA and GPU communication proposed by Bittner et. at. 2014 [26] takes place through the PCIe link. Thus, a communication link can be reliant on other communication links. This is reflected in the model through reliant links, denoted as *rl*. The presence of these reliant links is a condition for the communication link.

The nodes of the graph that represents a platform architecture have *docks*. These *docks* are the attachment points for communication links. It handles only one communication request at a time. Thus, when there are two or more communication links at a *dock*, they might receive multiple requests at the same time. This causes contention. Thus, they are used to represent contention in multiple communication links. Docks are defined in detail while describing tier2. A communication link has an element, denoted by *sh* that controls whether it can share a *dock* with other links. *sh* is Boolean, where true indicates that the communication link can share a dock and false otherwise. *sh* enables to explicitly specify the communication links that do not have contention or minimal contention.

Together with *ns*, *cs*, *pl*, *rl* and *sh*, there are two other elements, *id* and *dp* that defines a communication link. *id* is the unique identification of the communication

link. *dp* is Boolean to represent if the communication link is duplex or half-duplex. Hence, a communication link is defined as 7-tuple, $cl = (id, ns, cs, pl, rl, sh, dp), cl \in CL$. These seven elements of cl are defined as follows.

- id uniquely identifies a communication link.
- ns represents the nodes that cl connects. Therefore, ns is a pair of nodes, which is written as, $ns = (n, m), \forall (n, m) \in CN \cup MU$.
- cs is the function that returns time to transfer a certain amount of data. It is written as, $t = cs(d)$ where d is some amount of data and t is the delay in communication.
- pl represents the number of PCIe lanes a communication link requires. This attribute is optional. Thus, $pl \in Z^+ \cup \emptyset$.
- rl is a set of communication links that must be present for cl to exist in the platform architecture graph, where $rl \subseteq CL$ and $rl \neq cl$.
- sh represents if cl can share a *dock* with another communication link, where $sh \in B$. If cl can share a *dock*, then sh is true, else it is false.
- dp represents if cl is full-duplex or half-duplex, where $dp \in B$. If cl is full-duplex, then dp is true, else it is half-duplex, which is false.

Nodes (Compute engines)

A compute node cn represents a compute engine. Since there are three types of compute engines, its type can be either CPU, GPU or FPGA. Every compute node consists of a finite amount of resources. They represent the total resources that can be used for the execution of actors and channels. Its value is used for the design space exploration to estimate simultaneous execution of actors and channels. Furthermore, the compute nodes have parameters to govern the formation of a platform architecture graph. A parameter is an element whose value controls the connections between compute nodes. For example, a compute node of type CPU has an element *mpl*, which denotes the maximum number of PCIe. The rule associated with it limits the communication links, so that the *mpl* value is not exceeded. The parameters and the resources vary with the type of the compute node. Based on the type of compute node, sets of parameters and resources are defined. A compute node has five elements to denote its unique id, type, resources, parameters and its cost. Thus a compute node is defined as 5-tuple $ce = (id, ct, re, rq, pa, qe), ce \in CE$. The definition of these five elements are as follows.

- id uniquely identifies a given compute node.

- ct is the type of the compute engine, where $ct \in T$ and $T = \{fpga, gpu, cpu\}$.

- qe represents the capital cost of ce, where $qe \in R^+$.

- re is a tuple of resources, which depends on ct. Individual resource elements are defined in the following way.

$$re = \begin{cases} (core), \forall core \in Z^+, & \text{when } ct = cpu \\ (clb, bram, dsp), \forall (clb, bram, dsp) \in Z^+, & \text{when } ct = fpga \\ (nsm, shm, reg, th), \forall (nsm, shm, reg, th) \in Z^+, & \text{when } ct = gpu \end{cases}$$

  For *cpu*, *core* represents the number of cores. For *fpga*, *clb* represents the total complex logic blocks, *bram* represents the total block RAM and dsp represents the total block RAM. For *gpu*, *nsm* represents total streaming multiprocessor, *shm* represents total shared memory, *reg* represents total registers and *th* represents total threads.

  The *gpu* resource elements are based on the CUDA programming model [120]. This is because CUDA and OpenCL [121], the two widely used programming models for GPU, represents the resources differently. However, these resource elements can be replaced for another model. Nevertheless, a hybrid model cannot be used.

- rq is a function that represents the amount of time a compute engine consumes to allocate and deallocate resources; such as if r amount of resources are to be allocated to an actor of an application graph, then $rq(r)$ is the amount of time consumed for resource allocation.

- pa is a set of parameters that depend on ct. These parameters are elements with rules attached. They influence the connections of two or more cn. They are defined in the following way.

$$pa = \begin{cases} (mpl, mem, dcpu), \forall (mpl, mem) \in Z^+ \& dcpu \in B, & \text{when } ct = cpu \\ (dgpu, dfpga, both), \forall (dgpu, dfpga) \in Z^+ \& both \in B, & \text{when } ct = fpga, gpu \end{cases}$$

  mpl represents the maximum number of PCIe lanes that a CPU can support. The total number of PCIe lanes used by the communication links connected to a CPU cannot exceed the value of *mpl*. However, PCIe switch can be used to extend these lanes. This is represented in the model as *docks*. *Docks* are defined in tier2 when nodes are connected with one another to form the platform architecture graph. *mem* is the maximum system memory that the CPU can support. *dcpu* represents whether the CPU can support multi-CPU configuration. The value of *dcpu* represents the direct CPU connections that it can support. In a compute node of type CPU that does not allow multiple CPU connections, the *dcpu* value is set as zero. If two CPU models support multi-CPU configuration but their models are incompatible, then they are

prevented from getting connected by the lack of communication links between these CPU models.

When the compute engines are for type *fpga* or *gpu*, the number of direct connections that are possible amongst each other is represented by *dgpu* and *dfpga*. They are total number of direct connections possible with each type of compute nodes. If they do not support direct connections, these values are zero. Since direct connections between FPGA/GPU is a physical reality, this model envisages an FPGA or GPU is directly connected with both FPGA and GPU. If a compute node supports such connection, it is reflected through a Boolean attribute called *both*.

Even though the parameters of a compute node may support certain connections, the presence of respective communication links makes it possible to create a graph with such connections. Thus, the designer may impose further restrictions in creating platform architectures through the communication links.

Nodes (Memory units)

A memory node *mn* represents the available DRAM units in tier1. Every memory node represents a physical memory card in tier1. It is defined as a 3-tuple $mn = (id, sz, qs), mn \in MN$. The definition of the attributes of mn are as follows.

- *id* uniquely identifies the memory unit
- *sz* represents the size of mn in units of data.
- *qs* is the capital cost of the memory unit, where $qs \in R^+$.

5.2.3 Tier 2

Tier2 represents the platform architecture as a graph. It is composed of tier1 components. To represent a graph, *docks* are added to the compute nodes that forms the platform architecture graph. *Docks* are the attachment points of a communication link. A *dock* can have multiple communication links attached to it but handle them one at a time. Thus, connections with possibilities of contentions are represented as multiple communication links attached to the same dock. Memory nodes do not have docks of their own. They are combined together to form a system memory node, which has a dock for connection with a CPU.

After docks are introduced within the compute nodes and communication links of a platform architecture graph, they are redefined as *connected compute nodes* and

connected communication links, respectively. The tier2, which is a graph representing platform architecture is defined as 3-tuple, $T_2 = (CCN, CCL, SMN)$. CCN is the set of connected compute nodes that forms the platform architecture graph. CCL is the set of connected communication links that represents all the edges of the platform architecture graph. SMN is the set of system memory nodes. These sets are written as $CCN = \{ccn_1, ccn_2, \ldots ccn_m\}$, $CCL = \{ccl_1, ccl_2, \ldots ccl_n\}$ and $SMN = \{smn_1, smn_2, \ldots smn_o\}$ $\forall(m, n, o) \in Z^+$, where *ccn*, *ccl* and *smn* are the individual connected compute node, connected communication link and system memory node, respectively. These individual elements of the three sets are defined as follows.

- A connected compute node ccn_i extends a compute node cn_i with a set of *docks* D_i. Thus, ccn_i is defined as, $ccn_i = (cn_i, D_i)$,where $ccn_i \in CCN, cn_i \in CN$and $D_i \cap D_j = \emptyset, ccn_i \neq ccn_j$.

- A system memory node smn_i is made up of a set of memory nodes and a *dock*. It is defined as 2-tuple, $smn_i = (M_i, d_i)$, where $smn_i \in SMN, M_i \subseteq MN$ and d_i is the *dock*. The memory nodes in M_i are unique to smn_i. Thus, $M_i \cap M_j = \emptyset$,where $smn_i \neq smn_j$. The size of smn_i is calculated by adding the value of sz of all the individual memory nodes in M_i.

- A connected communication link ccl_i extends a communication link cl_i with two *docks* d_i^1 and d_i^2. Thus, ccl_i is defined as, $ccl_i = (cl_i, d_i^1, d_i^2)$, where $ccl_i \in CCL, cl_i \in CL$ and $d_i^1 \neq d_i^2$.

This definition of tier2 determines the platform architecture. In the next section, an example of PPG modelling is presented, which illustrates: (1) different architectures that can be expressed through parameterised platform graph and (2) how this model facilitates design space exploration.

5.2.4 Example of a parameterised platform graph (PPG)

In this section, the parameterised platform graph is demonstrated with an example. Initially, it is shown how the components in tier1 can be arranged to form distinct platform architecture graphs. This shows the effect of parameters in shaping the platform architecture graph. Afterwards, the selection of a platform architecture for a sample application-algorithm is demonstrated. This sample application-algorithm is a simple SDF graph. The design space that exists in tier1 for the SDF model is explored manually to find the platform architecture that provides best performance at the least capital cost. In this example the measure of better performance is a lower make-span. During this manual design space exploration, the architectural decisions necessary to deploy an SDF representation on a PPG tier2 instance are also identified. Thus, there are two parts in this subsection. The first part describes tier1 of an example PPG and the possible tier2. The second

Table 5.1: This table lists all the attribute values for the compute nodes of type CPU.

CPU		
Attribute	**Description**	**Value**
id	to uniquely identify	CPU1
cost	relative cost	1
mpl	maximum PCIe lanes	24
dcpu	number of direct CPU connections	0

Table 5.2: This table lists all the attribute values for the compute nodes of type GPU and FPGA.

GPU/FPGA				
Attribute	**Description**	**Value**		
id	to uniquely identify	GPU1	GPU2	FPGA1
type	whether GPU or FPGA	GPU	GPU	FPGA
cost	relative cost	1.5	1.5	3
dgpu	direct connection with another GPU	0	0	1
dfpga	direct connection with another FPGA	1	1	0
both	can have direct connection to both FPGA and GPU	false	false	false

part demonstrates a manual design space exploration for a sample SDF model.

PPG modelling example

In this example, tier1 consists of 4 compute nodes, one CPU, two GPUs of the same kind and one FPGA. There is only one memory node, so that the system memory in every platform architecture will have identical size. This reduces the number of platform architectures that only differ in system memory sizes.

Based on the definition of compute nodes, *pa* contains the list of parameters. These parameters influence the connections between the compute nodes. In Tables 5.1 and 5.2, the values of these parameters are listed. These values are derived from the specifications provided by their respective vendors and the direct FPGA–GPU connection proposed by Bittner. et.al. 2014 [26]. Since this direct connection is tested with one FPGA–GPU pair, the parameters that determine the maximum number of direct connections between them are set as one. For this reason, *dfpga* for the GPUs, *dgpu* for the FPGA are set as 1 and *parameter both* is set as False. In this example it is assumed that the resource allocation and deallocation delay is negligible, thus *rq* of the compute engines are not taken into account.

To connect the compute nodes, there are 6 communication links, referred as *L0*, *L1*, *L2*, *L3* and *L5*. *L0* represents the link between the CPU and the system

Table 5.3: This table lists the communication links with their attributes and their values.

id	compute nodes	pcie lanes	dependencies	sharing
L0	(CPU1, DRAM)	0	None	False
L1	(CPU1, GPU1)	16	None	True
L2	(CPU1, GPU2)	16	None	True
L3	(CPU1, FPGA1)	8	None	True
L4	(GPU1, FPGA1)	0	L1, L3	True
L5	(GPU2, FPGA1)	0	L2, L3	True

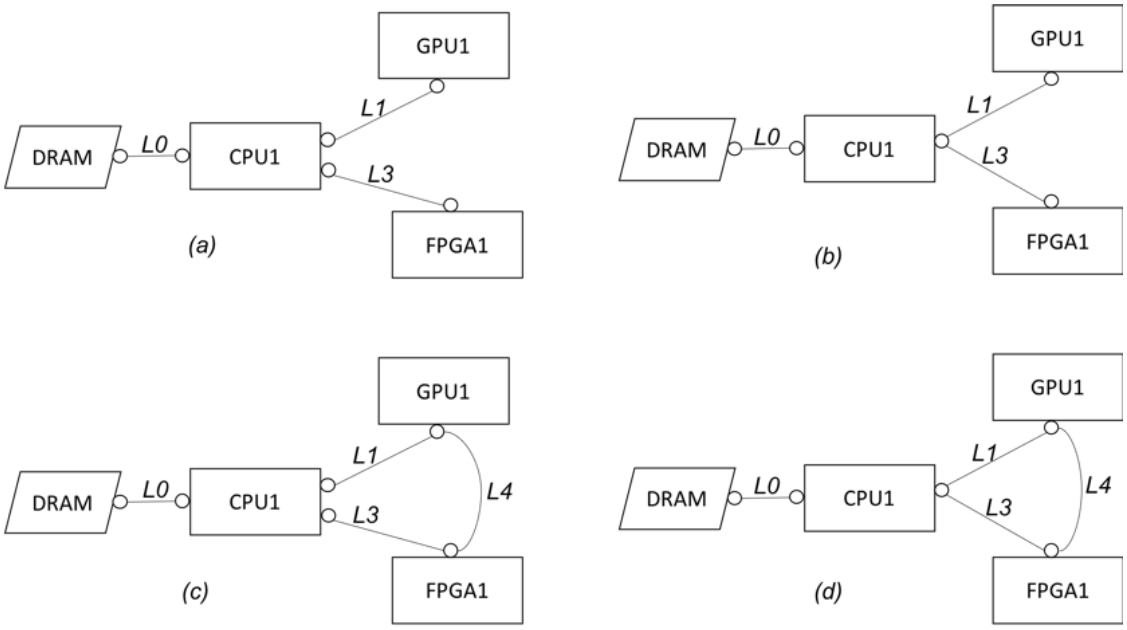


Figure 5.2: Platform architectures (PPG tier2) examples based on the constraints (PPG tier1) defined in Tables 5.1 and 5.2. These PPG tier2 consists of *CPU1*, *GPU1* and *FPGA1*.

memory. *L1* models a x16 PCIe connection between the CPU and GPU1. L2 is a similar connection but between the CPU and GPU2. *L3* is a x8 PCIe connection between the CPU and the FPGA. *L4* and *L5* are represents direct connections between the FPGA and a GPU. There are two communication links for two GPUs. These links are reliant on the PCIe infrastructure. Thus, their reliant links point to *L1*, *L2* and *L3*. The sharing condition *sh* is set true for all the links, as these links can potentially share through a PCI express switch. These details of the communication links that are involved with the formation of a platform graph are listed in Table 5.3.

Based on tier1 of this example, there are several platform architectures possible. Many of them are created by varying the connection topologies of the compute engines. Platform architectures with the same set of compute nodes and system memory have equal cost. Therefore, it becomes necessary to rule out the archi-

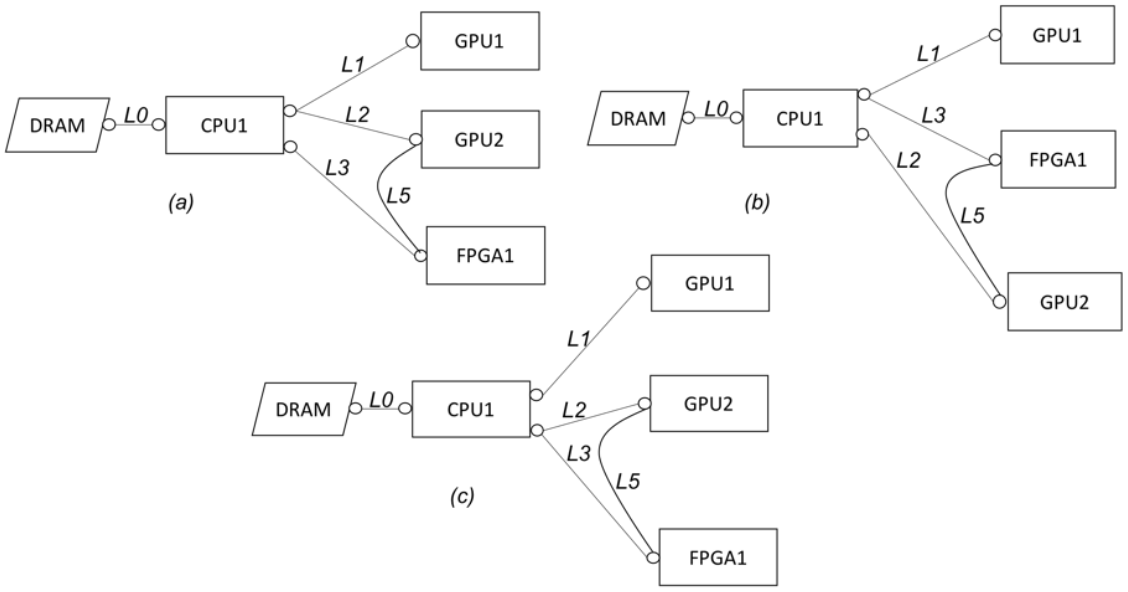


Figure 5.3: PPG tier2 models that consists of all the compute engines present in tier1, which is defined in Tables 5.1 and 5.2.

tectures with performance bottlenecks that have equivalent cost. Suppose the architecture instances in Figure 5.2, where there are four platform architectures from the same set of compute engines. These differences in topologies are created by the attaching *L1* and *L3* on the same dock and alternating the presence of *L4*, which is the direct link between the FPGA and the GPU. Sharing the same *dock*, models contention that can appear from the use of a PCIe switch, which can potentially impede performance. Furthermore, the use of a direct link *L4* can possibly provide a performance boost. Thus, using simple rules, instead of simulating the application-algorithm on every possible platform architecture, the ones that are capable of providing better performances can be identified.

Three platform architectures with all the compute nodes available in tier1 is shown in Figure 5.3. The primary differences between these architectures are the way a CPU dock is shared to connect the other compute nodes and the direct link between FPGA and GPU. Since the CPU has 24 PCIe lanes, attaching two GPUs and one FPGA exceeds them. Thus, a switch is necessary, which is modelled through a shared dock. It is also interesting to note that since *dfpga* and *dgpu* parameters are set to 1, the direct link cannot connect two GPUs with the FPGA. The performance of an application-algorithm on either of these platform architectures dependent on the architectural decisions. This is examined in the next part, where an SDF model is used as an example to conduct a manual design space exploration.

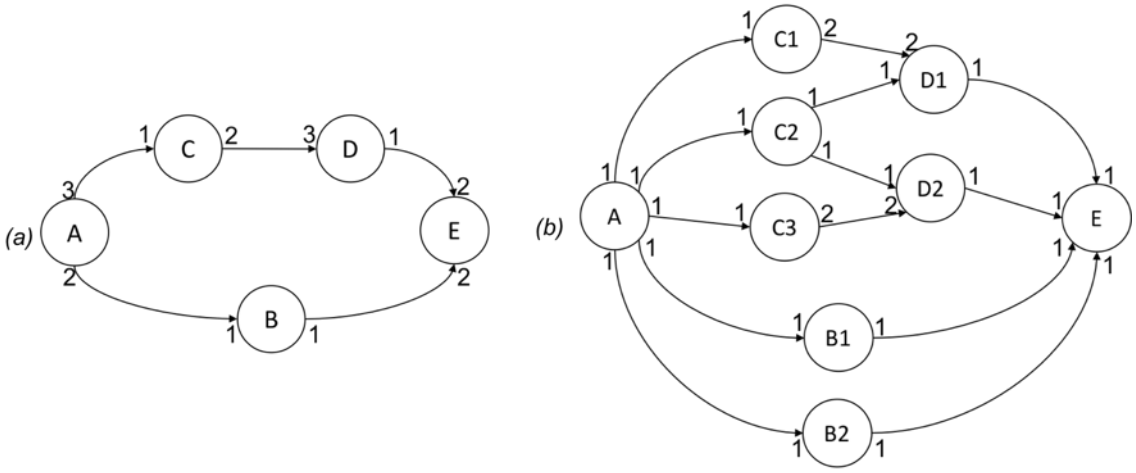


Figure 5.4: (a) A simple SDF graph, (b) with its equivalent homogeneous SDF graph.

Manual design space exploration with PPG

A manual design space exploration that does not use any particular intermediate data structure or any form of automation is demonstrated here. The purpose of this demonstration is to describe the design space exploration capabilities of PPG. This example does not claim to find any optimal design; rather it is used to explore some of the issues that will be further explored in chapter 6 which describes the design space exploration algorithm in detail.

The example PPG presented in the previous part is used to manually explore the design space for the simple SDF model in Figure 5.4(a). This exploration finds a platform architecture and the corresponding architectural decisions that results in the lowest makespan possible within the constraints of tier1. The exploration also tries to find a platform architecture that has lower capital cost. The SDF model consists of five actors and they have both task and data parallelisms. Task parallelism is evident between actors B and C, D. Whereas, data parallelism that exists in B, C and D. This becomes explicit after expanding the model to a homogeneous dataflow model, which is shown in Figure 5.4(b).

In order to conduct design space exploration, the execution timings and the resource usage of the actors on the available compute nodes are required. The execution timings are used to estimate the makespan. The resource consumption data determine the possibilities of concurrent actor execution on a compute engine. These are listed in table 5.4. Some execution times are infinite, which indicates that implementation of that actor is not available for a particular compute engine. The resource consumption information of the actors is tallied with the maximum resources in the compute engines to estimate if multiple actors can be placed together. In this example, the resource usage is taken as percentage of resources consumed by an actor on a compute engine.

Actor	CPU exe	GPU exe	FPGA exe	GPU resource %	FPGA resource %
A	10	20	20	40%	20%
B	20	10	20	90%	60%
C	∞	20	10	50%	20%
D	∞	20	10	40%	20%
E	10	20	20	50%	20%

Table 5.4: This table lists the execution time and resource usage percentage of actors on each compute engines. The actors are from the SDF model in Figure 5.4(a) and the compute engines are from the PPG tier1 in Tables 5.1 and 5.2. If a pre-engineered component of an actor on a compute engine is lacking, then the exec time is considered as infinite.

The execution times of the actors on the compute engines show that the best performance of A and E are on the CPU, B on one of the GPUs and C and D are on the FPGA. In order to achieve the data parallelism, they need to have separate instances in parallel. Based on the resource consumption data, simultaneous execution of two instances of B are achievable on the FPGA but not on one GPU. The second GPU can be brought in for this. However, this cannot reduce the make-span further, rather it will increase the cost of the platform architecture. The reason is A, C, D, E is the critical path. It consumes same time as the two instances of B executing sequentially on the GPU. The makespan can be further reduced if the critical path can be reduced. In this case, the critical path can be reduced by lowering the execution time of the actors. However, tier1 does not contain other compute engines to explore this option. Therefore, the platform architecture for lowest make-span consists of CPU1, GPU1 and FPGA1. This architecture was shown in Figure 5.2(a).

The actors placed on FPGA and GPU are controlled from the CPU. There is a delay due to this control from CPU. This delay consists of the data transfer time and the time required to activate the actors from the CPU. In this example, the delay is assumed to be uniform and denoted by t. The control of actors is sequential on the FPGA and GPU, as there is one communication link connecting them with the CPU. But, if the control of GPU and FPGA are conducted through separate threads, the make-span can be further reduced. This is illustrated using a Gantt chart in Figure 5.5. This improvement depends on the value of t. If t is very small, then the overhead of multiple threads can dwarf this decision of parallelly controlling the actors. Furthermore, for the actors C and D, it is assumed they communicate directly, rather than through the CPU. This communication time within the actors on the same compute engine is assumed to be negligible.

The DSE makes decisions that involve the selection of the actors for data parallelism, layout of actors on the compute engines and selection of the pre-engineered actor and channel. In this example, the selection of data parallel actors is the choice to sequentially execute B in a loop, while expanding C and D for parallel

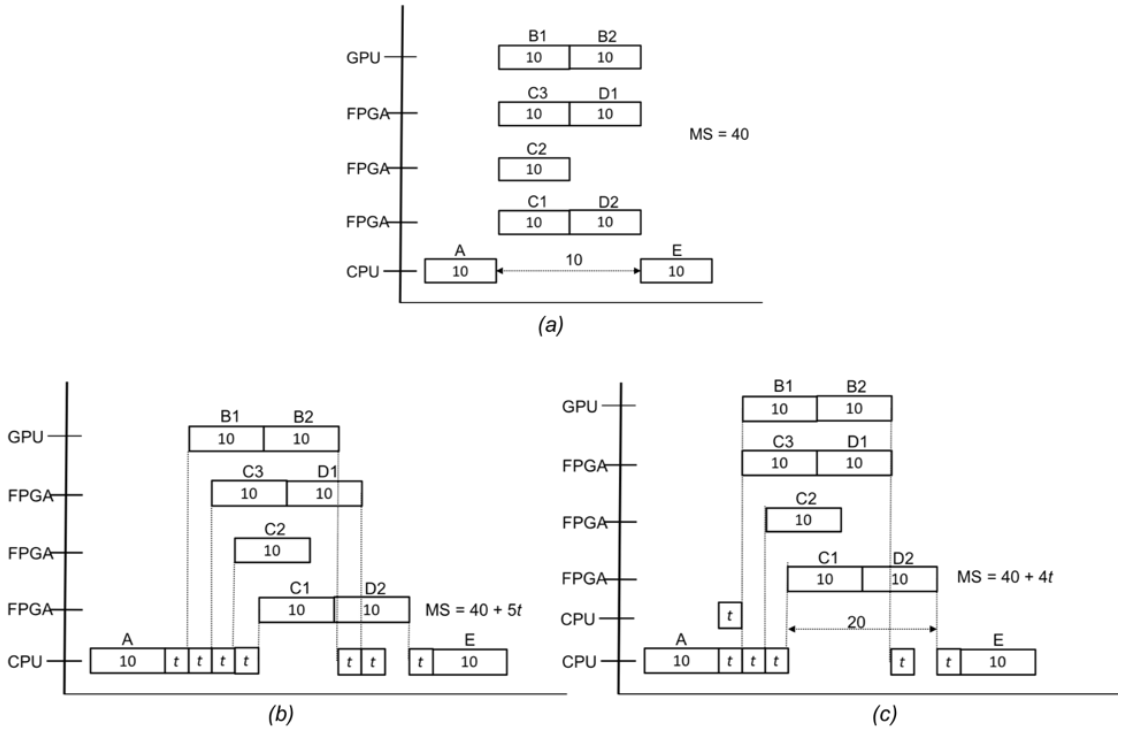


Figure 5.5: The layout of actors on CPU, GPU and FPGA is shown in (a) without the actor control delays. The actors placed on FPGA/GPU are controlled sequentially from the CPU $makespan = 40 + 5t$ is shown in (b), whereas in (c) they are controlled in parallel $makespan = 40 + 4t$. The value of t is the delay due to control of actors from CPU. In both (b) and (c) it is assumed that on the same compute engine, the actors have negligible communication delays, which is reflected in actors C and D. Note that the other two t are in sequence, as they are on the same compute engine, which is the FPGA.

deployment. Layout of actors is the decision to execute three instances of C simultaneously, followed by two instances of D. Thus, an instance of C and D are deployed sequentially. Resources can be optimised in this situation, as after C completes D starts.

The architectural decisions identified during this manual design space exploration can be summarised as follows:

1. choices on expanding a certain portion of a graph for data parallelism,
2. selection of a compute engine to place an actor,
3. implementation type of an actor and its channels,
4. choice of actor control from another compute engine and
5. choices on concurrent actor execution on a compute engine.

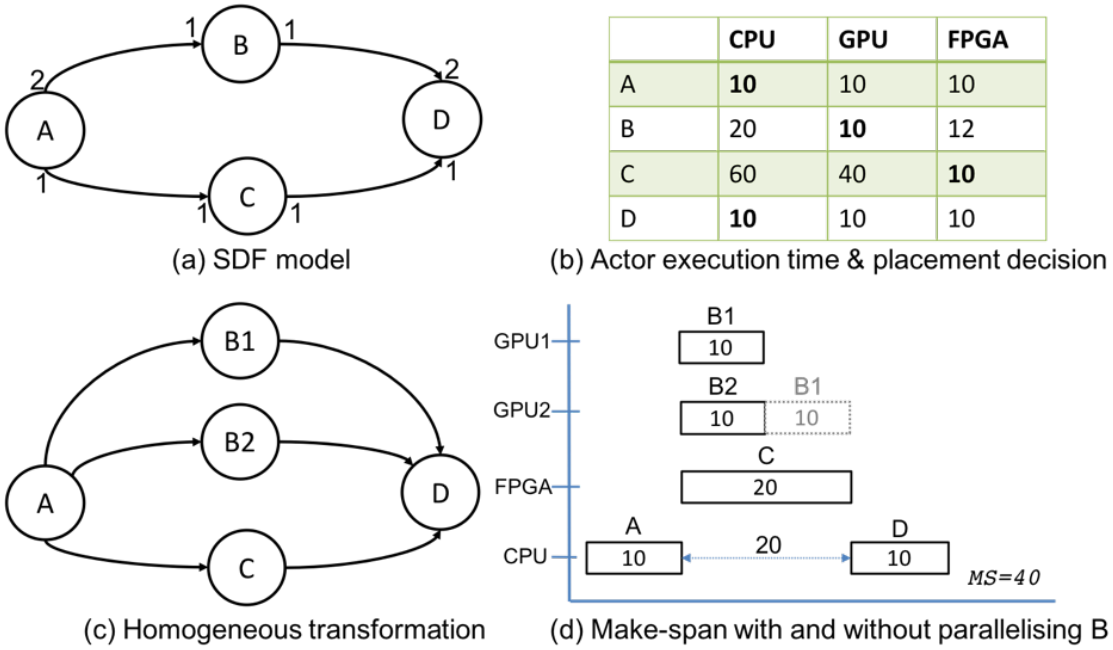
5.2.5 Defining architectural decisions

The architectural decisions that were identified in the previous subsection, while conducting a manual design space exploration, are detailed with informal examples in this subsection. The architectural decisions are defined in three parts. The first part presents the architectural decision to selecting data parallel actors for expansion. The second part presents three architectural decisions. They are the choice of compute engines for the placement of actors, specifying their implementation type and the decisions on how an actor placed on GPU/FPGA is controlled from CPU. The third part describes the choices on concurrent actor execution on a compute engine.

Actor selection for expansion

Task parallelism in an SDF model is explicitly represented, whereas data parallelism is exposed by expanding the model into its homogeneous counterpart. This example shows that there needs to be sufficient resources for the utilisation of all the parallelisms. However, utilising every data parallelism cannot always improve performance. Suppose the example in Figure 5.6, where an SDF graph with a data parallel actor B is expanded for parallel deployment. But, this does not lead to a better performance, due to the greater execution time of C. The performance is the same as that of executing the instances of B sequentially.

This decision leads to the selection of only certain data parallel actors for expansion. During deployment, the selected actors are deployed with parallelism. While other



	CPU	GPU	FPGA
A	**10**	10	10
B	20	**10**	12
C	60	40	**10**
D	**10**	10	10

Figure 5.6: Utilisation of data parallelism not yielding better performance: (a) the SDF model, (b) the actor execution timings and they are placed on the compute engine with the least execution time, (c) homogeneous transformation reveals the data-parallelism of B and (d) the make-span is same as that of executing both the instances of B in a loop.

actors, even with parallelism, are sequentially implemented. Hence, this decision is stated as follows.

Architectural decision–1: *Selection of certain data-parallel actors to achieve parallelism during deployment.*

Placement, implementation type and actor control

The placement decision is the selection of a compute engine to place an actor for deployment. Subsequently, this entails the decision of choosing a communication link or a compute engine to place its channels. The placement of a channel on a communication link takes place only for communications between two compute engines. This is illustrated in Figure 5.7. In other situations, a channel is placed on the compute engine. This compute engine is the one where the actors connected by the channel is placed. Thus, this placement decision is defined as follows.

Architectural decision–2: *Selection of a compute engine for the placement of an actor and choosing communication links for its channels.*

It is the performances of the actors on various compute engines combined with the delay in the communication links that drive the mapping decisions. An example of this decision is described in section 5.2.4. The outcome of this decision is passed

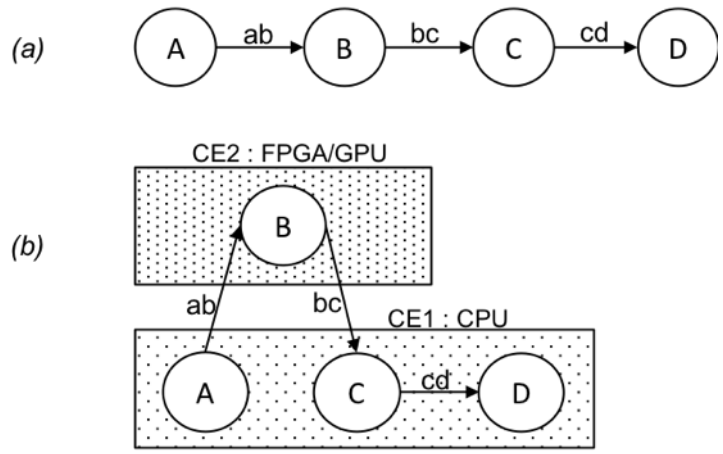


Figure 5.7: Actor and channel mapping: (a) a simple SDF model, (b) actor B is mapped on a compute engine ($CE2$) that is either GPU or FPGA, while the actors A, C and D are placed on the CPU ($CE1$). The mapping of B on $CE2$ requires to place channels ab and bc on the communication link that connects $CE1$ and $CE2$.

to the deployer, which then uses it to implement the actors and channels on the respective compute engines and communication links.

Based on the mapping decisions, the type of pre-engineered actors and channels are selected from the repository. There can be distinct implementations of an actor for the same type of compute engine. Distinct implementations exist to optimise an actor on the basis of its token size or on the model of a compute engine. The actor and channel implementation types are used to create instances of actors and channels for the deployment. This decision of choosing the type of implementation is stated as follows.

Architectural decision–3: *Choosing the implementation type of an actor and channel for deployment*

When an actor is placed on FPGA or GPU, its communications are generally controlled from the CPU. This control is the initiation of data transfers from the CPU. Every actor placed on FPGA/GPU can be implemented by controlling both its input and output from the CPU. This is an unique characteristic of agile heterogeneous computing that the actors on a compute engine are controlled from an external compute engine. An explicit representation of this control is shown in Figure 5.8. The dotted actors B_i and B_o represent the code that lies on the CPU to control the actor's communication. These actors are called *control actors*. They are defined later along with the definition of ArcSDF. The placement decision of control actors can impact performance.

Due to this control from another compute engine, there is an extra communication delay. Some of these extra delays can be avoided by enabling actors on the same FPGA/GPU to directly communicate. This is possible for multiple actors on the

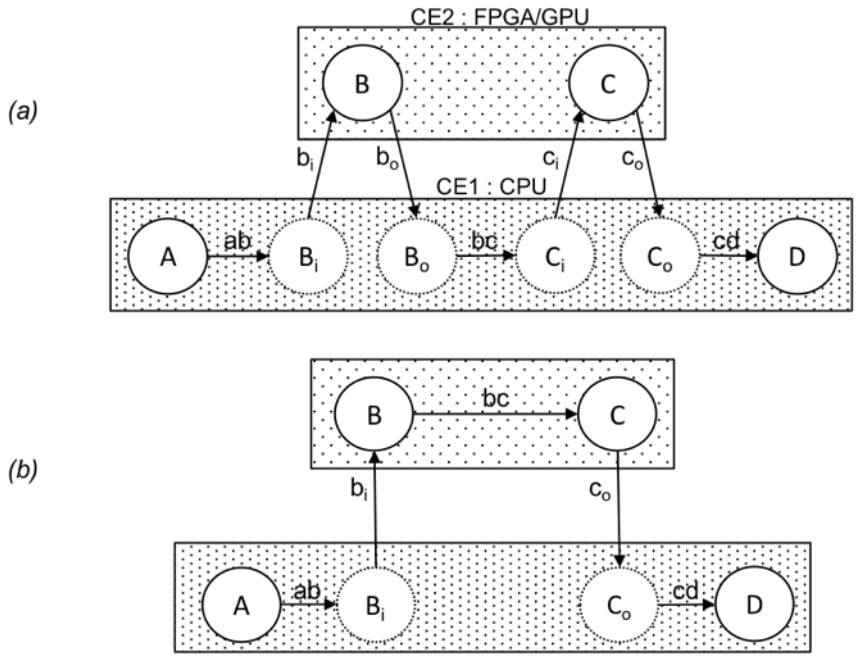


Figure 5.8: Two actors B and C of the SDF model from Figure 5.7 are mapped on the same CE. (a) Explicit representation of their communications controlled from the CPU and the extra delays due to it, (b) whereas a direct communication between actors B and C saves those extra delays. The dotted actors represent the code that is placed on the CPU to control the passage of data.

same compute engine that have channels amongst themselves, as shown in Figure 5.8(b). The decision of direct communication between actors on FPGA/GPU can be a result of a rule that implements it whenever possible. However, implementation of direct communications within a compute engine consumes resources. This implies such a rule might use resources that can be used elsewhere to boost performance. Thus, the choice of a direct communication is a part of the design space exploration. This architectural decision is stated as follows.

Architectural decision–4: *The choice of an actor being controlled from another compute engine.*

Concurrent actor execution on a compute engine

Multiple actors placed on the same compute engine can be deployed with distinct layouts. A layout is the arrangements of concurrent and sequential execution of actors. It determines resource usage and performance. Concurrent execution of all the data independent actors result in the best possible performance on the compute engine. However, this naive layout may not be deployable, as it might exceed the resources available on the compute engine. Moreover, a layout that combines sequential and concurrent executions may lead to the same performance

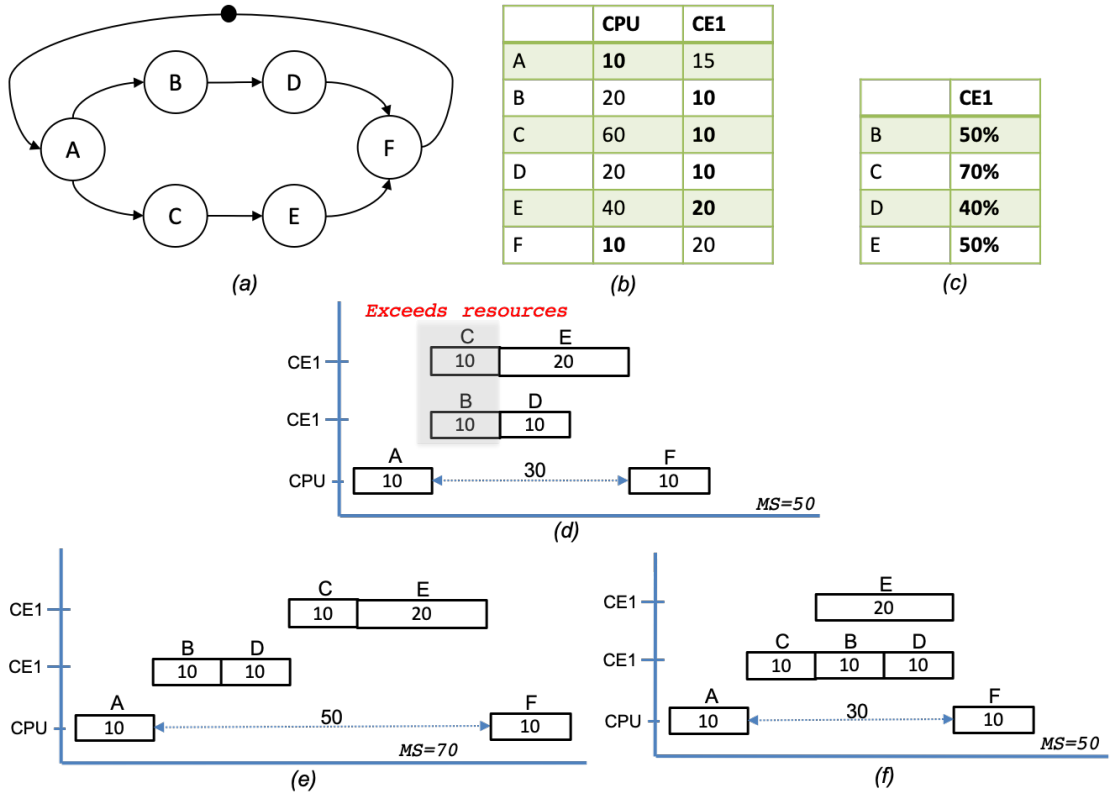


Figure 5.9: Actor layouts affecting performance and resource usage; (a) simple homogeneous SDF, (b) Performance data on CPU and another compute engine *CE1* that can be FPGA or GPU, (c) resource consumption of the actors mapped on *CE1*, (d) a naive layout that exceeds resources, (e) and (f) two distinct layouts within resource limits with varying performances.

with less resource usage. This is possible because not all data independent actors contribute towards performance improvement. Suppose the example in Figure 5.9, where actors *B*, *C*, *D* and *E* are placed on *CE1*. Amongst these actors, *B* and *C* have task parallelism. A naive layout with parallel execution of *B* and *C* is shown in Figure 5.9(d). This layout exceeds the resources of *CE1*. Therefore, these two actors need to execute sequentially on *CE1*. The choice of whether to execute *B* first or *C* creates variation in the make-span. A better decision is to execute *C* before *B*, so *E* which, consumes more time, can run simultaneously with *B* and *D*. A desirable outcome of this decision is to ensure a layout that is deployable with lowest make-span. Thus, this architectural decision is stated as follows.

Architectural decision–5: *Layout of actors governing their sequential and concurrent executions on a compute engine.*

This decision of actor layout directs the way resources of a compute engine are reused and shared amongst the actors. After an actor completes its execution, the resources are released for others. Either they are consumed fully by the next actor, or they are shared between the following concurrent actors. This is shown in the

example of Figure 5.9(f), where the resources of C are shared between B and E.

5.2.6 Conclusion

In this section a high-level representation for agile heterogeneous platforms called *Parameterised Platform Graph* (PPG) is presented. PPG consists of two tiers. The available components to create the platform architecture and their constraints are represented in tier1. A platform architecture instance is expressed in tier2. A sample design space exploration example is used to demonstrate how the model is likely to interact with the design space exploration algorithm. In this example, architectural decisions to deploy a SDF graph on a PPG tier2 instance are also identified. These architectural decisions will be used in the next section for the creation of the intermediate data structure (ArcSDF) required by the new design flow.

5.3 ArcSDF: Architecture augmented synchronous dataflow

A dataflow model capable of expressing architectural decisions together with the application-algorithm is envisaged to be the intermediate data structure of AhcFlow, so that it can be used for design space exploration and deployment. In this section, such a data structure, as a data flow representation called *architecture augmented synchronous dataflow* (ArcSDF) is presented. This representation uses four new constructs (*compute zone*, *interface*, *resource edge* and *control actor*) along with the basic actor and channel of SDF [57]. Amongst these four new constructs, compute zone is the primary one. It represents mapping, scheduling and concurrent actor layout. Whereas, the other constructs assist to express the operations concerning compute zones.

A compute zone represents a certain amount of resources of a compute engine that can only be used by the actors and channels mapped on it. Compute zones are sequential; only one actor on it can execute at a time. Actors on disparate compute zones can however execute concurrently. In order to make room for more actors, a compute zone can become inactive and release resource to a new compute zone. However, release of resources can incur a performance penalty. The dependencies between compute zones for resources are represented through resource edges. These features of compute zones allow the expression of an actor's resource consumption on the compute engine it is mapped and they also express the layout of actors in case of concurrent executions.

Due to the augmentation of architectural decisions with SDF, it is possible to analyse the following in addition to the traditional throughput calculation; (1) maximum resource usage of an ArcSDF representation, (2) optimisation to minimise the number of compute zones, and (3) searching the earliest time to map an actor on a platform architecture; some of these analyses are used in the design space exploration algorithm in chapter 6. This section is thus organised into three subsections. In the first subsection, an informal overview of ArcSDF is presented, where the representation is described with two examples. Then, in the second subsection, ArcSDF is defined comprehensively. Finally, in the third subsection, four different analyses with ArcSDF are described.

5.3.1 Overview

In this section, an overview of ArcSDF is presented using two examples. The first example is illustrated in Figure 5.10 which shows a complete ArcSDF representation for the application-algorithm on the platform architecture shown in Figure 5.11. This example was previously introduced in section 5.2.4. The second example shown in Figure 5.12 is the full ArcSDF graph for the application-algorithm

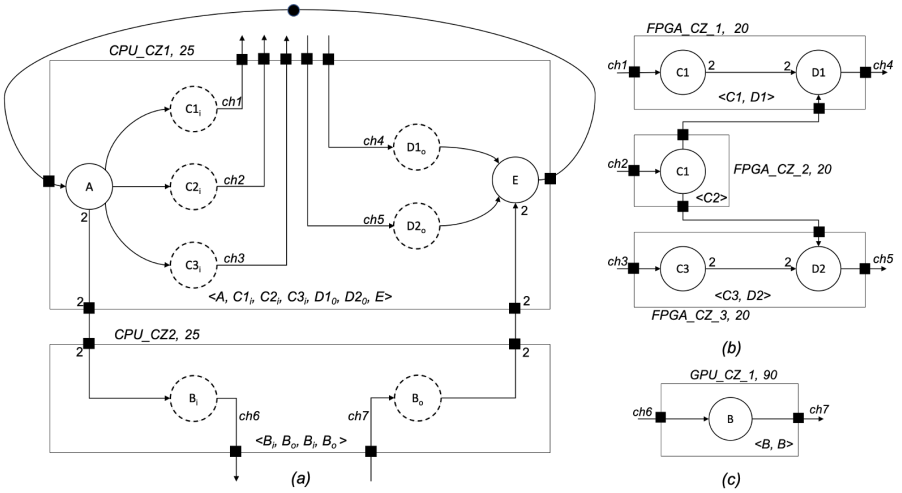


Figure 5.10: This figure shows the ArcSDF model representing the architectural decisions and the application-algorithm from the manual design space exploration example in section 5.2.4.

originally described in section 5.2.5 in Figure 5.9(a) with the actor concurrency decisions of Figure 5.9(f).

The main features of the example in Figure 5.10 will now be explained. The nodes of an ArcSDF graph correspond to the nodes (actors) of the application SDF. However, as can be observed, some of the nodes have been duplicated in the ArcSDF. This arises because nodes on GPU and FPGA require corresponding control actors on the CPU. These control actors are shown with dotted outlines. For GPU and FPGA there are separate control actors for input and output. Nodes are grouped together using boxes which represent compute zones. A compute zone is a part of a compute engine and its associated resources which can execute a set of actors sequentially. Separate compute zones can execute actors concurrently. Every compute zone is mapped to a compute engine chosen from those available in the PPG tier1. The diagram shows percentage resource consumption of the compute engine for each compute zone. In each compute zone, the sequential schedule of actor executions is shown in between angle brackets <>. Note that the schedule for the GPU is $<B,B>$ which indicates that actor *B1* will be executed followed by actor *B2*. This also prevents multiple self execution.

Edges can enter and exit a compute zone through interfaces (small black rectangles on the diagram). Interfaces have numbers called rates associated with them and these rates have the same meaning as actor rates on an SDF graph. Rates of 1 are omitted from the diagram. Edges in ArcSDF are either channels or resource edges. Channels are an extension of SDF channels, whereas resource edges expresses resource dependencies. Resource edges are described in the second example. Channels can be further classified as either internal or external to a compute zone. External channels connect interfaces. Internal channels are always inside of

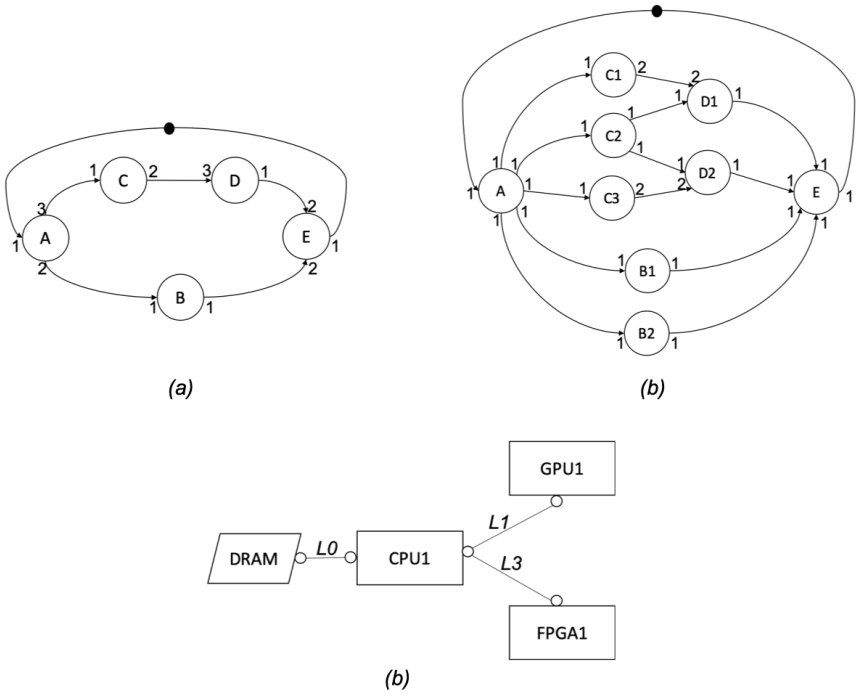


Figure 5.11: The application-algorithm and the platform architecture for the example in Figure 5.10. These are reproduced from section 5.2.4.

compute zones. All edges have unique labels inside the ArcSDF representation. However, on the diagram only external channel labels are shown to indicate connectivity. External channels that are external to a compute engine are mapped to communication links but internal channels do not require communication links because they are assumed to use shared memory implementations.

A simplified version of resource usage is used here, where an overall percentage of the resources are used. This can be easily replaced with the different types of resources of a compute engine.

The second ArcSDF example in Figure 5.12 illustrates resource edges that represents the resource dependencies between the compute zones as bold black arrows. Resource edges *RE1* and *RE2* between compute zones CE1_CZ1 and CE1_CZ2 expresses the decision to executing only actor C first. Then allowing the other actors to commence execution concurrently. In section 5.2.5 it is explained why this order of concurrency leads to better performance for this example.

For better readability, the ArcSDF graphs can be simplified by not showing the control actors and the interfaces. This will make the graph appear closer to the original application SDF and the make most of the architectural decisions more readable. Figure 5.13 shows the simplified diagram of the actual ArcSDF in Figure 5.12. Although this simplified diagram only lacks the control actor and interface related decisions, the majority of the decisions regrading mapping, scheduling and

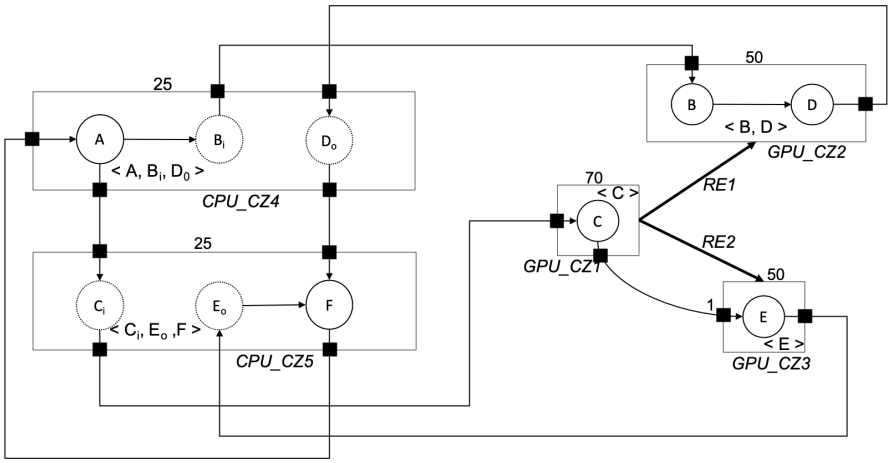


Figure 5.12: This figure shows the ArcSDF graph for the example previously shown in Figure 5.9. The actor concurrency decisions of Figure 5.9(f) are represented in this ArcSDF graph.

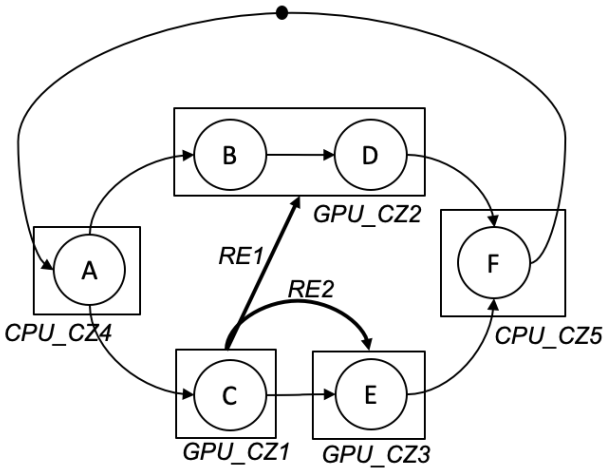


Figure 5.13: A simplified illustration of the complete ArcSDF graph shown in Figure 5.12. This simplified diagram only shows the mapping, scheduling and the actor concurrency decisions without the actor control decisions and the compute zone interfaces.

actor concurrency are visible.

5.3.2 Comprehensive definition of ArcSDF

In this section, a comprehensive definition of the ArcSDF graph is provided.

ArcSDF is created by extending the SDF graph to represent the architectural decisions. In this graph, actor firings are restricted by the availability of resources. The notion of resource is incorporated within the graph by enclosing actors and channels inside resource regions known as compute zones. A compute zone is imagined to contain a certain amount of the resources of a compute engine. This amount depend on the actors and channels enclosed in a compute zone. They are limited, such that only one actor can execute at a time. Thus, multiple actors in a compute zone are executed in a sequence. This sequence is known as the schedule of a compute zone. After the completion of its schedule, a compute zone releases its resources for others. The resource dependencies amongst compute zones are represented as edges, known as resource edges.

Resource edges together with compute zones incorporate the architectural decision of actor layout. The architectural decision of actor placement is implicitly expressed through the compute zones. This is because a compute zone represents resources of a particular compute engine. Thus, the actors and channels within it represents the placement decision. The architectural decisions of implementation type is represented through additional attributes of actors and channels. Control of actors are incorporated in the model through proxy actors known as control actors. Furthermore, the architectural decision on selection of data parallelism is represented through an implementation rule to restrict actor instances.

The example shown in Figure 5.14, is a partial ArcSDF model representing the architectural decision of actor layout on CE1 from Figure 5.12. The compute zones are depicted as rectangular boxes. The number above them represents the resources contained by the compute zones. This number is the percentage of resources consumed of CE1. The schedule is shown within the angle brackets. The resource edges amongst the compute zones are shown as bold arrows. The resource edges RE1 and RE2 represents the release of resource from CZ1 to its succeeding compute zones CZ2 and CZ3. Since compute zones represent a resource region, there are designated areas for the passage of data, known as interfaces. The solid rectangular boxes symbolise these interfaces. The number beside an interface represents the number of tokens released or consumed every time they fire.

Since ArcSDF is a dataflow representation, it is defined as a graph. An ArcSDF graph G is augmented with the architectural decisions to deploy an SDF model G_{sdf} on a platform architecture T_2. G is defined as 5-tuple (CZ, IF, AC, CH, RE), where CZ denotes a finite set of compute zones representing a set of sequential actors along with the resources used, IF is a finite set of interfaces for the passage of data through a compute zone, AC is the finite set of actors that is equivalent to that of G_{sdf}, CH is a finite set of channels expressing the data dependencies amongst the actors and RE is a finite set of resource edges expressing the resource

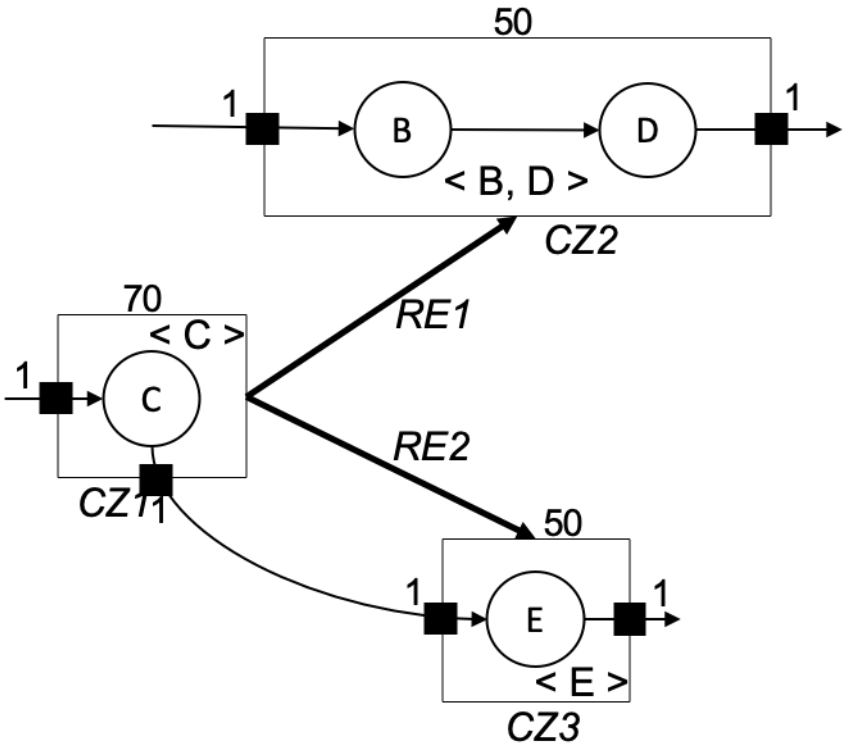


Figure 5.14: A partial ArcSDF model representing the actor layout on the compute engine CE1 of figure 5.7(f). The rectangular boxes denote the compute zones, where the resource amount is shown as the number and the schedules inside the angle brackets. This number is the percentage of resources a compute zone consumes of CE1. The bold arrows connecting the compute zones are the resource edges. They represent resource dependencies amongst the compute zones. The solid circles are the interfaces of a compute zone, which are used to pass data between compute zones. The number beside an interface is its rate.

dependencies between the compute zones. The details of compute zones, interface, actors, channels and resource edges are presented in six separate parts. At first they are described intuitively, then followed by a formal definition.

Compute zone

A compute zone represents a set of actors on a compute engine and the resources consumed by them. These resources are bounded by a compute zone, in a manner that they are only available for its actors and their input channels. The resources are just sufficient for the execution of the actors in a sequence. The sequence in which the actors execute is known as the schedule of the compute zone. When data parallelism is not expanded, a schedule incorporates the number of repetitions of its actors. A schedule of compute zones with data parallel actors is shown in Figure 5.15(b). Once all the actors on a compute zone are executed to fulfil its schedule, the compute zone is said to be completed. This completion indicates the release of its resources for succeeding compute zones, which are represented by the resource edges. A succeeding compute zone is created from the resources released by its preceding compute zones. Since the resources are released, the actors and the channels in the preceding compute zones are no longer available. The compute zones become inactive. Only the data released by the preceding actors are available. However, a compute zone without a resource edge will always occupy their resources. This enables compute zones with actors that have states to retain them.

Since ArcSDF graph is augmented from an SDF graph, it is periodic. This period begins with the start of the first compute zone and ends with the last compute zone. During the lifetime of the application-algorithm, this period is repeated for a very large number of times. Figure 5.15(c) illustrates the periodic nature of ArcSDF. An inactive compute zone becomes active again after the completion of a period. Thus, compute zones change from active to inactive in a periodic manner. There are chances of deadlocks if the periodic release of resources from a compute zone does not coincide with the data dependencies. For example, in Figure 5.16 the resource edge creates deadlock, as *CZ1* can never complete and *CZ2* can never start. In order to prevent the creation of such deadlocks, a condition is enforced between preceding and succeeding compute zones. This condition, known as the *precedence condition*, is that all the actors of the preceding compute zones must be data independent of the succeeding compute zones. Based on this condition, resource edges between *CZ1* and *CZ2* are not allowed. Thus, prevents deadlock. In the definition of resource edges, this precedence condition is integrated into the model.

A compute zone cz is written as a 6-tuple $cz = (A, CE, R, Sch, In, Out)$. Its individual elements and their execution semantics are described as follows.

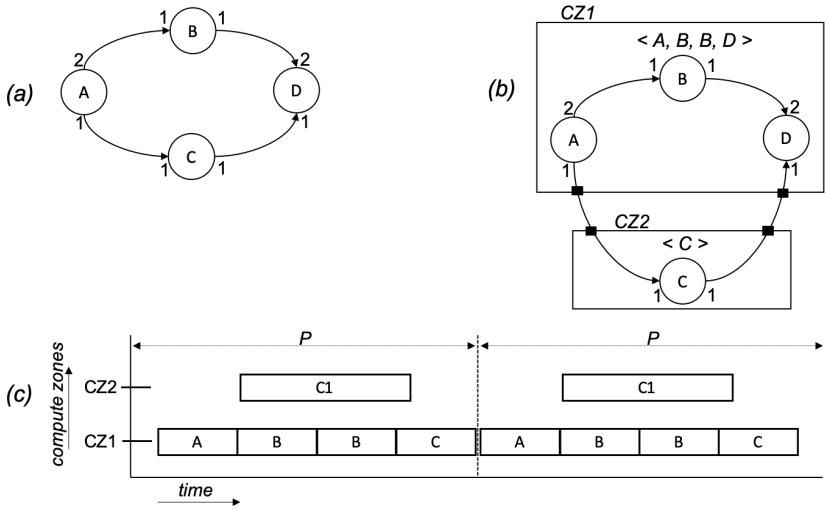


Figure 5.15: (a) An SDF model. (b) An ArcSDF instance of the SDF model, where the actor B in *CZ1* has a repetition of *2*. (c) A Gantt chart showing the period P.

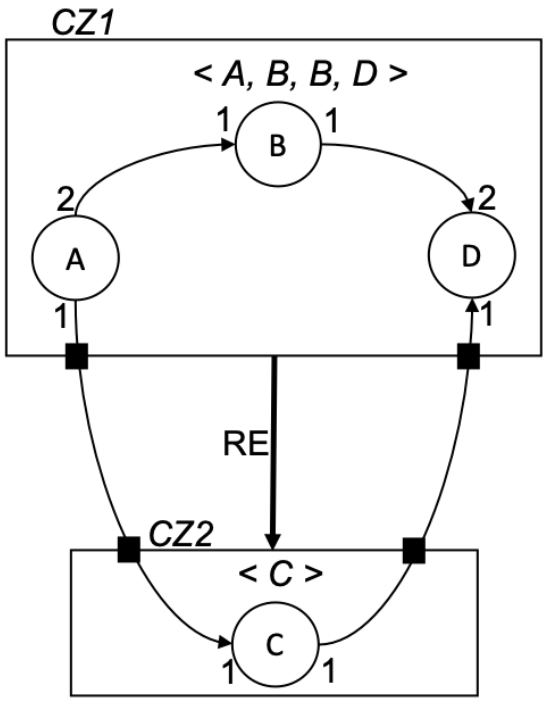


Figure 5.16: This figure shows deadlock due to the resource edge RE between compute zones *CZ1* and *CZ2* that have data dependent actors.

- The actor list A is a finite list of actors, such that $|A| \neq 0$ and $A_i \cap A_j = \emptyset$ where $A_i \neq A_j$ are actor lists of two distinct compute zones cz_i and cz_j.
- CE is the compute engine to which the compute zone is mapped.
- R represents the resources required to execute the actors in a sequence, such that $R \leq CE_{R,Max}$ and $R_{type} = CE_{R,type}$ where $CE_{R,Max}$ denotes the maximum usable resource in CE and $CE_{R,type}$ is its type of resource.
- Sch is the schedule of the compute zone, which is the order of execution for the actors in A.
- A compute zone cz is said to be completed when the actors in A have completed their execution based on the schedule Sch.
- If cz is connected to resource edges, then after its completion R is released for the succeeding compute zones.
- The period P of an ArcSDF graph G is the combination of all the schedules of its compute zones. It starts with the first compute zone and ends with the completion of the last compute zone. P is repeated for the life-time of the application-algorithm.
- In and Out represents the set of input and output interfaces, respectively, such that $In \cap out = \emptyset$.

Interface

Since a compute zone encloses a certain amount of resources in a compute engine, interfaces are vents for external communication. They take data in and out of a compute zone. Thus, they are of two types, input and output interfaces, depending on whether they bring data inside or send outside the compute zone. During the execution of an interface, it reads (writes) a certain number of tokens from an external (internal) channel to an internal (external) channel. The number of tokens an interface reads or writes in known as the rate of the interface. The input and output rates of an interface are equal. They execute whenever there are enough tokens at its input channel. Thus, interfaces can be observed as actors that operate on the threshold of compute zones. These actors have a special purpose of copying data in and out of the compute zones.

In order to maintain the SDF character of the model, there are two restrictions placed on the interfaces. Firstly, only one channel passes through an interface. Secondly, both the rates (input and output) of the interface are same as that of the rate which it's input channel is connected. This is shown in Figure 5.17. This restriction ensures that the rate of consumption and release of data tokens are

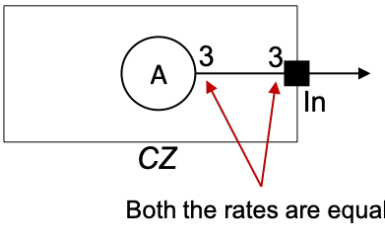


Figure 5.17: Interface rate is equal to the rate at which the actor releases tokens to the interface's input channel.

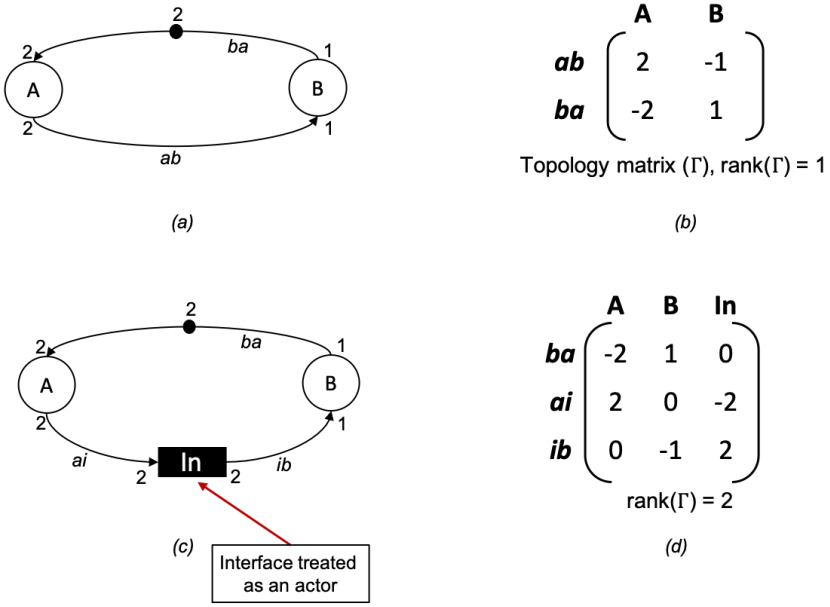


Figure 5.18: (a)An SDF graph, (b) its topology matrix with the rank of 1, which is one less than the number of actors, thus it can be scheduled and (c) insertion of an interface with the rates same as that of the input channel's actors rate and (d) the rank of the new topology matrix showing that it can still be scheduled.

consistent, so that the rank of the topology matrix of the SDF graph conforms with the schedule condition (see section 2.4.1). An example in Figure 5.18 illustrates an addition of interface to an SDF graph and the resulting rank of the topology matrix.

An interface i is formally defined as 2-tuple (r, t) where $i \in I$. Its individual elements and their execution semantics are described as follows.

- The rate r is the number of tokens the interface reads or writes to the compute zone and $r = r_{ch}$, where r_{ch} is the token release rate of the actor's port to which the channel ch is connected with the interface i.
- The type t determines whether data is written or read from the compute zone, where $t \in input, output$.
- Only one channel can pass through an interface.

Actor

Actors in ArcSDF are extended from their SDF definition by introducing an attribute to represent the implementation type and a rule to limit instantiation. The added attribute denotes the pre-engineered version of an actor for deployment, so that it is compatible with the compute engine it is placed. The additional rule allows only one instance for every actor. Thus, data parallel actors are expanded to represent concurrent deployment. Such expansions are conducted during design space exploration. Depending on the resource availability, some actors may not be expanded. However, if all actors are expanded to create an equivalent homogeneous dataflow model and there are not enough resources, then the actors are sequentially executed without a loop. This implies multiple instances of the data parallel actor is created but they are executed in a sequence. Figure 5.19 shows such a situation.

Expansion of all the data parallel actors are not necessary if there are limited resources for parallel deployment, the actor instance rule directs the deployer to create a single instance of every actor in the model. This also restricts the deployer from making multiple instances of a data parallel actor. Thus, the architecture decision of selecting data parallelism is incorporated in the model.

An actor a is defined as 3-tuple $(P_{in}, P_{out}, impl)$ where P_{in} and P_{out} are the set of input and output ports, respectively. The ports and the actor firing rules are defined in the SDF graph [57, 76]. The new additions to an actor in ArcSDF are defined here.

- *impl* represents the pre-engineered actor version that is chosen for deployment.

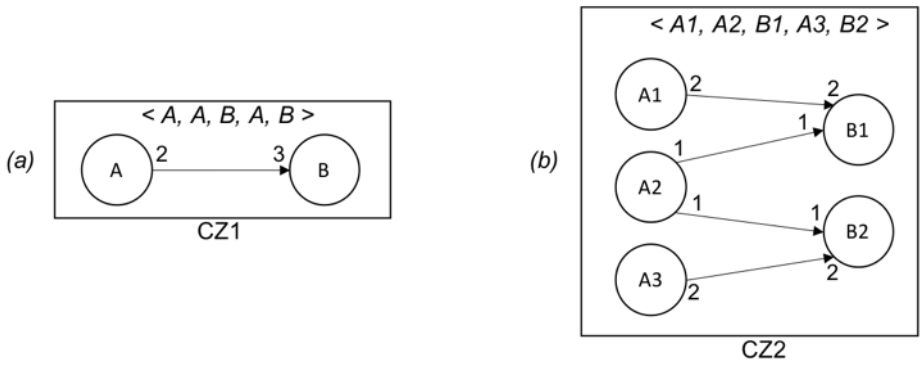


Figure 5.19: (a) A simple SDF graph with data parallelism that are not expanded. The same instance of A and B are executed multiple times. The schedule is shown in the angle brackets. (b) The former SDF graph but the data parallel actors are expanded. Multiple instances of A and B are executed. As they are on the same compute zone, they execute in a sequence. Thus, the make-span is same as the former one.

- The actors present in the model are instantiated only once. Thus, the same instance of a data parallel actor is called multiple times, unless it is explicitly expanded.

Control actor

Control actors, initially introduced in Figures 5.10 and 5.12, are proxies of usual actors that are placed on GPU or FPGA to represent its control from CPU. Since an actor is controlled during the input and output of its data, it can have two control actors. One control actor for its input and another for its output. Actors can also have just one control actor. The reason for this is that, multiple actors placed on the same FPGA/GPU can communicate directly. This eliminates the need for some of the control actors on CPU. Figure 5.12 shows control actors and the direct communication amongst B and D that are placed on CE1.

The input and output ports of a control actor are restricted to mimic the actual actor. For an input control actor, its input ports are equivalent to the actual actor's. They connect to the channels of the actual actor to receive the input data. Its output ports are also equivalent to its input ports, but the channels connected to it are used to send data to the actual actor. For an output control this is repeated to receive data from the actual actor. Then, send them to the respective channels. This restriction is integrated in the control actors.

Channel

Channels in ArcSDF are extended from its former SDF definition to add a placement decision, implementation type and enable them establish connection between compute zones. Channels in SDF join themselves itself to the actor's ports, thus connecting them. The actors that they are joined are usually distinct, unless the channel is loop-back. Similarly, a channel connects two distinct compute zones by connecting their interfaces. An interface reads (writes) tokens from (to) a channel. Channels that are outside to a compute zone are always connected to two interfaces. In other scenarios, either they are connected to an interface and a port, or both ends are connected to ports. A placement decision is denoted through an attribute pointing to the compute engine or the communication link where the channel is mapped. A channel is mapped on a communication link, when the two actors connected by it are on distinct compute engines. The implementations type of the channel is expressed through another attribute, which refers to the pre-engineered module that is used to implement the channel.

A channel c is defined as 4-tuple $(p_{in}, p_{out}, impl, map)$, $c \in C$. The individual elements are defined as follows.

- Since one end of the channel is connected to an input port or an input interface and the other end to the output counterparts, p_{in}, p_{out} represents them, where $p_{in} \in P_{in} \cup I_{in}$ and $p_{out} \in P_{out} \cup I_{out}$.
- The type of implementation is denoted by *impl*, which is used for deployment to retrieve the pre-engineered component from the repository.
- The *map* expresses the compute engine or the communication link upon which the channel is placed. Thus, $map \in CCL$ or $map \in CCN$, where CCL are the connected communication links and CCN are the connected compute nodes of the platform architecture (tier2).

Resource edges

A resource edge represents resource dependencies between compute zones. They indicate the release of resources of compute zones, so that succeeding compute zones are activated. These resource edges are described along with compute zones.

A resource edge e between two compute zones cz_1 and cz_2 is defined as the 2-tuple (cz_1, cz_2), $e \in RE$. Its direction and precedence condition are defined as follows.

- The direction is from cz_1 to cz_2, which represents that cz_2 is dependent on the resources of cz_1.
- e is allowed if all the actors in cz_2 are data independent of the actors in cz_1.

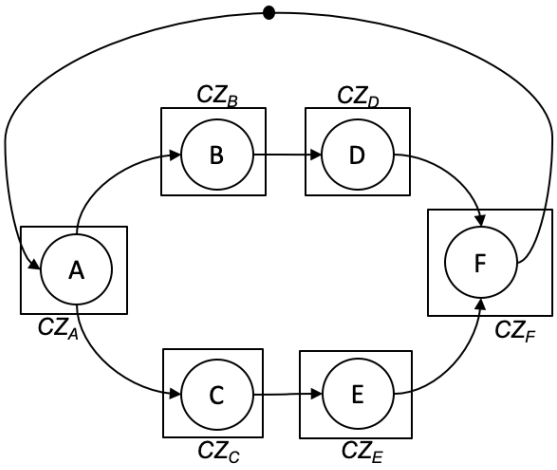


Figure 5.20: An initial-ArcSDF example graph with compute zones having one actor and no resource edges. This also an equivalent initial-ArcSDF of the previous ArcSDF graph shown in Figure 5.12.

initial-ArcSDF: preliminary ArcSDF graph

An initial-ArcSDF graph is defined as an ArcSDF graph where compute zones are occupied by only one actor and there are no resource edges. It is assumed that on a compute zone completion, which is also when all the actors on it have completed firing, the resources used by it are then released. An initial-ArcSDF example graph in Figure 5.20 shows that every compute zone is occupied by only one actor. It is clear that such representation lacks actor layout decisions. Thus, its direct implementation may lead to performance drawbacks.

The main purpose of initial-ArcSDF is however to be used for the design space exploration as a precursor to a complete ArcSDF graph to be transformed into a complete ArcSDF graph (see chapter 6). Once all the mapping and scheduling decisions of all the actors are available, some of the compute zones of the initial-ArcSDF are merged for a complete model. The actor start time, end time and where it is mapped is used by the design space exploration algorithm for optimisation to merge the compute zones of the initial-ArcSDF graph. An optimisation routine to merge the compute zones of an initial-ArcSDF is presented in the next subsection.

5.3.3 Analysis of ArcSDF

ArcSDF has several advantages over previous intermediate formats in its ability to support the analysis of the system beyond calculating throughput and makespan. The additional analysis supported by ArcSDF includes establishing the maximum resource usage, finding the earliest time slot to map an actor, and minimising the overheads introduced by the compute zones in an initial-ArcSDF graph. In this subsection, approaches for these additional analyses are presented along with a method to calculate the throughput and makespan of an ArcSDF graph. An overview of these analyses approaches are as follows:

1. The maximum resource usage of an ArcSDF graph is the amount of resources that it will require during its deployment. This analysis can be used for design space exploration to minimise resource usage. Also, it can be used to ascertain that sufficient resources are available for deployment. The maximum resource usage analysis is achieved by simulating the ArcSDF graph to calculate the maximum total resource usages of the concurrent actors on each of the compute engines.

2. When there are already mapped actors on a compute engine, the earliest time slot available to map another actor determines the suitability of a compute engine amongst others. This analysis is used within the design space exploration algorithm presented in the next chapter (Chapter 6). The earliest time slot analysis is based on searching an ArcSDF graph for sufficient free resources for the duration of the actor's execution.

3. The optimisation to minimise overheads due to compute zones is achieved by reducing the number of compute zones by merging them without compromising performance. This optimisation is developed for initial-ArcSDF, where every compute zone has one actor.

4. The throughput and makespan calculation of an ArcSDF graph is useful to estimate its performance, which finds its usage in design space exploration. This analysis is attained by first converting to its equivalent SDF. Then, applying previously published analysis approaches to calculate the equivalent SDF model's throughput and makespan.

In order to detail these analyses approaches, this subsection is divided into four parts. The first part describes the maximum resource usage analysis. The second part presents the analysis approach to find the earliest time slot to map an actor. The third part details the optimisation routine to merge compute zones. The fourth part describes a throughput and makespan calculation approach.

Maximum resource usages of an ArcSDF graph

```
INPUT:
   a. ArcSDF graph G
ALGORITHM:
   1. Create a list of is all the compute engines ce_j required by G as CE.
   2. Initialise a list R with its constituents r_j ← 0, where j ∈ CE.
   3. while an execution cycle of G is not complete do
   4.      Create segregated lists of all the fireable actors fl_j based on
           j, which is the compute engine they are mapped on.
   5.      for each fireable actor list fl_j do
   6.           temp ← Σ_{i=1}^{|fl_j|} cz_res_{i,j}, where cz_res_{i,j} are the resources consumed by
                the cz of ce_j on which actor i is mapped.
   7.           if temp > r_j then
   8.                r_j ← temp
   9.           end if
   10.     Simulate firing of all the fireable actors.
   11. end while
   12. return R
```

Figure 5.21: Pseudocode to calculate the maximum resource usages of an ArcSDF graph for all the required compute engines.

Maximum resource usage

The maximum resource usage of an ArcSDF graph is the highest amount of resources that the model will require on each compute engines during its execution. Since the compute zones with resource edges release resources after the completion of its schedule, the maximum resource usage of a compute engine is the highest amount of resources occupied by all the active compute zones. This value is calculated by simulating the execution of an ArcSDF graph to trace the resource usages.

In order to trace the resource usage, a skeleton of ArcSDF model is used, which is without the actual implementation of the actors and the channels. Dummy actor and channels are used that only accept and release dummy tokens to simulate the actor firing. A dummy token is a small sized primitive data-type, such as integer. During an actor firing, a number of dummy tokens that are defined by the actor's firing rate is accepted (released) as input (output) from the input (output) channels. An actor is fired whenever there are enough tokens available at its input channels. Just before an actor is fired, its resources usage is summed with other actors that can also fire. The resource usage of an actor on a compute engine is obtained from the compute zone it is mapped. For every actor firing, the total resource usage is traced. The trace is continued until the schedule of all the compute zones are completed, which marks the end of one execution cycle. The highest of the total resource usage of a compute engine is its maximum resource usage. Figure 5.21 describes a pseudocode for the calculation of maximum resource usages of all compute engines required.

Earliest time slot to map a new actor

The earliest time slot (*ets*) is found by searching all the available slots within a compute engine to map a new actor. It is assumed that the start and end times of the already mapped actors are available. The available time slots are searched through the ArcSDF graph containing the architectural decisions of the already mapped actors. If a new actor a is to be mapped at the desirable time dt_a on a compute engine ce_j for its execution duration $exe_{a,j}$, then a time slot closest to dt_a is the earliest time slot. This time slot needs to satisfy two conditions. The first condition is that the duration of the time slot is no less than dt_a which is the actor's execution time on ce_j. The second condition is that the time slot has enough resources $res_{a,j}$ for the execution of the actor a on ce_j.

The *ets* is found by first transforming the ArcSDF graph to its equivalent initial-ArcSDF graph, so that all the timing with the resource available slots are exposed. Then a slot is searched that is closest to the desirable start time and with enough available resources. The pseudocode to find *ets* is shown in Figure 5.22. Until line *8* the steps are to transform the ArcSDF graph to its equivalent initial-ArcSDF graph. Line *15* is the condition to find the time slot with the required amount of available resources within the already mapped actors. An example to describe the *ets* search is shown in Figure 5.23. The actor C is to be mapped on the compute engine *CE1* where two actors B and D are already placed on a compute zone. This example shows that the earliest time slot is from 20 to 30 time units, which is after B completes and resources are enough to execute D and C concurrently.

It is noted that due to the transformation of ArcSDF to initial-ArcSDF, the decisions of sequentially placing actors on a compute zone are lost. Also, it creates a large number of compute zones, which will increase the overheads of activating and deactivating a compute zone. To re-introduce the decisions of sequential actor execution within a compute zone and also to reduce the overheads due to the large number of compute zones, the initial-ArcSDF is transformed back to ArcSDF by applying an optimisation routine that merges certain compute zones. If there are several new actors to be mapped, then this might create many conversions between ArcSDF and initial-ArcSDF. This can however be avoided by applying the optimisation routine after all the new actors are mapped. An optimisation routine for initial-ArcSDF to reduce the number of compute zones is presented next.

Earliest time slot (ETS) search in an ArcSDF graph to map a new actor

INPUT:

a. ArcSDF graph G,
b. compute engine to map actor a as ce_j,
c. maximum usable resources of ce_j as max_j,
d. start time and end time matrix of actors mapped on ce_j as MAT,
e. desirable starting time of a as dt_a,
f. execution time of a on ce_j as $exe_{a,j}$ and
g. actor resource consumption matrix as RES, where $res_{a,j}$ is the resource consumption of actor i on compute engine ce_j.

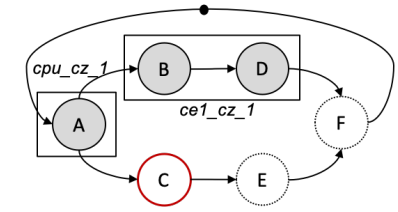
ALGORITHM:

1. Extract all the compute zones mapped on ce_j as G_{part}.
2. **for** each compute zone cz in CZ_j **do**
3. **for** each actor i in cz **do**
4. Map i on a new compute zone with its resource consumption.
5. Create associated interfaces and channels.
6. **end for**
7. Remove cz with its interfaces and resource edges.
8. **end for**
9. Sort the compute zones in G_{part} based on its starting time from MAT in ascending order. Note there is one actor in every compute zone of G_{part} as it is a partial initial-ArcSDF.
10. Create a sorted dictionary D of the starting and ending timings of all the actors in G_{part}, such that resource left at time t can be calculated as:

$$D[t] = max_j - \sum_{running\ cz} res_{i,j}$$

11. **if** $dt_a \geq$ last t in D **or** $dt_a + exe_{a,j} \leq$ first t in D **then**
12. **return** dt_a
13. **end if**
14. **for** each $t \geq dt_a$ to the second last t in D **do**
15. **if** $D[t]$ to $D[t + dt_a] \leq res_{a,j}$ **then**
16. **return** t
17. **end if**
18. **end for**
19. **return** *last t in D*

Figure 5.22: Pseudocode to find the earliest time slot (*ets*).



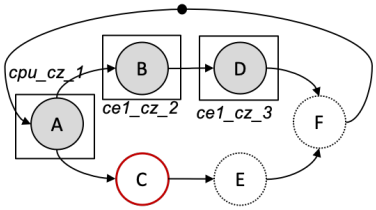
(a)

	CPU	CE1
A	**10**	15
B	20	**10**
C	60	**10**
D	20	**10**
E	40	**20**
F	**10**	20

(b)

	CE1
B	**70%**
C	**60%**
D	**40%**
E	**50%**

(c)



(d)

CPU	Start time	End time
A	0	10
CE1	**Start time**	**End time**
B	10	20
D	20	30

(e)

Time (t)	CE1 resources remaining
10	30%
20	40%
30	100%

(f)

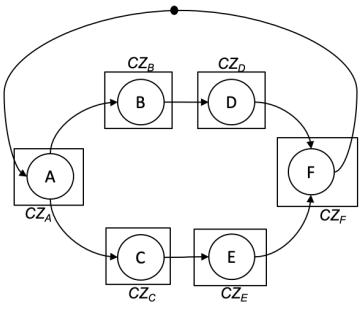
Figure 5.23: An example describing the earliest time slot (*ets*) search algorithm which is presented in Figure 5.22. The actor *C* marked in red is the new actor to be mapped on the compute engine *CE1* which already has actors *B* and *D* mapped and inside compute zone *ce1_cz_1*. It can be seen that the *ets* for this example is from 20 to 30 time units. The sub-figures are explained as follows: (a) the ArcSDF graph, (b) actor execution timings on the compute engines, (c) actor resource usages on *CE1*, (d) the equivalent initial-ArcSDF graph, (e) start and end timings of the already mapped actors and (f) resources remaining with time for CE1; it is the dictionary D in the *ets* search algorithm.

Compute zone overhead optimization

An optimisation routine to reduce the number of compute zones in an initial-ArcSDF is described here. The number of compute zones are reduced by merging them, so that actors that are sequential to one another and have similar resource usages are placed together in the same compute zone.

The merging of compute zone problem is visualised as a special case of knapsack problem [122], where the actors are to be placed inside a minimum possible number of compute zones while ensuring that the maximum resource of a compute engine is not exceeded. This problem is depicted in Figures 5.24, 5.25 and 5.26. The y-axis represents the start and end time of actors. Whereas, the x-axis represents the resource requirements of an actor. The width of an actor is its resource requirement. For a compute zone, its width equals to the actor within it that consumes maximum resources. The actors can slide across the x-axis. They can cross one another, so that they can be placed inside a suitable compute zone. This way of looking at the problem introduces three conditions; (1) within a compute zone, actors cannot overlap each other, which implies the actors that are placed inside a compute zone have distinct y-values, (2) the total width of the compute zones cannot exceed the maximum usable resource of the compute engine they are placed, and (3) every actor is placed inside a compute zone.

This problem is handled by merging the single actor compute zones with close end time and start time with the least resource usage difference. For example in Figure 5.25, compute zones R_B and R_D are merged when the first one finishes the second starts. Also, a difference of 10% was allowed. However, due to this difference the extra resources will be occupied for the duration of actor D. This resource expansion threshold can increase the overall resource usage of the ArcSDF graph. It is left to the designer to chose an appropriate resource threshold (RTH) for a given ArcSDF graph. After the compute zones are merged, a rule based approach is followed to add the resource edges among the compute zones. Resource edges are added to the next succeeding compute zones from the preceding ones. The pseudocode of this



CZ	Start time	End time	Resource usage %
CZ_B	20	30	50%
CZ_C	10	20	70%
CZ_D	30	40	40%
CZ_E	20	40	50%

(b)

Figure 5.24: An initial-ArcSDF with start, end timing and the actors resource usages on the compute engines they are mapped.

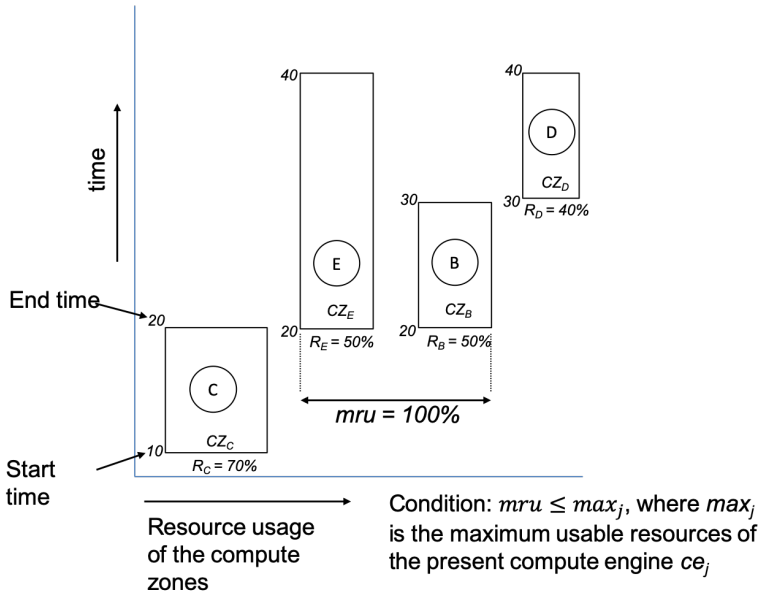


Figure 5.25: Compute zone merging problem as a special case of knapsack problem. mru is the maximum resource usage and max_j is the maximum resource that can be used for a compute engine j. For this example max_j is 100%.

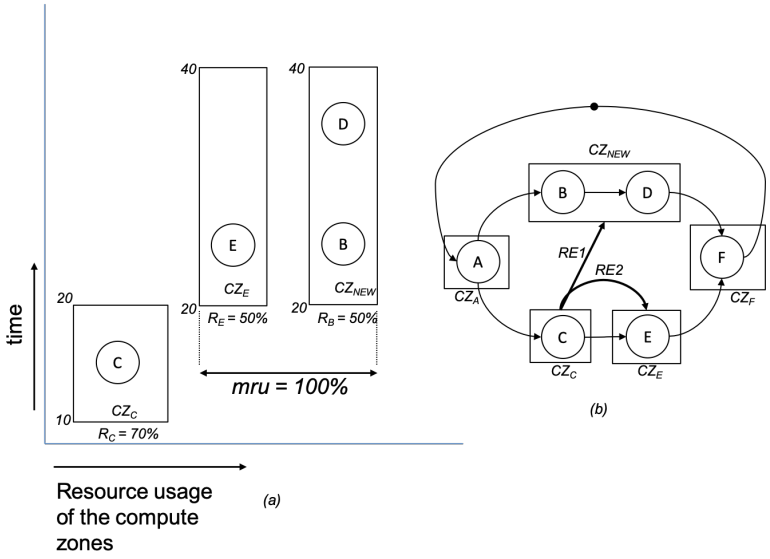


Figure 5.26: (a)This figure shows the merged compute zones where actors B and D were mapped before to create CZ_{NEW}. (b) The Arc-SDF graph with the merged compute zones.

Merging of compute zones in an initial-ArcSDF

```
INPUT:
   a. initial-ArcSDF graph iG,
   b. resource threshold RTH and
   c. start time (st), end time (et) and resource usage (ru) matrix of the
      mapped actors as MAT, where MAT_{st,j,i} is the st of actor i on compute
      engine j and similar notations are used for et and ru.
ALGORITHM:
   1. Create a list of is all the compute engines j required by iG as CE.
   2. Declare empty lists to store the optimised compute zones as OPT_j for
      all the compute engines in CE.
   3. for each j in CE do
   4.      Create a list of all the compute zones k' in j as CZ_j. The actor
           mapped on k' is k.
   5.      Sort CZ_j in ascending order of actor st and et.
   6.      Add the first compute zone in CZ_j to the empty OPT_j.
   7.      while all compute zones are not in OPT_j do
   8.          Select the next k' in CZ_j.
   9.          for each compute zone l' in OPT do
   10.             if MAT_{et,j,l} ≤ MAT_{st,j,k} then
   11.                   diff_{l,k} = MAT_{ru,j,l} − MAT_{ru,j,k}
   12.             end if
   13.         end for
   14.         if diff is not empty and min(diff) ≤ RTH then
   15.            Merge k' with l' by mapping k inside l' and deleting k'.
   16.            Create the schedule of k' based on st and et of the
                  mapped actors k and l.
   17.         else
   18.            Add k' to OPT_j.
   19.         end if
   20.         end for
   21.     end while
   22.     for each compute zone cz in OPT_j do
   23.         Find the next succeeding compute zones list CZ_succ.
   24.         Add resource edges from cz to all CZ_succ.
   25.     end for
   26. end for
   27. return iG
```

Figure 5.27: The compute zone merging algorithm.

optimisation routine to merge compute zones is presented in Figure 5.27.

ArcSDF throughput analysis

The throughput and makespan of an ArcSDF graph is calculated by converting it to an equivalent timed SDF graph [44, 123] with timing information, which is the execution time of the actors and the communication delays of the channels. Since an ArcSDF graph is formed after the mapping decisions, the timing information of an actor is its execution time on the compute engine it is mapped. Similarly the timing information of a channel is the communication delays on a compute engine or on a communication link for those channels that crosses between compute engines. Once the conversion to the equivalent SDF model is completed, standard analysis

ArcSDF to equivalent timed SDF

```
INPUT:
   a. ArcSDF model G,
   b. actor execution time matrix EXE and
   c. communication delay matrix COMM.
ALGORITHM:
   1.  for each resource edge re_i in G do
   2.      Add a restriction channel rc_i between the last actor a_l of cz_in
           and the starting actor a_s of cz_out, where cz_in and cz_out are the
           input and output compute zones of re_i.
   3.  end for
   4.  for each compute zone cz_i in G do
   5.      Find the list of actors par_i that can execute in parallel.
   6.      Add restriction channels rc_a between every actor a in par_i to
           adhering the schedule <cz_i>.
   7.      Get ce_i the compute engine where cz_i is mapped.
   8.      Annotate every actor a in cz_i with its execution time exe_a,i.
   9.      if channel ch in cz_i crosses between compute engines then
   10.       Annotate ch with its communication time comm_ch,cl, where cl is
             the communication link between the compute engines.
   11.     else
   12.       Annotate ch with zero
   13.     end if
   14. end for
   15. Remove all control actors, compute zones and interfaces in G while
       retaining all the actors and channels.
   16. return G
```

Figure 5.28: Pseudocode to convert an ArcSDF graph to its equivalent SDF representation. *EXE* is the actor execution timing data. *COMM* is the channel communication delay timing data.

algorithms for SDF [44, 60, 76, 123] are applied for the calculation of throughput and makespan.

The conversation of an ArcSDF model to an equivalent SDF is attained in three steps. Firstly, by enforcing the execution restrictions that exists due to the compute zones through extra restriction channels (RC). Then secondly, by removing the compute zones, interfaces and control actors from the graph while retaining all the actors and channels. The final step involves annotation of the execution timings of actors and channels. These steps are detailed in the algorithm presented in Figure 5.28.

The restriction channels (RC) are to enforce sequential execution between actors that can otherwise execute in parallel when there are sufficient resources. They contain tokens with negligible size, so that no communication delays are incurred because of an RC. Furthermore, an RC can be added in one of the following two situations; (1) enforce sequential execution between actors of disparate compute zones that have resource dependencies, and (2) ensure sequential execution of actors inside a compute zone.

It is noted that most of the published analysis algorithms for SDF requires it to be

converted to an equivalent DAG [60] but [76] has proposed an approach based on simulation, which can be directly applied to an SDF model (without conversion). For some SDF models that have high firing rates, the later approach is preferable, as for such SDF models the conversion to a DAG results in a large number of actors. This potentially increases the analysis time than directly applying the analysis algorithm on the SDF model. However, if data-level parallelism is to be explored, the actors will need to be expanded. Since data-level parallelisms can improve performance and applications with very high firing rates are rare, analysis by converting an SDF graph to an equivalent DAG is used in the design space exploration.

5.3.4 Conclusion

This section presented *architecture augmented synchronous dataflow* (ArcSDF) which is the intermediate data structure for the new design flow. It expresses the architectural decisions together with the application-algorithm. ArcSDF is created by augmenting SDF with for new constructs: *compute zones*, *interfaces*, *resource edges* and *control actors*. Amongst these new constructs, compute zone is the most prominent. It introduces resource usages within the SDF graph. A compute zone encloses the resource region of a compute engine and allows only one actor to be executed at a time. Thus more compute zones are required for concurrent executions. The other constructs complement the functionality of compute zones. After defining ArcSDF, its analytical capabilities were shown with new analyses, such as maximum resource usage calculation, merging of compute zones and finding the earliest time slot to map a new actor. These analyses approaches goes beyond the traditional throughput or makespan estimation.

5.4 Conclusion

This chapter presented a high-level agile platform model called *parameterised platform graph* (PPG) and a dataflow-based intermediate data structure called *architecture augmented synchronous dataflow* (ArcSDF). At first, the model for agile heterogeneous platforms was created, which provided the reference point to examine the architectural decisions. After the architectural decisions were defined, they were used to augment the SDF graph to introduce the notion of resources, as compute zones. The architectural decisions were incorporated through *compute zones*, its *resource edges*, *control actors* and *interfaces*. The ArcSDF model was described intuitively and through examples. A comprehensive definition of ArcSDF was presented with details for integration within a system and was demonstrated with examples. Both the models, PPG and ArcSDF are an integral part of AhcFlow - the new design flow for agile heterogeneous computing. After the definition of ArcSDF, its utility for analysis was presented. Three new analysis approaches apart from the traditional throughput analysis of dataflow graphs were described. The first analysis is the calculation of the maximum resource usage of the compute engines required by the ArcSDF representation. The second analysis finds the earliest time slot where enough resources are available to map a new actor into an ArcSDF that already has mapped and scheduled actors. The third analysis is an optimisation routine to merge compute zones. Finally, the fourth analysis is the calculation of throughput. The next chapter presents a design space exploration algorithm for AhcFlow that uses ArcSDF as an intermediate data structure and uses the PPG to express agile platform architectures.

Chapter 6

Agile mapping and scheduling algorithm

Contents

6.1 Introduction

In chapter 4, a new design flow for agile heterogeneous computing called AhcFlow was proposed. This chapter is focused on the details of a new design space exploration (DSE) algorithm for AhcFlow and the evaluation of a prototype that embodies the algorithm. The new design space exploration algorithm is denoted as agile mapping and scheduling (AMS) algorithm. The AMS algorithm incorporates enhancements of two other published algorithms; (1) the *heterogeneous earliest-finish-time* (HEFT) algorithm [108] and (2) an algorithm formerly developed for workflow scheduling with rental budget constraints [8], which is denoted as the Gain/Loss algorithm in this chapter. These algorithms have been previously summarized in section 2.5.4 of chapter 2. The AMS algorithm described in this chapter fulfils the following requirements that are unique to agile heterogeneous computing:

1. A mapping and scheduling algorithm that considers resource usage, so that concurrent executions of actors on the same compute engine can be taken into account.

2. Capabilities for the specialised connectivity topology of agile heterogeneous platforms, where the compute engines are not always directly connected to one another, rather their connections can go via CPU.

3. Mapping and scheduling decisions of the application-algorithm must directly influence the formation of the platform architecture, which is created with the platform components represented as a parametrised platform graph (PPG).

4. Capital cost based metric for exploration of the optimal platform architecture rather than rental cost.

In order to cater for these unique requirements, the AMS algorithm consists of the following major innovations:

1. The original *heterogeneous earliest-finish-time* (HEFT) algorithm is extended to resource-HEFT (rHEFT) that allows resources to be considered for concurrent actor executions and also incorporate the specialised connectivity of an agile heterogeneous platform. The original actor ranking and compute engine selection algorithms, that are internal to HEFT, are enhanced by taking resource into consideration.

2. The AMS algorithm consists of deterministic and random steps. The deterministic steps gradually form a platform architecture based on local mapping and scheduling decisions. However, these local deterministic decisions can lead the design space exploration to a local minimum, which is avoided by introducing global random steps. The global steps shift the exploration to a random location of the design space, from where the deterministic steps resume.

3. Supports formation of platform architecture through expansion or reduction mode. In the expansion mode, the AMS algorithm gradually adds compute engines to an initial platform architecture, whereas in the reduction mode, compute engines are removed progressively from the initial platform architecture.

This chapter is organised into two major parts. In the first part (section 6.2), the overall AMS algorithm is initially outlined. Then, its major modules are separately described, which includes rHEFT and the algorithms for expansion and reduction modes. In the second part (section 6.3), a prototype of the AMS algorithm is first shown to be feasible. Then, the prototype is used to evaluate rHEFT and the overall AMS algorithm. The framework used for the evaluation is discussed in section 6.3.1. The rHEFT evaluation shows that the enhancements of actor ranking and compute engine allocations of rHEFT significantly improves the quality of mapping and scheduling decisions as compared with the original actor ranking and compute engine allocations of HEFT. The comparison is conducted over a wide range of synthetic DAGs on multiple fixed platform architectures that are represented as PPG tier 2. This is covered in section 6.3.2. Finally, in section 6.3.3, the AMS algorithm which includes both the expansion and reduction modes of exploration together with rHEFT is evaluated using synthetic and real DAGs taken from the literature. The AMS algorithm is shown to produce better platform architectures than random platform architectures. The AMS algorithm evaluation also shows that the expansion mode performs better as compared to the reduction mode.

6.2 The agile mapping and scheduling (AMS) algorithm

List-based algorithms, such as HEFT closely consider the detailed properties of the application-algorithm, which results in a low complexity algorithm with good mapping and scheduling decisions [104, 108]. However, they have two major limitations concerning agile heterogeneous computing: (1) they are limited to fixed platforms and (2) they have inadequate support for concurrent executions on the same compute engine. In this section, a list-based agile mapping and scheduling algorithm (AMS), created around HEFT for agile platforms, is presented. The AMS algorithm is created in two stages. In the first stage, HEFT is extended to resource-HEFT (rHEFT) that caters for concurrent actor execution on the same compute engine and specialized connectivity topologies of platform architecture. Then in the second stage, AMS is created around rHEFT together with the capability for expansion or reduction of the initial platform architecture (PPG tier2). Expansion refers to addition of more compute engines to the initial platform architecture, whereas reduction refers to an removal of compute engines from the current platform architecture. The AMS algorithm either performs expansion or reduction. This selection of the mode of exploration (expansion or reduction) is dependent on the budget and the initial platform architecture. If the budget is higher than the capital cost of the initial platform architecture, then PPG operates in the expansion mode, else otherwise, it chooses the reduction mode.

In both the modes (expansion or reduction), AMS allows for the introduction of randomness in choices for the possibility of escaping from local minimums in platform architectures. The initial platform architecture provides the first design point, which is changed gradually. This gradual change is deterministic which reaches an end; either due to the budget limit or the platform constraints (PPG tier1). The randomness can then reset the starting point of the design space exploration. In order to direct the next starting point in an optimal direction, the random change is guided by the history of the previous iterations.

This section is organised into five subsections. In the first subsection, an overview of the AMS algorithm structure is presented. Its two key modules; the global platform architecture update (GPAU) module and the local platform architecture expansion (LPAE) module are introduced. Then, the second and third subsections present two enhancements of HEFT that incorporate resource and topology information, respectively. The last two subsections detail LPAE and GPAU. After detailing GPAU, the pseudocode of the full AMS algorithm is also described and its application is demonstrated with examples to show both the modes of design space exploration.

6.2.1 Overview

In this subsection, the overall structure of the AMS algorithm is described. Figure 6.1 illustrates the inputs, the output and the components of the AMS algorithm in a hierarchical manner.

There are five distinct inputs for the AMS algorithm; (1) the application-algorithm in the form of dataflow acyclic graph (DAG), (2) the parametrised platform graph (PPG) consists of the platform constraints (tier1) with the initial platform architecture (tier2), (3) the maximum budget that can be spent to form the platform architecture, (4) performance/resource consumption data of every actor and channel on various components in PPG tier1 and (5) the maximum iteration count for the AMS algorithm. The maximum iteration count puts a limit on the number of random design space points to be explored. It must be noted that the first design point is always set by the initial platform architecture defined in the PPG.

At the highest-level of the hierarchy, there are four major modules; the global platform architecture update (GPAU), the local platform architecture expansion (LPAE), the exploration history (EH) and the final ArcSDF generation (FASG).

The global platform architecture update (GPAU) is the first module that sets the mode of the AMS algorithm. This mode can be either expansion or reduction. Expansion mode is switched on when the capital cost of the initial platform architecture (PPG tier2) is lower than the maximum budget. In this mode, the second module, which is the local platform architecture expansion (LPAE) tries to expand the initial platform architecture with more compute engines for better performance. The expanded platform instance however, needs to conform with the platform constraints (PPG tier1) and its capital cost needs to be within the maximum budget. On the other hand, reduction mode is switched on when the initial platform architecture's capital cost is higher than the maximum budget. In the reduction mode, the GPAU gradually removes compute engines or swaps with less expensive compute engines until the budget is met. There will be several platform architectures which meet the budget but result in different makespans. The platform architecture with the minimum makespan is selected, so that impact on the performance is minimal. During reduction, the expansion is disabled and hence LPAE defaults to rHEFT.

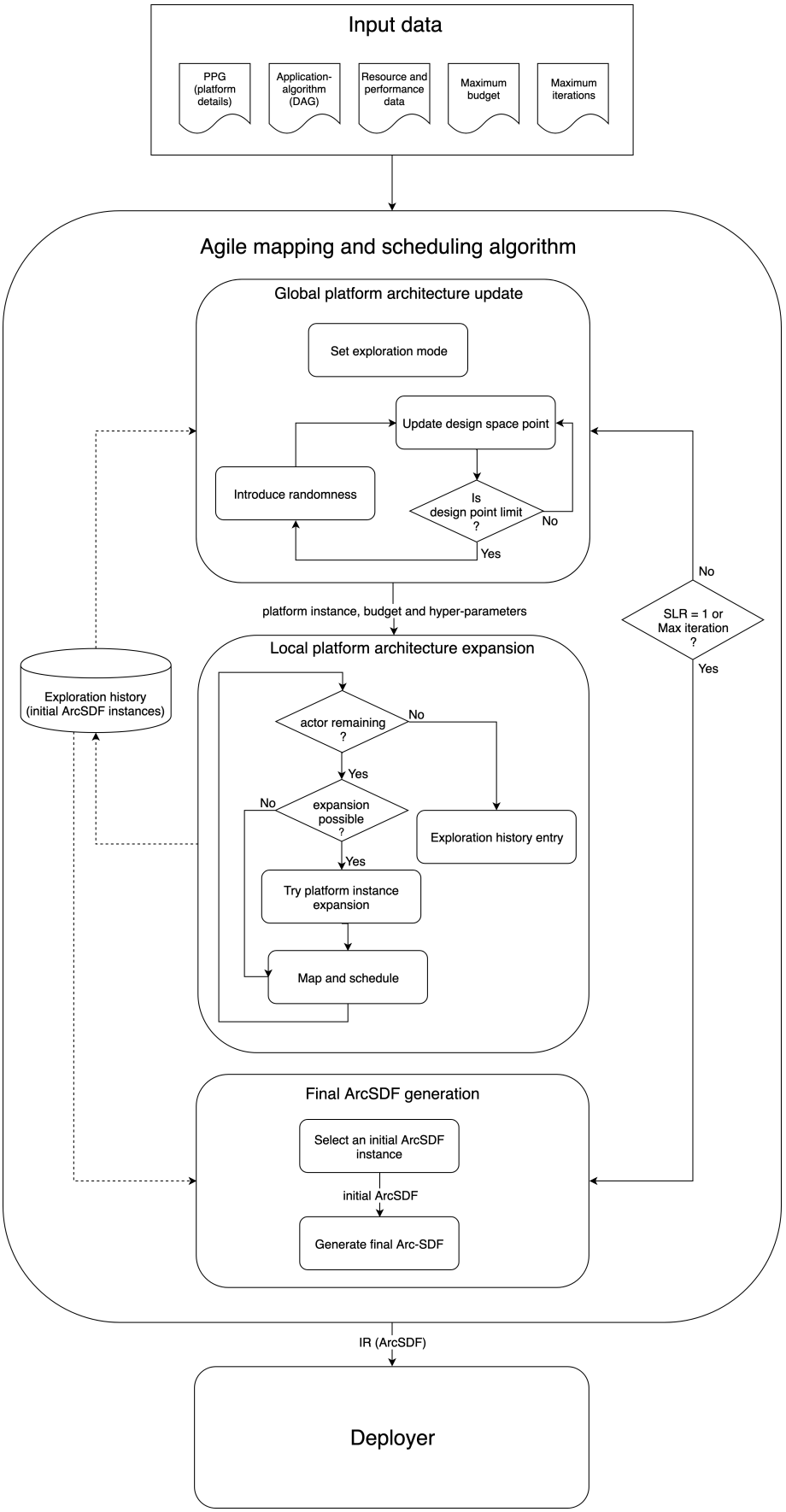


Figure 6.1: The structure of the AMS algorithm.

Another important functionality of the GPAU is to introduce randomness during the gradual change of design points. This function of introducing randomness is effective in both the modes of reduction and expansion. The gradual changes to the initial design point is initially deterministic until it reaches the constraints defined in the PPG tier1. In the expansion mode, this happens when the budget reaches to the maximum or the platform constraints limit any further addition of compute engines. When in the reduction mode, the limit is reached once the platform instance cannot be reduced any more; that is the platform instance is reduced to just a CPU. When one of these limits are reached, the GPAU selects a random platform instance that fulfils the constraints of PPG. In this situation, the GPAU also selects a random set of hyper-parameters, within the limits set by the user, to be used inside the LPAE. The complete list of hyper-parameters are contained in the table 6.1. The meaning and significance of these hyper-parameters are introduced gradually through the chapter.

The inputs for the GPAU module are the PPG representation, maximum budget and the maximum iteration value. Based on these inputs, the GPAU outputs a platform instance, budget and a set of hyper-parameters. These three outputs are used along with the DAG and the performance/resource consumption data, by the second module (LPAE) to produce the mapping and scheduling decisions. These mapping and scheduling decisions are targeted for the input platform instance. However, if the budget is higher than the capital cost of the platform instance, the LPAE may expand the platform instance. In the reduction mode, the budget for the LPAE is less than the capital cost of its platform instance. Thus, only mapping and scheduling decisions are explored. The LPAE outputs a possibly expanded platform instance and a rudimentary ArcSDF model (initial ArcSDF[1]). In the final ArcSDF generation (FASG) module, a chosen initial ArcSDF undergoes optimisation and model completion routines to generate a complete ArcSDF model.

The LPAE module is a synthesis of rHEFT (enhanced HEFT for concurrent actor execution and specialised topology) and a modification of the Gain/Loss algorithm of Sakellariou et al. 2007 [8]. LPAE is responsible for: (1) searching the optimal mapping and scheduling decisions for a platform instance and (2) conducting expansion of the initial platform architecture when in the expansion mode. In reduction mode, the expansion facilities of LPAE are disabled and the algorithm defaults of rHEFT. In expansion mode, changes in the platform instance are deeply embedded in rHEFT. This allows to explore several options without the need to find the mapping and scheduling decisions multiple times. LPAE expands a platform instance by locally optimising the mapping and scheduling decision of an actor by incorporating a new compute engine in the platform instance. The available compute engines are placed in a ranked list which is scored on the basis of their contribution in lowering the makespan. This score considers three factors; acceleration of the present actor, estimated acceleration of the remaining actors and the dampening

[1]The initial ArcSDF should not be confused with the initial platform instance. The initial ArcSDF considers every actor to be inside one unique compute zone. Its applications are explained in chapter 5 in section 5.3.

Table 6.1: This table lists the hyper-parameters in the AMS algorithm.

Name	**Denotation**	**Description**
Resource factor	RF	RF is used to calculate actor rank in rHEFT. It determines the weighting given to the resource consumption of actors.
Mapping threshold	MT	MT is used in rHEFT for the selection of compute engines. It signifies the trade-off between resource consumption and the compute engine acceleration.
Local factor	ce_{local}	ce_{local} is used within LPAE during expansion mode for the creation of dynamic compute engine rank. It determines the weighting of local gain for a compute engine.
Global factor	ce_{global}	ce_{global} is used within LPAE during expansion mode for the creation of dynamic compute engine rank. It determines the weighting of global gain for a compute engine.
Communication factor	ce_{comm}	ce_{comm} is used within LPAE during expansion mode for the creation of dynamic compute engine rank. It determines the negative weighting of communication delay for a compute engine.

impact of communication links. A set of three hyper-parameters are used to scale these three factors. Then the highest scoring compute engine, determined by summing the scaled values of these three factors, is selected to be incorporated into the platform. Once the budget is reached, expansion stops and control is transferred to the GPAU. It is possible that halfway through the LPAE execution, the budget is reached and then further expansion stops but the remaining actors are mapped and scheduled as per the rHEFT algorithm.

The third module, which is the exploration history (EH) module records previous platform instances, mapping and scheduling decisions from the LPAE as a history. A record consists of the initial ArcSDF, the platform instance, makespan and the capital cost. At the end of the AMS algorithm, the user selects one of the records from the history, which is then sent to the final ArcSDF generation (FASG) module.

The fourth module is called the final ArcSDF generation (FASG) module. This module is responsible for the selection of an optimal record from the EH. Control moves to this module after the AMS algorithm reaches its maximum iteration limit or the best theoretical makespan is attained within budget. The user is involved in the first step of selecting an optimal record from the set of optimal records based on the makespan length and the capital cost of the platform instance. From this selected record, the initial ArcSDF is retrieved. The FASG then applies a peephole [124] like compute zone optimisation routine (see section 5.3.3) to reduce the number of compute zones to combine compute zones where the actors in these zones are sequential and use similar resources. The final step of FASG is to complete the ArcSDF model by inserting the so called pre-engineered control actors, which manage communication between GPU/FPGA and their controlling CPU. The output of FASG is thus a complete ArcSDF model. This ArcSDF model is the output of the AMS algorithm that is used for deployment.

In the next subsection, the resource-HEFT part of the AMS algorithm is detailed.

6.2.2 rHEFT1: Resource conscious mapping and scheduling

There are two versions of rHEFT. This subsection presents the first version of the new resource-HEFT (rHEFT-1) algorithm that allows parallel execution of actors on the same compute engine. rHEFT-1 is created by mapping and scheduling actors on compute engines with multiple compute zones. In the following subsection 6.2.2 rHEFT-2 is described that caters for specialised topologies.

Recall that in ArcSDF, the *compute zones* are partitions of a compute engine that allow multiple actors to execute in parallel. rHEFT-1 is created by first introducing compute zone resources from ArcSDF. rHEFT-1 redefines the earliest finish time (EFT) by using the earliest time slot algorithm (see section 5.3.3). Then, the underlying ranking and allocation algorithms are enhanced, so that the significance of resource consumption is taken into account.

It is noted that enabling multiple actors to execute concurrently on the same compute engine necessitates scheduling of communication links. The reason is that multiple actors on the same compute engine might need to communicate with actors on another compute engine. In order to simplify the introduction of resources, it is assumed that there are all-to-all communication links amongst the compute engines and that the communication links can be shared without compromising communication speed. This assumption permits the original earliest start time (EST) and earliest finish time (EFT) functions to be used only with the resource modifications. However, communication links are generalised in the second version of (rHEFT-2), which is presented in the next subsection (6.2.2) to consider specialised connectivity constraints of agile heterogeneous computing platforms.

The creation of rHEFT-1 is described in this subsection in three parts. In the first part, the basic introduction of resources is presented and its capability to consider parallel execution of actors on the same compute engine is demonstrated with two examples. Then in the second and third part, the ranking and allocation algorithms are improved, respectively. The improvement is achieved by considering actor resource usage.

rHEFT1: Introducing compute zone resources in HEFT

Resource consumption is introduced by using the earliest time slot (*ets*) algorithm (see section 5.3.3) to find the earliest time to map an actor on a compute engine. The *ets* algorithm finds the space in a compute engine where enough compute zone resources are available for a certain duration of time to execute an actor. Since *ets* operates on initial-ArcSDF [2], an intermediate representation (IR) in the form of an initial-ArcSDF model is used within HEFT to capture the mapping and

[2]In an initial-ArcSDF, every actor is mapped onto an different compute zone, so that all available free resources are utilised during the search of earliest slot time.

scheduling decisions. Each time an actor is mapped and scheduled on a compute engine, it's earliest start time (EST) and the earliest finish time (EFT) are added to the IR, along with the compute engine to which it is mapped. For the next actor, the updated IR is used again by the *ets* algorithm to find the next EST on the compute engine. This continues until all the actors are mapped. Figure 6.2 shows the pseudocode of rHEFT-1. It is noted that the ranking and allocation algorithm are resource agnostic. They are similar to that in the original HEFT algorithm (see section 2.5.4), with differences arising in the allocation algorithm for the incorporation of *est*. The pseudocode of the new allocation algorithm is shown in figure 6.3 details the inclusion of *ets*.

Resource-based heterogeneous earliest first (rHEFT) algorithm

```
REQUIRED:
   a. application DAG G,
   b. set of compute engines CE,
   c. actor execution time matrix EXE,
   d. actor resource usage matrix RES,
   e. communication delay matrix COMM,
   f. resource-based upward ranking algorithm rank_up and
   g. resource-based earliest finish time (REFT) allocation algorithm
ALGORITHM:
   1. Calculate rank_up for all nodes (actors) by traversing G upward,
      starting from the exit node.
   2. Sort the nodes in a list L by non-increasing order of rank_up values.
   3. Initialise IR an initial-ArcSDF model with zero compute zones for
      every compute engines in CE.
   4. while there are unscheduled nodes in L do
   5.      Select the first node n_i in L and remove it.
   6.      for each compute engine ce_k do
   7.           Calculate eft_i,k, est_i,k ← REFT(n_i, p_k, IR, EXE, RES, COMM, G)
   8.      end for
   9.  Assign n_i to ce_j that minimises eft value of n_i.
   10. Update IR to record est_i,j and eft_i,j
   11. end while
   12. return IR
```

Figure 6.2: Resource-HEFT (rHEFT-1) algorithm with original ranking and allocation approach. The $rank_{up}$ algorithm was previously described while reviewing the original HEFT algorithm in section 2.5.4. This algorithm calls REFT in line 7 which is shown in 6.3.

Resource-based earliest finish time (REFT) algorithm to find the Earliest finish time (EFT) and earliest start time (EST) on a compute engine

REQUIRED:
- a. node n_i to be allocated,
- b. compute engine ce_j for allocation,
- c. actor execution time matrix *EXE*,
- d. actor resource usage matrix *RES*,
- e. communication delay matrix *COMM*,
- f. application-algorithm DAG *G*,
- g. ongoing decisions as initial-ArcSDF model *IR*, and
- h. algorithm to find the earliest time slot *(ets)*

ALGORITHM:

1. $et_i \leftarrow ets(IR, ce_k, exe_{i,k}, res_{i,k})$
2. Find all preceding nodes pre_i of n_i from *G*
3. **for** all preceding nodes p_k in pre_i **do**
4. From *IR*, retrieve compute engine ce_k where p_k is mapped and it's completion time ct_k.
5. Calculate possible start time $st_k \leftarrow ct_k + comm_{k,j}$.
6. **end for**
7. From all the predecessor's *st* find the maximum, which is the start time (st_i) of n_i.
8. $est_i \leftarrow max(st_i, et_i)$
9. $eft_i \leftarrow est_i + exe_{i,j}$
10. **return** eft_i, est_i

Figure 6.3: Resource-based earliest finish time (REFT) algorithm called from rHEFT-2 in 6.2. The earliest time slot (*ets*) algorithm was described in section 5.3.3.

The algorithm of rHEFT-1 that introduces resources in the original HEFT algorithm is demonstrated using two examples. They are illustrated in figures 6.4 and 6.5. The first example (Ex1) shows the mapping and scheduling of a DAG consisting of six actors on a platform architecture with a *CPU* and a *GPU*. The ranking of the actors based on their computation times on the *CPU* and the *GPU* is shown in figure 6.4(c). In the same figure, the mapping and scheduling decisions are shown consisting of the time each actor starts on a compute engine and their end time. The mapping and scheduling decisions are illustrated through a Gantt chart in figure 6.4(f). This shows that actors C and D are executing concurrently on the *GPU*. Also, B and E show some amount of concurrent executions on the *GPU*.

The platform architecture of the second example (Ex2) is constituted of one of each type of compute engines and executing a different DAG is used. The scheduling and mapping decisions demonstrates the ability of the resource-HEFT algorithm in enabling actors B and C to execute simultaneously on different compute engines.

The impact of the communication delays can be seen from the table with the mapping and scheduling timings. In the second example (Ex2) (figure 6.5), actor E, which is mapped on the *GPU* needs data from C on the *CPU* and from B on the *FPGA*. Amongst these two actors, C takes the longest to complete. Thus, the starting time of E is when C completes plus the communication delay.

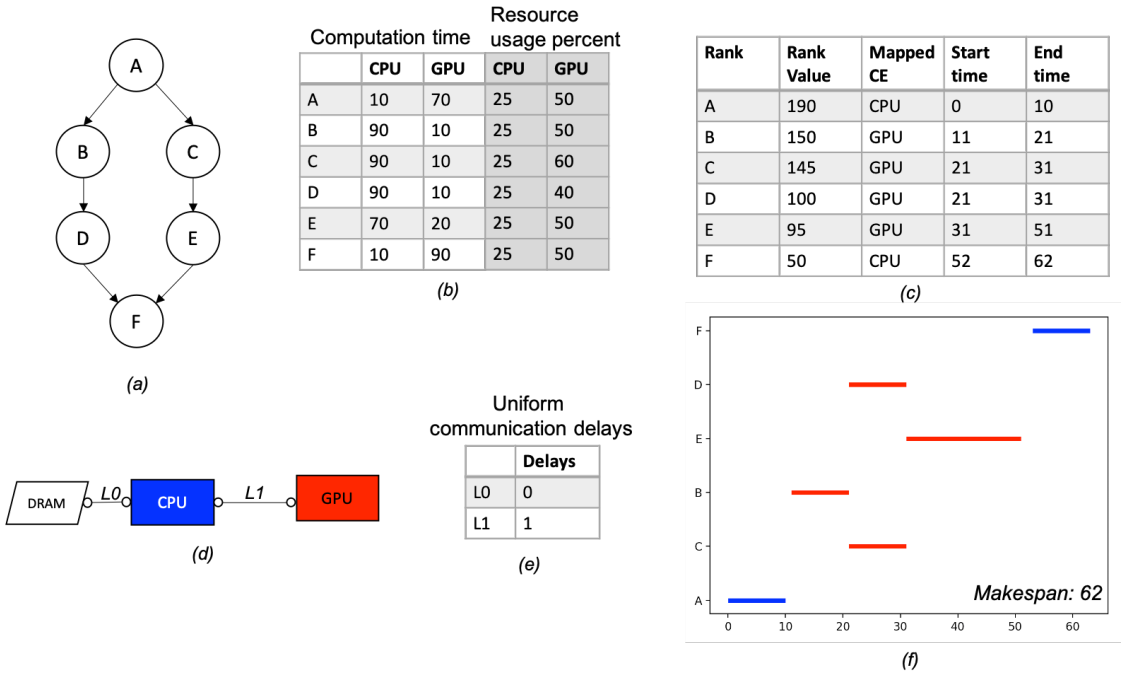


Figure 6.4: Ex1 Resource-HEFT (rHEFT-1) enabling concurrent actor execution on the same compute engine. The platform architecture consists of one CPU and one GPU.

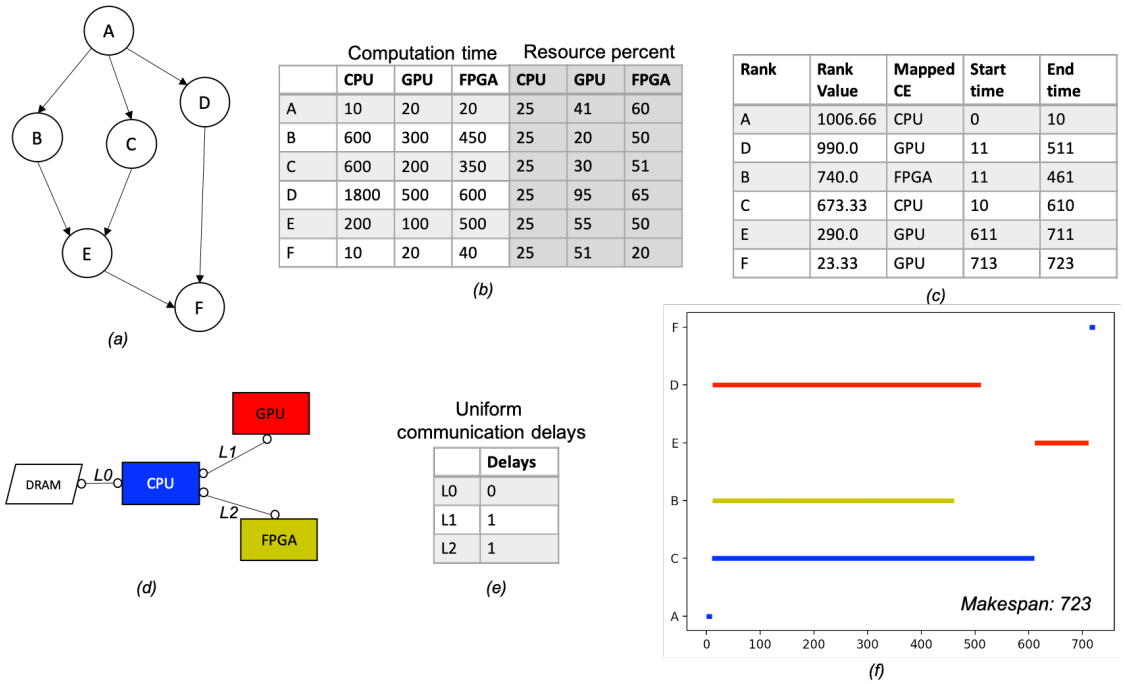


Figure 6.5: Ex2 Resource-HEFT (rHEFT-1) enabling concurrent actor execution on the same compute engine. The platform architecture consists of one of each type of compute engines.

Although execution of multiple actors can be taken into account, better mapping

and scheduling decisions are also possible if resource usage of actors are considered. The next two parts of this subsection revisits the ranking and allocation algorithms to incorporate resource usage.

rHEFT-1a: Enhanced actor ranking and compute engine selection algorithm

Although rHEFT-1 enabled concurrent execution of actors on the same compute engine by introducing compute zone resources with the original HEFT algorithm, the internal actor ranking and the compute engine selection algorithms were resource agnostic. In this part, the actor ranking and the compute engine selection algorithms are enhanced to make them resource conscious. Initially, the actor ranking algorithm is enhanced to consider resources. Then, the compute engine selection algorithm is improved to incorporate resource usage of actors. The improvements are illustrated by applying these enhanced algorithms on the previous examples (Ex1 and Ex2). It is acknowledged that all DAGs on all platform architectures will not result in an improvement. The examples used are illustrative only. An elaborate evaluation will be conducted in section 6.3.2, which will show that there is a significant improvement of average makespan over a very large number of randomly generated DAGs, as compared with the original actor ranking and compute engine selection algorithms.

rHEFT-1a: Resource conscious actor ranking

rHEFT-1a is based on a heuristic that if actors with higher resource requirements and with longer execution times are mapped first, then the remaining resources can be better utilised for rest of the actors. An actor with higher resource usage needs a lot of resources continuously for the length of its execution. Compute engines with already placed actors might not have an earlier slot time for a new actor with higher resource requirement. Moreover, if such an actor has longer execution time and has to wait significantly for placement on the compute engine, then it may profoundly increase the makespan.

The original HEFT actor ranking algorithm (see section 2.4.3) only prioritised actors with higher execution times. Whereas, in resource conscious ranking, the actor's resource usage value is also taken into account, such that actors with higher resource usage and longer execution times are prioritised. This is explained in the next paragraph. The detailed algorithm is contained in figure 6.6.

In the resource conscious ranking, the actors are ranked by traversing the graph upwards. The weight of every actor is the sum of its average execution time with a weighting of its average resource consumption value. The weighting is derived from a hyper-parameter called *resource factor* (RF), which is set by the designer to signify the influence of resource usage on the rank. Figure 6.6 shows the equation

to calculate this weighting. It is noted that a very high RF will dominate the rank with only resource consumption values. This is not desirable as placing actors with higher execution times are more critical. The average execution time and the average resource consumption values are calculated from the inputs of the AMS algorithm, where the performance and resource of all actors on every compute engines exists.

Resource conscious upward rank

$rank_{up,res}$ is calculated as upward rank starting from the exit node (actor). The $rank_{up,res}$ of node (actor) n_i can be recursively written as follows;

$$rank_{up,res}(n_i) = w_i + \max_{n_i \in succ(n_i)} \left(\overline{comm_{i,j}} + rank_{up,res}(n_j) \right)$$

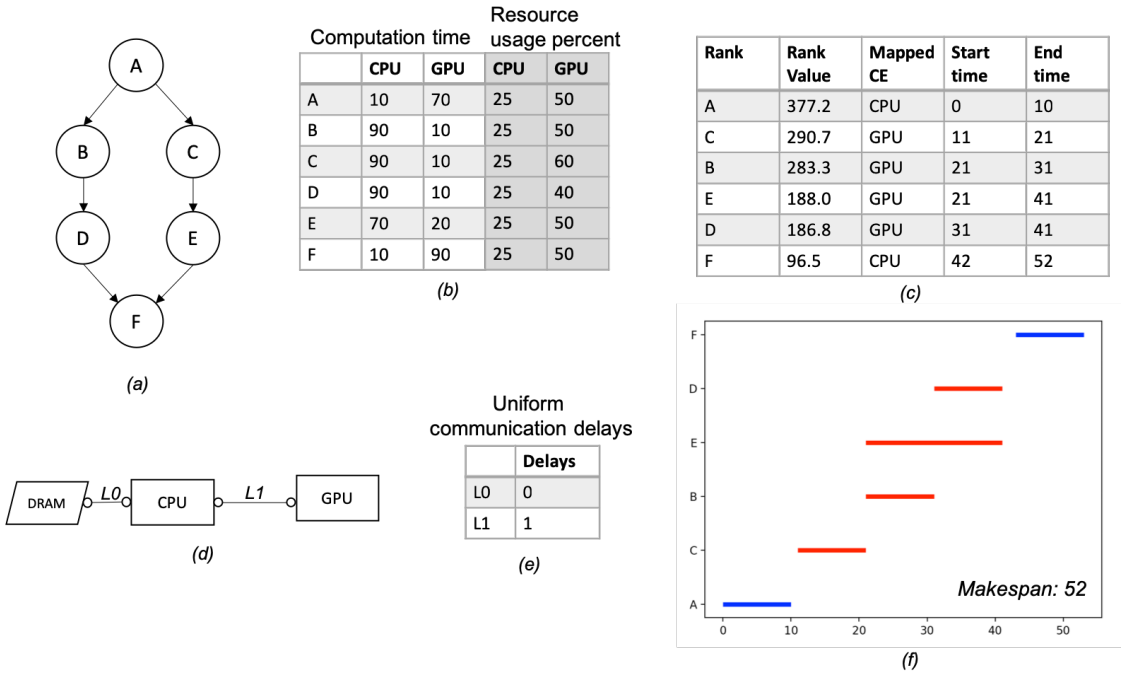
w_i is the weight of node (actor) n_i. It is calculated as the average execution time on all the compute engines added with a factor of the actor's average resource consumption. Formally, w is defined as follows;

$$\overline{exe_i} = \sum_j^P exe_{i,j} / |P|$$
$$\overline{res_i} = \sum_j^P exe_{i,j} / |P|$$
$$w_i = \overline{exe_i} + RF * \overline{res_i}$$

$succ(n_i)$ returns all the succeeding actors of n_i. Therefore, the max block finds the maximum rank of all the succeeding actors with their average communication delay. The average communication delay, $\overline{comm_{i,j}}$ is the average communication time of the channel between n_i and n_j on all the communication links.

Figure 6.6: An enhanced actor ranking for rHEFT-1a by considering resource usage. This replaces the ranking algorithm used in figure 6.2.

The new ranking algorithm is demonstrated with an example Ex3 of figure 6.7, where $RF = 2.4$. This example is based on the same input data that was previous example Ex1 presented in figure 6.4. The improvement in the makespan can be seen by comparing both the figures. This improvement is due to the change in the ranking, where actor C is given extra priority, as it consumes more resources. The new rank is shown in figure 6.7(b).



	CPU	GPU	CPU	GPU
A	10	70	25	50
B	90	10	25	50
C	90	10	25	60
D	90	10	25	40
E	70	20	25	50
F	10	90	25	50

Rank	Rank Value	Mapped CE	Start time	End time
A	377.2	CPU	0	10
C	290.7	GPU	11	21
B	283.3	GPU	21	31
E	188.0	GPU	21	41
D	186.8	GPU	31	41
F	96.5	CPU	42	52

	Delays
L0	0
L1	1

Figure 6.7: Ex3 An example to demonstrate that part of rHEFT-1a where resource conscious upward ranking is introduced. The makespan from this example can be compared with the previous example Ex1 in figure 6.4.

In fact the application of this new ranking algorithm does not improve the other example Ex2 shown in figure 6.5. This will require enhancement in the allocation algorithm, which is the topic of the next part.

rHEFT-1a: Resource conscious compute engine selection

The second modification to rHEFT-1 internally is to introduce an enhanced compute engine selection algorithm that reflects compute zones as resources in compute engines. The major change is in step 9 of the rHEFT-1 algorithm, which previously selected a compute engine based on EFT. Now, in rHEFT-1a, it selects a compute engine with maximum resource performance value (rpv). The computation of the rpv is added to the algorithm shown in the steps 9 to 17 in Figure 6.8.

The idea behind this new algorithm is, compute engines that can provide better actor performance with fewer resources are given priority. The new compute engine selection algorithm consists of two steps. They are repeated for every actor to be mapped. In the first step, a list of compute engines, where the actor to be mapped can have similar performances to the best possible one, is formed. Then, in the second step, the compute engine with least resource requirements for the actor is selected. In order to ensure that a significant amount of performance is not compromised for resources, a hyper-parameter called mapping threshold (MT) is used in the first step to form the list of compute engines with similar performances.

The value of the mapping threshold, MT is set by the designer. If the application

DAG has a lot of parallel executions, then a higher value of MT is preferred, as compute engines may increase the number of concurrent actor executions. A very small MT will make the algorithm behave like the original HEFT's compute engine selection algorithm. Whereas, a very large MT value will consider all the compute engines and can lead to sub-optimal mapping and scheduling decisions. The reason, when every compute engine is taken into account for acceleration to resource usage calculation, it will result in the choice of a compute engine that might have a very low resource usage value but high earliest finish time. Furthermore, the value of MT is also dependent on the numerical sizes of the executing time of actors. The designer needs to take these into account to set the value of MT.

Those compute engines on whom the EFT of the actor to be mapped falls within EFT_{MIN} to $EFT_{MIN} + MT$ are selected. Amongst this list of compute engines, the one with maximum acceleration to resource usage ratio is selected for the actor to be mapped. Acceleration is normalised to CPU times, which is calculated by dividing the EFT on CPU with the EFT on the compute engine. Pseudocode of this new compute engine selection algorithm is shown in figure 6.8.

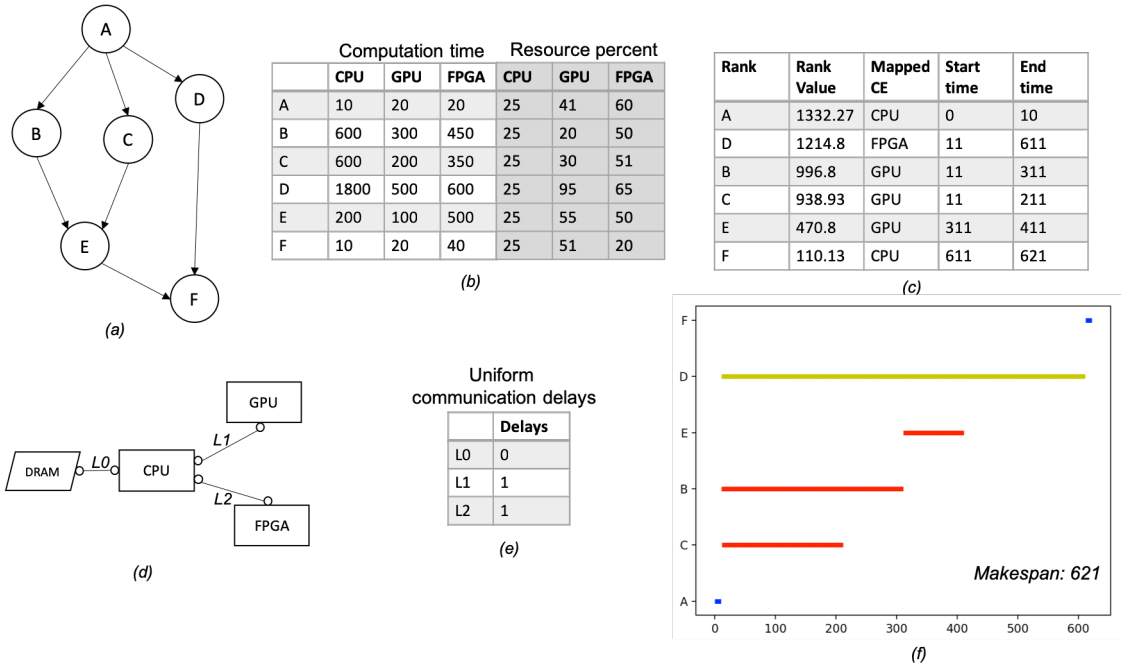
rHEFT with resource-based compute engine selection algorithm

```
REQUIRED:
   h. application DAG G,
   i. set of compute engines CE,
   j. actor execution time matrix EXE,
   k. actor resource usage matrix RES,
   l. communication delay matrix COMM,
   m. resource-based upward ranking algorithm rank_up,res and
   n. resource-based earliest finish time (REFT) allocation algorithm
ALGORITHM:
   1. Calculate rank_up,res for all nodes (actors) by traversing G upward,
      starting from the exit node.
   2. Sort the nodes in a list L by non-increasing order of rank_up,res
      values.
   3. Initialise IR an initial-ArcSDF model with zero compute zones for
      every compute engines in CE.
   4. while there are unscheduled nodes in L do
   5.     Select the first node n_i in L and remove it.
   6.     for each compute engine ce_k do
   7.         Calculate eft_i,k, est_i,k ← REFT(n_i, p_k, EX, RES, COMM, IR, G)
   8.     end for
   9.     From all the previously calculated eft, find eft_min the minimum
          eft.
   10.    for each compute engine ce_k do
   11.       diff_k ← eft_k - eft_min
   12.       if diff_k <= MAP_TH then
   13.          rpv_k ← (eft_CPU - eft_k)/res_i,k
   14.       else
   15.          rpv_k ← -1.0
   16.       end if
   17.    end for
   18.    Assign n_i to ce_j that maximises the rpv value of n_i.
   19.    Update IR to record est_i,j and eft_i,j
   20. end while
```

Figure 6.8: Pseudocode of the overall rHEFT with the resource conscious compute engine selection algorithm.

The previous example, Ex2 in figure 6.5 is used again to demonstrate the improvement due to resource conscious compute engine selection. Example Ex4 shown in Figure 6.9 shows the new mapping the scheduling decisions. In this example, the MT value is set as 120 and the RF is set as 2.4. The makespan is improved due to the local performance compromise of D by mapping it on $FPGA$, which creates more space for B and C to execute simultaneously. Since rHEFT-1 assumes communication links to be all-to-all connected with no-loss sharing of the links (no wait time), the specialised connectivity of agile heterogeneous computing platform has not been taken into account yet.

This concludes discussions of rHEFT-1a. In the next subsection rHEFT-2 is described, which includes all of rHEFT-1a plus capabilities to connect limited topologies connectivities in many real systems.



	CPU	GPU	FPGA	CPU	GPU	FPGA
A	10	20	20	25	41	60
B	600	300	450	25	20	50
C	600	200	350	25	30	51
D	1800	500	600	25	95	65
E	200	100	500	25	55	50
F	10	20	40	25	51	20

Rank	Rank Value	Mapped CE	Start time	End time
A	1332.27	CPU	0	10
D	1214.8	FPGA	11	611
B	996.8	GPU	11	311
C	938.93	GPU	11	211
E	470.8	GPU	311	411
F	110.13	CPU	611	621

Figure 6.9: Ex 4: An enhanced compute engine selection algorithm for rHEFT by considering resource usage. The makespan from this example can be compared with the previous example Ex2 in figure 6.5.

6.2.3 rHEFT-2: Specialised connectivity topology

In the previous subsection, resource consumption of actors was added into the original HEFT, so that parallel execution of actors is allowed on a compute engine. This enhancement, called the first version of resource-HEFT (rHEFT-1a) was achieved by assuming that every compute engine is connected to one another and that communication links are always available (no wait time). In this subsection, the second version of resource-HEFT (rHEFT-2) is presented that relaxes the previous connectivity assumption.

rHEFT-2 encompasses the specialized connectivity topologies targeted in the new design flow for agile heterogeneous computing proposed in chapter 4. This design flow has allowance for a much richer range of interconnections between compute engines than envisaged in the original HEFT algorithm. The communication links between compute engines are shared amongst concurrently executing actors, which can result in waiting time, as the communications links are sequential. Furthermore, every compute engine may not be directly connected to one another. For example, communication between *GPU* and *FPGA* may take place through the CPU, in fact in most platform architectures direct links will not be available.

The pseudocode of the rHEFT-2 algorithm is shown in figure 6.10 and the pseudocode to calculate the earliest start time with specialised connectivity topology is shown in figure 6.11. The basic structure of rHEFT-2 is similar to rHEFT-1a but,

there are two main additions: (1) communication links are modified to include waiting time and (2) the algorithm is made aware of the fastest route available in the connectivity topology. These additions are demonstrated with an example where the impact on mapping and scheduling decisions due to these additions are discussed. Therefore, this subsection is organised into two parts. The first part presents the additions of specialised connectivity topology and demonstrates them with an example. Then the second part presents a discussion on the implications of mapping and scheduling decisions due to the allowance of specialised connectivity topology.

rHEFT-2: Resource-HEFT for specialised communication topology

```
REQUIRED:
   a. application DAG G,
   b. platform architecture PA,
   c. actor execution time matrix EXE,
   d. actor resource usage matrix RES,
   e. communication delay matrix COMM,
   f. resource-based upward ranking algorithm rank_up,res,
   g. communication and resource-based earliest finish time (REFT_COMM)
      allocation algorithm
ALGORITHM:
   1. Extract the set of constituent compute engines CE from PA.
   2. Calculate rank_up,res for all nodes (actors) by traversing G upward,
      starting from the exit node.
   3. Sort the nodes in a list L by non-increasing order of rank_up,res
      values.
   4. Initialise IR an initial-ArcSDF model with zero compute zones for
      every compute engines in CE.
   5. while there are unscheduled nodes in L do
   6.     Select the first node n_i in L and remove it.
   7.     for each compute engine ce_k do
   8.        R_MIN,k, eft_i,k, est_i,k ← REFT_COMM(n_i,ce_k,PA,EXE,RES,COMM,IR,G)
   9.     end for
   10.    From all the previously calculated eft, find eft_min the minimum
          eft.
   11.    for each compute engine ce_k do
   12.       diff_k ← eft_k - eft_min
   13.       if diff_k <= MAP_TH then
   14.          rpv_k ← (eft_CPU - eft_k)/res_i,k
   15.       else
   16.          rpv_k ← -1.0
   17.       end if
   18.    end for
   19.    Assign n_i to ce_j that maximises the rpv value of n_i.
   20.    Assign input channels of n_i to routes in set R_MIN,j.
   21.    Update IR to record est_i,j and eft_i,j
   22. end while
```

Figure 6.10: rHEFT-2 algorithm for specialised connectivity topology for agile heterogeneous computing. The resource based early finish time computation (REFT) has been modified to take into account of the communication ($REEFT_{COMM}$) which is shown in figure 6.11

Communication and resource-based earliest finish time ($REFT_{COMM}$) allocation algorithm

```
INPUT:
   a. the node n_i to be allocated,
   b. the compute engine ce_j for allocation,
   c. platform architecture PA,
   d. actor execution time matrix EXE,
   e. actor resource usage matrix RES,
   f. communication delay matrix COMM,
   g. the application DAG G,
   h. ongoing scheduling and mapping decision in IR and
   i. algorithm to find the earliest time slot (ets)
ALGORITHM:
   1.  et_i ← ets(IR, ce_j, exe_{i,j}, res_{i,j})
   2.  Find all preceding nodes pre_i of n_i from G
   3.  for all preceding nodes p_k in pre_i do
   4.     From IR, retrieve compute engine ce_k where p_k is mapped and it's
          completion time ct_k.
   5.     List all available routes L_R in PA from ce_k to ce_j.
   6.     for route r in L_R do
   7.        Calculate eft_r ← max(ct_k, est_r) + comm_{k,j,r}
   8.     end for
   9.     Select the route r_MIN with minimum eft_r as eft_{r,MIN}.
   10. end for
   11. From all the predecessor's eft_{r,MIN} find the maximum, which is the
       start time (st_i) of n_i.
   12. Create a set a routes R_MIN for all the predecessors.
   13. est_i ← max(st_i, et_i)
   14. eft_i ← est_i + exe_{i,j}
   15. return R_MIN, eft_i, est_i
```

Figure 6.11: $REFT_{COMM}$: Earliest start time algorithm that considers the specialised communication topology.

Shared communication link

Since actors can execute in parallel on a compute engine, their need to communicate with other actors mapped on different compute engines might coincide. Communication links, until now were treated to be always available without any waiting. However, according to the definition of a communication link in the parametrised platform graph (PPG), they are restricted to allow the communication of only one channel (actor-to-actor communication) at a time. This implies that if a communication link is already occupied, then new channels will be queued. Therefore, the communication delay of a channel is the sum of the data transfer time and the wait time due to the previous channels. Data transfer time is calculated using the token size of the channel and communication link bandwidth. Both of these values, the token size and the bandwidth of communication links are obtained from the application dataflow model and the platform architecture (PPG tier2), respectively.

Since communication links now have a wait time, the communication delays of

an actor's input channels need to be calculated for its mapping. This was not necessary previously, as the communication delay of a channel on a communication link was always same. The earliest start time of a channel on a communication link is calculated in the same way EST of an actor is done in the original HEFT, which is by finding the earliest time slot available on a processor. Once the start time of a channels is it is added with the data transfer time to compute the communication delay.

The start time of an actor on a compute engine is dependent on the communication delays of its input channels. The channel with the maximum communication delay determines the start time of the actor. This leads to its earliest finish time on the compute engine, which is then used to select the compute engine for the mapping of the actor. This shows that a faster compute engine with a lot of resources may not be chosen due it its slow or already queued communication links. The algorithm to calculate the earliest finish time considering delays in a shared-communication link ($REFT_{COMM}$) is shown in figure 6.11. Multiple routes between two compute engines may be available in the connectivity topology of the platform architecture. The route with the least communication delay is used for $REFT_{COMM}$. The fastest route may not be chosen, as it may have channels already queued. After a compute engine is selected, the actor's input links are mapped on the communication links with minimum delay.

Example Ex5 shown in figure 6.12 demonstrates the wait time on communication links and the selection of routes. The computation time on a compute engine and the resource utilisation is the same as in Ex4 in Figure 6.9. The Gantt chart in the Figure shows the channel allocation on the communication links and their queuing. *L3* is the direct connection between the *FPGA* and *GPU*. It is chosen for the channel *DC* instead of the communication links *L1* and *L2* that goes via the *CPU*.

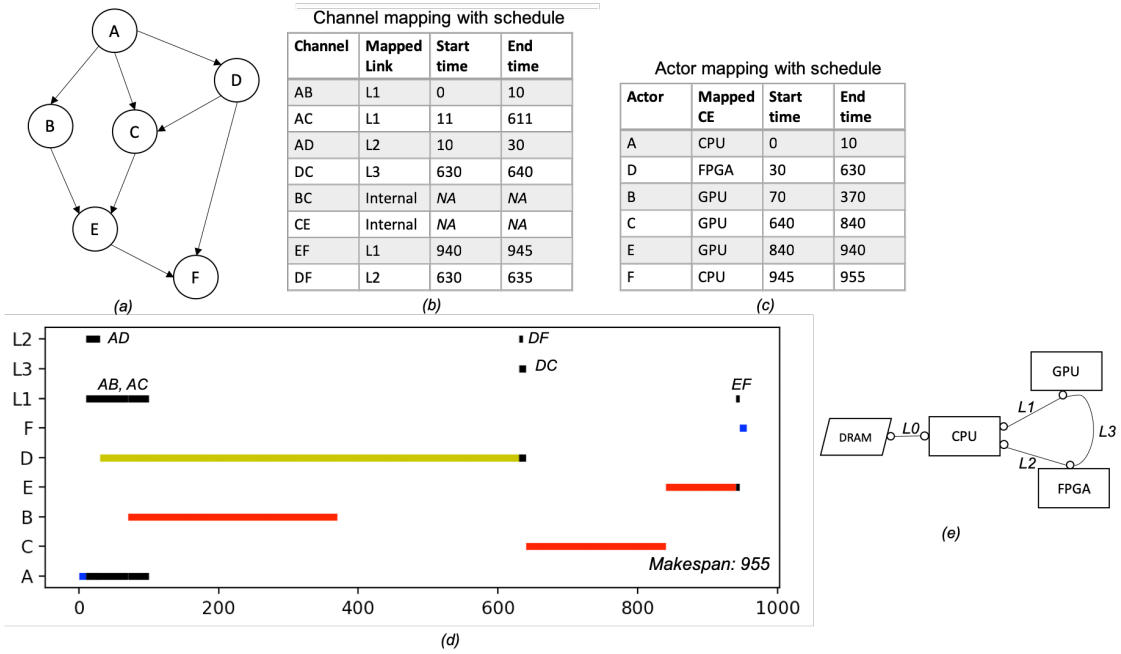


Figure 6.12: Ex5: This example shows the waiting on a communication links and selection of a route that has with lower communication delay. The connectivity topology consists of three links; *L1*, *L2* and *L3*. The compute engines *GPU* and *FPGA* can are connected through *L1* and *L2* as indirect links and also via *L3*. The channel allocation on the communication links are shown as black lines in the Gantt chart.

Mapping and scheduling decision impact

The main implications of generalising the communication links by allowing routes and link sharing is that the delays due to the output channels of an actor can become unpredictable. Although the output channels are considered by the preceding actors to which they are input channels, it is sometimes not sufficient. This can be explained better with an example (Ex6) in figure 6.13. A simple DAG with four actors *A*, *B*, *C* and *D* is used in this example. There are two platform architectures. The first has one of each compute engine kind and the communication links are through *CPU* (no direct connections). The other has one *CPU* and a *GPU*. The performances of actor *A* and *D* are better on the *CPU*, *B* on the *GPU* and *C* on the *FPGA*. In the first platform architecture, after *A* is mapped on *CPU*, *B* is then mapped on *GPU*, next *C* on *FPGA* and finally *D* on the *CPU*. This results to a longer makespan than on the smaller platform architecture. The reason is that the output channel from *B* to *D* was not taken into account while mapping it. Since *B* has a channel *BD* that takes a very long time through the PCIe link, the makespan is increased.

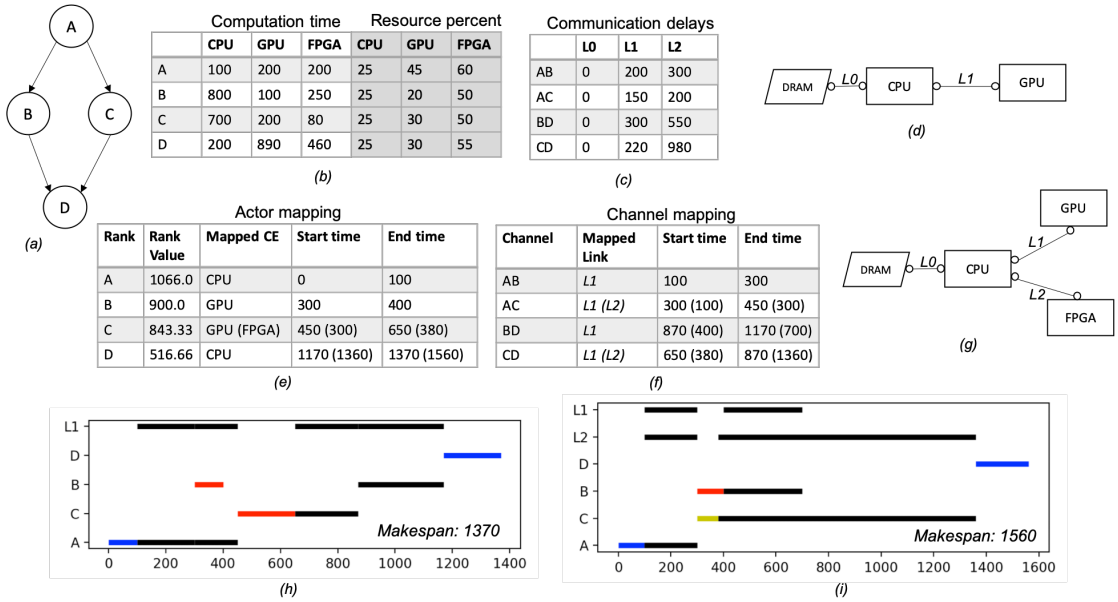


Figure 6.13: Ex6: This example shows that a larger platform can lead to a higher makespan than a smaller platform, due to unpredicted output channel delays.

This example shows that more costly architectures are not necessarily faster. If the expansion had proceeded rapidly it might have jumped from one CPU to one CPU plus GPU and FPGA. without first considering a single GPU. Thus this example supports the arguments that there should be gradual expansion in the architecture. A similar situation is possible under the reduction scenario. So, gradual reduction is also recommended. This concept of gradually changing the platform architecture is applied in the next two subsection for the construction of local platform architecture expansion (LPAE) and global platform architecture update (GPAU).

6.2.4 Local platform architecture expansion

The local platform architecture expansion (LPAE) module of the AMS algorithm, introduced in the overview 6.2.1, is presented in this subsection. As described in the overview, this module is a synthesis of the enhanced versions of HEFT (rHEFT-2), explained in the previous two subsections and a modified version of the Gain/Loss algorithm [8]. Since the original Gain/Loss algorithm was created for rental cost, it is modified to suit the conditions of capital cost for agile heterogeneous computing. The synthesis of rHEFT-2 with the modified Gain/Loss algorithm enables the expansion of the platform architecture in conjunction with the mapping and scheduling decisions of the application-algorithm. As each actor is mapped, the benefits of a new compute engine is evaluated. Based on this evaluation, new compute engines are added to expand the platform architecture instance. Thus, the result of the LPAE module is a possibly expanded platform architecture instance

and an initial-ArcSDF representation containing the mapping and scheduling decisions. These results are used by the GPAU that eventually controls the overall execution of the AMS algorithm. This subsection is divided into three parts. In the first part, the synthesis of rHEFT with the Gain/Loss algorithm is described. Then, in the second part, the pseudocode of LPAE is presented. Finally, in the third part, LPAE is demonstrated with an example.

rHEFT-2 synthesis with the Gain/Loss algorithm

The original Gain/Loss algorithm is based on the idea to gradually increase the rental cost by placing actors on compute engines where it can lower its computation time as compared to what is currently mapped. The actor with the highest computation gain with minimum cost increase is mapped first. This is continued for all the actors until the budget limit is reached. This idea however, cannot be applied for agile platforms, as using a compute engine for a long length of time increases the capital cost to the same extent, as if it was used for a smaller timespan.

The gain idea is realised in LPAE by adding a new compute engine into the platform instance with every actor mapping within rHEFT-2. This is done if the addition has the potential to locally reduce the present actor's computation time as well as globally reduce computation time of all the other un-mapped actors. Recall that in rHEFT-2 actors are first ranked and then they are selected one by one for mapping on the existing platform instance. However, if adding a new compute engine reduces the computation time of the current actor and also has the potential to reduce the computation timings of other un-mapped actors, then it can be added to the existing platform instance. In the next paragraph the effect of adding another compute engine is evaluated. The evaluation involves three hyper-parameters (ce_{local}, ce_{global} and ce_{comm}) listed previously in table 6.1.

The selection of the new compute engine during every actor mapping in rHEFT-2 is achieved through a ranked list of compute engines. Unlike the actor rank in rHEFT-2, the compute engine rank is dynamic. It is updated with the growth of the platform architecture every-time a new actor is selected for mapping. The ranked list of compute engines is called the *dynamic compute engine rank* (DCER). The compute engines considered within DCER are all the ones that are possible given the platform constraints (PPG tier1). If the compute engine with the highest non-zero rank value in DCER can improve the performance of the actor to be mapped in rHEFT-2, then it is selected for the expansion of the platform instance.

The DCER is created by adding three different factors for a compute engine and the actor to be considered for mapping. These three factors are; local reduction time (*local*), global reduction time (*global*) and communication impediment time (*comm*). The *comm* is a negative value that reflects the extra communication time due to the addition of an extra compute engine. *local* is the difference between the

earliest finish time (EFT) on the existing platform architecture instance (PI) minus the EFT once the new CE is added to the platform. *global* is the sum of all possible accelerations of the unmapped actors after the addition of the new compute engine. *comm* is the average delay of all the communication links connecting the new compute engine. Each of these factors are multiplied by the three different hyper-parameters, respectively, to determine the weight of each factor for the *DCER* value.

The DCER value of a compute engine CE_k and actor A_i is defined as follows.

$$local_{i,k} = EFT_{i,pi} - EFT_{i,k} \tag{6.1}$$

$$global_k = \sum_{j \in unmapped} \frac{EXE_{i,cpu}}{EXE_{i,k}} \tag{6.2}$$

$$comm_k = \frac{\sum \bar{link}_k}{|link_k|} \tag{6.3}$$

$$dcer_k = local_k * ce_{local} + global_k * ce_{global} - comm_k * ce_{comm} \tag{6.4}$$

Where $EFT_{i,pi}$ is the EFT of A_i on the PI, $EFT_{i,k}$ is the EFT of A_i on the new CE_k, *unmapped* refers to all the actors that are not yet mapped, $EXE_{i,cpu}$ is the execution time of A_i on the CPU, similarly $EXE_{i,k}$ is the execution time of A_i on CE_k, $link_k$ is the average communication delays on a link connected to CE_k and ce_{local}, ce_{global} and ce_{comm} are the hyper-parameters to determined the weights of local, global and communication factors. The LPAE pseudocode presented in the next part shows the usage of DCER in expanding the platform architecture instance.

LPAE pseudocode

The LPAE pseudocode is presented in three Figures 6.14, 6.15 and 6.16. The first Figure 6.14 shows the overall LPAE algorithm. It calls the $allocate_{rheft}$ (see Figure 6.15)function twice to map the actor A that is to be mapped. The first call is to calculate the EFT of A on the existing platform architecture instance (PI). The EFT call is necessary to evaluate whether another compute engine is required to be added in the PI. The function $select_{ce}$ (see Figure 6.16) is called to evaluate the remaining compute engines in PPG tier1 and then based on the evaluation, it may select from the remaining compute engines. After this evaluation and a possible expansion of the PI, $allocate_{rheft}$ is called again (the second call) to actually map the actor. The pseudocode further details the steps of updating the intermediate representation (IR), which is an initial-ArcSDF instance to capture the mapping and scheduling decisions required in the underlying rHEFT algorithm.

The connectivity topology formed while adding new compute engines favours direct connection links for better performance. When a new compute engine is added to the platform architecture instance, it is first checked that if there are available PCIe lanes to connect a new FPGA or GPU with a CPU. *Docks*[3] are used to accommodate more compute engines when PCIe lanes are occupied. A similar type of compute engine that was already connected is chosen to share a switch with the new compute engine. Also, direct connections are added whenever possible.

[3]Docks are defined in the parametrised platform graph (PPG) it refers to the sharing of a communication link with another compute engine, which can be similar to a switch.

The local platform architecture expansion (LPAE) algorithm

REQUIRED:

a. application DAG *G*,
b. platform architecture instance *pi*,
c. actor execution time matrix *EXE*,
d. actor resource usage matrix *RES*,
e. communication delay matrix *COMM*,
f. resource-based upward ranking algorithm $rank_{up,res}$,
g. communication and resource-based earliest finish time ($REFT_{COMM}$) allocation algorithm

ALGORITHM:

1. Set *rf* of *hapr* to $rank_{up,res}$.
2. Set *mt* of *hpar* to $allocate_{rheft}$.
3. Set ce_{local}, ce_{global} and ce_{comm} of *hpar* to $select_{ce}$.
4. Calculate $rank_{up,res}$ for all nodes (actors) by traversing *G* upward, starting from the exit node.
5. Sort the nodes in a list *L* by non-increasing order of $rank_{up,res}$ values.
6. Initialise *ir* an initial-ArcSDF model with zero compute zones for every compute engines in *pi*.
7. **while** there are unscheduled nodes in *L* **do**
8. Select the first node n_i in *L* and remove it.
9. Predict earliest finish time on the existing platform instance, $eft_p \leftarrow allocate_{rheft}(n_i, pi, ir, map=False)$.
10. Select new compute engine, $ce_{new} \leftarrow select_{ce}(n_i, eft_p)$.
11. **if** ce_{new} is not *None* **then**
12. Expand *pi* with ce_{new}.
13. **endif**
14. Update *ir* to incorporate ce_{new}.
15. $allocate_{rheft}(n_i, pi, ir, map=True)$.
16. **return** *ir*, *pi*

Figure 6.14: The overall algorithm of local platform architecture expansion (LPAE). rHEFT-2 is embedded in this algorithm. This algorithm requires the $allocate_{rheft}$ algorithm, shown in figure 6.15 to map and schedule actors and $select_{ce}$, shown in figure 6.16 to select compute engines for expansion of the initial platform architecture.

$allocate_{rheft}$: Selection of compute engine for the allocation of an actor. This algorithm is based on rHEFT-2.

REQUIRED:

a. application DAG G,
b. platform architecture PA,
c. actor execution time matrix EXE,
d. actor resource usage matrix RES,
e. communication delay matrix $COMM$,
f. resource-based upward ranking algorithm $rank_{up,res}$,
g. communication and resource-based earliest finish time ($REFT_{COMM}$) allocation algorithm

ALGORITHM:

1. **for** each compute engine ce_k **do**
2. $R_{MIN,k}$, $eft_{i,k}$, $est_{i,k} \leftarrow REFT_{COMM}(n_i, ce_k, pi, EX, RES, COMM, IR, G)$
3. **end for**
4. From all the previously calculated eft, find eft_{min} the minimum eft.
5. **for** each compute engine ce_k **do**
6. $diff_k \leftarrow eft_k - eft_{min}$
7. **if** $diff_k <= mt$ **then**
8. $rpv_k \leftarrow (eft_{CPU} - eft_k)/res_{i,k}$
9. **else**
10. $rpv_k \leftarrow -1.0$
11. **end if**
12. **end for**
13. **if not** map then
14. Select ce_j that maximises the rpv value of n_i.
15. **return** $eft_{i,j}$
16. **else**
17. Assign n_i to ce_j that maximises the rpv value of n_i.
18. Assign input channels of n_i to routes in set $R_{MIN,j}$.
19. Update IR to record $est_{i,j}$ and $eft_{i,j}$
20. **endif**

Figure 6.15: The algorithm to select a compute engine for mapping and scheduling of an actor. This is based on the rHEFT algorithm.

$select_{ce}$: The algorithm to select a compute engine based on dynamic compute engine ranking (DCER).

REQUIRED:

a. application DAG *G*,
b. platform architecture instance *pi*,
c. actor execution time matrix *EXE*,
d. actor resource usage matrix *RES*,
e. communication delay matrix *COMM*,
f. resource-based upward ranking algorithm $rank_{up,res}$,
g. communication and resource-based earliest finish time ($REFT_{COMM}$) allocation algorithm

ALGORITHM:

1. Create a list of unused compute engines CE_{un} from *tier1*.
2. Create a list of actors that are not mapped, A_{un}
3. **for** each compute engine ce_k in CE_{un} **do**
4. $global_k \leftarrow \sum(exe_{j,cpu}/exe_{j,k})$, where *j* is an unmapped actor in A_{un}.
5. $local_k \leftarrow eft_p - exe_{i,k}$
6. **if** $locak_k <= 0$ **then**
7. $local_k \leftarrow 0$
8. **endif**
9. $comm_k \leftarrow$ average comm delays
10. $dcer_k \leftarrow global_k * ce_{global} + local_k * ce_{local} - comm_k * ce_{comm}$
11. **end for**
12. Select ce_{high} with highest *dcer*, within the budget *b* and its local rank, *local* is not zero.
13. **if** rank of $ce_{high} <= 0$ **then**
14. $ce_{high} \leftarrow None$
15. **endif**
16. **return** ce_{high}

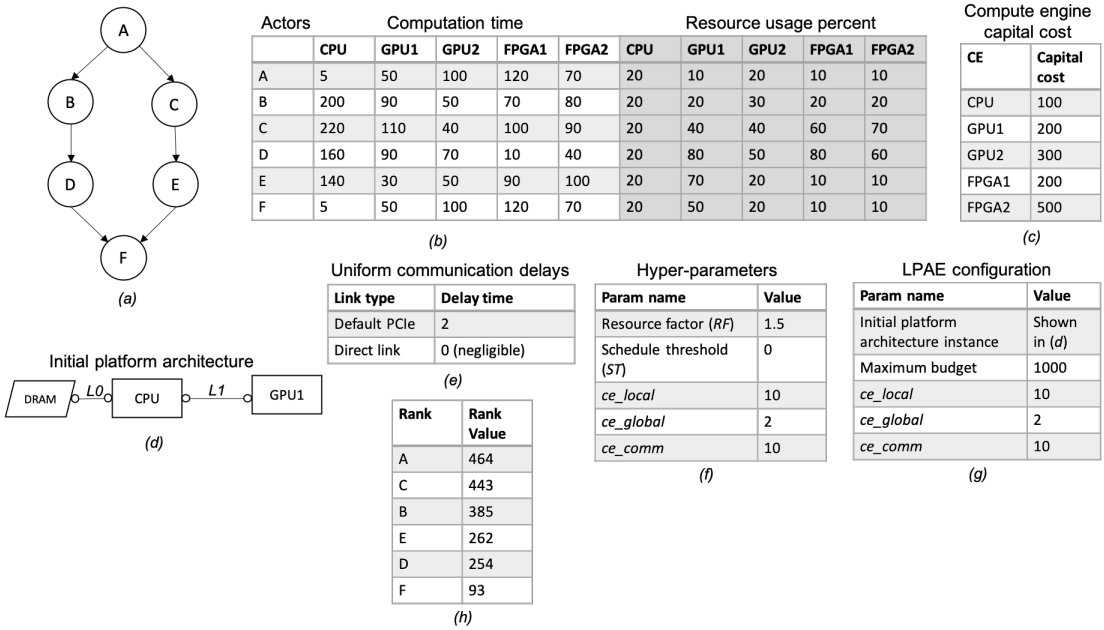
Figure 6.16: The algorithm selects a compute engine for expansion of the platform architecture.

Example

LPAE is demonstrated through an example (Ex7) shown in Figures 6.17, 6.18 and 6.19. This example shows how compute engines are added in an existing platform architecture instance (PI) within the rHEFT algorithm. The PI consists of one CPU and one GPU, however there are one more GPU and two FPGAs available in the PPG tier1. The application DAG consists of 6 actors. Figure 6.17 shows all the inputs necessary for LPAE and the hyper-parameters values. It also contains the actor rank values. Based on the rank values, actors are considered for mapping in the following order: *A, C, B, D, E and F.*

All the execution steps of LPAE are illustrated in Figure 6.18. The budget for the expansion is $1000 that includes the cost of the initial platform architecture. This leaves $700 for the addition of new compute engines. The first actor to be considered for mapping is A, which is based on the actor rank. In *step 1*, the DCER values of the remaining compute engines are calculated. Since A has the least EFT on CPU, already a part of PI, the *local* values are zero. This prevents from any additional compute engine from being selected. The next step maps actor C. Since the EFT of C is lower on a compute engine not part of the PI, the *local* values are positive and so are the DCER values. The compute engine with the highest DCER value is selected, which is GPU2 and it is added to the PI. Figure 6.19(b) shows the expanded platform architecture. Since the connectivity topology is rule based, it automatically adds direct connections whenever possible. It is interesting to note that before the addition of GPU2 the EFT of C was 117 and after it is added, the EFT reduces to 47.

Theses steps are continued for all the actors. No expansion occurs, until *step 5*, where FPGA1 is added for actor D. The final expanded platform architecture is shown in Figure 6.19(c). The mapping and scheduling decisions are shown as a Gantt chart in Figure 6.19(d). It also shows the makespan on the final platform architecture. The communication delays of 2 time units are shown as d blocks.



	CPU	GPU1	GPU2	FPGA1	FPGA2	CPU	GPU1	GPU2	FPGA1	FPGA2
A	5	50	100	120	70	20	10	20	10	10
B	200	90	50	70	80	20	20	30	20	20
C	220	110	40	100	90	20	40	40	60	70
D	160	90	70	10	40	20	80	50	80	60
E	140	30	50	90	100	20	70	20	10	10
F	5	50	100	120	70	20	50	20	10	10

CE	Capital cost
CPU	100
GPU1	200
GPU2	300
FPGA1	200
FPGA2	500

Link type	Delay time
Default PCIe	2
Direct link	0 (negligible)

Param name	Value
Resource factor (*RF*)	1.5
Schedule threshold (*ST*)	0
ce_local	10
ce_global	2
ce_comm	10

Param name	Value
Initial platform architecture instance	Shown in (*d*)
Maximum budget	1000
ce_local	10
ce_global	2
ce_comm	10

Rank	Rank Value
A	464
C	443
B	385
E	262
D	254
F	93

Figure 6.17: Ex7: An example illustrating LPAE. The configurations are shown in this figure.

Actor to be mapped: A			
Estimated EFT = 5 on CPU			
	GPU2	FPGA1	FPGA2
Global	15.6 x 2	22.87 x 2	10.66 x 2
Local	0 x 10	0	0
Comm	4 x 10	4 x 10	4 x 10
DCER	**0**	**0**	**0**
Compute engine selected: NONE **Remaining budget = 700**			
Finally mapped on CPU and EFT = 5			

Step 1

Actor to be mapped: C			
Estimated EFT = 5 + 2 + 110 = 117 on GPU1			
	GPU2	FPGA1	FPGA2
Global	15.55 x 2	22.82 x 2	10.61 x 2
Local	60 x 10	10 x 10	20 x 10
Comm	4 x 10	4 x 10	4 x 10
DCER	**671.1**	**185.64**	**261.22**
Compute engine selected: GPU2 **Remaining budget = 400**			
Finally mapped on GPU2 and EFT = 5 + 2 + 40 = 47			

Step 2

Actor to be mapped: B		
Estimated EFT = 5 + 4 + 50 = 59 on GPU2		
	FPGA1	FPGA2
Global	20.63 x 2	8.17 x 2
Local	0 x 10	0 x 10
Comm	4 x 10	4 x 10
DCER	**0**	**0**
Compute engine selected: None **Remaining budget = 400**		
Finally mapped on GPU2 and EFT = 59		

Step 3

Actor to be mapped: E		
Estimated EFT = 47 + 0 + 30 = 77 on GPU1		
	FPGA1	FPGA2
Global	17.78 x 2	5.65 x 2
Local	0 x 10	0 x 10
Comm	4 x 10	4 x 10
DCER	**0**	**0**
Compute engine selected: None **Remaining budget = 400**		
Finally mapped on GPU1 and EFT = 77		

Step 4

Actor to be mapped: D		
Estimated EFT = 59 + 0 + 70 = 129 on GPU2		
	FPGA1	FPGA2
Global	16.24 x 2	4.1 x 2
Local	56 x 10	26 x 10
Comm	4 x 10	4 x 10
DCER	**632.48**	**308.2**
Compute engine selected: FPGA1 **Remaining budget = 200**		
Finally mapped on FPGA1 and EFT = 59 + 4 + 10 = 73		

Step 5

Actor to be mapped: F	
Estimated EFT = 77 + 2 + 5 = 84 on CPU	
	FPGA2
Global	0 x 2
Local	0 x 10
Comm	4 x 10
DCER	**0**
Compute engine selected: None **Remaining budget = 200**	
Finally mapped on CPU and EFT = 84	

Step 6

Figure 6.18: Ex7: Six steps of the example in Figure 6.17 to select compute engines for platform architecture expansion.

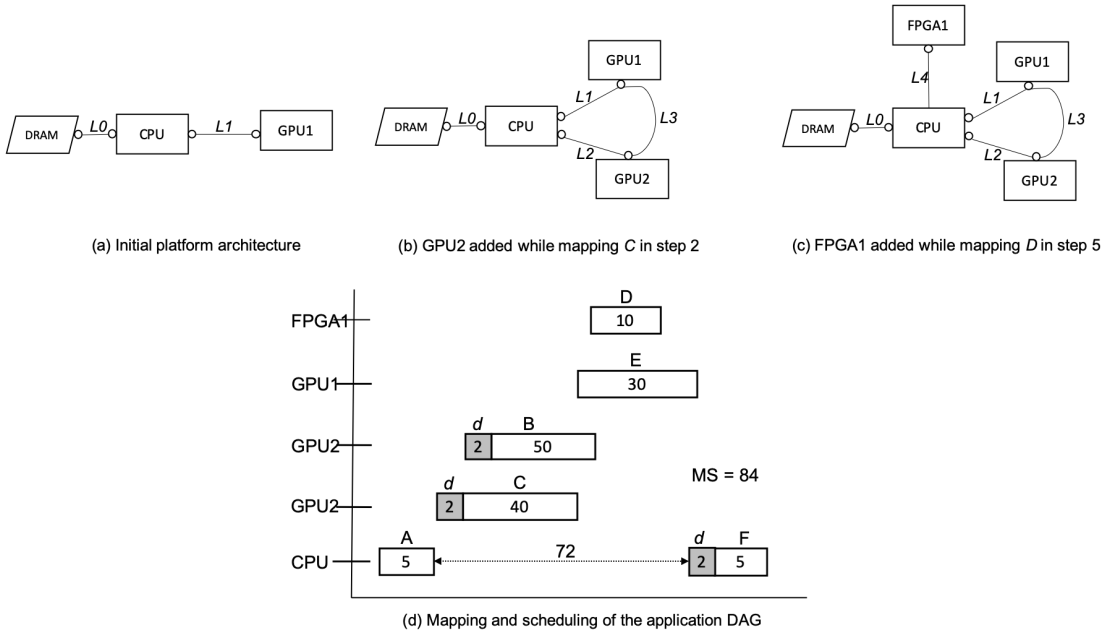


Figure 6.19: Ex7: The expansion of the platform architecture from its initial architecture for the example in Figure 6.17. The resulting mapping and scheduling decisions are also shown as a Gantt chart.

6.2.5 Global platform architecture update

This subsection presents the global platform architecture update (GPAU) module that was previously introduced in the AMS algorithm overview (see section 6.2.1). GPAU controls the overall execution of the AMS algorithm by initially setting the mode of design space exploration to either expansion or reduction and by introducing randomness to restart exploration from a new design point when limits are reached for the expansion or reduction of the earlier platform architecture instances. This subsection is organised into three parts. The first part presents pseudocode of the overall AMS algorithm and the GPAU module to explain how GPAU controls the execution of AMS. The second part first describes the expansion mode of GPAU and then demonstrates it with an example. The third part details the reduction mode of exploration, along with an example.

AMS and GPAU pseudocode

The AMS algorithm pseudocode is presented in Figure 6.20. This pseudocode calls both GPAU and LPAE. It shows that the GPAU and LPAE are called in a loop that iterates until the number of "random design point changes" equals to the maximum random iteration itr_{MAX}. In every iteration, GPAU is executed followed by LPAE and their outcomes are stored inside the exploration history (EH). GPAU provides the budget b, the platform architecture instance PI to start the iteration, hyper-parameters $hpar$ and a boolean value for random update br. Amongst these values, pi, $hpar$ and b are used by LPAE, which return a possibly expanded platform architecture xpi and the intermediate representation ir. br is used to identify whether GPAU has changed the exploration path randomly. This value is used to track the number of random changes to end the loop. After reaching itr_{MAX} the record with the lowest makespan (r_{best}) is retrieved from EH. r_{best} contains the IR (initial-ArcSDF model) and the platform architecture (PPG tier2) to be used for deployment.

The GPAU pseudocode shown in Figure 6.21 consists of an initialisation phase and two exploration modes; expansion and reduction. During initialisation, the mode of exploration is decided based on the initial platform architecture's (pi_0) capital cost and the maximum budget (b_{max}) allocated for exploration. The exploration mode is set to *expansion* if the capital cost of pi_0 is less than b_{max}, so that there is extra budget to expand the platform architecture. If b_{max} is less than the capital cost of pi_0, then reduction mode is set to remove compute engines to lower the capital cost. It is noted that each mode has two stages; deterministic and random. When the steps inside the deterministic mode are exhausted, the exploration shifts to a random design point, which restarts the deterministic steps. *limit_reached* is the function that decides whether the limits have been reached for the deterministic steps. The pseudocode of *limit_reached* is described in Figure 6.23. If the limits are reached then, the function *random_point* is called to find the random design

Structure of the agile mapping and scheduling (AMS) algorithm

REQUIRED:

a. Global platform architecture update *(gpau)* algorithm
b. Local platform architecture expansion *(lpae)* algorithm
c. Exploration history storage *(EH)* and
d. Maximum random iteration, itr_{MAX}

ALGORITHM:

1. Initialise *lpae*, *gpau*, *eh* and $itr_{rnd} \leftarrow 0$
2. **while** $itr_{rnd} < itr_{MAX}$ **do**
3. Execute $gpau\,(eh)$ for budget *b*, platform instance *pi*, hyper-parameters *hpar* and a boolean value for random update *br*, written as $(b,\ pi,\ hpar,\ br) \leftarrow gpau\,(eh)$.
4. **if** *br* is true **then** increment itr_{rnd} by 1.
5. Execute $lpae\,(b,pi,hpar)$ for initial-ArcSDF *ir* and a possibly expanded platform instance *xpi*, written as $(ir,xpi) \leftarrow lpae\,(b,pi,hpar)$.
6. Record $(xpi,\ hpar,\ ir,\ br)$ in *EH*.
7. end while
8. Find the record r_{best} in *EH* with the lowest makespan.
9. Retrieve platform architecture PA_{best} and initial-ArcSDF IR_{best} from r_{best}.
10. Run compute zone optimisation routine to form a complete ArcSDF model, IR^c_{best} from IR_{best}
11. **return** IR^c_{best}, PA_{best}.

Figure 6.20: The high-level pseudocode of the AMS algorithm.

point. The pseudocode of *random_point* is described in Figure 6.22. Details of how these functions operate during expansion and reduction are described in the next two parts of the subsection.

Global platform architecture update (GPAU) algorithm

REQUIRED:

a. Maximum random iteration, itr_{MAX}
b. $(ce_{local,0}, ce_{global,0}, ce_{comm,0}, rf_0, mt_0)$
c. platform architecture constraints tier1,
d. Maximum budget b_{MAX}
e. *check_limit*
f. *random_point*

INITIALISE:

1. **if** capital cost of pi_0 < b_{MAX} **then**
2. $mode \leftarrow expansion$
3. expansion budget, $b_x \leftarrow$ capital cost of pi_0
4. **else**
5. $mode \leftarrow reduction$
6. expansion budget, $b_x \leftarrow 1$
7. **endif**
8. create a list of hyper-parameters, *hpar* with the initial values. $hpar \leftarrow list(ce_{local,0}, ce_{global,0}, ce_{comm,0}, rf_0, mt_0)$

ALGORITHM:

1. $limit\_reached \leftarrow check\_limit(eh, mode, tier1, b_{MAX})$
2. **if** *limit_reached* **then**
3. $pi, hpar \leftarrow random\_point(eh, mode, tier1, b_{MAX})$
4. $rnd \leftarrow True$
5. **if** mode is expansion **then**
6. $b_x \leftarrow$ capital cost of *pi*
7. **endif**
8. **else**
9. $rnd \leftarrow False$
10. **if** mode is expansion **then**
11. increment b_x by the minimum capital cost of the unused compute engines in tier1.
12. **else**
13. from the last record in *eh* find the least used compute engine ce_{least}.
14. reduce *pi* by removing ce_{least}.
15. **endif**
16. **endif**
17. **return** $(b_x, pi, hpar, rnd)$

Figure 6.21: Pseudocode of the global platform architecture update (GPAU) algorithm.

Random design point(random_point) update algorithm

REQUIRED:

a. Maximum random iteration, itr_{MAX}
b. $(ce_{local,0}, ce_{global,0}, ce_{comm,0}, rf_0, mt_0)$
c. platform architecture constraints tier1,
d. Maximum budget b_{MAX}
e. *check_limit*
f. *random_point*

ALGORITHM:

1. From *eh* select the record r_{best} with least makespan.
2. Create a list of compute engines ce_{best} that are used in the platform architecture of r_{best}.
3. Create a list of unused compute engine ce_{un} from the compute engines in *tier1* by subtracting ce_{best}.
4. **if** mode is expansion **then**
5. Randomly choose compute engines from ce_{best} and ce_{un} to create a platform architecture instance *pi* with the capital *cost* < b_{MAX}.
6. **else**
7. Randomly choose compute engines from ce_{best} and ce_{un} to create a platform architecture instance *pi* with the capital *cost* >= b_{MAX}.
8. **endif**
9. Generate random values with the minimum and maximum limits of all the five individual hyperparameters within *hpar*.
10. **return** pi, hpar

Figure 6.22: Pseudocode of the *random_point* function.

Design space limit checking (limit_reached) algorithm

REQUIRED:

a. Maximum random iteration, itr_{MAX}
b. $(ce_{local,0}, ce_{global,0}, ce_{comm,0}, rf_0, mt_0)$
c. platform architecture constraints tier1,
d. Maximum budget b_{MAX}
e. *check_limit*
f. *random_point*

ALGORITHM:

```
1.  if mode is expansion then
2.    if capital cost of pi >= b_MAX or
3.              pi cannot have more compute engines then
4.      limit_reached ← True
5.    else
6.      limit_reached ← False
7.    endif
8.  else
9.    if pi consists of only a CPU from tier1 then
10.     limit_reached ← True
11.   else
12.     limit_reached ← False
13.   endif
14. endif
15. return limit_reached
```

Figure 6.23: Pseudocode of the *limit_ reached* function.

Expansion mode

Although the platform architecture expansion takes place in LPAE, the expansion budget, the starting platform architecture and the hyper-parameters are decided in GPAU. The expansion budget b is gradually increased, so that during expansion in LPAE, the starting platform architecture is also gradually expanded. Gradual expansion ensures that smaller architectures and lower capital cost compute engines are considered even when the maximum budget b_{max} is high. The importance of smaller architectures was discussed in section 6.2.3, where it was shown that large platform architectures can sometimes incur communication penalty than its smaller counterpart.

The expansion mode starts by calling the *limit_ reached* function (see Figure 6.23) to check whether the limits of platform expansion has been reached. There are three limits to be checked; (1) the expansion budget b equals to the maximum budget b_{max}, or (2) PPG tier1 has no more compute engines remaining, (3) the available links between compute engines are exhausted. The second limit is due to the connectivity constraints of PPG tier1. If any of these three limits are reached, the exploration shifts to a new design point by calling the *random_ point* function.

In the *random_ point* function, a random starting platform architecture instance *pi* is created and along with it, a random set of hyper-parameters within the minimum and maximum range provided by the designer are also generated. While randomly creating *pi*, it is ensured that it's capital cost is much lower than the maximum budget b_{max}. This allows more expansion to take place in the LPAE. The generation of a new random design point works as follows. The previous history of the architecture is looked through to find the one with the lowest makespan. Then the compute engines that form the lowest makespan are placed into the selection pool. The history of the other architectures is also looked through to create a list of compute engines that were never used before. These are added to the selection pool. Finally, a random selection from the pool of compute engines is made to create the random design starting point. For each possible random choice from the pool the budget is checked and only those choices that meet the budget are kept.

When the expansion mode of exploration is within the limits, the deterministic steps consist of increasing the expansion budget b gradually. The amount of budget to be increased is a random value between the lowest cost compute engine left in PPG tier1 and b_{max}. The starting platform architecture instance (pi_0) is not changed in these deterministic steps, only b is increased. This ensures that in each iteration the LPAE expands platform architecture with increasing budget, so that smaller and larger platform architectures are properly explored. The results from the LPAE are the possibly expanded platform architecture, *xpi* and the intermediate representation *IR* containing the mapping and scheduling decisions are recorded in the exploration history along with the hyper-parameters *hpar*. This iteration is continued in the AMS algorithm for itr_{max} times.

The expansion mode of exploration is demonstrated through an example (Ex8) that is based on inputs previously used to illustrate the LPAE algorithm (see Figure 6.17 (Ex7). Unlike the previous example, the initial platform architecture instance (pi_0) and the set of hyper-parameters *hpar* are controlled by GPAU. The result of running GPAU for $itr_{max} = 3$ and $b_{max} = 850$ is shown in table 6.2. The first (pi_0) consists of only one CPU and the *hpar* values are shown in step 0. These values are provided by the user. It is noted that in the 0^{th} step, the expansion budget is same as the (pi_0) capital cost, so that expansion does not take place. Since (pi_0) consists of only one CPU, all the actors are mapped on the CPU, where the makespan is 370. In the second step the (pi_1) is expanded allowing the inclusion of FPGA1 and the makespan reduced to 204. In the next step, (the 2nd step) three compute engines (FPGA1, GPU1, GPU2) are added and the makespan reduces to 90. It is interesting to note that in the previous example (Ex6) this (pi_2) resulted in the lowest makespan but due to different hyper-parameter values, the mapping and scheduling decisions are not the same. The design points are reset in steps 3 and 5. These two corresponding rows are marked as bold. In step 4, the lowest makespan is achieved.

Table 6.2: Ex7: GPAU expansion example.

	GPAU				**LPAE**		
Steps	**Budget**	**Platform architecture**	**Hyper-parameters**	**Random bool**	**Expanded platform**	**Platform cost**	**MS**
0	100	CPU1	RF: 2.8, MT: 30, local: 15, global: 5, comm:5	False	CPU1	100	370
1	344	Same as above	Same as above	False	FPGA1, CPU1	300	204
2	872	Same as above	Same as above	False	FPGA1, CPU1, GPU1, GPU2	800	90
3	**300**	**GPU1, CPU1**	**RF: 2.2, MT: 22, local: 19, global: 8, comm:3**	**True**	**GPU1, CPU1**	**300**	**242**
4	850	Same as above	Same as above	False	GPU1, GPU2, FPGA1, CPU1	800	84
5	**300**	**FPGA1, CPU1**	**RF: 3.5, MT: 48, local: 17, global: 9, comm:4**	**True**	**FPGA1, CPU1**	**300**	**204**
6	627	Same as above	Same as above	False	FPGA1, CPU1, GPU2	600	102
7	830	Same as above	Same as above	False	FPGA1, CPU1, GPU2, GPU1	800	86

Reduction mode

The reduction mode of exploration takes place solely in GPAU. LPAE is only for the mapping and scheduling decisions, so that the makespan of every platform architecture instances can be calculated. First step is to find whether the reduction limit has reached, which is when the platform architecture is just left with the CPU. This is checked by calling the *limit_reached* function. After the reduction limit is reached, *random_point* function is called to create a new design point by randomly forming a starting platform architecture instance *pi* and also by randomly creating new values for the hyper-parameters. *pi* is created from a combination of the compute engines compute engines, compute engines from tier1 which are not yet used and compute engines used in the previous iteration. At first, the compute engines with lowest makespan from the previous iteration are preferred, but if the capital cost of this *pi* is below the maximum budget b_{max}, compute engines from the previous iterations are also included. It is noted that although all the hyper-parameters are changed, resource factor (RF) and mapping threshold (MT) are only used in the LPAE, as it defaults to rHEFT-2 during reduction. The others, ce_{local}, ce_{global} and ce_{comm} are not used, as they are applicable only when choosing a new compute engine during expansion.

Up till the time of the one CPU limit is reached, the limit of reduction is not reached, the algorithm is in the deterministic stage, where the platform architecture instance is reduced gradually. The compute engine removed from the architecture is determined as follows. The total compute time of every actors is evaluated. The compute engine where the sum of execution time is the lowest is removed. After removal of the this least used compute engine, LPAE is called to produce the makespan and the intermediate representation containing the mapping and scheduling decisions. The result of LPAE and GPAU are then stored in the exploration history. Removal of compute engines continues until the reduction limit is reached, then a new design point is randomly created. This iteration continues for the maximum iteration count itr_{max}.

The example previously shown in Figure 6.17 (Ex7) is used as an input to the new example Ex8 for the reduction mode of exploration. The initial platform architecture consists of CPU, GPU2, FPGA1 and FPGA2 and its capital cost is 1100. The maximum budget is 850, the same value was used to demonstrate expansion mode. There are three deterministic steps to reduce the *pi* to a CPU. In these three steps, compute engines are reduced one by one. In step three a platform instance is randomly generated based on the previous history as described in section 6.2.1. It can be seen that the makespan keeps to increasing. In step 1, the platform architecture instance is randomly generated. Its capital cost is 1300 and the makespan is 90, which is the lowest so far. However, in the next step after removing FPGA2, the makespan is still 90. This shows that a smaller platform architecture can produce a lower makespan than a larger one. This platform instance architecture is similar to the one where makespan of 84 was attained previously. This is due to different

RF and MT values that changes the mapping and scheduling decisions. Another reset happened in step 8 but none of the subsequent deterministic architectures resulted in a better makespan.

Table 6.3: GPAU reduction example.

	GPAU					LPAE (rHEFT)
Steps	**Budget**	**Platform architecture**	**Cost**	**Hyper-parameters**	**Random bool**	**MS**
0	1	FPGA1, FPGA2, GPU2, CPU1,	1100	RF: 2.8, MT: 30, local: 15, global: 5, comm:5	False	106
1	1	FPGA1, GPU2, CPU1	600	Same as above	False	106
2	1	CPU1	100	Same as above	False	370
3	**1**	**GPU1, GPU2, FPGA1, FPGA2, CPU1**	**1300**	**RF: 3.6, MT: 44, local: 18, global: 2, comm:1**	**True**	**90**
4	1	GPU1, GPU2, FPGA1, CPU1	800	Same as above	False	90
5	1	GPU1, GPU2, CPU1	600	Same as above	False	134
6	1	GPU2, CPU1	400	Same as above	False	127
7	1	CPU1	100	Same as above	False	370
8	**1**	**GPU1, FPAG1, FPGA2, CPU1**	**1000**	**RF: 2.2, MT: 27, local: 20, global: 4, comm:5**	**True**	**138**
9	1	FPAG1, FPGA2, CPU1	800	Same as above	False	200
10	1	FPAG1, CPU1	300	Same as above	False	204
11	1	CPU1	100	Same as above	False	370

6.2.6 Conclusion

The design space exploration algorithm for agile heterogeneous computing, known as agile mapping and scheduling algorithm (AMS) was presented in this section. The creation of AMS was described in stages and how the unique requirements of agile heterogeneous computing were fulfilled was explained and demonstrated with

examples. The unique requirements consist of allowing concurrent actor executions on the same compute engine, catering to the specialised connectivity topology constraints of an agile platform architecture, formation an optimised platform architecture at the same time as considering the mapping and scheduling decisions and capital cost based exploration rather than rental cost.

There are two main stages in which AMS was created. In the first stage, a widely used and efficient list-based scheduling algorithm called HEFT [7] was extended to consider actor resource usage and specialised connectivity topology of an agile heterogeneous computing architecture. This new version of HEFT is called resource-HEFT (rHEFT-2). In the second stage, rHEFT-2 was combined with an adapted version of Gain/Loss algorithm [8] to create the expansion and reduction modes of exploration. Since the original Gain/Loss paper is a rental cost based optimisation on cloud like infrastructures, it was adapted to incorporate capital cost. In expansion mode, a smaller platform architecture is expanded by adding more compute engines. Whereas, in reduction mode, a large platform architecture is reduced by removing compute engine. In both the modes, addition and removal of compute engines are gradual and are influenced by the mapping and scheduling decisions. In the next section, the overall AMS algorithm incorporating rHEFT-2 is evaluated using a range of synthetic and real DAGs.

6.3 AMS algorithm evaluation

The previous section presented the agile mapping and scheduling (AMS) algorithm for the new design flow called AhcFlow that was presented in chapter 4. This section presents an evaluation of the AMS algorithm. Since the AMS algorithm comprises of a new version of HEFT that considers resources (rHEFT), the evaluation first assesses rHEFT with synthetic datasets to study the improvements due to the enhanced resource oriented ranking and allocation algorithms. Then, the evaluation focuses on the overall AMS algorithm, where real published and synthetic DAGs are used to study the formation of the platform architectures. This study compares performance for the expansion and reduction modes of exploration and compares the results with randomly generated platform architectures.

In order to perform the evaluation, a framework and prototype of the AMS algorithm is first developed with facilities to generate synthetic datasets and store the design space exploration results in a database. Then, the framework is used for the evaluation experimentations. Therefore, this section is organised into three subsections. The first subsection describes the prototype of the AMS algorithm. The second subsection evaluates enhanced resource conscious ranking and compute engine selection algorithms with rHEFT-1, which is closest to the original HEFT algorithm. Finally, the third subsection evaluates the overall AMS algorithm by comparing the two modes of exploration with each other and with randomly generated platform architectures.

6.3.1 Evaluation framework

This subsection describes an evaluation framework consisting of a prototype of the AMS algorithm and the functionalities to generate datasets, run experiments and then store the results for interpretation. Figure 6.24 shows the software components of the evaluation framework. It depicts the layered and hierarchical form of the evaluation framework. Dependencies between the components exist as layers, where a high-level component is placed on top of a low-level component. Organisation of the components are hierarchical, where smaller ones are combined to form a larger component. There are three key components; AMS prototype, data generator and evaluation driver. Thus this subsection is divided into three parts to detail each key component separately.

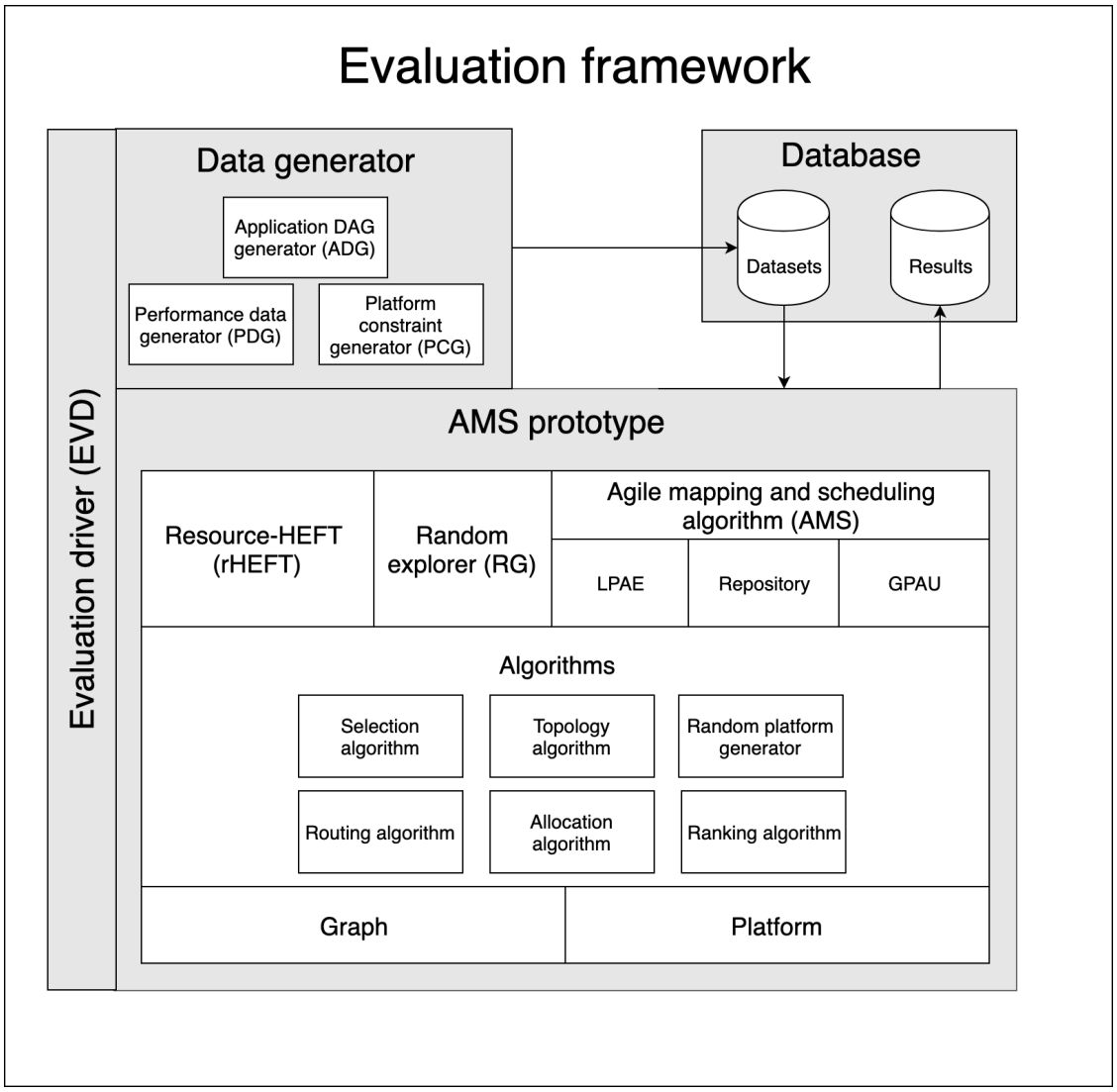


Figure 6.24: The structure of the evaluation framework and the AMS prototype.

AMS prototype

A prototype of the agile mapping and scheduling (AMS) algorithm is developed as a plug-n-play structure, so that different configurations of rHEFT and AMS can be created by just changing their underlying components. For example, to compare the original ranking and allocation algorithm of rHEFT with the enhanced versions, two instances of rHEFT are created but with different ranking and allocation algorithms. In one of the instance, the original algorithm (pseudocode in Figure 6.3) is used, whereas in the other instances, enhanced algorithms (pseudocode in Figures 6.6 and 6.8) are applied.

In order to create this flexibility, the AMS prototype is developed as high-level components for rHEFT-2 and AMS with their fundamental algorithms constructed

from low-level components called *Algorithms*. There are five internal components; *ranking algorithms*, *allocation algorithms*, *selection algorithms* and *routing algorithms*. Each of these internal components have various versions. Therefore, different configurations of rHEFT and AMS can be easily created by choosing distinct combinations of the basic versions of these algorithms.

At the lowest level, there are two components; *graph* and *platform*. The *graph* component is responsible for graph-based operations consisting of SDF to HSDF (DAG) conversion, initial-ArcSDF optimisation routine to generate a complete ArcSDF, ArcSDF to initial-ArcSDF conversion and the implementation of the earliest time slot algorithm (see section 5.3.3). The *platform* component is responsible for the parametrised platform graph (PPG). This component is primarily used to ensure that the formation of platform architecture instances (PPG tier2) conforms to the platform constraints (PPG tier1).

The structure of the evaluation framework in Figure 6.24 shows that the *algorithms* components are directly used by *rHEFT-2* and the Random explorer (*RG*) but, the *AMS* components have another layer containing the *LPAE*, the *GPAU* and the *EH* components. This extra layer is developed mainly for possible future extensions of the LPAE and the GPAU algorithms, so that their later enhancements can be developed without changing the AMS algorithm and the underlying components. These possible future extensions are not part of this thesis.

The three high-level components; *rHEFT*, *RE* and *AMS* are exposed to the outside world. They have application programmers interfaces (API) that are used to create instances of rHEFT and AMS algorithms, which are then used for experimentation. The *RG* component is used to generate random platform architectures from the platform constraints in PPG tier1. The purpose of this component is to compare random platform architectures with the ones generated by the AMS algorithm. In order to establish a fair comparison, a list of N random platform architectures are generated, then the best one is selected. The value of N is decided based on the random iteration count of AMS. The next part describes the generation of datasets used in the evaluation.

Data generator

The data generator produces data for the evaluation experiments. It consists of three internal components; the application DAG generator (*ADG*), platform constraint generator (*PCG*), and the performance data generator (*PDG*). The DAG generator produces synthetic graphs for the application-algorithm. The platform constraint generator produces synthetic instances of the available components for a platform architecture and their constraints in the form of a parametrised platform graph (PPG) tier1 model. The final component, which is the performance data generator produces synthetic computation, communication and resource usage

data for actors executing on various compute engines.

The DAG generator is capable of producing five kinds of DAGs; synthetic random DAGs [7, 104] and four kinds of real DAGs. The published DAGs include Laplace equation solver [101, 125], fast Fourier transformation graph [101], and LU decomposition [101, 125]. These published DAGs are chosen because they are widely used in the literature of heterogeneous mapping and scheduling algorithms [8, 104, 126]. The characteristics of random DAGs depend on various attributes, which can be freely chosen, whereas for the published DAGs, attributes are are pre-determined, except for their size. Table 6.4 lists all the attributes that affect the characteristics of randomly generated DAGs. A large number of DAGs were generated using a combination of these attributes for the evaluation. For every DAG, a set of performance data that consists of: computation, communication and resource usage data is produced by the performance data generator (PDG). Another table 6.5 lists the attributes that affect the value of these data. How performance data is generated based on these parameters are explained as follows.

- Before the performance data generation can start, the compute engine and the communication link details are required. The number, type and unique ids of each compute engines are necessary. Similarly, the types of communication links are required. These details are present in PPG tier1 or tier2. For the experiments with rHEFT, fixed platform architectures were used, so just tier2 instances were used. The tier2 instances were manually created by using the *platform* component in the *AMS* component. The overall AMS algorithm platforms are variable, so tier1 was used. The platform constraint generator (PCG) contains scripts to generate PPG tier1.

- Computation time of an actor on a compute engine is generated relative to the computation time of the actor on the slowest CPU. The computation time of an actor A_i on the slowest CPU is written as $EXE_{i,CPU}$. For every actor, this value is generated as a uniform distribution with the limits of MIN_{CPU} and MAX_{CPU}. Based on the percentage of compute engines that can accelerate, the compute engines are divided into two groups, one that can accelerate and the rest will decelerate. Since the purpose of compute engines are for acceleration, the value of ACC_{PER} is supposed to be high. The computation time on compute engines that will accelerate is a random value within $MAX_{ACC} \times EXE_{i,CPU}$ and $MIN_{ACC} * EXE_{i,CPU}$. Similarly, for the compute engines that will decelerate is within $MIN_{DECC} \times EXE_{i,CPU}$, $MAX_{DECC} \times EXE_{i,CPU}$.

- Communication delays of a channel on a communication link (CL_i) is based on the average computation time and the *computation communication ratio* (CCR). The average compute time AVG_{COMP} of every actors on each compute engines are calculated. It is then divided by (CCR) to find the average communication time, AVG_{COMM}. The AVG_{COMM} is used to produce the communication time of each channels on the available communi-

cation links. The amount of communication delay to be assigned to the slow and fast links is decided by PER_{SLOW}. The delay for a slow link is $D_{i,SLOW} = \frac{AVG_{COMM} \times PER_{SLOW}}{|CL_{SLOW}|}$, where CL_{SLOW} is the set of all slow links. The delay of a fast link is also calculated using the same approach. Bandwidth is indirectly inferred from the CCR value and the average actor computation time. These parameters for the generation of the synthetic DAGs are summarised in table 6.4. The parameters that affect the computation, communication and resource usage values of the DAGs are in table 6.5.

- The percentage of resources required to execute an actor A_i on a compute engine CE_j is randomly generated within the range $RES_{BASE} \times DIFF_{MIN}$ and $RES_{BASE} \times DIFF_{MAX}$. The RES_{BASE} value of an actor is generated from a uniform random distribution with the limits of RES_{MIN} and RES_{MAX}.

Table 6.4: The parameters for the generation of synthetic application DAG that are used for the evaluation of the AMS algorithm.

Parameter name	**Parameter description**
NODE	Number of nodes (actors) in the DAG
FAT	Determines the maximum number of concurrent actors
JUMP	Maximum number of levels that a channel crosses to connect actors
AFFINITY	Maximum number of channels that an actor connects between two consecutive levels
REGULAR	Distribution regularity among the DAG levels

The next part will describe the highest-level component called the evaluation driver (*EVD*) that is used by the user to run the experiments.

Table 6.5: The parameters for the generation of synthetic DAG's performance and resource usage data that are used in the evaluation.

Parameter name	Parameter description
CPU_{MIN}	Minimum computation time when mapped onto the slowest CPU
CPU_{MAX}	Maximum computation time when mapped onto the slowest CPU
ACC_{MIN}	Minimum acceleration ratio
ACC_{MAX}	Maximum acceleration ratio
DEC_{MIN}	Minimum deceleration ratio
DEC_{MAX}	Maximum deceleration ratio
PER_{ACC}	Percentage of actors that can be accelerated in FPGA and GPU
CCR	Computation to communication ratio
PER_{SLOW}	Communication delay share assigned to the slow links rest are given to the fast links
RES_{MIN}	Minimum percentage of resource usage for the base value of an actor
RES_{MAX}	Maximum percentage of resource usage for the base value of an actor
$DIFF_{MIN}$	Minimum difference of resource from an actor's base value
$DIFF_{MAX}$	Maximum difference of resource from an actor's base value

Evaluation driver

The evaluation driver (*EVD*) component is primarily responsible to run the experiments. The high-level APIs of *rHEFT* and *AMS* are used to configure different versions of the algorithms. It consists of the functionalities to initialise the database, orchestrate data generation followed by storage and then running different configurations of rHEFT and AMS algorithms. Databases are refreshed for every run of an experiment unless the user specifies to retain the older data and the results. DAGs are stored with an index of each DAG uniquely identified by an *id*. The performance data for each DAG identified by its *id* is stored in the database. After running the algorithm (either a version of rHEFT or AMS), the results are stored. When all the DAGs are completed, the results are retrieved for interpretation. The next two sub-section details the usage of evaluation driver to set-up the experiments for evaluation and then discusses the results.

6.3.2 rHEFT-2 evaluation

In this subsection, the enhanced ranking and compute engine selection algorithms that incorporate resource usage of actors is compared with the original algorithms

that do not take resources into account. For this comparison, the AMS prototype is used to create two instances of rHEFT; one with the enhanced algorithms and the second with the original algorithms. In both the instances, shared communication links are used. The algorithm of the first instance is identical to the pseudocode in Figure 6.10. The algorithm for the second instance is similar to figure 6.2 but with shared communication links. The datasets used in this evaluation are adopted from the original HEFT paper [7] with the addition of resource data and the datasets are more exhaustive as three distinct platform architectures with varying sizes and topologies are used. This subsection is divided into two parts. The first part details the generation attribute values to generate the datasets and the metric used for the evaluation. The second part presents the experiential results.

Experimental configuration

rHEFT evaluation was conducted on a large number of synthetic DAGs. Based on the possibilities of concurrent actor executions, DAGs were grouped into high and low density DAGs. A high density DAG has more actors at each level and they consume less resources, as compared to a low density DAG. High density DAGs were generated with high *FAT* values and low resource usage values. Whereas low density DAGs were generated from low *FAT* values with higher resource usage. The rest of the generation attribute values of high and low density DAGs were same. Table 6.6 and 6.7 lists the generation attributes values of DAGs and their performance data, respectively.

Table 6.6: The parameter values for the generation of synthetic application DAG.

Parameter name	**Parameter value**
NODE	20, 40, 60, 80, 100, 120, 140, 160, 180 and 200
FAT	For low density 0.25 and 0.45, for high density 0.65 and 0.85
JUMP	1, 3 and 3
AFFINITY	1, 3 and 5
REGULAR	0.5 and 0.9

These experimental configurations were adopted from the original HEFT paper [7] with the following four modifications; (1) the generation of resource data for every actor, (2) the usage of three different platform architectures with varying sizes, (3) running experiments separately for low and high density DAGs, and (4) average computation to communication ratio (*CCR*) values that are higher than the original paper. A higher CCR value is used to emulate the agile heterogeneous computing scenario, where average computational times are generally higher than communication, so as to gain an acceleration advantage. Even if the computational time reduces significantly on GPU or FPGA, the total time including communication must be lower than the computation time on the CPU, which implies that average computation time is higher than communication.

Table 6.7: The parameter value for the generation of synthetic DAG's performance and resource usage data.

Parameter name	**Parameter value**
CPU_{MIN}	10
CPU_{MAX}	200
ACC_{MIN}	1.5 and 2.5
ACC_{MAX}	8.0 and 12
DEC_{MIN}	0.75 and 1.25
DEC_{MAX}	1.5 and 2.25
PER_{ACC}	70 and 95
CCR	5, 0.5 and 10
PER_{SLOW}	75 and 90
RES_{MIN}	for high density 10 and for low density 50
RES_{MAX}	for high density 50 and for low 95
$DIFF_{MIN}$	10
$DIFF_{MAX}$	30

The configuration details of (1) the fixed platform architecture used, (2) the range of values for DAG generation attributes, (3) the range of values for the DAG's performance and resource consumption data and (4) the hyper-parameter configurations are described as follows.

1. The three fixed platform architectures used varied in the number of constituent compute engines. The first platform architecture is named *mini* that consists of one of each compute engine kind. The second platform architecture is named *big*, as it consists of two GPU, one FPGA and one CPU. The third platform architecture is the largest. It consists of three GPUs, two FPGA and one CPU. This platform architecture is named *huge*. The connection topologies of mini, big and huge are as follows. In the *mini* platform, there are no direct connections between FPGA and GPU are connected to the CPU. In the *big* platform architecture, both the GPUs are directly connected. In the *huge* platform architecture, two GPUs have a direct connection and the two FPGAs have a direct connection as well. There are no direct connections between GPU and FPGA.

2. A large number of high and low density DAGs were generated by using an extended range of values for the generation attributes. Table 6.6 and 6.7 contain the values used for the generation attributes. Based on theses values, around 140,000 DAGs with a unique set of attribute values were generated. In order to reduce biases, the experiments were repeated 25 times. In every repetition, DAGs were re-generated. Therefore, $25 \times 140,000$ DAGs were used for the evaluation of rHEFT.

3. The attribute values used for the performance data encompass a wide accel-

eration range. It also includes deceleration. However, only a small number of actors were allowed to reduce their performance on GPU or FPGA, which is evident from the PER_{ACC} values in table 6.7. A wide range of values were also used for the resource usage data. It is noted that when actors consume less resources, there are more chances of concurrent execution on a compute engine, as compared to actors with large resource consumption percent.

4. The rHEFT-2 algorithm requires two hyper-parameters; resource factor (RF) and mapping threshold (MT). They determine the influence of an actor's resource usage on the mapping and scheduling decisions. Recall from section 6.2.2 that RF determines the impact of resource usage in the actor ranking. Whereas, MT is the trade-off between actor acceleration and resources required to achieve it. In order to study how these hyper-parameters change the quality of mapping and scheduling decisions, experiments are run with different RF and MT values.

The original and the enhanced rHEFT algorithms (with different MT and RF values) were executed on all the DAGs for the three different platform architectures. This was repeated 25 times for random DAGs with common attributes and then averaged. The metrics used to record performance are described in the next part.

Metrics

The metric used for comparing the two instances of rHEFT (original and enhanced) is schedule length ratio (SLR). This metric is chosen because they are widely applied in relevant literature for the comparison of mapping and scheduling decisions [7, 104, 109]. The SLR definition is modified to incorporate resources within compute engines.

SLR is defined as the ratio of the algorithm's makespan with the theoretical minimum makespan. The theoretical minimum makespan is obtained by mapping the actors on the compute engines where it's computation time is least and the communication delays are assumed to be negligible. It is also assumed that the compute engines have infinite resources, allowing all possible concurrent executions. SLR can be written as,

$$SLR = \sum_{i \in CP_{MIN}} EXE_{i,min} \tag{6.5}$$

where CP_{MIN} is the critical path of the DAG after mapping the actors i on the compute engines providing minimum computation time. The minimum computation time of actor i is CP_{MIN} is denoted as $EXE_{i,min}$. The average SLR of several DAGs were used in the evaluation.

It is noted that the minimum theoretical makespan for agile heterogeneous computing can be much lower than actual ones, resulting in higher SLR values than

previously published papers [7, 104, 109]. This is because of the higher variation of computation times than traditional mapping and scheduling problems, where the variations are considered only on different kinds of multiprocessors that led to differ computation times with smaller range. Whereas, on FPGAs and GPUs the computation times can be lower by up to 12 times of a CPU (see the data generation parameters in table 6.6) and 6.7. Higher SLR can be attributed to another reason, which is, the theoretical minimum makespan do not include the delay due to communication delays. This further increases the actual makespan than the theoretical minimum value that can result in a higher SLR. The next part presents the SLR and SU values that resulted after running rHEFT with the original and enhanced algorithms.

Results

The experimental results for rHEFT using the three different platform architectures (*mini*, *big* and *huge*) are shown in Figure 6.25. These platform architectures were defined in the previous part. It can be seen that with the increase in platform architecture size, the difference in makespan increases between the original and the enhanced algorithm. The enhanced algorithms resulted in much lower makespan with the increase in the platform architecture size. Also, high density DAGs resulted in better performance.

It can be further seen that reduction of makespan only occurs within a certain region of the chart called the *gain region*. This region starts from a lower actor number and lasts until a higher actor number. As the number of actors increase, chances of concurrent execution also increases. This causes the enhanced algorithms to improve more with a higher number of actors. However, with too many nodes, the makespan becomes similar to the original algorithms. Since too many actors increases the communication waiting times, the over all performance drops.

Since the performance improves more with a higher number of actors and also for high density DAGs, it can be inferred that the enhanced algorithms perform better with resource competition. This inference can be backed by another observation that the gain region starts with a higher number of actors with the increase of platform architecture size. For example with the *huge* platform the gain starts from 140 actors, whereas for *mini* it is 80. As the sizes of platform increases, competition for resources are not created until a higher number of actors are to be mapped.

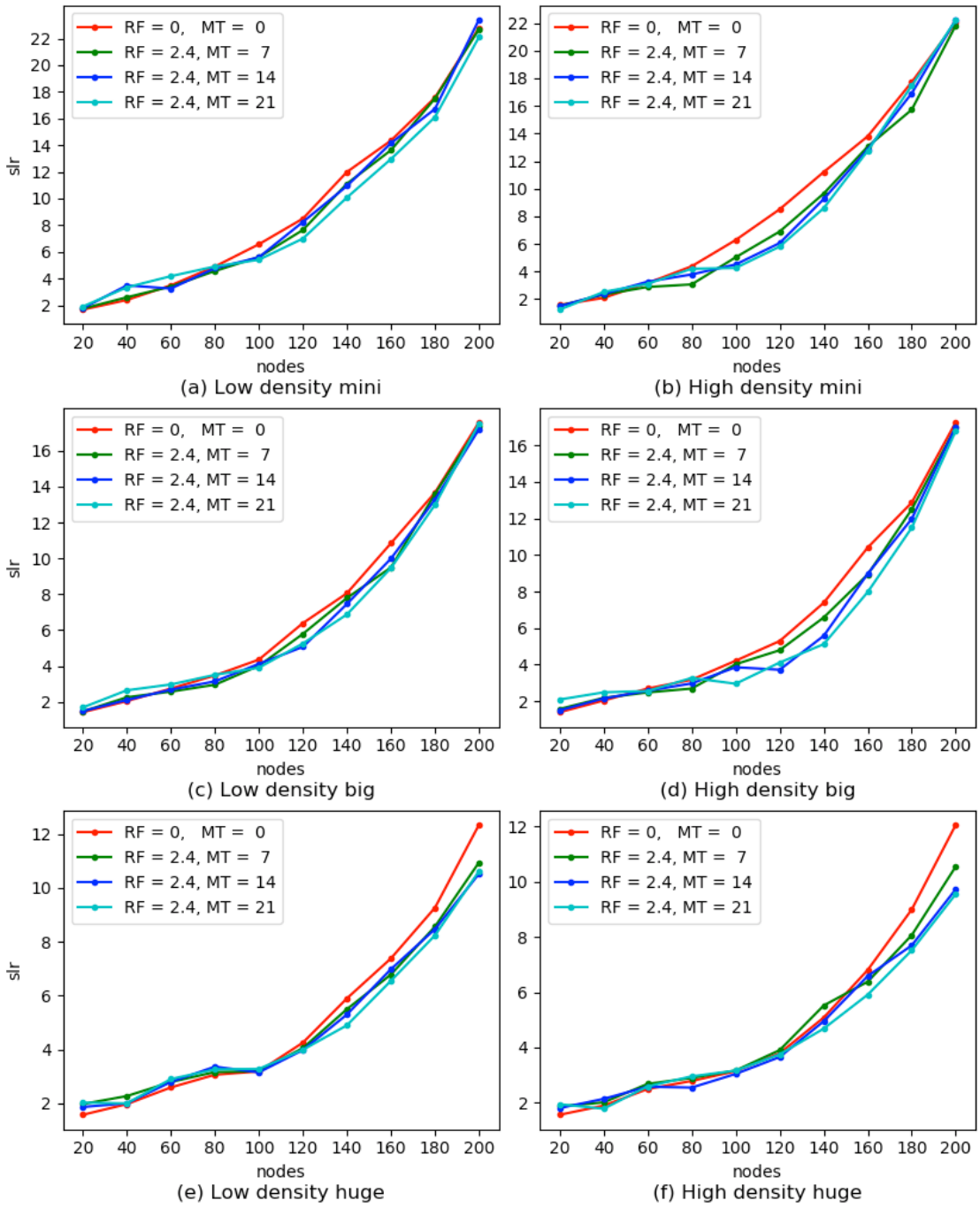


Figure 6.25: The SLR values of random DAGs after executing the original and enhanced versions of ranking and compute engine selection algorithms for different resource factor (RF) and mapping threshold (MT) values.

6.3.3 AMS evaluation

In the previous subsection, it was shown that when there is competition for resources, the rHEFT enhancements improve the mapping and scheduling decisions. However, rHEFT is a part of the AMS algorithm. An overall evaluation of the AMS algorithm with a large set of available compute engines and DAGs is presented here. In this evaluation, the expansion and reduction modes of design space exploration are comparatively studied. Also, they are compared with randomly generated platform architecture within a similar budget (capital cost). The comparison of the two exploration modes and the random platform architecture selection is adapted from the evaluation method proposed in the Gain/Loss paper [8], where experiments are run over a range of budget. The experimental results show that both the modes of exploration (expansion and reduction) perform significantly better than random platform generation. It is also established that among both the modes, the average performance of *expansion* is better. This subsection is divided into three parts. In the first part, the details of the experimental set-up is described. Then in the second part the metrics used for the evaluation are defined. Finally, in the third part, the experimental results are presented.

Experimental configuration

The platform architecture generated by the AMS algorithm depends on the following: (1) platform constraints, (2) maximum budget, (3) initial platform architecture instance, (4) random iteration count and (5) application-algorithm DAG. In order to evaluate the expansion and reduction modes of exploration with each other and with randomly generated platform architectures, these dependencies are set in a way such that they are unbiased for every application DAG. The configuration of these five dependencies are described as follows.

1. The platform constraints are in the form of parametrised platform graph (PPG) tier1. It consists of 3 types of GPU, 3 types of FPGA and one CPU. There are 5 units for every type of FPGA and GPU. By default communi-

Table 6.8: The platform architecture constraints for the AMS experiments.

Compute Engine	No. of Types	No. of units of each type	Unit cost relative to CPU cost	Direct connections	Total compute engines
GPU	3	2	type-1: 1.5 type-2: 2.0 type-3: 2.5	5	NA
FPGA	3	2	type-1: 2.0 type-2: 2.5 type-3: 3.0	5	NA
CPU	1	1	NA	4	4

cation links are through CPU. However, direct connection are also allowed. Every GPU or FPGA can have a maximum of two direct communication links, whereas the CPU can have 4 compute engines directly connected. However, more compute engines can be connected through *docks* (switches). The capital cost of each compute engine type are relative to that of the CPU. The lowest cost of a compute engine is equal to cost of the CPU, whereas the most expensive is 10 times of its cost. Table 6.8 shows the compute engines available for design space exploration. This platform constraints are used for every application DAG in this evaluation.

2. The budget range for the evaluation is adopted from the Gain/Loss paper [8], where equally spaced values between the lowest and the highest budget are used. Since the Gain/Loss paper uses rental cost, the main difference in the budget range calculation is in the formulation of the lowest and the highest budget. The highest budget is the capital cost of the best platform architecture PI_{BEST} on which the application DAG attains its best performance (lowest makespan). It is calculated from the expansion mode of exploration, where the initial platform architecture is the smallest architecture consisting only of the CPU. Whereas, the lowest budget is the cost of the smallest platform architecture, which is just the capital cost of one CPU. Therefore the budget range of an application DAG i is defined as follows:

$$B_i = B_{CPU} + k \times (B_{highest,i} - B_{CPU}) \tag{6.6}$$

   where k is 0.2, 0.4, 0.6 and 0.8.

3. Two random iteration counts, 10 and 25 are used in the experiments. A lower random iteration count shows efficiency in finding the optimal platform architecture instance.

4. The initial platform architecture instance (PPG tier2) for expansion mode is the smallest architecture, which is just the CPU. This ensures uniformity for every application DAG. For the reduction, the best platform architecture (PI_{BEST}) is the initial platform architecture. PI_{BEST} is the result of previous exploration using the expansion mode to calculate the highest budget $B_{highest}$. It is noted from the budget range equation, the highest budget is always less than the capital cost of PI_{BEST} which satisfies the condition of the reduction mode of the maximum budget to be always lower than the capital cost of the initial platform architecture.

5. The application DAGs used in this evaluation consists of 90, 100 and 110 node counts. The rest of the generation parameters values for the random DAGs are identical to table 6.6 and 6.7. A relatively large number of nodes are used so that PI_{BEST} has at least 6 compute engines including the CPU. This ensures that when the maximum budget is changed, at least one compute engine can be added or removed, depending on the mode of exploration.

The two exploration modes along with the random platform architecture generation are executed for all the DAGs. Experiments are repeated for the two different random iteration counts. Each experiment is repeated for 25 times and the average of the metrics are used. The metrics used are defined in the next part.

Metric

The quality of the platform architecture produced by either reduction, expansion or random exploration for an application DAG G within a certain budget (B_{max}) is reflected by the makespan of G on the chosen platform architecture instance PI_{chosen}. PI_{chosen} is the one that conforms to B_{max} and results in the lowest makespan amongst other platform architectures. Therefore, the metric used to evaluate the quality of the platform architecture is based on makespan. An adaptation of the metric, *average normalised difference* (AND) defined in the Gain/Loss paper [8] is used to compare the three different variations of agile platform architecture exploration. Since AND is normalised difference averaged over the number of DAGs, the adaptation of normalised difference (ND) is first defined, then adapted AND is described.

The adapted normalised makespan is defined as follows:

$$ND_k = \frac{MS_{ex} - MS_{cpu}}{MS_{best} - MS_{cpu}} \tag{6.7}$$

, where k stands for the budget range of the exploration (defined in equation 6.6), MS_{ex} is the makespan on the platform architecture produced by the experiment, MS_{cpu} is the makespan on the platform architecture consisting only of the CPU and MS_{best} is the makespan on the best platform architecture possible within the PPG tier1. It is assumed that MS_{best} is the lowest makespan. This implies that AND varies between 0 and 1, as MS_{ex} can only get close to MS_{best}. A higher value of AND, closer to 1 signifies better platform architecture. The normalised makespan is calculated for a DAG with a maximum budget denoted by k.

The adapted AND is defined as follows,

$$AND_k = \frac{1}{N} \sum_{i=1}^{N} \frac{MS^i_{ex} - MS^i_{cpu}}{MS^i_{best} - MS^i_{cpu}} \tag{6.8}$$

, where N is the total number of DAGs used. For each DAG, AND was calculated by running the three variations of exploration; expansion, reduction and random platform generation. Each variation of exploration resulted in 4 AND values, where every value of AND corresponds to MS_{ex} determined by k (see equation 6.6).

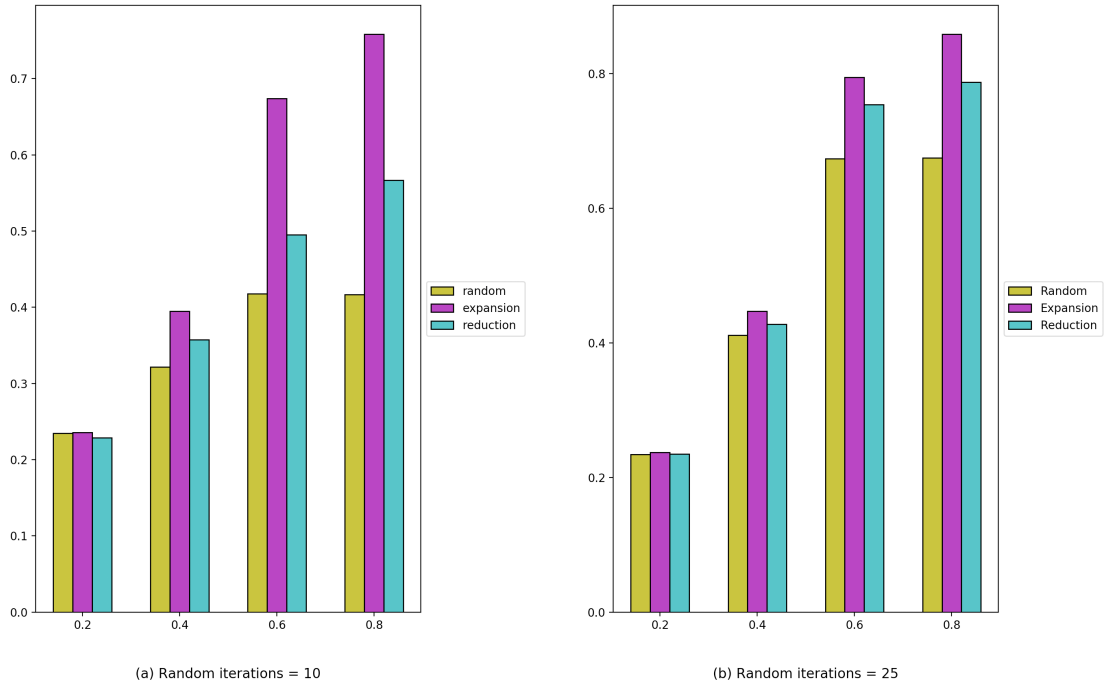
In order to study the impact of random iteration count (itr_{max}), the experiments are conducted with two different values of itr_{max}. The first is lower, $itr_{max} = 10$

and the second is higher $itr_{max} = 25$. Based on itr_{max}, the number of samples for the random platform generations is determined. The random samples are five times of itr_{max}. This ensures a fair comparison between the random platform generation with the two modes of exploration. These experimental results are presented next.

Results

The experimental results of running expansion, reduction and random modes of platform generation are presented in Figures 6.26, 6.27, 6.28 and 6.29. Each figure is dedicated for a type of DAG that are either random [7, 104], Laplace equation solver [101, 125], fast Fourier transformation graph [101] and LU decomposition [101, 125]. The figures are in the form of histogram showing the AND values in the y-axis for every k values (see equation 6.8) that represents the maximum budget. This budget represented by the k values are in the x-axis. These figures show that expansion and random modes of exploration performs significantly better than random platform generation. However, in some cases, when the budget is low, the random generation results are closer to the other modes of exploration. This is due to a reduced number of compute engines present for the creation of a platform architecture with a low budget. As the best platform architecture in random generation is selected amongst $itr_{max} \times 5$ samples, there is a higher chance of finding the optimal solution with less number of compute engines to form the platform architecture.

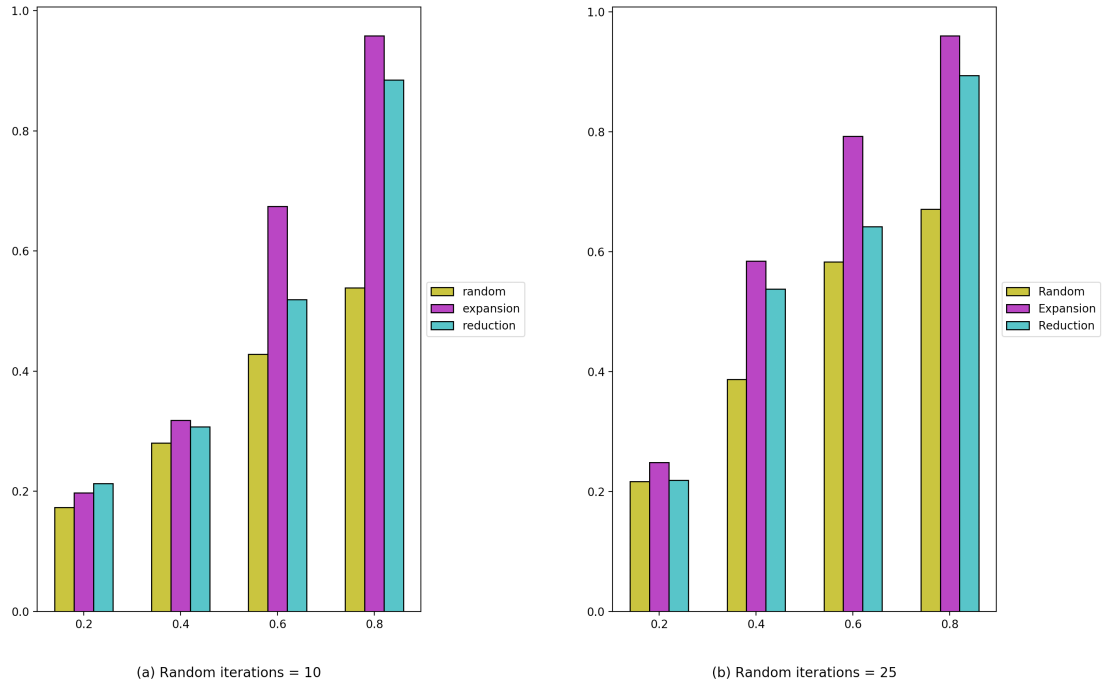
Among expansion and reduction modes of exploration, the expansion approach performs better. This can be attributed to the fact that in the reduction mode, the removal of a compute engines do not consider replacing it with a cheaper variant. Whereas, the expansion mode gradually builds the platform architecture by accessing all the available compute engines. This inference can be used for the optimisation of the reduction mode of exploration in the future work. However, the of lower and higher random iteration counts show that the results of random exploration tends to get closer with the expansion mode.



(a) Random iterations = 10

(b) Random iterations = 25

Figure 6.26: Average normalised difference (AND) for random DAGs [7,104]. Three modes of exploration: expansion, reduction and random platform architectures are compared with each other for random iteration counts of 10 and 25.



(a) Random iterations = 10

(b) Random iterations = 25

Figure 6.27: Average normalised difference (AND) for Laplace equation solver DAGs [101, 125]. Three modes of exploration: expansion, reduction and random platform architectures are compared with each other for random iteration counts of 10 and 25.

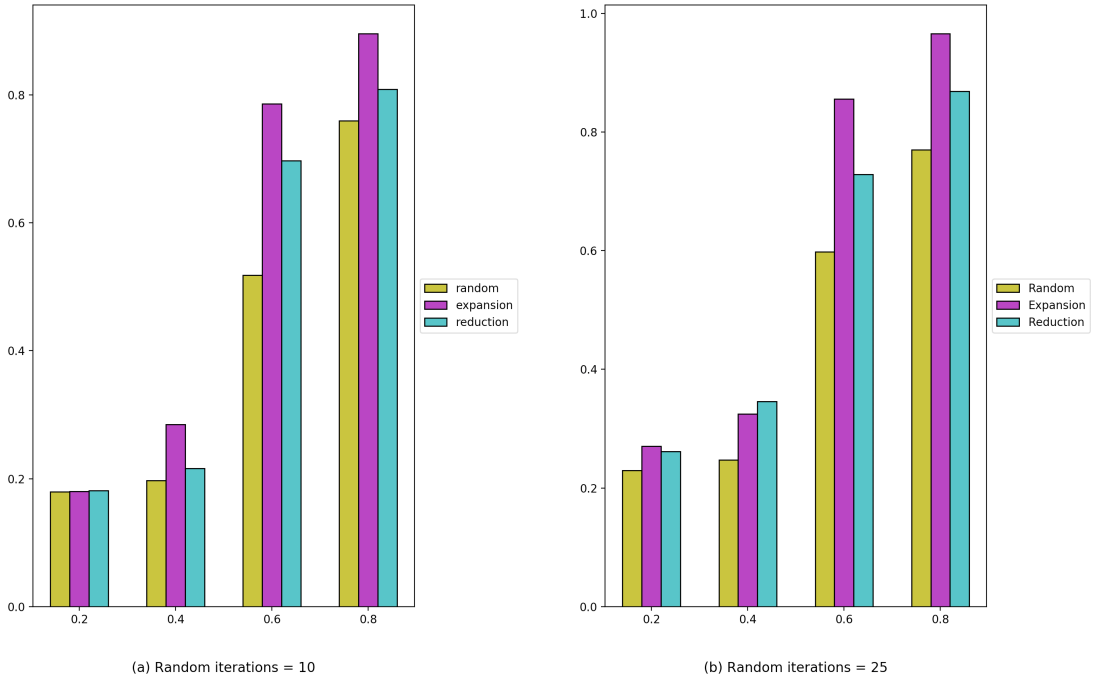


Figure 6.28: Average normalised difference (AND) for Fourier transformation DAGs [101]. Three modes of exploration: expansion, reduction and random platform architectures are compared with each other for random iteration counts of 10 and 25.

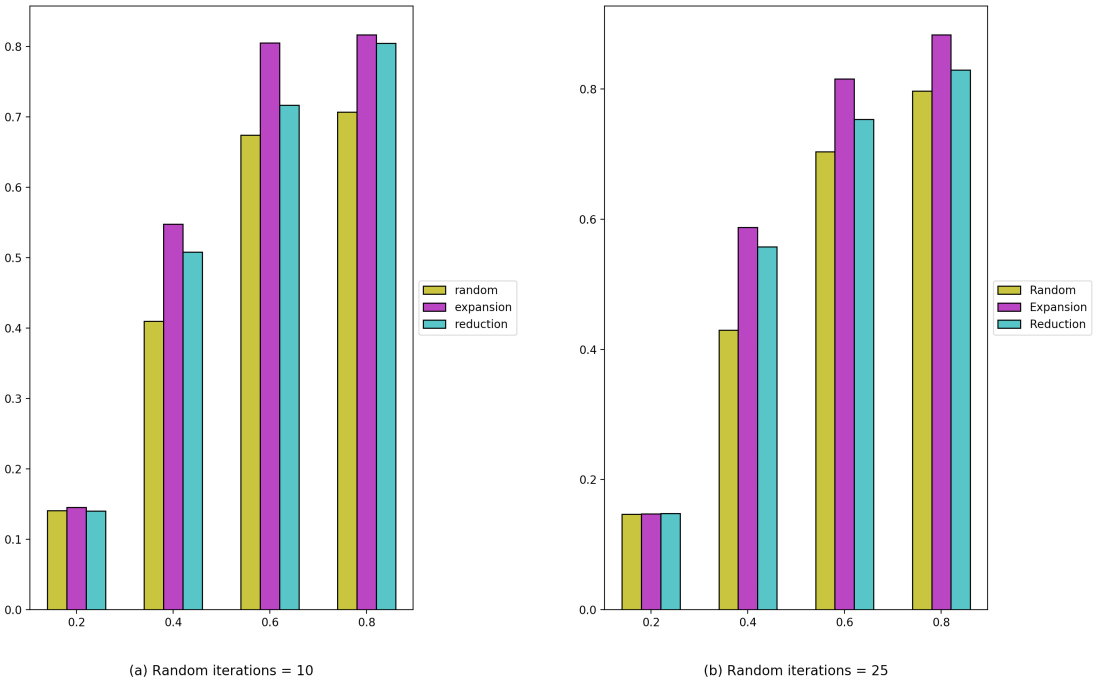


Figure 6.29: Average normalised difference (AND) for LU decomposition DAGs [101,125]. Three modes of exploration: expansion, reduction and random platform architectures are compared with each other for random iteration counts of 10 and 25.

6.3.4 Conclusion

In this section, the evaluation of the agile mapping and scheduling algorithm (AMS) was presented. At first, a prototype of AMS was developed as a part of an elaborate evaluation framework, which was later used to run experiments with different versions of the rHEFT and AMS. Although rHEFT constitutes AMS, it was evaluated separately, as it lays the foundation of considering resource usage for design space exploration.

The performance of rHEFT consisting of the enhanced resource conscious ranking and compute engine selection algorithms were compared with the original ranking and compute engine selection algorithms. A large number of synthetic DAGs and three different platform architecture with varying sizes were used for the experiments with rHEFT. The results showed that the enhanced algorithms reduce makespan significantly at the *gain region* than the original ones. *Gain region* denotes the area in the graph where the enhanced algorithms perform better. This region varies with size of the platform architecture and the value of the hyperparameters. Since the gain region shows more difference in makespan with larger number of actors and high density DAGs, it was inferred that when there is competition for resources, the enhanced (resource conscious) algorithms performs better.

After evaluating rHEFT, AMS was evaluated to find which of the two modes of exploration (expansion or reduction) are better. Furthermore, both the modes were compared with randomly generated platform architectures. The random generation of the platform architecture selected the best architecture among a certain number of samples. In order to compare fairly, the sample numbers were based on the maximum random iteration count (itr_{max}). The experimental results showed that the random mode of platform generation always perform below the other two modes. Amongst expansion and reduction, the expansion mode perform better. But, the difference of performance between expansion and reduction are evident when itr_{max} is lower. With a higher itr_{max}, results of both the modes are almost similar. It was inferred that the AMS algorithm can find optimal results and the expansion approach of exploration is more efficient than the reduction mode. However, if the designer needs to use the reduction mode of exploration to reduce an existing platform architecture's capital cost, then it can be used with higher itr_{max} to increase the chances of better results.

6.4 Conclusion

In this chapter, a design space exploration algorithm called agile mapping and scheduling algorithm (AMS) was presented. AMS is created for the new design flow of agile heterogeneous computing, where an optimised platform architecture needs to be formed while considering the mapping and scheduling decisions of an application-algorithm. AMS combines design space exploration with deterministic and random steps. Initially the exploration starts from a designer provided design point and proceeds with deterministic steps. However, the deterministic steps leads the exploration to an edge of the design point that causes an impasse. The random steps moves the design point to a new location to resume the deterministic steps. The results of the exploration are stored in a database and they are used to guide the random steps for the selection of a new design point. The number of times a design point to be moved randomly is decided by the designer, which is typically based on the size of the design space.

AMS was created in two stages. Each stage of the AMS development was described with examples and pseudocode. In the first stage a widely used and efficient mapping and scheduling algorithm called HEFT was enhanced to allow the unique requirements of agile heterogeneous computing, which resulted in resource-HEFT (rHEFT) that can consider resource usage of actors and specialised topology of agile platforms. Then, in the second stage, rHEFT was synthesised with an enhanced rental budget optimisation algorithm [8] for work-flow schedule. The enhancement was to enable capital cost-based optimisation, rather than rental cost. The second stage resulted in the AMS algorithm that consists of two modes of exploration; expansion and reduction. In the expansion mode, a small platform architecture is expanded by adding more compute engine. Whereas, in the reduction mode, compute engines are removed from a large platform.

After describing the AMS algorithm, it was evaluated with an extensive range of application DAGs. In order to conduct experiments for the evaluation, a framework was created, which consisted of a prototype of AMS and other facilities of data generation and storage. The evaluation first experimented with rHEFT, then the overall AMS algorithm was considered.

It is noted that while rHEFT was created, the actor ranking and compute engine selection algorithms, which are the constituents of HEFT, were enhanced to make them resource conscious. Thus, in the rHEFT evaluation, the resource conscious algorithms were compared with the original algorithms. It was found that the enhanced algorithms provide significantly better mapping and scheduling results when there is competition for resources. There will be competition for resources when there are many actors for concurrent executions and less resources available for all the concurrent executions. Since the enhanced algorithms is capable to prioritise actors for the best available compute engine for concurrent executions, they result in better performance.

The purpose of the overall AMS evaluation was to study the exploration results of the expansion and reduction modes of explorations and to compare both the modes of exploration with randomly generated platform architectures. For a fair comparison, the random platform generation is sampled over an extended number of random platform architectures. The results showed that the random platform generation performed poorly as compared to both the modes of exploration. Furthermore, on average, the expansion mode of exploration performs better than the random mode. However, when the number of random design point selections are increased, the results of reduction mode of exploration is comparable with the expansion.

Chapter 7

Case study with a multi-object visual tracking application

Contents

7.1 Introduction

The previous chapter introduced the agile mapping and scheduling (AMS) algorithm, which is central to the new design flow (AhcFlow) for exploring the design space. A number of synthetic and some real application DAGs were used to explore the predicted performance from the AMS algorithm prior to actual deployment on a real platform architecture. In this chapter a published visual tracking algorithm (CACTuS [112]) which has been previously implemented as a hand crafted heterogeneous system is used to demonstrate the practical utility of AMS and AhcFlow as an design space exploration tool and to illustrate the automated deployment features of AhcFlow.

The chapter is organized as follows. In the next section 7.2, the CACTuS tracking algorithm is introduced. This introduction includes the SDF representation of CACTuS, its conversion to a DAG and the actor performance and resource consumption data taken from a previous published hand crafted implementation of CACTuS on a heterogeneous platform not using AhcFlow. Also included is a data set of performance and resource usage of actors in CACTuS obtained from experiments conducted as part of this research. The following section 7.3 compares the performances of two hand crafted published implementations of CACTuS with that achieved with the AMS algorithm inside AhcFlow. This section shows that the AMS algorithm is able to achieve similar predicted performance results as the hand crafted implementations. Section 7.4 uses the AMS algorithm to explore the design space for agile heterogeneous implementations of CACTuS that goes beyond the architectures assumed in the published hand crafted implementations. This section provides insight into the best architectures for larger versions of CACTuS and especially investigates the impact of direct communication links between GPUs on multi-GPU architectures for CACTuS. This design space exploration is conducted with actor performance and resource usage data collected as part of this research based on an Intel i7 CPU and Nvidia GTX 960 GPUs[1]. In section 7.5 the deployment parts of AhcFlow are described in association with a simple example DAG. The deployment of a version of CACTuS is also demonstrated. A complete AhcFlow design flow deployment shows that the predicted performance of the AMS algorithm is maintained after deployment.

7.2 CACTuS visual tracking application

In this section, the CACTuS visual tracking application is introduced. This application is used to demonstrate and evaluate the complete AhcFlow design flow from the DAG representing CACTuS to deployment, including the AMS mapping

[1]Direct GPU to GPU links were not available for this research but reasonable estimates were used to simulate the links.

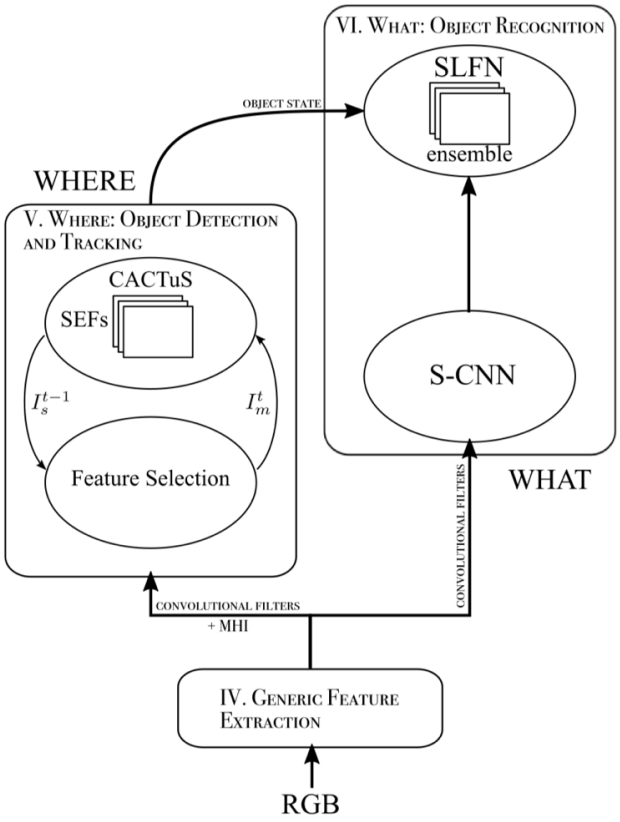


Figure 7.1: The CACTuS-FL application [112] showing multiple shape estimating filters (SEFs) within the object detection and tracking block, numbered as (V).

and scheduling algorithm described in the previous chapter. CACTuS is a complex algorithm whose detailed inner workings will not be all explained here. The reader is referred to the numerous previous publications on the topic [112, 127–130]. It should be noted that in recent times, CACTuS has been enhanced with feature detection using machine learning (the so called CACTuS-FL as presented in Figure 7.1). The version used here is a tracking only subset of CACTuS-FL where generic feature extraction, feature selection and object recognition are assumed to be carried out prior to starting. This core component of CACTuS-FL is referred as CACTuS [112].

One of the most important aspects of CACTuS is the possibility to scale the algorithm based on the number of so called Shape Estimating Filters (SEFs) and the size of each video frames, called image size. For the purposes of this work, a single SEF can be viewed as capable of tracking a single object, so the algorithm can be scaled from tracking a single object (1 SEF) to tracking a large number of objects simultaneously (multiple SEFS). A large image can have more SEFs as compared to a smaller image. As the number of SEFs increases, so does the computational requirements of the algorithm. In this chapter five different configurations of image sizes and SEF numbers are used. They are detailed later in table 7.1.

The first step in making CACTuS ready for processing by AhcFlow is to create a DAG representing CACTuS, which is done by transforming an SDF graph of CACTuS to its DAG equivalent. The second step is to gather the performance and resources used by each actor (within the CACTuS DAG) on each compute engine. There are two sets of performance and resource data used in this chapter. The first set is obtained from the hand-crafted implementation in [129] and the second is measured as a part of this research. These two steps are detailed in the following two subsections. The first subsection, presents the CACTuS SDF graph and then its equivalent DAG. Then, the second subsection details the two sets of performance and resources datasets.

7.2.1 Dataflow model

A synchronous dataflow (SDF) graph[2] of the CACTuS algorithm, showcasing all of its constituent actors and channels is illustrated in Figure 7.2. The dotted rectangle encloses the actors that are within each SEF. Based on the number of SEFs, which is denoted by N in the figure, the actors within the dotted rectangle will be fired N times for every image. In SDF semantics it can be said that the rate of firing of all the actors within the dotted rectangle equals to N. The solid large dots are the initial tokens that are used to start the application. The actors *Predict Velocity (PV)*, *Predict Shape (PP)* and *Form Likelihood Ratio (FLR)* are the starting actors. This can be used to transform to an equivalent DAG using the algorithm presented in [60]. A CACTuS DAG will consist of $18 \times 2 + 3$ actors (ignoring the duplicate actors), where the 3 additional actors are external to the SEFs. This implies that every time a SEF is added, 18 actors are also added. An equivalent DAG of a single SEF SDF graph is shown in Figure 7.3.

It is interesting to note that the internal actors of a SEF form an homogeneous synchronous dataflow (HSDF) graph, as every actor consumes and releases one token at there input and output channels. The model also reveals that apart from data-level parallelism amongst SEFs, the actors within a SEF has task-level parallelism. It is up to the design space exploration to find which parallelisms are essential for performance. Appropriate compute engine selection for an actor will be essential when the number of SEFs are high and the available resources are not enough to exploit all the available parallelisms.

[2]In this SDF graph, the actors that just duplicate an output of an actor to more than one channel are not shown, rather an output channel is simply divided.

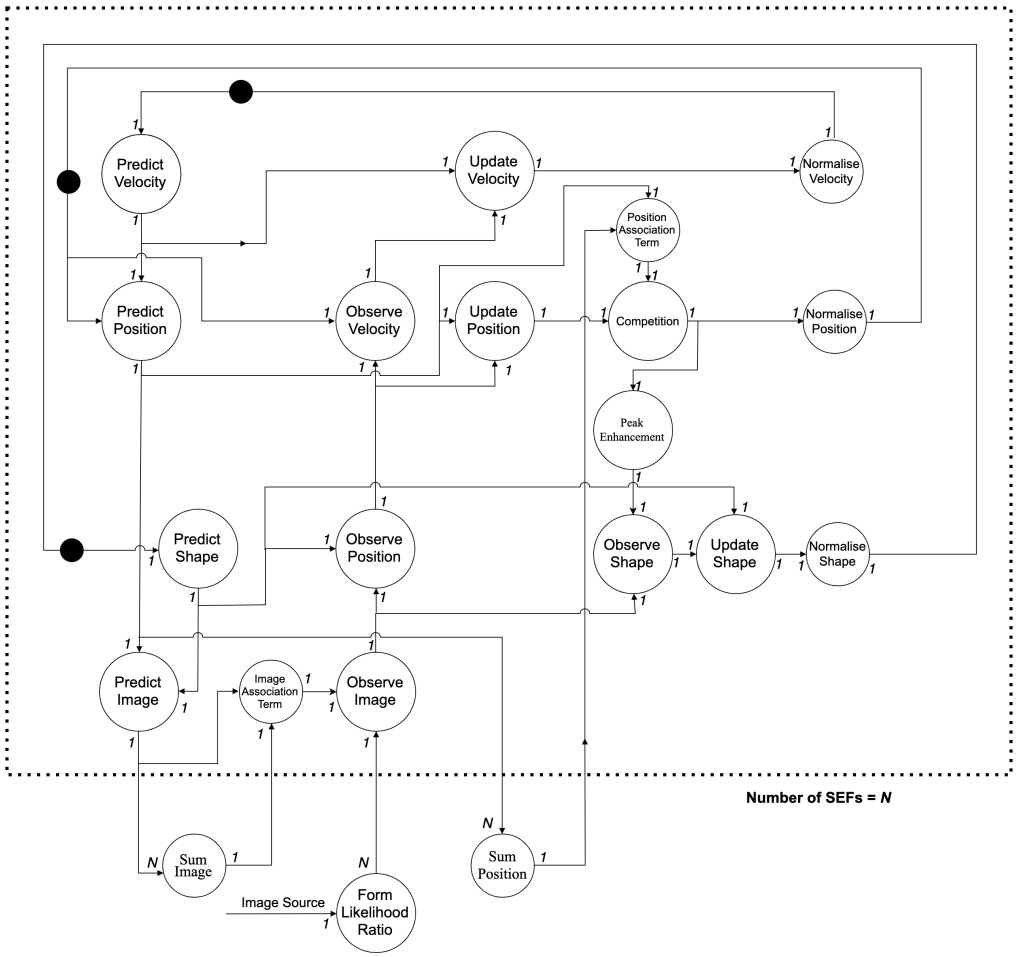


Figure 7.2: The SDF model of CACTuS. The actors shown are explained in table 7.2.

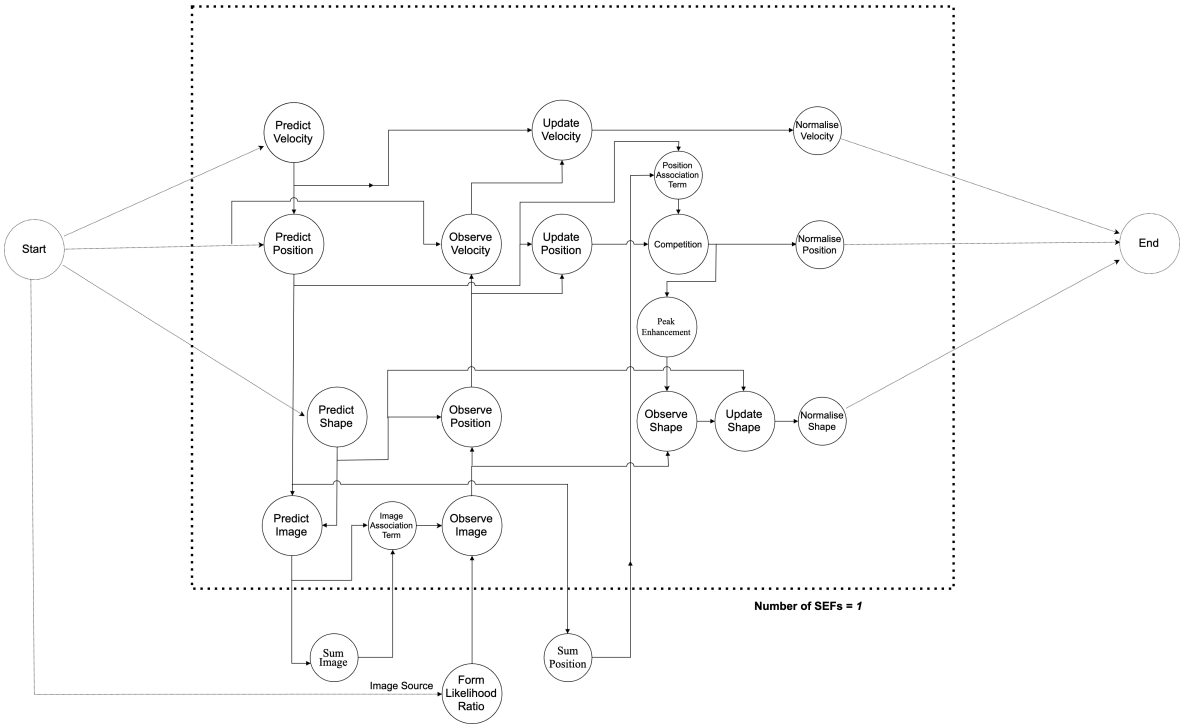


Figure 7.3: The DAG model of CACTuS with 1 SEF.

7.2.2 Pre-engineered components

In this section, the actor execution timings and resource usage data for the hand crafted versions of CACTuS constructed by Milton 2017 [129] and the additional data gathered as a part of this research using an Intel i7 CPU and Nvidia GTX960 GPU are presented. These datasets are specific to CACTuS configurations with varying number of SEFs and image sizes. As the computation complexity of the actors are dependent on the image size and the SEF numbers, the execution timing and the resource usages of the actors vary. There are five different configurations taken into account. These configurations are listed in table 7.1. Milton's data for configuration *C1* is shown in table 7.5. The data collected as a part of this research for configuration *C1* is presented in tables 7.3 and 7.4. Data for the rest of the configurations are listed in Appendix-A. The actor execution timings and the communication delays are in micro seconds. The resource consumption shows the percentage of the compute engine occupied by the actor.

Milton's actor execution timings were inclusive of communication delays and they were measured after all the SEFs were completed, so this dataset is extrapolated for individual actor execution timing. This is done by dividing the reported timing by the number of SEFs. Also, since Milton implemented one actor at a time on a compute engine, the resource consumption for every actor on a compute engine is considered to be 100%. Furthermore, as the communication delays are included within the actor execution timings, the communication delays of the channels are

Table 7.1: The five CACTuS configurations are detailed in this table. It consists of the number of SEFs and the actor input data token sizes in pixels for each of the five configurations. In the table N refers to number of SEFs, Z is predict velocity kernel size, S is the shape kernel size, I is the image size, V is the velocity kernel size and X is the position kernel size.

Configuration	**N no. of SEFs**	**Z (pixels)**	**S (pixels)**	**I (pixels)**	**V (pixels)**	**X (pixels)**
C1	16	7×7	13×13	127×127	27×27	115×115
C2	16	9×9	55×55	511×511	28×27	457×457
C3	64	9×9	27×27	511×511	29×27	485×485
C4	64	11×11	55×55	1023×1023	47×47	913×913
C5	256	11×11	111×111	1023×1023	47×47	968×968

Table 7.2: The CACTuS actors with their input data token sizes in pixels and their complexities in big-$\mathcal{O}$ notation.

Actor name	**Denotation**	**Operation**	**Input data pixel sizes**	**Complexity**
Predict Velocity	PV	Convolution	V, Z	$\mathcal{O}(V \times Z)$
Predict Position	PP	Convolution	X, V	$\mathcal{O}(X \times V)$
Predict Shape	PS	Convolution	S, Z	$\mathcal{O}(S \times Z)$
Predict Image	PI	Convolution	X, S	$\mathcal{O}(X \times S)$
Observe Velocity	OV	Cross-correlation	X, X	$\mathcal{O}(X^2)$
Observe Position	OP	Cross-correlation	I, S	$\mathcal{O}(X)$
Observe Image	OI	Per-element multiplication	I, I	$\mathcal{O}(I)$
Image Association Term	IAT	Per-element multiplication with summation	I, I	$\mathcal{O}(NI)$
Update Velocity	UV	Per-element multiplication	V, V	$\mathcal{O}(V)$
Update Position	UP	Per-element multiplication	X, X	$\mathcal{O}(X)$
Position Association Term	PAT	Per-element multiplication with summation	X, X	$\mathcal{O}(NX)$
Competition	COM	Per-element multiplication	X, X	$\mathcal{O}(X)$
Peak Enhancement	PE	Max function	X, X	$\mathcal{O}(X)$
Observe Shape	OS	Extraction	S, S	$\mathcal{O}(1)$
Normalise Velocity	NV	Per-element division	V	$\mathcal{O}(V)$
Normalise Position	NX	Per-element division	X	$\mathcal{O}(2X)$
Update Shape	US	Per-element multiplication	S, S	$\mathcal{O}(S)$
Normalise Shape	NS	Per-element division	S	$\mathcal{O}(2S)$
Sum Position	SUMX	Summation	N × X	$\mathcal{O}(N \times X)$
Sum Image	SUMI	Summation	N × I	$\mathcal{O}(N \times I)$
Form Likelihood Ratio	FLR	Reading from file	None	$\mathcal{O}(1)$

Table 7.3: The execution timings and the resource consumption percentage of the CACTuS actors for configuration *C1* obtained in this work. Infinite (∞) represents the actors that do not have a pre-engineered implementation on the compute engine. This is replaced with a very large number in the experiments. The timings are in micro seconds (μ). The resource consumptions are expressed as the percentage of the usable compute engine resources.

Actor	**CPU execution timing (μ)**	**CPU resource consumption (%)**	**GPU execution timing (μ)**	**GPU resource consumption (%)**
PV	8	90	∞	∞
PP	22	90	192	12
PS	9	90	28	12
PI	898	90	261	12
OV	837	90	239	12
OP	129	90	314	12
OI	11	90	72	12
IAT	19	25	68	12
UV	12	90	47	12
UP	29	90	59	12
PAT	17	25	70	12
COM	11	25	51	12
PE	2	25	∞	∞
OS	62	90	62	12
NV	2	90	∞	∞
NX	2	90	∞	∞
US	2	90	∞	∞
NS	2	25	22	12
SUMX	2	25	∞	∞
SUMI	2	25	∞	∞
FLR	349	90	∞	∞

Table 7.4: The channel communication delays in micro-seconds (μ) for CACTuS configuration *C1*.

Communication Channel	Delay timing μ	Communication Channel	Delay timing μ
PV to PP	150	OI to OP	600
PV to UV	150	OI to OS	600
PP to SUMX	482	IAT to OI	600
PP to PAT	482	UV to NV	150
PP to UP	482	UP to COM	482
PPto PI	482	PAT to COM	482
PS to PI	135	COM to PE	482
PS to OP	135	COM to NX	482
PS to US	135	PE to OS	5
PI to IAT	600	OS to US	135
PI to SUMX	600	US to NS	135
OV to UV	150	SUMX to PAT	482
OP to OV	482	SUMI to IAT	600
OP to UP	482	FLR to OI	600

Table 7.5: The published hand-crafted actor timing results by Milton [129]. Infinity (∞) refers to the actors that were not considered to be implemented on the compute engine. It is noted that the original results were in milliseconds for 16 SEFs. These timings were extrapolated for individual actors by converting the timing to microsecond (μ) and then dividing by the number of SEFs.

Actor	**CPU execution timing (μ)**	**GPU execution timing (μ)**	**FPGA execution timing (μ)**
PV	6	∞	∞
PP	19	200	∞
PS	6	21	∞
PI	975	256	∞
OV	850	225	1080
OP	125	213	∞
OI	6	56	∞
IAT	13	75	∞
UV	13	38	∞
UP	31	63	∞
PAT	13	69	∞
COM	6	62	∞
PE	2	∞	∞
OS	44	44	∞
NV	2	∞	∞
NX	2	∞	∞
US	2	∞	∞
NS	2	13	∞
SUMX	2	∞	∞
SUMI	2	∞	∞
FLR	349	∞	∞

Table 7.6: A summary of the throughput of the published hand crafted implementations of CACTuS configuration *C1*.

Publication	**Published Throughput (fps)**	**CACTuS Configuration**	**Compute engines used**
Milton [129]	60.98	C1 (16 SEF, I = 127 $\times$127)	CPU:1, GPU:1, FPGA:1
Webb [130]	41.25	C1 (16 SEF I = 127$\times$ 127)	CPU:1, GPU:1

assumed to be negligible for this dataset. This assumption is compatible with AMS, due the implementation of one actor at a time on a compute engine. The reason, as one actor executes on a compute engine, there are no communication queues, so the assumption of negligible communication delays when they are added in the actor execution timings will be valid for AMS.

The dataset that is collected as a part of this research are obtained by measuring the timings of individual actors and by profiling their resource usages on the CPU and the GPU. The actors with a GPU implementation and the multi-threaded CPU implementation were executed separately by varying the input token sizes based on the CACTuS configuration to measure their resource usages. GPU resource usages were measured by CUDA profiler [131]. For the CPU, the resource consumption is an approximation of its individual core usage, which is calculated as the average over several executions. For the GPU, a range of resources were considered, which includes threads per block, registers per thread and shared memory. CUDA profiler showed that for *C1*, *C2* and *C3*, the usages of these resources were very low. However, performance deteriorated when more that four streams with a combination of actors from the afore-mentioned SEF configuration were executed. This led to the assumption that the actors on a GPU consumes 12.5% of its resources. For configuration *C4* and *C5* performance deteriorated with two streams, so it is assumed that 50% of GPU resources are consumed by the actors of those configurations.

The actor execution timings were measured by recording the timings before and after an actor's execution. For this purpose CACTuS was implemented on a single thread and without streams, so that one actor executes at a time on a compute engine. Communication delays between CPU and GPU for various channels were calculated by varying the token sizes. A dummy kernel on the GPU was implement for the measurement of the communication delay. From the execution timings in table 7.3 it can be seen that not every actor has been benchmarked on each compute engine. If an actor is not implemented it's execution timing is taken as infinite (a very large number).

Another published hand crafted implementation of CACTuS by Webb [130] has been used along with Milton's datasets for the comparison of performance predicted by the AMS algorithm and the published results. However, Webb has not published individual actor performance and resource usage data. To overcome this handicap, data collected as part of this research is used for the comparison, as the compute engines used by Webb are identical to the ones used in this research. The summary of throughput in frames per second (fps) of both the published hand crafted implementations of CACTuS is shown in table 7.6.

7.2.3 Conclusion

In this section the CACTuS application-algorithm is introduced. An SDF model of CACTuS is presented, which reveals its task and data parallelisms. The compute performance and the resource consumption of actors as implemented by Milton and as measured as part of this research work are described.

Table 7.7: This table compares two published hand-crafted implementations of CACTuS with the predicted performance from the AMS algorithm. Since the performance metric of AMS is makespan, throughput is calculated as 1 by makespan.

Publication	Published Throughput (fps)	AMS Predicted throughput (fps)	CACTuS Configuration	Compute engines used
Milton [129]	60.98	61.4	*C1*	CPU:1, GPU:1, FPGA:1
Milton [129]	17.30	17.2	*C2*	CPU:1, GPU:1, FPGA:1
Webb [130]	41.25	47.70	*C1*	CPU:1, GPU:1

7.3 The CACTuS application with AhcFlow: comparison with published results

An important consideration for testing the validity of the AMS algorithm is to compare the performance achieved with it as compared with the published hand-crafted implementations. In this section, the predictions made by the AMS algorithms with the measured performances of Milton [129] and Webb [130] are presented. Milton has used two platform architectures; one consisting of a CPU, a GPU and an FPGA and the other consisting of a CPU and a GPU. Whereas, Webb has used a fixed platform architecture consisting of a CPU and a GPU.

Milton's implementation allows only a single actor to execute on a compute engine at a time. To match this decision using the AMS algorithm which allows concurrent execution of actors on a single compute, the resource usage of actors were artificially constrained by assigning the actor resource usage data to be 100%, so that concurrent actor executions are not considered. This restriction is however removed for the comparison with Webb's implementation and the implementation developed in this work.

For the comparison with Milton, his actor timing data were used. As noted above 100% resource consumption with Milton's data were used. Detailed actor timing data are not reported by Webb. However, separate experiments were performed to create actor timing and resource consumption to allow the comparison with Webb.

Table 7.7 summarises the results of the comparison. The throughput predictions made by the AMS algorithm closely match the performance achieved Milton and Webb. The throughput of Milton's implementation are a closer match as compared with Webb's implementation. This may be expected because Milton's implementation doesn't have complex concurrent scheduling.

The details of the mapping and scheduling decisions that are made by AMS compared to Webb's implementation are shown in Figures 7.4 and 7.5 and in table 7.8. Figure 7.4 shows the time-line of actor executions on the CPU and the GPU and data transmission activity on the link (L1) between CPU and the GPU. A coloured

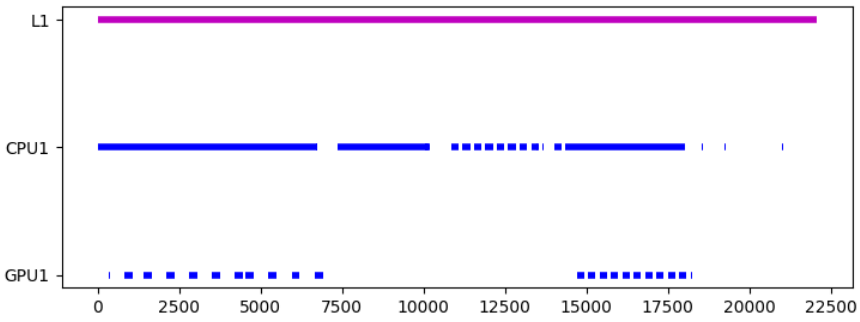


Figure 7.4: Occupancy of the CPU and the GPU by the CACTuS actors configuration *C1*.

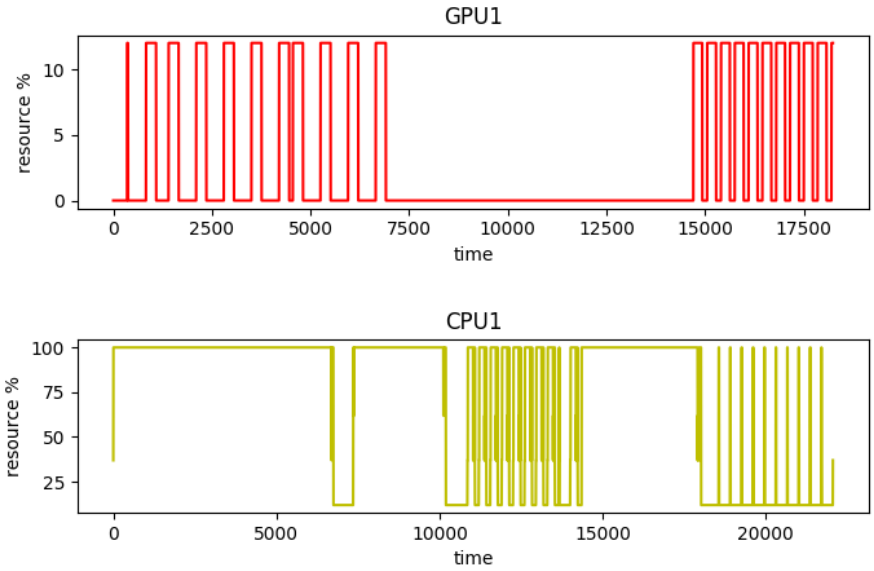


Figure 7.5: Resource usage of CACTuS actors configuration *C1* on the CPU and the GPU.

part of the time-line in Figure 7.4 indicates an actor executing on a compute engine but doesn't indicate the resource consumption of the actor on that compute engine. Figure 7.5 provides the same information as 7.4 but in addition has resource consumption. Table 7.8 provides the corresponding mapping of actors on to each compute engine.

In this section the study is limited to 16 SEFs with 127 × 127 pixel images on just two platform architectures. In the next section, larger images and higher numbed SEFs are considered on an agile heterogeneous platform that can consist of up to 8 GPUs.

Table 7.8: Mapping decisions of the actors for CACTuS configuration *C1*. The actors that are not mentioned were mapped to on the CPU.

Actors	**Total**	**Number mapped on the CPU**	**Number mapped on the GPU**
Predict Shape	16	15	1
Predict Image	16	6	10
Update Velocity	16	15	1
Observe Velocity	16	6	10

7.4 Design space exploration for the CACTuS application

In this section, the utility of the AMS algorithm is illustrated through a series of studies to explore the design space for the CACTuS application-algorithm. As previously noted, CACTuS can be scaled by introducing additional shape estimating filters (SEF) and by increasing the image size. The study in this section illustrates the agile architecture exploration capabilities of the AMS algorithm.

Table 7.10 shows the design space exploration studies that are presented in this section. Five different CACTuS configurations representing dissimilar combinations of SEFs and Image sizes (see table 7.1) are used in this design space exploration study. The platform constraints consists of one Intel i7 quad-core CPU and a maximum of 8 GTX 960 GPUs. It is assumed that the GPUs can have direct GPU to GPU communication links with 3 other GPUs. Since these direct links are much faster than the usual PCIe [132], it is assumed that the communication delay of the channels are one fifth of their estimated delay on PCIe links. It is acknowledged that the GTX 960 GPU doesn't support GPU to GPU links. However, these links are becoming available in more recent GPUs like in NVLINK [28]. The table 7.10 shows nine different capital cost budget points that were used in the AMS algorithm for each CACTuS configuration. The AMS algorithm explores many different platform architectures but outputs the one with the lowest makespan, for each budget. However, the three lowest makespan are reported here. The hyper-parameters for AMS used for this exploration are in table 7.9.

Because of the large number of cases involved in the design space exploration studies (5 studies by 9 budgets per study) the results will be presented in an overview first, then some interesting cases will be presented in more detail. This section is thus organised into two subsections. The first subsection presents the overview of the design space exploration. Then, the second subsection describes few interesting platform architectures and the mapping and scheduling decisions with resource usage details.

Table 7.9: The hyper-parameters of the AMS algorithm for the design space exploration of CACTuS.

Name	**Denotation**	Value
Resource factor	RF	2.4
Mapping threshold	MT	15
Local factor	ce_{local}	7
Global factor	ce_{global}	2
Communication factor	ce_{comm}	2

Table 7.10: The design space exploration studies showing the CACTuS configurations, overview of the platform constraints (PPG tier 1) and the budget points taken into account.

Study No.	**CACTuS configuration from table T.a**	**Platform constraints (PPG tier1)**	**Budget Points (Capital cost units)**
1	C1 (16 SEFs, I = 127×127)	1 CPU (cost: 100, max up to GPU: 8) 8 GPUs (unit cost: 200, max direct connections: 3)	100 (CPU) 300 (CPU, GPU: 1) 500 (CPU, GPU: 2) 700 (CPU, GPU: 3) 900 (CPU, GPU: 4) 1100 (CPU, GPU: 5) 1300 (CPU, GPU: 6) 1500 (CPU, GPU: 7) 1700 (CPU, GPU: 8)
2	C2 (16 SEFs, I = 511×511)		
3	C3 (64 SEFs, I = 511×511)		
4	C4 (64 SEFs, I = 1023×1023)		
5	C5 (256 SEFs, I = 1023×1023)		

7.4.1 Exploration results overview

Figure 7.6 is a scatter plot (Pareto diagram) that shows all the platform architecture instances for each configuration of CACTuS (*C1*, *C2*, *C3*, *C4* and *C5*) described in table 7.1. The y-axis is the schedule length ratio (SLR) as a metric of performance (equation 6.5). The x-axis is the capital cost of a platform architecture instance, which is the sum of constituent compute engines' capital cost. In order to distinguish the plots of different CACTuS configurations, the plots are of different colour and shape. There are multiple plots with the same colour and capital costs representing platform architecture instances with the same number for GPUs. They have different connection topologies due to different GPU to GPU connections.

The key observations from the platform architectures formed by AMS as shown in Figure 7.6 are:

1. **Major initial performance improvement:** The makespan value drops dramatically for the initial few GPU additions. This observation is common across all the configurations of CACTuS. For example, in *C3* the makespan reduces from 141.53 to 86.76 with the increase of capital cost just by 200, which is due to the addition of only one GPU. However, this significant drop of makespan is not continued with further addition of GPUs beyond three or four. This can be attributed to the actors that must be mapped on CPU, or whose execution time on the CPU is lower than the GPU. Also, with more GPUs the communication delays continue to increase.

2. **Connectivity topology impact:** The differences in makespan on the different platform architecture instances of the same capital cost are greater for *C3*. These differences are attributed to different GPU to GPU connections. In the next subsection, further details are investigated on how direct connections improve performance for certain configurations of CACTuS.

3. **Configuration with larger image size yields better performance:** The final observation from Figure 7.6 is that the initial performance improvements as compared to CPU only are greater when the image sizes are larger. It can be seen that *C4*, which is frame size 511×511 starts with an SLR of 40.16 and it drops to 10.55 with the addition of just one GPU. Whereas, *C1* only drops from 45.88 to 30.46 with the addition of one GPU. This can be attributed to better acceleration by GPU for larger images, as compared to smaller images.

Since diversity of makespan with various collectivity topologies is more for the *C3* configuration, two platform architecture instances with the same capital cost of 900, which is 4 GPUs, are examined in the next subsection. Along with these two platform architecture instances, another two instance for *C3* with just one GPU and 7 GPUs are also examined to gain insight into the dramatic drop in makespan

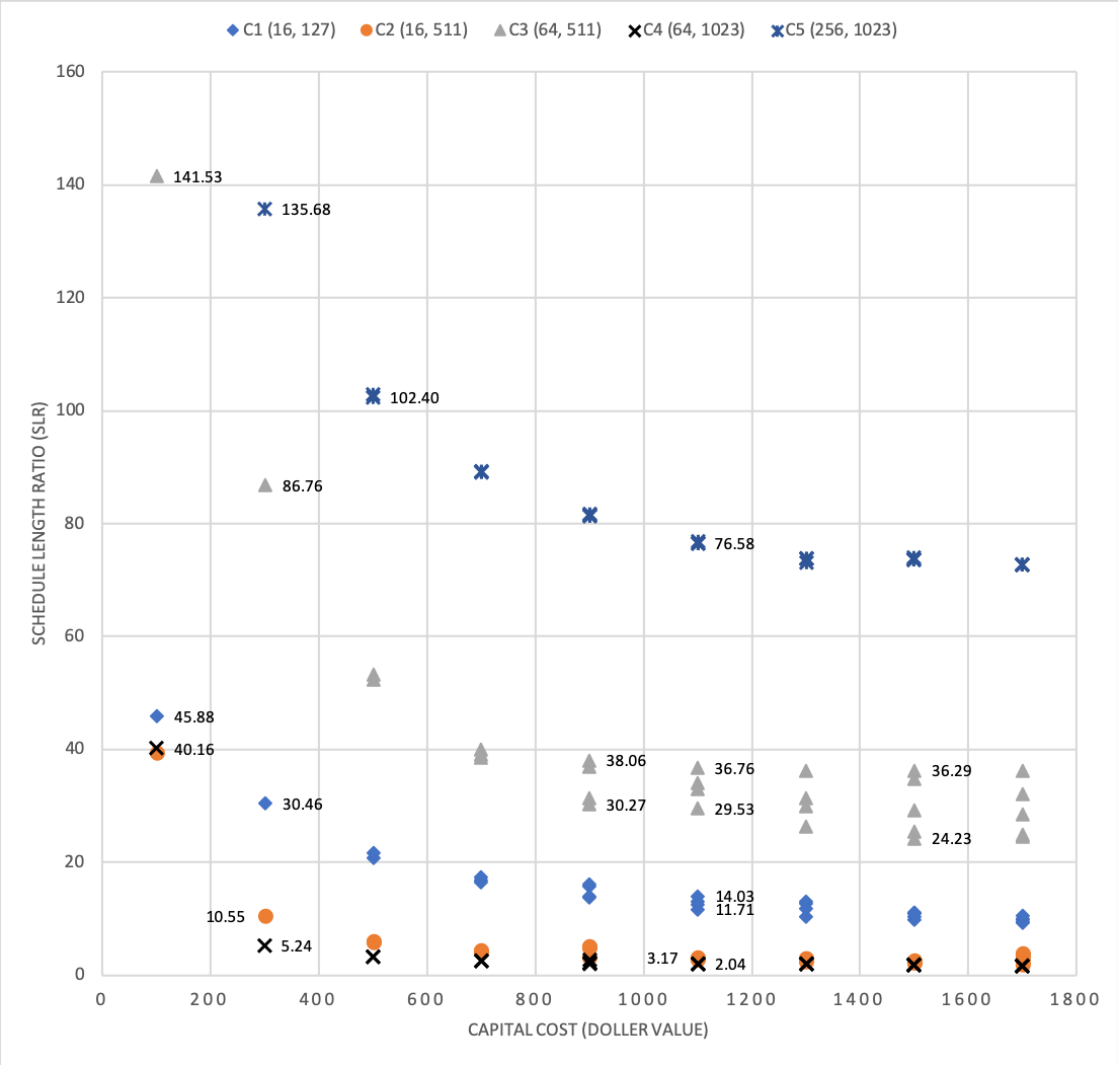


Figure 7.6: Overview of the CACTuS design space exploration studies. Five different configurations are considered with nine capital budget points. These configurations are different combinations of SEF numbers and data token sizes, which are detailed in table 7.1. Y-axis represents the schedule length ratio (SLR) (equation 6.5 in section 6.3.2) and the x-axis represents the capital cost. For configuration *C5*, the makespan on the platform architecture with just the CPU is 760.78. Because of this large value, this particular makespan is not shown in the figure.

from 86.76 to 30.27, as compared to 24.23 with another 3 GPUs (total 7) being further added. A platform architecture instance for *C1* is also examined in the next subsection as a comparison with a configuration of smaller data token (image) sizes.

7.4.2 Resource usage with mapping and scheduling decisions

In order to gain further insights to the design space exploration of CACTuS, five cases that are listed in table 7.11 are examined here. A case is an platform architecture instance that the AMS algorithm has explored. For each of these cases, the occupancy of the compute engines, resource usages and the actor distribution on the compute engines are studied. The first case is of CACTuS configuration *C1* and the rest of the cases are from *C3*. The C3 configuration is focused here because it's design space exploration shows larger differences in SLR than the other configurations.

Table 7.11: The platform platform architecture instances that are discussed in detailed. These platform architecture instances were explored by AMS for CACTuS.

Case No.	**CACTuS Config**	**Architecture Instance**	**Capital cost**	**SLR**
1.	C1	GPU: 4, CPU:1 No direct connection	900	17.64
2.	*C3*	GPU: 1, CPU:1 No direct connection	300	86.76
3.	*C3*	GPU: 4, CPU:1 No direct connection	900	38.06
4.	*C3*	GPU: 4, CPU:1 6 direct connections	900	30.27
5.	*C3*	GPU: 7, CPU:1 No direct connection	1500	36.29

Case 1: CACTuS configuration *C1* on a platform of 4 GPU and 1 CPU with no direct connection amongst the GPUs

In this first case, it can be seen from table 7.12 that two new actors *Update Position* (UP) and *Observe Position* (OP) are mapped on some of the GPUs. They are new since actors were not considered for GPU with an architecture of one GPU (see section 7.3). From the compute engine occupancy in Figure 7.7 and the resource usage Figure 7.8, it can be seen that the CPU usage drops after around 2500 μs. This is due to the mapping of more actors on the GPUs and also due to the distribution of communication tasks to the concurrent links. Another observation is that load amongst the GPUs are reasonably even, but they are not utilised much.

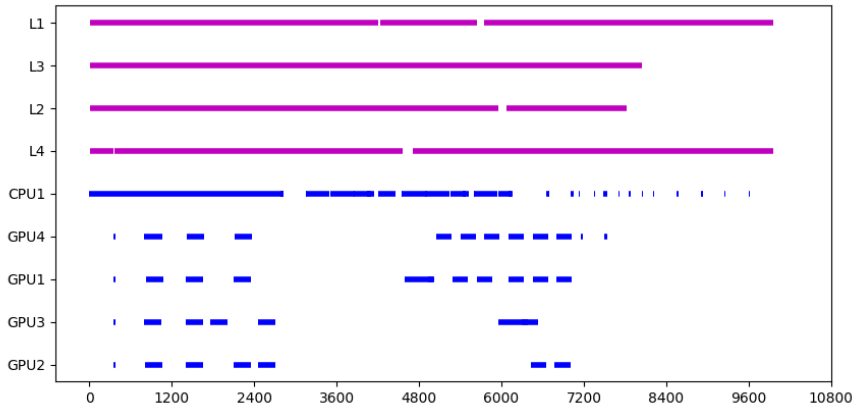


Figure 7.7: Occupancy of the CPU and 4 GPUs by the CACTuS C1 actors.

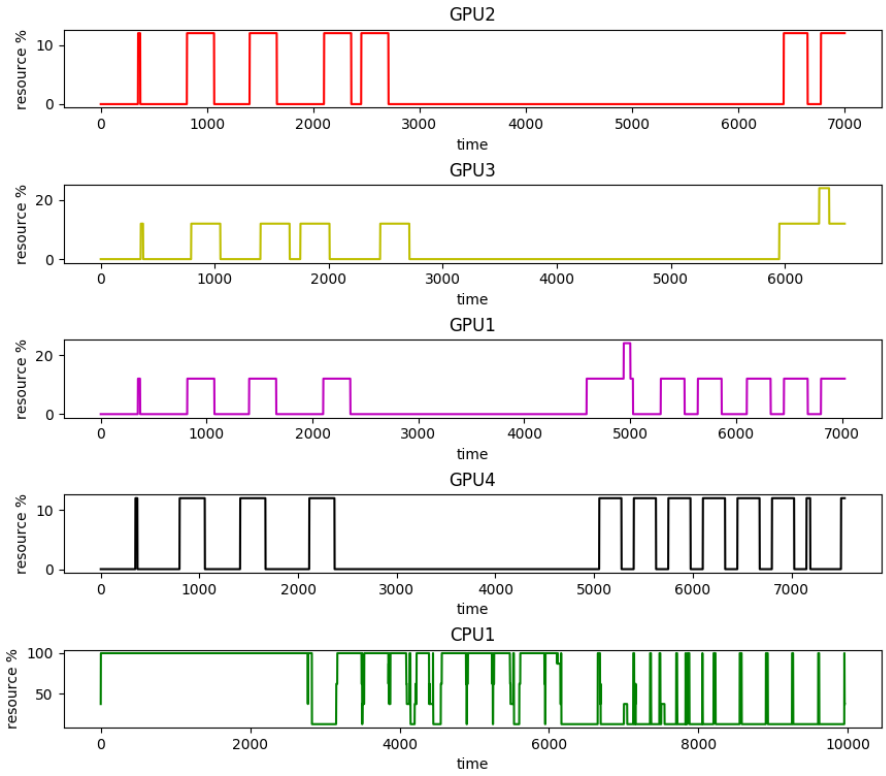


Figure 7.8: Resource usage of CACTuS C1 actors on the CPU and 4 GPUs with no direct link (Case 1).

Table 7.12: Mapping decisions of the actors for CACTuS configuration C1 on the CPU and 4 GPUs (Case 1). Only those kind of actors that are mapped on GPUs are listed here. The actors that are not listed here are 100% mapped on the CPU.

Actors	**CPU**	**GPU1**	**GPU2**	**GPU3**	**GPU4**
PS	75	6.25	6.25	6.25	6.25
PI	12.5	18.75	25	25	18.75
UV	87.5	0	0	0	12.5
UP	93.75	6.25	0	0	0
OV	0	37.5	12.5	12.5	37.5
OP	87.5	6.25	0	6.25	0

Case 2: CACTuS configuration *C3* on a platform of 1 GPU and 1 CPU

Since this is CACTuS *C3*, there are 64 SEFs of 1155 actors, which is a large number of actors and the sizes of images (data tokens) are also bigger than the previous case. It can be seen from table 7.13 that more actors are added on the GPU than with CACTuS *C1*. This may be due to the better acceleration of the actors on GPU as compared to smaller images of *C1*. The GPU usage from Figures 7.9 and 7.10 shows that it is fairly occupied and the maximum resource usage has reached more than 20% of the GPU resources. It will be interesting to see how the actors are distributed with more GPUs.

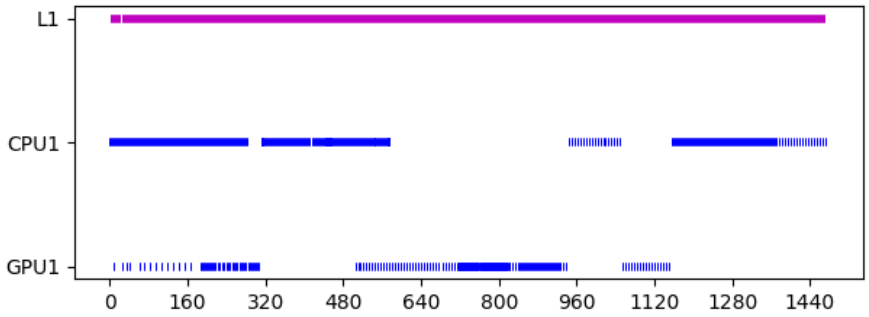


Figure 7.9: Occupancy of the CPU and 1 GPU by the CACTuS C3 actors (Case 2).

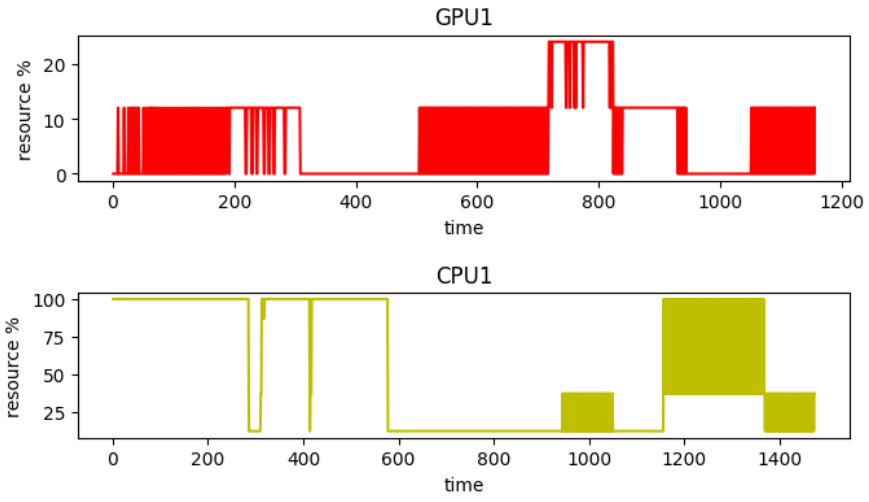


Figure 7.10: Resource usage of CACTuS C3 actors on the CPU and 1 GPU (Case 2).

Table 7.13: Mapping decisions of the actors for CACTuS configuration C2 on the CPU and 1 GPU (Case 2). Only those kind of actors that are mapped on GPUs are listed here. The actors that are not listed here are 100% mapped on the CPU.

Actors	**CPU**	**GPU1**
PAT	30	70
COM	17.2	82.8
PP	30	70
PI	17.2	82.8
IAT	17.2	82.8
OI	17.2	82.8
UP	17.2	82.8
OV	17.2	82.8
OS	17.2	82.8
OP	17.2	82.8

Case 3: CACTuS configuration *C3* on a platform of 4 GPUs and 1 CPU with no direct connections amongst the GPUs.

From the actor distribution table 7.14 it can be seen that although there has been no new actor type being mapped on a GPU, all the same actor types that were mapped in case 2 are now almost completely mapped on one of the 4 GPUs, with the exception of *Observe Shape* (OS), where only 9.38% are still mapped on the CPU. Thus, more new actors have moved to GPU. Also, the communication load is shared amongst four links that executes them concurrently, as seen in Figures 7.11 and 7.12. This reveals the reason for the drastic reduction of SLR due to the addition of the initial GPUs. Furthermore, the resource usage Figure 7.12 shows that the GPUs are load balanced uniformly.

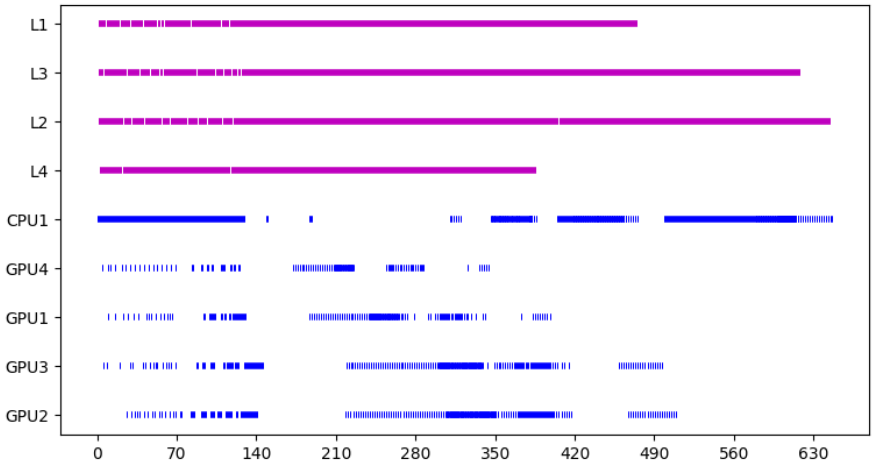


Figure 7.11: Occupancy of the CPU and 4 GPU no direct links by the CACTuS C3 actors (Case 3).

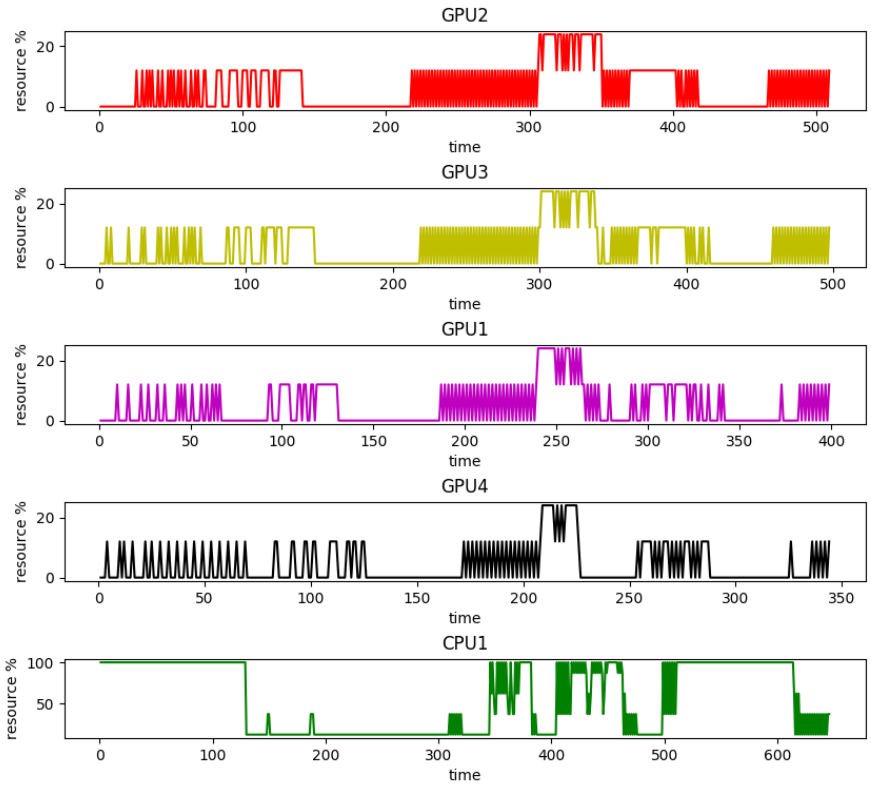


Figure 7.12: Resource usage of CACTuS C3 actors on the CPU and 4 GPU with no direct links (Case 3).

Table 7.14: Mapping decisions of the actors for CACTuS configuration C3 on the CPU and 4 GPUs with no direct links (Case 3). Only those kind of actors that are mapped on GPUs are listed here. The actors that are not listed here are 100% mapped on the CPU.

Actors	**CPU**	**GPU1**	**GPU2**	**GPU3**	**GPU4**
PAT	0	25	25	25	25
COM	0	20.31	37.5	32.81	9.37
PP	0	23.44	25	25	26.56
PI	0	20.3	34.37	31.25	14.06
IAT	0	20.31	34.38	31.25	14.06
OI	0	20.31	34.38	31.25	14.06
UP	0	20.31	37.5	32.81	9.37
OV	0	20.31	34.38	31.25	14.06
OS	9.38	15.63	34.38	31.25	9.37
OP	0	20.31	34.38	31.25	14.06

Case 4: CACTuS configuration *C3* on a platform of 4 GPUs and 1 CPU with six direct connections (all-to-all) amongst the GPUs.

The actor distribution table 7.15 shows that the actors are distributed in the same way as it was for the previous case. Also the pattern in which the resources shown in Figure 7.14 are used is similar to the previous case. The performance gain can thus be attributed only to the direct communications between the GPUs. The usage of the direct connections (D1, D2, D3, D4, D5, and D6) are shown to be used in Figure 7.13. However, this performance gain is not significant as compared to the initial GPU addition improvements.

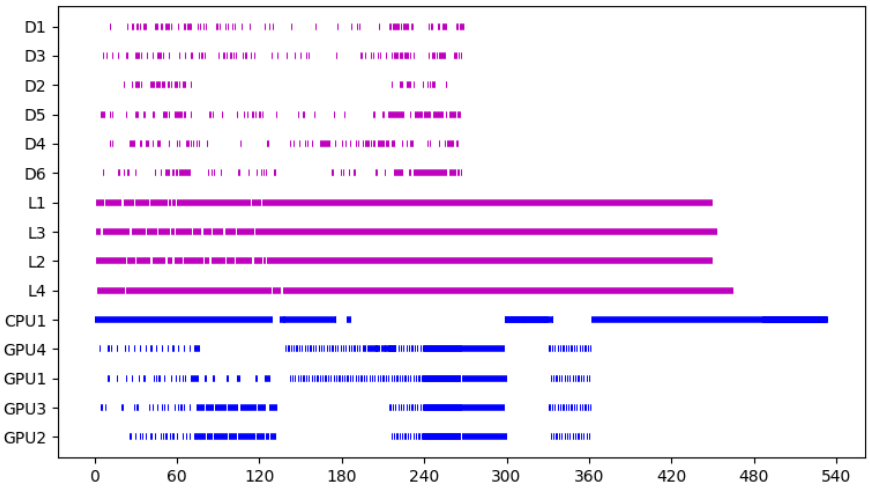


Figure 7.13: Occupancy of the CPU and 4 GPU all connected by the CACTuS C3 actors (Case 4).

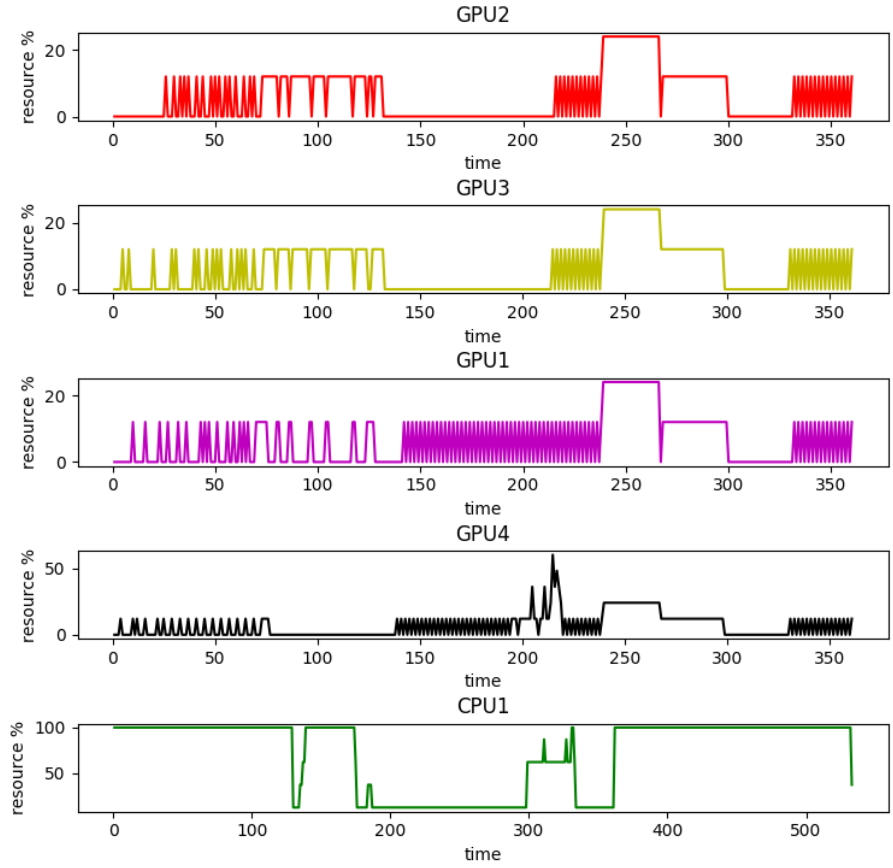


Figure 7.14: Resource usage of CACTuS C3 actors on the CPU and 4 GPU all connected (Case 4).

Table 7.15: Mapping decisions of the actors for CACTuS configuration C3 on the CPU and 4 GPUs with no direct links (Case 4). Only those kind of actors that are mapped on GPUs are listed here. The actors that are not listed here are 100% mapped on the CPU.

Actors	**CPU**	**GPU1**	**GPU2**	**GPU3**	**GPU4**
PAT	0	25	25	25	25
COM	0	20.31	37.5	32.81	9.37
PP	0	23.44	25	25	26.56
PI	0	20.3	34.37	31.25	14.06
IAT	0	20.31	34.38	31.25	14.06
OI	0	20.31	34.38	31.25	14.06
UP	0	20.31	37.5	32.81	9.37
OV	0	20.31	34.38	31.25	14.06
OS	9.38	15.63	34.38	31.25	9.37
OP	0	20.31	34.38	31.25	14.06

Case 5: CACTuS configuration *C3* on a platform of 7 GPUs and 1 CPU with no direct connections amongst the GPUs.

It is interesting to see that with 7 GPUs there is a reduction in the actors being mapped on GPUs but there is only a slight improvement in performance as compared to the architecture instance of 4 GPUs. The actor distribution table 7.16 shows the reduction in the actors being mapped on GPUs. The improvement in performance (lowering of SLR) is due to the sharing of extra communication links, which freed more CPU time, allowing actors that has low acceleration on GPU to be mapped on CPU. The actor occupancy Figure 7.15 shows more CPU usage while concurrent communication links are busy. Resource usage Figure 7.16 on the other hand shows that some of the GPUs are barely used. Therefore, this case explains why addition of further GPUs (beyond 3 or 4 GPUs) only improves performance slightly.

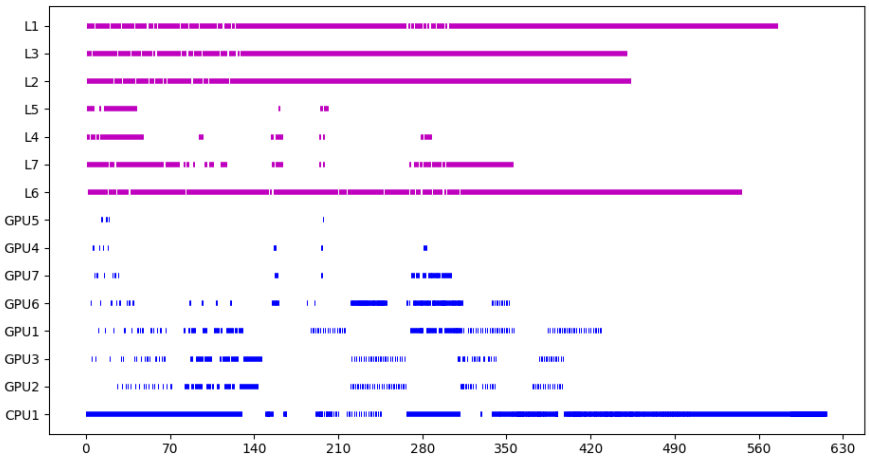


Figure 7.15: Occupancy of the CPU and 7 GPU with no direct links by the CACTuS C3 actors (Case 5).

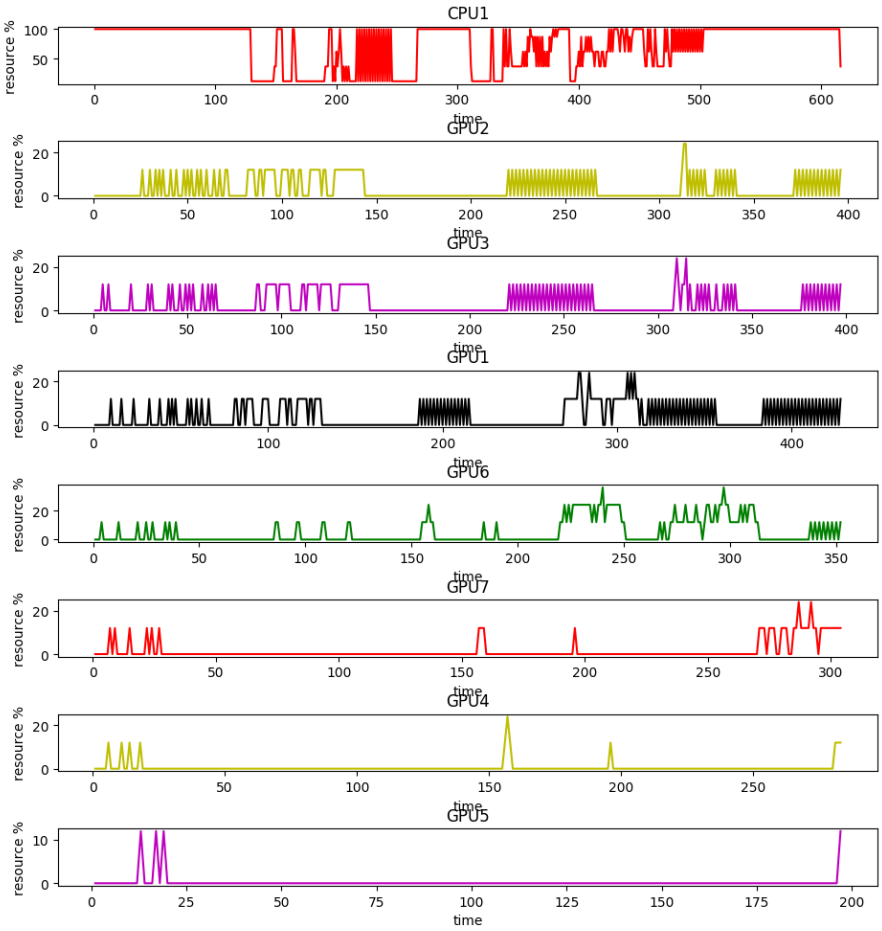


Figure 7.16: Resource usage of CACTuS C3 actors on the CPU and 7 GPU with no direct link (Case 5).

Table 7.16: The percentage of actors mapped on the compute engines of the platform architecture (Case 5). Only those kind of actors that are mapped on GPUs are listed here. The actors that are not listed here are 100% mapped on the CPU.

Actors	**CPU**	**GPU1**	**GPU2**	**GPU3**	**GPU4**	**GPU5**	**GPU6**	**GPU7**
PAT	96.88	0	0	0	1.56	1.56	0	0
COM	3.16	20.31	17.19	35.93	21.88	0	0	1.56
PP	0	23.44	23.44	20.31	12.5	9.38	6.25	4.69
PI	0	35.94	34.38	23.44	6.25	0	0	0
IAT	6.25	35.94	34.38	23.44	0	0	0	0
OI	95.31	1.56	0	1.56	0	0	0	0
UP	1.56	20.31	18.75	35.94	21.88	0	1.56	0
OV	0	1.56	3.13	21.88	53.13	17.18	3.13	0
OP	0	1.56	3.12	21.88	53.12	17.19	3.13	0

Summary

From the compute engines resource usages of the first case and the rest of the cases, it is inferred that GPU resources are better utilised with larger data token (image size). The larger resource usage is due to more concurrent actor executions on GPU. This can be seen from the actor distribution tables, where the critical actors like Observe Velocity (OV), Predict Image (PI) and few more actors are completely mapped on the GPUs. Whereas, for the first case these critical actors are shared amongst CPU and the GPUs. This is due to higher communication delay with the GPU mapped actors that will not be compensated by the acceleration gained on the GPUs. This explains the reason for better performance improvements with larger image sizes. On the other hand, with the addition of more GPUs like four in total, although there are not a lot of new actors mapped on these GPUs, performance improves drastically due to the sharing of the communication tasks amongst the available concurrent links. An important observation is that in spite of the direct links help, performance is not improved much[3]. Performances are also not improved much with further addition of GPU, like in the last case, some the additional GPUs are rarely used.

7.4.3 Conclusion

This section presented the design space exploration results of five different configurations of CACTuS with varying SEF numbers and image sizes. The platform architecture constraints allow up to 8 GPUs which are capable of direct communications links and one CPU. This exploration showed the utility of the AMS algorithm to study the impact of direct communication links with the expansion of the platform architecture for better performance of CACTuS with different configurations. The design space exploration showed the following:

- Performance improves greatly with the addition of first few GPUs. The number of few GPUs depends upon the CACTuS configuration. However, after the initial GPUs, the improvement in performance reduces with further addition of GPUs.

- With larger image sizes, GPU resources are better used due to possibilities of concurrent actor executions, as the communication link penalty is compensated with the acceleration gained on GPU. Due to this phenomena, the performance improvement with the initial GPUs are much greater for larger image sizes.

- Direct connections improve performance but they are not as significant as

[3]GPUs with direct links are currently much more expensive. This extra cost is not reflected in the cost metric of the study. If the extra cost was considered the performance advantage would be overwhelmed by the extra capital cost.

compared with an addition of the initial GPUs. Further, the improvement of performance due to the direct connections is more profound with only configuration *C3* of CACTuS.

In the next section the deployment that follows design space exploration is described.

7.5 Application deployment within AhcFlow

In this section, the deployment technique is presented. It transforms the ArcSDF representation output from the AMS algorithm to create a runtime image for the heterogeneous platform. An overview of the deployment task has been detailed in section 4.4. In summary, the deployer creates a runtime image of the application-algorithm from the pre-engineered components of the actors and the communication channels. Vendor specific tool-flows are necessary to support deployment. A complex software and hardware infrastructure is necessary to achieve this. For this reason, the deployment details described here are limited to CPUs and GPUs only. However, there is no inherent reason why deployment cannot be extended to FPGA based compute engines. To assist in the exposition of the deployment technique, a simple ArcSDF representation example with 6 actors on to a platform architecture consisting of one CPU and one GPU is used. After describing the deployment technique with the simple example, a version of 16 SEF CACTuS is deployed, which is then verified as correct based on tracking accuracy.

This section is organised into five subsections. In the first subsection, an overview of the deployment technique and the example that is used to describe the deployment technique are described. The overview of the deployment technique introduces the following three key steps: (1) parsing and validation, (2) skeleton code generation and (3) injection of pre-engineered actors and launch. Each of these steps are detailed individually in the subsequent three subsections. Finally, in the fifth subsection, the deployment of CACTuS with a configuration of 16 SEFs is presented.

7.5.1 Deployment technique overview

The deployment technique entails reading of the ArcSDF model to create an ArcSDF C++ object, which is then combined with pre-engineered components to create the runtime. The pre-engineered components are written to follow an application programmers interface. This API is used to realise the constructs of the ArcSDF representation, which includes *compute zones*, *interfaces*, *control actors* and *resource edges*. The deployer and its internal components are shown in Figure 7.17. There are three steps involved in the deployment and they are: parsing and validation, skeleton code generation and lunching after pre-engineered actor injection.

A simple application example targeted on a heterogeneous platform constituted of one GPU and one CPU is used to illustrate the deployment technique. The example application detects persons from a scene by running concurrent correlation functions. The scene is divided into four equal parts to concurrently run the correlation functions. Bounding boxes are then drawn over the peaks generated by

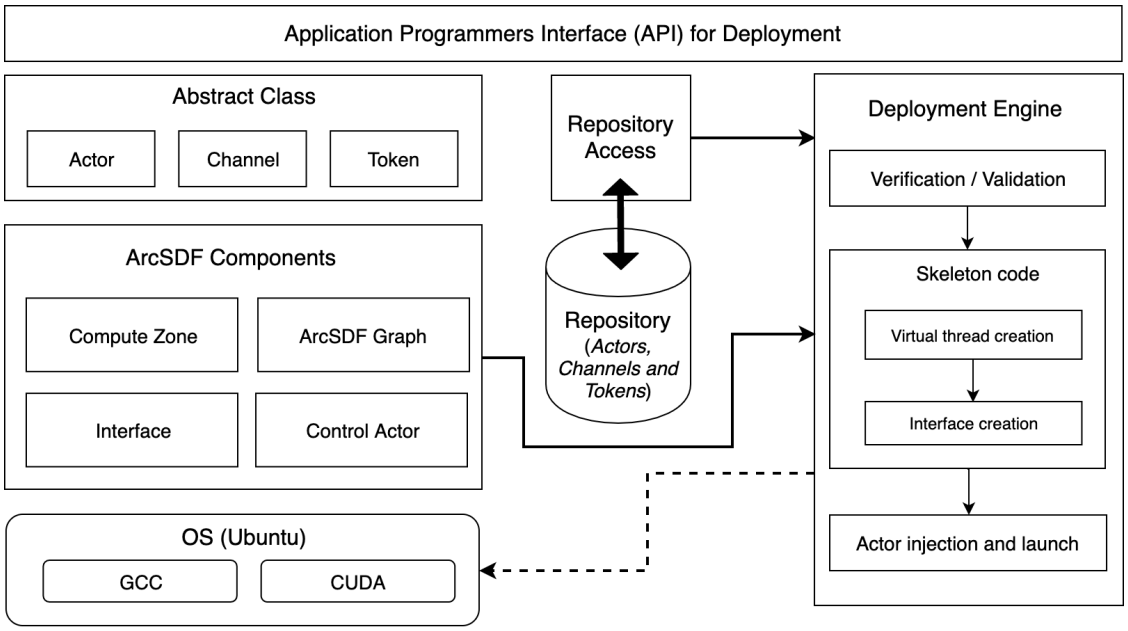


Figure 7.17: This figure illustrates the structure of the deployment module of the AhcFlow design flow.

the correlation function. The final step is stitching the scene together for display. The SDF graph of this example and its equivalent DAG is shown in Figure 7.18(a) and (b), respectively. The platform architecture instance to deploy this application is shown in Figure 7.18(c). An ArcSDF graph for this application and the target platform architecture, which contains the architectural decisions is shown in Figure 7.19. This ArcSDF example will be used to describe all the steps of the deployment. Recall that the AMS algorithms generates the ArcSDF model in the form of a JSON [133] data structure, which the main input for the deployer. A JSON structure snippet representing for example ArcSDF graph is shown in Figure 7.20.

The SDF model of the example application consists of six actors. The *frame source* (f) actor reads the video frames of size 288 $\times$ 384 from a camera, which is then segmented by the *segmentation* (s) actor to split it into four equal parts of size 144 $\times$ 192. Each of these parts of the frame are processed concurrently to detect persons. This is performed by the *correlation* (c) function, which essentially runs a correlation function with a kernel of a walking person. The result is a set of coordinates. Since correlation is an computationally expensive function, the video frames are divided into 4 parts and each part is executed in parallel. After the peaks are found, the images are stitched back by the *merge* (m) actor, which is stored to the disk by the *display* (e) actor.

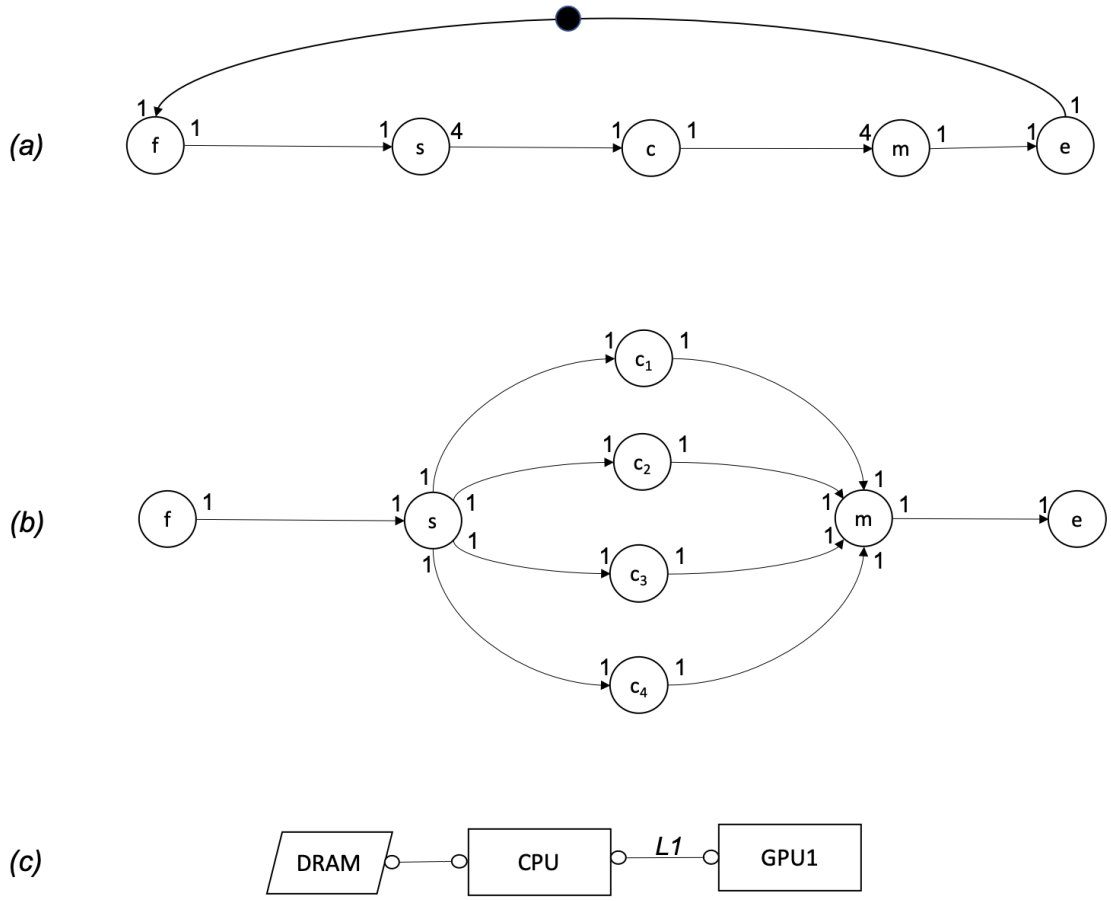


Figure 7.18: This figure illustrates the example used to describe the deployment technique. The sub-images are: (a) the application SDF graph, (b) equivalent DAG and (c) platform architecture instance.

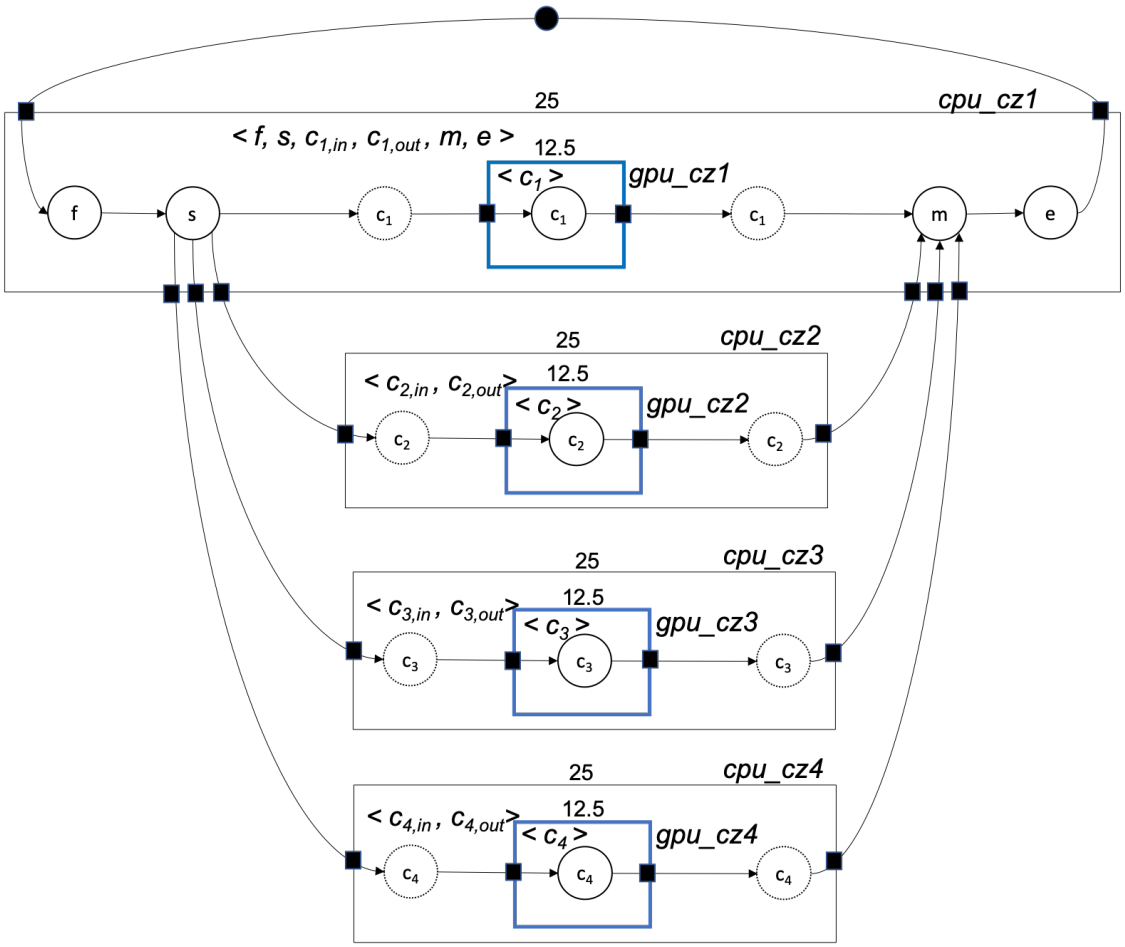


Figure 7.19: The ArcSDF example to explain the deployment technique. The SDF graph of the example and its equivalent DAG is shown in Figure 7.18

Figure 7.20: Snippet of the JSON file of the ArcSDF representation which is shown in Figure 7.19.

7.5.2 Parsing and Validation

In this subsection, the first step in the deployment is described. The ArcSDF representation for deployment is first analysed to check that the compute engines, where the actors are placed, exists and the pre-engineered actors and the channels supporting each compute engines are present in the repository. A second validation check establishes that it is a valid ArcSDF graph and the maximum resources required will not exceed the resources of the existing platform architecture. This validation ensures that the model is deployable. As the ArcSDF graph is validated and tested for compatibility on the platform, it is parsed, which brings to the next stage of creating the internal ArcSDF skeleton object that will drive the deployment.

In Figure 7.20, the example ArcSDF JSON file illustrates the identification of the compute engines, actors and channels. Two compute engines, the CPU and the GPU (marked in red boxes) are identified and the validation layer finds if their types match with the one present in the existing platform architecture. Pre-engineered components, comprising the actors (marked in green) and the channels (marked in blue), are checked if they exists within the repository. Since actor C is mapped on the GPU, its GPU implementation existence is checked, whereas rest of the actor's CPU implementation presences are checked. Communication link types required by the channels are checked for their presence. In this example only PCIe connection is necessary.

After it is ensured that the ArcSDF representation is deployable, the ArcSDF graph is first checked for consistency. This is achieved by initially converting the ArcSDF graph to an equivalent SDF (using the algorithm presented in Figure 5.28). Then confirm the SDF graph's consistency by checking if the repetition vector is not equal to the null-vector. The concept of SDF consistency is explained in section 2.4.1. In the *deployer*, this check is performed by using the SDF3 tool [31]. After the consistency check, it is verified whether the ArcSDF graph is within the maximum resource usage limit of the platform architecture. This is conducted by running the maximum resource usage algorithm from section 5.3.3. After these checks, the deployment then progress to the next stage of creating the internal ArcSDF skeleton object.

7.5.3 Skeleton code generation

In this deployment stage, a skeleton ArcSDF object is created. It consists of the compute zones and their communication channels. The compute zones of the ArcSDF representation are transformed into virtual threads. Each virtual thread executes the schedule of it's compute zone infinitely. Some virtual threads end up as real threads on the CPU, whilst others will end up as a combination of a real-thread on the CPU and a stream on the GPU or equivalent on the FPGA. In

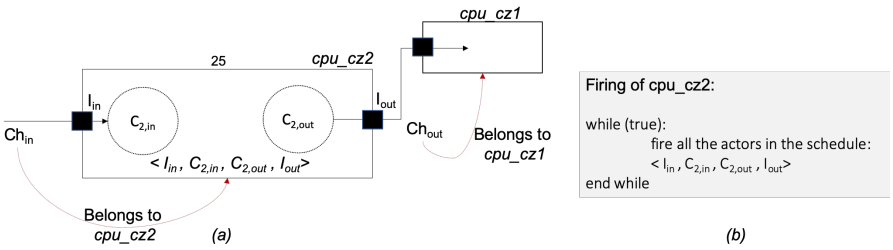


Figure 7.21: Details of how channels that are external to a compute are included within the skeleton object. The two parts of the figure are: (a) shows that input channels are included within the compute zone and (b) is the pseudocode to execute the actors within a compute zone sequentially that includes the interfaces.

the current version of the deployer, it is assumed that actors mapped to the FPGA and GPU are associated with control actors on the CPU. These virtual threads are ordered on the basis of their resource edges. The preceding compute zones are given a lower number, whereas the succeeding compute zones are given a higher number. These numbers are used later to order the launch of the virtual threads.

The skeleton refers to the structure of virtual threads without the internal pre-engineered actors and the channels being instantiated. The skeleton however includes the interfaces and the communication channels that are associated with an interface. Since the interfaces are for external communication that are concurrent, necessary communication infrastructure are created first before the instantiation of the other internal components that will execute in sequence (recall that the actors within a compute zone executes sequentially). External channels that are inputs to a compute zone are attached with it. Figure 7.21(a) illustrates the attachment of the external communication channels, which creates a self-contained concurrent block capable to communicating with other compute zones. It is noted that the interfaces are also include as a part of the schedule, as shown in Figure 7.21(b). A loop infinitely executes the schedule by first finding a list of fireable actors and interfaces. Then firing them one by one. The code snippet in Figure 7.22 shows a CPU virtual thread that infinitely loops the schedule of a compute zone. In this code snippet, compute zone is represented by a class named *Group*. The *fire()* method is executed infinitely during the life time of the application.

Highlighted sections of the example ArcSDF JSON file in Figure 7.20 shows the compute zones, interfaces and the selected channels that are of importance for the ArcSDF skeleton. The compute zones *cpu_cz1*, *cpu_cz2*, *cpu_cz3* and *cpu_cz4* are transformed to four CPU virtual threads, on the other hand, *gpu_cz1*, *gpu_cz2*, *gpu_cz3* and *gpu_cz4* are transformed to two CUDA streams. The virtual thread numbers associated with *cpu_cz1* and *cpu_cz2* are both 1 as there are no resource edges between them. There are no virtual thread numbers for *cz_gpu_1*, *cz_gpu_2*, *cz_gpu_3* and *cz_gpu_4* as they are CUDA streams. The example also shows an interface and the external channel with more details regarding the data token size of the channel. This information is used to instantiate a FIFO

```
320  void Group::fire(){
321
322      _temp.clear();
323      _temp = _totalActors;
324      while(_temp.size()){
325          auto fa = getFirableActor();
326  //      for(auto itr = fa.begin(); itr != fa.end(); ++itr){
327          if( fa.size()){
328              auto itr = fa.begin();
329              auto start = chrono::high_resolution_clock::now();
330
331              (*itr)->fire();
332
333              auto end = chrono::high_resolution_clock::now();
334              auto time = chrono::duration_cast<chrono::microseconds>(end-start).count();
335              cout << (*itr)->getName()<< '\t'<< (*itr)->getActorType()<< '\t'<< (*itr)->getComputeEngine().first
336                      << '\t' << (*itr)->getDomain() <<" Frame time taken is : " << time / 1000.0 << "\tmili" << en
337
338  //          cout  << fa.size() << '\t' <<  syscall(__NR_gettid) << '\t' << (*itr)->getName()<< " Fired, Frame Cou
339              _temp.erase(find(_temp.begin(), _temp.end(), (*itr)->getName()));
340          }
341          if(!_temp.size())
342              ++_counter;
343      }
344  }
```

Figure 7.22: The fire function of a CPU compute zone which is called in an infinite loop during the life cycle of the application. In line 325, *getFirableActor()* returns ready to fire actors ordered as the compute zone schedule.

channel with the required token type to connect compute zones. The actual actors and the actual channels internal to the virtual threads are next added, which is the next stage of deployment and are detailed in the following subsection.

7.5.4 Injection of pre-engineered actors and launch

After the formation of the ArcSDF skeleton object, it acts as a framework into which the pre-engineered actors and the communication channels are injected. Some optimisation occurs for communication channels between actors within the same compute zone. Once the pre-engineered code is being injected, the runtime is started by launching the real threads on the CPU. Correct synchronisation is achieved automatically due to the inherent qualities of ArcSDF.

The communication channel optimisation is achieved by replacing the channels that are internal to a CPU compute zone and is not connected to an interface with memory references. This avoids the necessity of expensive FIFO implementations. The blue box enclosing *Ch_f_0* in the example ArcSDF JSON (see Figure 7.20) shows an internal channel that can undergo the channel optimisation. This optimisation is followed by the actual actor injection, which is done by replacing the previous dummy actors with the actual pre-engineered actors.

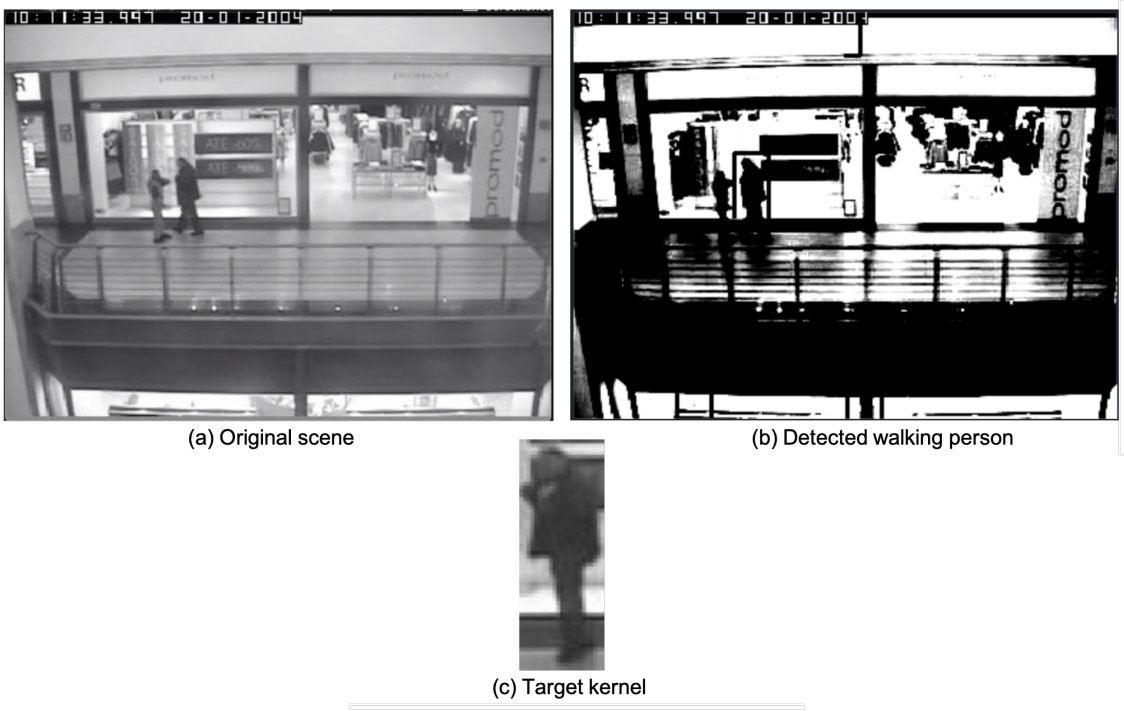


Figure 7.23: (a) The original shopping mall scene, (b) visual output from the detection example and (c) target kernel.

Injection of the actors is followed by the launch of the CPU threads that corresponds to virtual threads of CPU. The CPU threads are started in ascending order of the virtual thread number. The GPU and FPGA virtual threads are handled by one or more CPU threads. For instance in the afore-mentioned ArcSDF example, the virtual thread of *gpu_cz1* is realised as a CUDA stream, which is handled by a CPU thread that corresponds to *cpu_cz1*. There is a delay in which the threads are launched sequentially. This delay time allows each compute zone to occupy its required resources. The delay time is provided by the designer.

In the ArcSDF example, the CPU threads *th1*, *th2*, *th3* and *th4* are launched sequentially without any delay, as there are no resource edges used in this example. If there were a resource edge between *cpu_cz1*, *cpu_cz2* , *cpu_cz3* and *cpu_cz4*, then the first thread would have got more priority in occupying the resources than the later thread. To apply this priority the delay between the thread launches would have been applied. Further, streams *st1*, *st2* , *st3* and *st4* are assigned to the *th1*, *th2*, *th3* and *th4*, respectively. The actual realisation of a resource edge, which release resource after the completion of the compute zone is dependent on the underlying tools that govern the respective compute engines.

The launch of the example ArcSDF showed that the application was successfully deployed. This was seen visually by the detection of a person in the a shopping-mall. An instance of a person being detected is shown in Figure 7.23. The results of CACTuS deployment is described in the next subsection.

Table 7.17: Tracking performance of the deployment of CACTuS *C1*.

False Tracks (NFT)	Track Error (MTE)	Track Latency (MTL)	Track Completeness (MTC)	Predicted Throughput (fps)	Measured Throughput (fps)
1	1.62	16	87%	41.65	36.92

7.5.5 Deployment of CACTuS using AhcFlow

In this section, the deployment results of CACTuS configuration *C1* which is 16 SEFs with 127×127 pixel sized images on a platform architecture of one CPU and one GPU are described. A video of 160 frames with a range of tracking scenarios as described in [127] was used to measure the tracking accuracy. This video was created along with its corresponding ground-truth data containing the coordinates of the centroid of the objects. The tracking accuracy of the deployment of CACTuS in table 7.17 shows that the CACTuS application was correctly deployed. The values of the metrics used to measure the tracking performances are relatively similar to the previous published implementations by Milton [129] and Webb [130]. Furthermore, the measured throughput is close the predicted throughput (1/makespan) by the AMS algorithm.

The metrics used to measure the tracking performances are explained as follows:

1. Number of false tracks (NFT) are the total number of objects tracked that were not part of the ground-truth data.

2. Mean tracking error (MTE) is calculated as the average of the pixels between the tracked object's centroid with the centroid of the object recorded in the ground-truth data.

3. Mean track latency (MTL) is the mean of the number of frames elapsed before a SEF makes the object as its primary target.

4. Mean track completeness (MTC) is the mean of the percentage of the total distance that the algorithm accurately follows the target.

7.5.6 Conclusion

The deployment technique of the AhcFlow design flow was described in this section. It was explained how the output of the AMS algorithm, which is an ArcSDF graph, is used by the deployer to create a runtime image of the application-algorithm on the platform architecture instance. The structure of the deployer framework was described along with its four key steps in the creation of the runtime form the abstract ArcSDF graph. These steps are verification and validation, skeleton object generation and injection of the pre-engineered actors and channels with the launch

of the compute zones. These steps were described with a simple target detection example. After describing the steps in the deployment technique, CACTuS was automatically deployed using the aforementioned deployment technique from the AMS algorithm on a platform architecture instance of one CPU and one GPU. This automatic CACTuS deployment was checked for correctness and its actual throughput was measured. It was found that the automatically deployed CACTuS produced efficient tracking results comparable with published hand crafted implementations of CACTuS. Furthermore, the measured throughput was close to the AMS predicted performance value.

7.6 Conclusion

In this chapter, a complex multi-object visual tracking application called CACTuS [112, 127] was used to demonstrate the complete AMS design flow along with the automated deployment features. Initially, an overview of the CACTuS DFG was presented to describe its constituent actors, how the application can be scaled on the basis of filter called shape estimating filter (SEF) and the profiled timings and resource usages of the pre-engineered actors and channels of CACTuS. These actor timings and their resource usages were used in the AMS algorithm to predict makespan and to explore suitable platform architecture instances.

After describing the CACTuS application with the profiled actor timings and resource requirements, a comparison to AMS estimated performance with published hand-crafted implementations of CACTuS was presented. The comparison showcased the predicted performance matched closely with the published throughput. This comparison was limited to smaller data token sizes (image pixel size) of CACTuS and the platform architecture was fixed to only one GPU.

In order to consider more variety of platform architectures, an exhaustive design space exploration of five different configurations of CACTuS with varying data token sizes and number of SEFs on a PPG tier 1 (agile platform constraint) of one CPU and eight GPUs were conducted. The utility of AMS algorithm was shown during the study of direct connections impact between GPUs. This design space exploration produced three key observations: (1) performance improves greatly with the first few addition of 3 or 4 GPUs but this improvement plummets with further GPU additions, (2) CACTuS configuration with larger sized images have better concurrent GPU usages and (3) although direct connections improves performance, they are not significant as expected and this slight improvement is not consistent across all the CACTuS configurations.

The automatic deployment from the abstract ArcSDF representation was described next. The framework that conducts this automated deployment was described with its key steps consisting of verification and validation, skeleton ArcSDF object generation and pre-engineered actor injection. These steps were described with a simplified object detection application. It was shown how this application was automatically deployed from ArcSDF and produced correct results. A configuration of CACTuS was also automatically deployed on a platform architecture of a GPU and a CPU. The measured tracking results were consistent with the published results of CACTuS. Also, the predicted performance of CACTuS closely matched with the measured throughput resulting from the automated deployment. This demonstrated the feasibility of AhcFlow.

Chapter 8

Conclusion and future work

Contents

8.1 Introduction

This research has aimed to create a design flow for agile heterogeneous computing, where the platform architecture is a design variable and is constituted of three disparate compute engines (FPGA, GPU and CPU). A new design flow called AhcFlow that can support platform architecture as a design variable and handle concurrent actor executions on compute engines like FPGA, GPU and CPU is created in this research. It was achieved by generalising the Y-chart approach to enable design space exploration with variable platform architecture. In order to attain the generalisation of Y-chart, a new model called parameterised platform graph (PPG) was created to represent agile platforms and a new design space exploration algorithm called agile mapping and scheduling (AMS) algorithm was developed. Concurrent actor execution on the same compute engine was made possible by creating a new dataflow-based intermediate data structure called architecture augmented synchronous dataflow (ArcSDF). This representation is used within the new design flow to enable analysis of resource consumption and to foster automation between design space exploration and deployment. Since ArcSDF represents the architectural decisions, it is used to automatically produce the runtime code for the deployment. A case-study using a computationally intensive multi-object tracking application is also conducted in this research.

PPG is a high-level platform representation that is capable of representing variable platform architecture and concurrent actor execution on the same compute engine. These are unique requirements of agile heterogeneous computing and were not addressed in previous research literature of high-level platform architectures. PPG consists of two tiers. The first tier (also called tier 1) represents the available platform components and their constraints. The second tier (also called tier 2) represents a platform architecture instance. Tier 2 is used to reproduce a platform architecture physically. PPG is used within the design space exploration algorithm called AMS to find a suitable tier 2 that meets the constraints of tier 1. PPG is a highly abstract representation and yet possesses the necessary details for reproducibility to be used for design space exploration. PPG has been described in detail and demonstrated using examples. Its feasibility is validated through an implementation within the AMS prototype and is used for the case-study.

ArcSDF extends synchronous dataflow (SDF) representation to express architectural decisions, which allows it to exhibit more analytical strengths than its parent SDF. It can be actively used within design space exploration to access resource usage, prediction of concurrent executions and estimate throughput, which is inherited from the parent SDF representation. ArcSDF is used in AhcFlow as an intermediate data structure with the design space exploration algorithm and also to automate deployment; showing that the architectural decisions in ArcSDF is reproducible. The ArcSDF representation extends the boundary of decidable dataflow models to increase their expressive and analytical strengths. In this research, ArcSDF is presented and algorithms to showcase some of its analytical

strengths are developed. The ArcSDF representation is validated by implementation within the AMS prototype and also through its usage in the case-study.

The AMS algorithm addresses the research gap of a lack of design space exploration algorithms for variable platform architectures. AMS explores the design space to find an optimal platform architecture within a certain capital cost budget and also finds the architectural decisions to deploy the application-algorithm onto it. The output is an ArcSDF representation that can be used for deployment. Furthermore, AMS allows concurrent actor execution on compute engines like GPU by enabling concurrency on the same compute engine. This is accomplished by incorporating ArcSDF within the algorithm. A large number of synthetic and real-application specific DAGs were used to evaluate AMS. A prototype of AMS was created to conduct the evaluation experiments. The experimental results showed that AMS is capable of finding platform architectures to produce optimal performance with the capital cost budget. Finally, the overall design flow (AhcFlow) was applied for a case-study. The AMS prototype was used along with a deployment module to conduct the case-study.

In order to describe the afore-mentioned research contributions in detail, the research questions that were defined in Chapter 3 will be revisited in the next section; following which, in the final section, suggested future work will be discussed.

8.2 Research questions revisit

The research questions that were presented previous in Chapter 3 are revisited in this section. For every research question, a discussion is presented to summarise the research contributions made while addressing them.

1. **Is it feasible to create a design flow for agile heterogeneous computing where architectural exploration is closely integrated into mapping and scheduling decisions?**

This is the overarching research question of this thesis, which was primarily answered in Chapter 4, where a new design flow for agile heterogeneous computing called *AhcFlow* was conceived. In order to realise AhcFlow, specifications of the following new elements were described: a new agile platform representation called *Parameterised platform graph* (PPG), a new intermediate data structure called *Architecture augmented synchronous dataflow* (ArcSDF), a design space exploration algorithm called *Agile mapping and scheduling algorithm* (AMS) and a deployment technique. The representations; PPG and ArcSDF were presented in Chapter 5, along with their analysis algorithms. The AMS algorithm was described in Chapter 6. The validation of the new design flow was conducted in parts, in Chapters 6 and 7.

AhcFlow was conceived through the generalisation of the Y-chart design methodology by incorporating a representation of the platform constraints (PPG) rather than a fixed platform architecture. This allowed close exploration of platform architecture instances with the mapping and scheduling decisions. An intermediate data-structure (called ArcSDF) was proposed to support this close exploration of platform architecture instances. The new design flow uses ArcSDF for analysis to find suitable platform architecture instances and the mapping and scheduling decisions. The use of ArcSDF for analysis is mainly done by the design space exploration algorithm (AMS), which relies on ArcSDF to estimate makespan, resource usage and capital cost of the platform architecture instance.

Another innovation of this new design flow is the automation of deployment by using the intermediate data-structure. Since it contains the architectural decisions that include the mapping and scheduling information, it is parsed during deployment to automate the creation of a runtime on the designated platform architecture. The runtime is created by using vendor specific tool-flows of the compute engines that constitutes the platform architecture.

While describing AhcFlow in Chapter 4, the requirements for PPG, ArcSDF and AMS were also conceived. These requirements were later used in Chapters 5 and 6 to define PPG, ArcSDF and AMS.

2. **How to incorporate in a design flow, for agile heterogeneous computing, a constraint based platform definition?**

A new way to represent agile heterogeneous platform at a higher abstraction, called *Parameterised platform graph* (PPG) was created in Chapter 5 to answer this question. PPG incorporates the necessary details for design space exploration, thus it has the capabilities to express the platform constraints and represent instances of platform architectures possible within the constraints. PPG was validated by using it to construct the prototype of the new design flow. PPG was implemented and used for design space exploration in Chapter 6, where synthetic datasets were used for evaluation. Further, in Chapter 7 it was shown that PPG can be used for real platform architectures.

The ability to express the details required for design space exploration, the platform constraints and the architecture instances was achieved through the following two features of PPG:

- PPG is divided into two tiers. PPG tier1 represents the platform constraints, whereas PPG tier 2 expresses a platform architecture instance. PPG tier 1 is used by the design space exploration algorithm, whereas different instances PPG tier2 are explored for the selection of the final platform architecture (PPG tier2) for deployment.

- PPG tier 2 that represents a platform architecture instance is a high-level

graph that can be analysed to estimate performance, resource usage and capital during design space exploration.

It is important to note that PPG was used as a reference to define the architectural decisions to be expressed by the intermediate data-structure (ArcSDF). These decisions were an important aspect of the design space exploration algorithm (AMS).

3. **How to represent simultaneously the application, architecture, mapping and scheduling decisions, and deployment details of an agile heterogeneous computing solution?**

A novel data-structure called *architecture augmented synchronous dataflow* (ArcSDF) was developed to answer this research question in Chapter 5. ArcSDF was created by augmenting the original SDF graph to express the architectural decisions, which were defined by using PPG tier 2 as a reference. Four new constructs were defined for the creation of ArcSDF. The most important one being the *compute zone*, which groups actors for sequential execution and also expresses resource consumption by the group. Compute zones are part of a compute engine and their resource usage depends upon the actors mapped onto it. Once all the actors inside the compute zones complete their execution, the compute zone may release the resources for a new compute zone. The other constructs are *interfaces*, *resource edge* and *control actors*. ArcSDF is validated by implementing it within the AMS prototype in Chapter 6 and using it for deployment in Chapter 7. The analysis algorithm resulting from ArcSDF was used with the design space exploration algorithm which was evaluated with synthetic and real application dataflow graphs.

It is further shown in Chapter 5 that ArcSDF is capable of maximum resource usage by an ArcSDF representation, calculation of the earliest time slot to map a new actor and optimisation to reduce the number of compute zones used, which was not possible just with the parent SDF representation. These analyses were later used for the design space exploration.

4. **How to best perform mapping and scheduling in the context for this new design flow for agile heterogeneous computing?**

This research question is answered in Chapter 6 by creating a new design space exploration algorithm called the *agile mapping and scheduling* (AMS) algorithm that evolves the platform architecture from the mapping and scheduling decisions. This algorithm is created by advancing a widely accepted list scheduling heuristic algorithm called HEFT to consider concurrent actor executions on a compute engine and by advancing the Gain/Loss algorithm for capital cost. AMS was evaluated in two phases. In the first phase, only static platform architectures were considered to evaluate advanced HEFT. This evaluation was conducted with a very large number of synthetic dataflow acyclic graphs. It was found that the advanced HEFT performed significantly better than the version closer to its original counterpart. In the second phase of evaluation, the overall AMS algorithm was evaluated with a large

number of synthetic and real application dataflow acyclic graphs. This evaluation compared two approaches of the AMS algorithm (called expansion and reduction) against a random brute-force approach. The experimental results showed that in general both the approaches of AMS performed significantly higher than the brute force approach. Also, the expansion mode of exploration performs better for larger sized graphs with higher concurrency. AMS was further evaluated through an exhaustive case study with a complex visual tracking application called CACTuS in Chapter 7, where the design space exploration was compared with published hand crafted results. Furthermore, the predictions from AMS were shown to match with actual deployed results.

5. **How can an agile heterogeneous design flow incorporate decisions concerning the sharing of the compute engine resources amongst multiple concurrent actors?**

This question intersects the previous research questions. It has been mainly addressed by enabling ArcSDF to express resource consumption through *compute zones*, which allowed to analyse whether more than one actor can execute simultaneously on a compute engine. The resources are the computation resources that are expressed through the PPG representation. Resource usage needed to be incorporated within AMS, so that the design space exploration can consider concurrent actor executions. This was achieved by advancing the original HEFT algorithm to allow resources to be considered. In Chapter 7, the AhcFlow *deployer* also took the resources expressed by the ArcSDF into account, which shows the feasibility of this technique to incorporate resource sharing decisions. The evaluation of this technique was further conducted in Chapter 6 along with the AMS algorithm by using synthetic and real application DAGs to compare the benefits of the resource-based enhancements.

8.3 Future work

The research results presented in this thesis, shows the future directions of agile heterogeneous computing design flows are towards: (1) improving accuracy of the design space exploration outcomes and (2) increasing automation in deployment through an intermediate data structure (like ArcSDF). Improving accuracy in the design space exploration algorithm refers to lessening the difference between the predicted and the actual performance. It also refers to the inclusion of other objectives that might interest a designer, such as power usage. On the other hand, increasing automation refers to deployment with minimal designer's effort. This section presents a discussion on these two future research directions.

A more detailed platform architecture representation is expected for the improvement of accuracy in design space exploration results. This is so that the platform architecture representation can express detailed aspects of communication delays

and computation timings. Examples of these details may include the time required to access a certain type of memory, or the preparation time for a kernel to start execution. These details need to be included in such a manner that the complexities of representing the platform architecture components are not significantly increased. This is significant because if a designer's involvement increases, then the purpose of a design flow is defeated. In order to take advantage of the platform architecture details, it is also important to evolve the design space algorithm (AMS). Currently, the AMS algorithm assumes that communication requests on the same physical link are queued but with more architectural details, this idealistic behaviour needs to adapt.

A detailed platform architecture representation can also include power usage, so that designers can conduct exploration with power consumption as an objective during design space exploration. The challenge here is again including the details of power within the design flow (AhcFlow) without raising the representation complexity. This will also necessitate a significant enhancement of the AMS algorithm. It will be an interesting research direction to introduce more than two objectives in the AMS algorithm and measure its performance with the two modes of explorations (expansion and reduction).

The second future direction is increasing automation in deployment through an intermediate data structure, in the likeness of ArcSDF, which was presented in this thesis. The automated deployment conducted by the AhcFlow *deployer* heavily relied on vendor specific tools, which requires the designer to manually install and configure these tools. It is envisioned that the future research goals will extend the *deployer* into a uniform hardware and software layer, where intermediate datastructures like ArcSDF can be deployed with reduced human interventions. The deployer layer will be closely integrated with the compute engine hardware. It will read the ArcSDF instance to implement the application across all the compute engines and the designer need not be concerned about interacting with every toolsets for each compute engines. In order to pursue research in this direction, support from vendors to reveal their GPU and FPGA architectures would be necessary.

Appendix A

CACTuS computation and communication timings

Table A.1: This table summarises the execution timings and the resource consumption percentage of the CACTuS actors for configuration C2 (see table 7.1) obtained in this work. Infinite (∞) represents the actors that do not have a pre-engineered implementation on the compute engine. This is replaced with a very large number in the experiments. The timings are in micro seconds (μ). The resource consumptions are expressed as the percentage of the usable compute engine resources.

Actor	**CPU execution timing (μ)**	**CPU resource consumption (%)**	**GPU execution timing (μ)**	**GPU resource consumption (%)**
PV	6	90	13	12
PP	3262	90	2419	12
PS	81	90	75	12
PI	10388	90	2944	12
OV	9863	90	3475	12
OP	8700	90	3000	12
OI	413	90	94	12
IAT	1381	25	656	12
UV	44	90	56	12
UP	1263	90	988	12
PAT	1388	25	1013	12
COM	906	25	657	12
PE	6	25	∞	∞
OS	823	90	699	12
NV	17	90	∞	∞
NX	19	90	∞	∞
US	8	90	∞	∞
NS	8	25	700	12
SUMX	1138	25	∞	∞
SUMI	908	25	∞	∞
FLR	572	90	∞	∞

Table A.2: This table summarises the channel communication delays in microseconds (μ) for CACTuS configuration *C2*. List of configurations are in table 7.1.

Communication Channel	Delay timing μ	Communication Channel	Delay timing μ
PV to PP	780	OI to OP	1800
PV to UV	780	OI to OS	1800
PP to SUMX	2300	IAT to OI	600
PP to PAT	2300	UV to NV	150
PP to UP	2300	UP to COM	2400
PP to PI	2300	PAT to COM	2400
PS to PI	700	COM to PE	2300
PS to OP	700	COM to NX	2300
PS to US	700	PE to OS	7
PI to IAT	2800	OS to US	700
PI to SUMX	1800	US to NS	700
OV to UV	760	SUMX to PAT	2300
OP to OV	2400	SUMI to IAT	1800
OP to UP	2400	FLR to OI	1800

Table A.3: This table summarises the execution timings and the resource consumption percentage of the CACTuS actors for configuration C3 (see table 7.1) obtained in this work. Infinite (∞) represents the actors that do not have a pre-engineered implementation on the compute engine. This is replaced with a very large number in the experiments. The timings are in micro seconds (μ). The resource consumptions are expressed as the percentage of the usable compute engine resources.

Actor	**CPU execution timing (μ)**	**CPU resource consumption (%)**	**GPU execution timing (μ)**	**GPU resource consumption (%)**
PV	22	90	52	12
PP	3333	90	605	12
PS	34	90	1458	12
PI	8581	90	1575	12
OV	8200	90	2120	12
OP	8252	90	1814	12
OI	278	90	264	12
IAT	1314	25	298	12
UV	19	90	66	12
UP	1255	90	1095	12
PAT	1233	25	1109	12
COM	969	25	248	12
PE	6	25	∞	∞
OS	735	90	323	12
NV	17	90	∞	∞
NX	21	90	∞	∞
US	8	90	∞	∞
NS	8	25	700	12
SUMX	2797	25	∞	∞
SUMI	1891	25	∞	∞
FLR	572	90	∞	∞

Table A.4: This table summarises the channel communication delays in micro-seconds (μ) for CACTuS configuration *C3*. List of configurations are in table 7.1.

Communication Channel	Delay timing μ	Communication Channel	Delay timing μ
PV to PP	780	OI to OP	1800
PV to UV	780	OI to OS	1800
PP to SUMX	2300	IAT to OI	600
PP to PAT	2300	UV to NV	150
PP to UP	2300	UP to COM	2400
PP to PI	2300	PAT to COM	2400
PS to PI	700	COM to PE	2300
PS to OP	700	COM to NX	2300
PS to US	700	PE to OS	7
PI to IAT	2800	OS to US	700
PI to SUMX	1800	US to NS	700
OV to UV	760	SUMX to PAT	2300
OP to OV	2400	SUMI to IAT	1800
OP to UP	2400	FLR to OI	1800

Table A.5: This table summarises the execution timings and the resource consumption percentage of the CACTuS actors for configuration C4 (see table 7.1) obtained in this work. Infinite (∞) represents the actors that do not have a pre-engineered implementation on the compute engine. This is replaced with a very large number in the experiments. The timings are in micro seconds (μ). The resource consumptions are expressed as the percentage of the usable compute engine resources.

Actor	**CPU execution timing (μ)**	**CPU resource consumption (%)**	**GPU execution timing (μ)**	**GPU resource consumption (%)**
PV	27	90	7	12
PP	39451	90	8068	50
PS	493	90	239	12
PI	168874	90	34676	50
OV	157119	90	39477	50
OP	142020	90	35067	50
OI	1655	90	436	50
IAT	22139	25	4710	50
UV	133	90	196	12
UP	5041	90	1954	50
PAT	22159	25	9350	50
COM	3616	25	1291	50
PE	24	25	∞	∞
OS	823	90	11757	12
NV	76	90	∞	∞
NX	76	90	∞	∞
US	53	90	∞	∞
NS	24	25	611	12
SUMX	10190	25	∞	∞
SUMI	7271	25	∞	∞
FLR	785	90	∞	∞

Table A.6: This table summarises the channel communication delays in microseconds (μ) for CACTuS configuration *C4*. List of configurations are in table 7.1.

Communication Channel	Delay timing μ	Communication Channel	Delay timing μ
PV to PP	1180	OI to OP	1800
PV to UV	1180	OI to OS	1800
PP to SUMX	3700	IAT to OI	2100
PP to PAT	3700	UV to NV	870
PP to UP	3700	UP to COM	3800
PP to PI	3700	PAT to COM	3800
PS to PI	920	COM to PE	3800
PS to OP	920	COM to NX	3800
PS to US	920	PE to OS	22
PI to IAT	3900	OS to US	940
PI to SUMX	4000	US to NS	940
OV to UV	1180	SUMX to PAT	3900
OP to OV	3800	SUMI to IAT	3700
OP to UP	3800	FLR to OI	3700

Table A.7: This table summarises the execution timings and the resource consumption percentage of the CACTuS actors for configuration C4 (see table 7.1) obtained in this work. Infinite (∞) represents the actors that do not have a pre-engineered implementation on the compute engine. This is replaced with a very large number in the experiments. The timings are in micro seconds (μ). The resource consumptions are expressed as the percentage of the usable compute engine resources.

Actor	**CPU execution timing (μ)**	**CPU resource consumption (%)**	**GPU execution timing (μ)**	**GPU resource consumption (%)**
PV	49	90	7	12
PP	44348	90	6437	50
PS	421	90	209	12
PI	146607	90	22315	50
OV	198535	90	59982	50
OP	117369	90	28980	50
OI	1495	90	393	50
IAT	22057	25	4693	50
UV	164	90	242	12
UP	5667	90	2599	50
PAT	19539	25	9487	50
COM	4065	25	1431	50
PE	27	25	∞	∞
OS	823	90	12800	12
NV	76	90	∞	∞
NX	85	90	∞	∞
US	13	90	∞	∞
NS	6	25	150	12
SUMX	40759	25	∞	∞
SUMI	29084	25	∞	∞
FLR	785	90	∞	∞

Table A.8: This table summarises the channel communication delays in microseconds (μ) for CACTuS configuration *C5*. List of configurations are in table 7.1.

Communication Channel	Delay timing μ	Communication Channel	Delay timing μ
PV to PP	1180	OI to OP	1800
PV to UV	1180	OI to OS	1800
PP to SUMX	3700	IAT to OI	2100
PP to PAT	3700	UV to NV	870
PP to UP	3700	UP to COM	3800
PP to PI	3700	PAT to COM	3800
PS to PI	920	COM to PE	3800
PS to OP	920	COM to NX	3800
PS to US	920	PE to OS	22
PI to IAT	3900	OS to US	940
PI to SUMX	4000	US to NS	940
OV to UV	1180	SUMX to PAT	3900
OP to OV	3800	SUMI to IAT	3700
OP to UP	3800	FLR to OI	3700

Bibliography

[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in Heterogeneous Computing," *Sci. Program.*, vol. 18, pp. 1–33, Jan. 2010.

[2] J. W. S. Liu and A.-T. Yang, "Optimal Scheduling of Independent Tasks on Heterogeneous Computing Systems," in *Proceedings of the 1974 Annual Conference - Volume 1*, (New York, NY, USA), pp. 38–45, ACM, 1974.

[3] R. Inta, D. J. Bowman, and S. M. Scott, "The "Chimera": An Off-the-shelf CPU/GPGPU/FPGA Hybrid Computing Platform," *Int. J. Reconfig. Comput.*, vol. 2012, pp. 2:2–2:2, Jan. 2012.

[4] P. Meng, M. Jacobsen, and R. Kastner, "FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping," in *2012 International Conference on Field-Programmable Technology*, pp. 37–42, Dec. 2012.

[5] M. Alawieh, M. Kasparek, N. Franke, and J. Hupfer, "A high performance FPGA-GPU-CPU platform for a real-time locating system," in *2015 23rd European Signal Processing Conference (EUSIPCO)*, pp. 1576–1580, Aug. 2015.

[6] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers, "A Methodology to Design Programmable Embedded Systems," in *SpringerLink*, pp. 18–37, Springer, Berlin, Heidelberg, July 2001.

[7] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, pp. 3–14, Apr. 1999.

[8] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling Workflows with Budget Constraints," in *Integrated Research in GRID Computing* (S. Gorlatch and M. Danelutto, eds.), pp. 189–202, Boston, MA: Springer US, 2007.

[9] D. Menascé and V. Almeida, "Cost-performance Analysis of Heterogeneity in

Supercomputer Architectures," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, (Los Alamitos, CA, USA), pp. 169–177, IEEE Computer Society Press, 1990.

[10] M. Ercegovac, "Heterogeneity in supercomputer architectures," *Parallel Computing*, vol. 7, pp. 367–372, Sept. 1988.

[11] C. Ncube, "The NCUBE Family of High-performance Parallel Computer Systems," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues - Volume 1*, C^3P, (New York, NY, USA), pp. 847–851, ACM, 1988.

[12] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The Network Architecture of the Connection Machine CM-5 (Extended Abstract)," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, (New York, NY, USA), pp. 272–285, ACM, 1992.

[13] D. Watson, H. Siegel, J. Antonio, M. Nichols, and M. Atallah, "A Framework for Compile-Time Selection of Parallel Modes in an Simd/spmd Heterogeneous Environment," pp. 57–64, IEEE, 1993.

[14] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in Heterogeneous Computing," *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010.

[15] S. Undy, M. Bass, D. Hollenbeck, W. Kever, and L. Thayer, "A low-cost graphics and multimedia workstation chip set," *IEEE Micro*, vol. 14, pp. 10–22, Apr. 1994.

[16] M. Awaga, "3D Graphics Geometry Processor for PC," *IEICE TRANSACTIONS on Electronics*, vol. E81-C, pp. 733–736, May 1998.

[17] M. Macedonia, "The GPU enters computing's mainstream," *Computer*, vol. 36, pp. 106–108, Oct. 2003.

[18] C. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: A framework and analysis," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pp. 306–317, Nov. 2002.

[19] B. Neelima and P. S. Raghavendra, "Recent trends in software and hardware for GPGPU computing: A comprehensive survey," in *2010 5th International Conference on Industrial and Information Systems*, pp. 319–324, July 2010.

[20] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed, "A heterogeneous platform with GPU and FPGA for power efficient high performance computing," in *2014 International Symposium on Integrated Circuits (ISIC)*, pp. 220–223, Dec. 2014.

[21] P. E. Ross, "Why CPU Frequency Stalled," *IEEE Spectrum*, vol. 45, pp. 72–72, Apr. 2008.

[22] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, and J. Lemeire, "Comparing and combining GPU and FPGA accelerators in an image processing context," in *2013 23rd International Conference on Field Programmable Logic and Applications*, pp. 1–4, Sept. 2013.

[23] S. Skalicky, S. Lopez, and M. Lukowiak, "Distributed execution of transmural electrophysiological imaging with CPU, GPU, and FPGA," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–7, Dec. 2013.

[24] K. H. Tsoi and W. Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, (New York, NY, USA), pp. 115–124, ACM, 2010.

[25] R. Inta, D. J. Bowman, S. M. Scott, R. Inta, D. J. Bowman, and S. M. Scott, "The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform," *International Journal of Reconfigurable Computing, International Journal of Reconfigurable Computing*, vol. 2012, 2012, p. e241439, 2012/03/20, 2012/03/20.

[26] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA communication Via PCI express," *Cluster Computing*, vol. 17, pp. 339–348, June 2014.

[27] Y. Thoma, A. Dassatti, and D. Molla, "FPGA2: An open source framework for FPGA-GPU PCIe communication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec. 2013.

[28] "NVDIA NVLink Fabric." https://www.nvidia.com/en-au/data-center/nvlink/, Dec. 2018.

[29] I.-H. Chung, B. Abali, and P. Crumley, "Towards a Composable Computer System," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, (New York, NY, USA), pp. 137–147, ACM, 2018.

[30] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *2014 6th European Embedded Design in Education and*

Research Conference (EDERC), pp. 36–40, Sept. 2014.

[31] S. Stuijk, M. Geilen, and T. Basten, "A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pp. 548–555, Sept. 2010.

[32] A. Enrici, L. Apvrille, and R. Pacalet, "A Model-Driven Engineering Methodology to Design Parallel and Distributed Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, pp. 34:1–34:25, Jan. 2017.

[33] T. Stefanov, A. Pimentel, and H. Nikolov, "Daedalus: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips," in *Handbook of Hardware/Software Codesign* (S. Ha and J. Teich, eds.), pp. 1–36, Dordrecht: Springer Netherlands, 2016.

[34] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, (San Francisco, CA, USA), pp. 1–11, IEEE, Feb. 2015.

[35] Y. Liu and B. C. Schafer, "HW acceleration of multiple applications on a single FPGA," in *2014 International Conference on Field-Programmable Technology (FPT)*, pp. 284–285, Dec. 2014.

[36] Y. Wen and M. F. O'Boyle, "Merge or Separate?: Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms," in *Proceedings of the General Purpose GPUs*, GPGPU-10, (Austin, TX, USA), pp. 22–31, ACM, 2017.

[37] R. A. Q. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P. M. C. de Carvalho, and L. M. A. Drummond, "Maximizing the GPU resource usage by reordering concurrent kernels submission," *Concurrency and Computation: Practice and Experience*, Nov. 2017.

[38] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, pp. 323–364, June 1977.

[39] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, pp. 125–141, Sept. 1990.

[40] E. A. Lee and S. Neuendorffer, "Actor-Oriented Models for Codesign," in *Formal Methods and Models for System Design: A System Level Perspective* (R. Gupta, P. L. Guernic, S. K. Shukla, and J.-P. Talpin, eds.), pp. 33–56, Boston, MA: Springer US, 2004.

[41] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. A. Vissers, "A Methodology to Design Programmable Embedded Systems - The Y-Chart

Approach," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, (London, UK, UK), pp. 18–37, Springer-Verlag, 2002.

[42] B. Kienhuis, E. Deprettere, K. Vissers, and P. V. D. Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 338–349, July 1997.

[43] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, eds., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[44] S. Stuijk, *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Technische Universiteit Eindhoven, Jan. 2007.

[45] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 24:1–24:25, May 2008.

[46] Y. Sorel, "Massively parallel computing systems with real time constraints: The "Algorithm Architecture Adequation" methodology," in *Proceedings of the First International Conference on Massively Parallel Computing Systems (MPCS) The Challenges of General-Purpose and Special-Purpose Computing*, pp. 44–53, May 1994.

[47] R. Collobert, S. Bengio, and J. Marithoz, "Torch: A Modular Machine Learning Software Library," Nov. 2002.

[48] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[49] A. Vedaldi and K. Lenc, "MatConvNet: Convolutional Neural Networks for MATLAB," in *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, (Brisbane, Australia), pp. 689–692, Association for Computing Machinery, Oct. 2015.

[50] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *arXiv:1512.01274 [cs]*, Dec. 2015.

[51] J. Zhan and J. Zhang, "Pipe-Torch: Pipeline-Based Distributed Deep Learn-

ing in a GPU Cluster with Heterogeneous Networking," in *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*, pp. 55–60, Sept. 2019.

[52] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing, "Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines," *arXiv:1512.06216 [cs]*, Dec. 2015.

[53] X. Liu, H.-A. Ounifi, A. Gherbi, W. Li, and M. Cheriet, "A hybrid GPU-FPGA based design methodology for enhancing machine learning applications performance," *Journal of Ambient Intelligence and Humanized Computing*, June 2019.

[54] S. Mouselinos, V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, "TF2FPGA: A Framework for Projecting and Accelerating Tensorflow CNNs on FPGA Platforms," in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–4, May 2019.

[55] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, pp. 99–112, Feb. 2006.

[56] G. Kahn, "The Semantics of Simple Language for Parallel Programming," in *IFIP Congress*, pp. 471–475, 1974.

[57] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.

[58] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *, 1995 International Conference on Acoustics, Speech, and Signal Processing, 1995. ICASSP-95*, vol. 5, pp. 3255–3258 vol.5, May 1995.

[59] S. Ha and H. Oh, "Decidable Signal Processing Dataflow Graphs," in *Handbook of Signal Processing Systems* (S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, eds.), pp. 907–937, Cham: Springer International Publishing, 2019.

[60] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Jan. 2009.

[61] T. M. Parks, "Bounded Scheduling of Process Networks," Tech. Rep. UCB/ERL-95-105, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, Dec. 1995.

[62] A. Verma and S. Kaushal, "Cost-Time Efficient Scheduling Plan for Executing Workflows in the Cloud," *Journal of Grid Computing*, vol. 13, pp. 495–506, Dec. 2015.

[63] I. Casas, J. Taheri, R. Ranjan, L. Wang, and A. Y. Zomaya, "GA-ETI: An enhanced genetic algorithm for the scheduling of scientific workflows in cloud environments," *Journal of Computational Science*, vol. 26, pp. 318–331, Aug. 2016.

[64] J. J. Durillo, H. M. Fard, and R. Prodan, "MOHEFT: A multi-objective list-based method for workflow scheduling," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 185–192, Dec. 2012.

[65] M. Akbari, H. Rashidi, and S. H. Alizadeh, "An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems," *Engineering Applications of Artificial Intelligence*, vol. 61, pp. 35–46, May 2017.

[66] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[67] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, June 1987.

[68] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero, R. Valencia, and G. Palermo, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, pp. 55–78, Jan. 2014.

[69] S. Lecomte, S. Guillouard, C. Moy, P. Leray, and P. Soulard, "A co-design methodology based on model driven architecture for real time embedded systems," *Mathematical and Computer Modelling*, vol. 53, pp. 471–484, Feb. 2011.

[70] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid, "MARTE to ΠSDF transformation for data-intensive applications analysis," in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, pp. 1–8, Oct. 2014.

[71] O. Labbani, J.-L. Dekeyser, P. Boulet, and É. Rutten, "Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach," in *International Workshop MARTES: Modeling and Analysis of Real-Time and Embedded Systems*, Oct. 2005.

[72] B. P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[73] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet, "Efficient multicore scheduling of dataflow process networks," in *2011 IEEE Workshop on Signal*

Processing Systems (SiPS), pp. 198–203, Oct. 2011.

[74] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. Nezan, "An Open Framework for Rapid Prototyping of Signal Processing Applications," *EURASIP Journal on Embedded Systems*, vol. 2009, p. 598529, Dec. 2009.

[75] Y. Chen and H. Zhou, "Buffer minimization in pipelined SDF scheduling on multi-core platforms," in *17th Asia and South Pacific Design Automation Conference*, pp. 127–132, Jan. 2012.

[76] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," in *Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006*, pp. 25–36, June 2006.

[77] H. Hong, H. Oh, and S. Ha, "Hierarchical Dataflow Modeling of Iterative Applications," in *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, (New York, NY, USA), pp. 39:1–39:6, ACM, 2017.

[78] K. Desnos, M. Pelcat, J.-F. Nezan, S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pp. 41–48, July 2013.

[79] H. N. Tran, S. S. Bhattacharyya, J.-P. Talpin, and T. Gautier, "Toward Efficient Many-core Scheduling of Partial Expansion Graphs," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, (Sankt Goar, Germany), pp. 100–103, ACM, 2018.

[80] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, and F. Fruth, "Partial Expansion Graphs: Exposing Parallelism and Dynamic Scheduling Opportunities for DSP Applications," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pp. 86–93, July 2012.

[81] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, pp. 24–35, Jan. 1987.

[82] H. H. Wu, C. C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A Model-Based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 70–81, May 2011.

[83] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi,

and S. Stuijk, "Liveness and Boundedness of Synchronous Data Flow Graphs," in *2006 Formal Methods in Computer Aided Design*, pp. 68–75, Nov. 2006.

[84] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Transactions on Signal Processing*, vol. 45, pp. 1605–1618, June 1997.

[85] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Schedule-Extended Synchronous Dataflow Graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 1495–1508, Oct. 2013.

[86] C. Zebelein, C. Haubelt, J. Falk, T. Schwarzer, and J. Teich, "Representing mapping and scheduling decisions within dataflow graphs," in *Proceedings of the 2013 Forum on Specification and Design Languages (FDL)*, pp. 1–8, Sept. 2013.

[87] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: A seamless flow of graphs transformations," in *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings*, pp. 123–132, June 2003.

[88] M. Pelcat, J. F. Nezan, J. Piat, J. Croizer, and S. Aridhi, "A System-Level Architecture Model for Rapid Prototyping of Heterogeneous Multicore Embedded Systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, (nice, France), p. 8 pages, Sept. 2009.

[89] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1997.

[90] M. Pelcat, A. Mercat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J. Nezan, W. Hamidouche, D. Ménard, and S. S. Bhattacharyya, "Reproducible Evaluation of System Efficiency With a Model of Architecture: From Theory to Practice," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 2050–2063, Oct. 2018.

[91] P. Kutzer, J. Gladigau, C. Haubelt, and J. Teich, "Automatic generation of system-level virtual prototypes from streaming application models," in *2011 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, pp. 128–134, May 2011.

[92] Z. J. Jia, A. D. Pimentel, M. Thompson, T. Bautista, and A. Núñez, "NASA: A generic infrastructure for system-level MP-SoC design space exploration," in *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, pp. 41–50, Oct. 2010.

[93] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, Dec. 1999.

[94] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384–393, June 1975.

[95] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, eds., *Handbook of Signal Processing Systems*. Boston, MA: Springer US, 2010.

[96] V. J. Amuso and J. Enslin, "The Strength Pareto Evolutionary Algorithm 2 (SPEA2) applied to simultaneous multi- mission waveform design," in *2007 International Waveform Diversity and Design Conference*, pp. 407–417, June 2007.

[97] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 138–153, June 1990.

[98] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, vol. 5, pp. 23–32, Jan. 1988.

[99] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Comput.*, vol. 18, pp. 244–257, Apr. 1989.

[100] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 175–187, Feb. 1993.

[101] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, May 1996.

[102] E. Ilavarasan, P. Thambidurai, and R. Mahilmannan, "Performance Effective Task Scheduling Algorithm for Heterogeneous Computing System," in *The 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)*, pp. 28–38, July 2005.

[103] H. Arabnejad and J. G. Barbosa, "List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 682–694, Mar. 2014.

[104] S. AlEbrahim and I. Ahmad, "Task scheduling for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 73, pp. 2313–2338, June 2017.

[105] G. Xie, G. Zeng, L. Liu, R. Li, and K. Li, "High performance real-time scheduling of multiple mixed-criticality functions in heterogeneous distributed embedded systems," *Journal of Systems Architecture*, vol. 70, pp. 3–14, Oct. 2016.

[106] G. Xie, R. Li, and K. Li, "Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems," *Journal of Parallel and Distributed Computing*, vol. 83, pp. 1–12, Sept. 2015.

[107] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 107–, Apr. 2004.

[108] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, Mar. 2002.

[109] K. R. Shetti, S. A. Fahmy, and T. Bretschneider, "Optimization of the HEFT Algorithm for a CPU-GPU Environment," in *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 212–218, Dec. 2013.

[110] E. N. Alkhanak and S. P. Lee, "A hyper-heuristic cost optimisation approach for Scientific Workflow Scheduling in cloud computing," *Future Generation Computer Systems*, vol. 86, pp. 480–506, Sept. 2018.

[111] I. Crnkovic, "Software Engineering Research," 2008. http://www.idt.mdh.se/kurser/ct3340/archives/ht08, (retrieved 17/11/2015) Mälardalen University.

[112] S. C. Wong, V. Stamatescu, A. Gatt, D. Kearney, I. Lee, and M. D. McDonnell, "Track Everything: Limiting Prior Knowledge in Online Multi-Object Recognition," *IEEE Transactions on Image Processing*, vol. 26, pp. 4669–4683, Oct. 2017.

[113] M. F. Akbar, E. U. Munir, M. M. Rafique, Z. Malik, S. U. Khan, and L. T. Yang, "List-Based Task Scheduling for Cloud Computing," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 652–659, Dec. 2016.

[114] L. Pezzarossa, M. Schoeberl, and J. Sparsø, "Reconfiguration in FPGA-based multi-core platforms for hard real-time applications," in *2016 11th International Symposium on Reconfigurable Communication-Centric Systems-on-*

Chip (ReCoSoC), pp. 1–8, June 2016.

[115] K. Desnos and J. Heulot, "PiSDF: Parameterized & Interfaced Synchronous Dataflow for MPSoCs Runtime Reconfiguration," in *1st Workshop on MEthods and TOols for Dataflow PrOgramming (METODO)*, (Madrid, Spain), ECSI, Oct. 2014.

[116] X. Cai, M. R. Lyu, K.-F. Wong, and R. Ko, "Component-based software engineering: Technologies, development frameworks, and quality assurance schemes," in *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, pp. 372–379, 2000.

[117] F. Xie, G. Yang, and X. Song, "Component-based hardware/software coverification for building trustworthy embedded systems," *Journal of Systems and Software*, vol. 80, pp. 643–654, May 2007.

[118] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada, "A New Specification of Software Components for Embedded Systems," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pp. 46–50, May 2007.

[119] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Springer Science & Business Media, Dec. 2012.

[120] "CUDA Programming." http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Sept. 2018.

[121] "OpenCL programming." http://opencl.org/about/, July 2017.

[122] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[123] I. E. Bennour and J. Abderrazek, "Timed-SDF patterns for applications throughput analysis," in *2016 11th International Design Test Symposium (IDT)*, pp. 187–192, Dec. 2016.

[124] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[125] A. Radulescu and A. J. C. van Gemund, "Fast and effective task scheduling in heterogeneous systems," in *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556)*, pp. 229–238, May 2000.

[126] R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 111–, Apr. 2004.

[127] S. Wong, *Algorithms and Architectures for Visual Tracking.* PhD thesis, University of South Australia, May 2010.

[128] A. Milton, S. Wong, D. Kearney, and S. Lemmo, "The CACTuS Multi-object Visual Tracking Algorithm on a Heterogeneous Computing System," in *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand*, (New York, NY, USA), pp. 19–24, ACM, 2014.

[129] A. Milton, *Heterogeneous Computing for Bayesian Multi-Object Visual Tracking.* PhD Thesis, University of South Australia, Adelaide, Jan. 2017.

[130] D. Webb, *GPGPU Multi Object Bayesian Tracking with an Embedded System on a Chip.* Masters Thesis, University of South Australia, Adelaide, Dec. 2015.

[131] "CUDA profiler." http://docs.nvidia.com/cuda/profiler-users-guide/index.html, Sept. 2017.

[132] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. Barker, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2019.

[133] "The JSON Data Interchange Syntax." http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, Apr. 2016.