Mapping and Scheduling on Multi-core Processors using SMT Solvers

ACKNOWLEDGEMENTS

Foremost I would like immensely thank Oded Maler who is my thesis supervisor, but much more. He persistently helped me throughout my PhD. and provided motivation, research directions to carry out my work. But more importantly he gave me unparalleled freedom to explore my varied interests while choosing a topic for my thesis. He made me understand research and many other things in general and broader sense. He has been a continuous source of inspiration and motivation, to carry out the work, especially in times when things were obscure. He always provided me more than enough time and guidance where it was utmost necessary and ensured that I remained focussed on my topic. It was a privilege to work at Verimag especially with Oded, since he always provided a healthy and relaxed work atmosphere. He always blended humor with our daily chores at Verimag which teared down undesired tensions and frustrations. I had a great time at Grenoble working under his supervision. He was like a friend when we met over many lunches or dinners, with his entertaining and knowledgeable discussions on diverse topics. I will forever cherish the time that we shared during these years.

I am deeply thankful to Peter Poplavko who inspired me to work on dataflow models. Although according to Oded, he *hijacked* me to this domain. Peter played a very important role in my thesis. He was providing me a continuous feedback on my work. He enabled me to understand how to persistently attack a problem. He was actively guiding me throughout all my PhD. I thank him for all our discussions, code reviews, publication writing exercises etc. in which he invested a lot of his time. Especially the feedback on my thesis even when he was on vacation was of great help. I am confident that I would never have been able to complete this thesis without him.

I would like to thank the jury members and the reviewers for attending my thesis defense. I especially thank them for their valuable feedback on the manuscript. The feedback gave me an additional perspective to my own work. I am very happy to have such a jury.

I also want to thank the group at Technical University of Eindhoven who supported me for an internship. It was a very different and wonderful experience to work there. Prof. Sander Stuijk and his team were immensely supportive during my work. A few names I would like to mention are Hamid Reza Pourshaghaghi, Francesco Comaschi, Sebastian Moreno, Rosilde Corvino, Karthik Chandrasekar. Out of my office I made many friends in Eindhoven who accompanied me in several evenings for dinner, playing cards, skating and many other things. I will always remember the time that I spent with Deepak Jain, Srivathsa Bhat, Sushil Shirsath, Sandip Pawar, Aroa Izquierdo and many others.

I want to thank all my colleagues in Verimag. Sophie Quinton, Irini-Eleftheria Mens, Abhinav Srivastav, Alexios Lekidis and many of them. We always crack jokes on PhD. life and make fun of our situations which seemed to be going nowhere. I believe I had a great working environment, sharing my views, talks, presentation etc. with my friends. I would like to convey my thanks to the administrative staff which was a life-line for me in France. I still remember my struggles as a non-french speaking foreigner and without their help I would have been nowhere.

A special thanks goes to Dorit Maler. I enjoyed the time that we spent with her in informal meetings. We shared such memorable discussions, that I would cherish for rest of my life. I

still remember how confident I felt after I talked with her just before my defense. She is a wonderful person, and I feel lucky to know her closely.

Finally, I want to thank most supportive and loveable parents, who have been there for me always, irrespective of all the things. I am indebted to them to the last bit for being there and bringing the best out of me. Along with them, I thank my sisters and extended family for their support even if I am far away for a long time. I want to thank my wife "Vrushali", for her constant support in good and bad times. She accompanied me without any complain irrespective of my negligence towards her. I am grateful to have such companion. I dedicate this thesis to my parents and my wife.

Abstract

In order to achieve performance gains in the software, computers have evolved to multi-core and many-core platforms abounding with multiple processor cores. However the problem of finding efficient ways to execute parallel software on these platform is hard. With a large number of processor cores available, the software must orchestrate the communication, synchronization along with the execution of the code. Communication corresponds to the transport of data between different processors, which either can be handled transparently by the hardware or explicitly managed by the software. Synchronization is a requirement of proper selection of start time of computations e.g. the condition for software tasks to begin execution only after all its dependencies are satisfied.

Models which represent the algorithms in a structured and formal way expose the available parallelism. Deployment of the software algorithms represented by such models needs a specification of which processor to execute the tasks on (*mapping*) and when to execute them (*scheduling*). Mapping and scheduling is a hard combinatorial problem to solve with a huge design space containing exponential number of solutions. In addition, the solutions are evaluated according to different costs that need to be optimized, such as memory consumption, time to execute, static power consumption, resources used etc. Such a problem with multiple costs is called a *multi-criteria* optimization problem. The solution to this problem is not a unique single solution, but a set of incomparable solutions called *Pareto* solutions. In order to track multi-criteria problems, special algorithms are needed which can approximate the Pareto solutions in the design space.

In this thesis we target a class of applications called *streaming* applications, which process a continuous stream of data. These applications typically apply similar computation on different data items. A common class of models called *dataflow* models conveniently expresses such applications. In this thesis, we deal with mapping and scheduling of dataflow applications on many-core platforms. We encode this problem in form of logical constraints and present it to *satisfiability modulo theory* (SMT) solvers. SMT solvers, solve the encoded problem by using a combination of search techniques and constraint propagation to find an assignment to the problem variables satisfying the given cost constraints.

In dataflow applications, the design space explodes with increased number of tasks and processors. In this thesis, we tackle this problem by introducing symmetry reduction techniques and demonstrate that symmetry breaking accelerates search in SMT solvers, increasing the size of the problem that can be solved. Our design-space exploration algorithm approximates the Pareto front of the problem and produces solutions with different cost trade-offs. We validate these solutions by executing them on a real multi-core platform.

Further we extend the scheduling problem to the many-core platforms which are assembled from multi-core clusters connected by network-on-chip. We provide a design flow which performs mapping of the applications on such platforms and automatic insertion of additional elements to model the communication. We demonstrate how communication with bounded memory can be performed by correctly modeling the flow-control. We provide experimental results obtained on the 256-processor Kalray MPPA-256 platform.

iii

Multi-core processors have typically a small amount of memory close to the processor. Generally application data does not fit in the local memory. We study a class of parallel applications having a regular data access pattern, with large amount of data to be processed by a uniform computation. Such applications are commonly found in image processing. The data must be brought from main memory to local memory, processed and then the results written back to main memory, all in batches. Selecting the proper granularity of the data that is brought into local memory is an optimization problem. We formalize this problem and provide a way to determine the optimal transfer granularity depending on the characteristics of application and the hardware platform. Further we provide a technique to analyze different data exchange mechanisms for the case where some data is shared between different computations.

Applications in modern embedded systems can start and stop dynamically. In order to execute all these applications efficiently and to optimize global costs such as power consumption, execution time etc., the applications must be reconfigured at runtime. We present a predictable and composable way (executing independently without affecting others) of migrating tasks according to the reconfiguration decision.

Keywords: Multi-core, many-core, dataflow, mapping, scheduling, SMT solver

Résumé

Dans l'objectif d'augmenter les performances, l'architecture des processeurs a évolué vers des plate-formes "multi-core" et "many-core" composées de multiple unités de traitements. Toutefois, trouver des moyens efficaces pour exécuter du logiciel parallèle reste un problème difficile. Avec un grand nombre d'unités de calcul disponibles, le logiciel doit orchestrer la communication et assurer la synchronisation lors de l'exécution du code. La communication (transport des données entre les différents processeurs) est gérée de façon transparente par le matériel ou explicitement par le logiciel.

Les modèles qui représentent les algorithmes de façon structurée et formelle mettent en évidence leur parallélisme inhérent. Le déploiement des logiciels représentés par ces modèles nécessite de spécifier placement (sur quel processeur s'exécute une certaine tâche) et l'ordonnancement (dans quel ordre sont exécutées les tâches). Le placement et l'ordonnancement sont des problèmes combinatoires difficile avec un nombre exponentiel de solutions. En outre, les solutions ont différents coûts qui doivent être optimisés : la consommation de mémoire, le temps d'exécution, les ressources utilisées, etc. C'est un problème d'optimisation multi-critères. La solution à ce problème est ce qu'on appelle un ensemble Pareto-optimal nécessitant des algorithmes spéciaux pour l'approximer.

Nous ciblons une classe d'applications, appelées applications de streaming, qui traitent un flux continu de données. Ces applications qui appliquent un calcul similaire sur différents éléments de données successifs, peuvent être commodément exprimées par une classe de modèles appelés modèles de flux de données. Le problème du placement et de l'ordonnancement est codé sous forme de contraintes logiques et résolu par un solveur Satisfaisabilité Modulo Théories (SMT). Les solveurs SMT résolvent le problème en combinant des techniques de recherche et de la propagation de contraintes afin d'attribuer des valeurs aux variables du problème satisfaisant les contraintes de coût données.

Dans les applications de flux de données, l'espace de conception explose avec l'augmentation du nombre de tâches et de processeurs. Dans cette thèse, nous nous attaquons à ce problème par l'introduction des techniques de réduction de symétrie et démontrons que la rupture de symétrie accélère la recherche dans un solveur SMT, permettant ainsi l'augmentation de la taille du problème qui peut être résolu. Notre algorithme d'exploration de l'espace de conception approxime le front de Pareto du problème et produit des solutions pour différents compromis de coûts. De plus, nous étendons le problème d'ordonnancement pour les plate-formes "many-core" qui sont une catégorie de plate-forme multi coeurs où les unités sont connectés par un réseau sur puce (NoC). Nous fournissons un flot de conception qui réalise le placement des applications sur de telles plate-formes et insert automatiquement des éléments supplémentaires pour modéliser la communication à l'aide de mémoires de taille bornée. Nous présentons des résultats expérimentaux obtenus sur deux plate-formes existantes : la machine Kalray à 256 processeurs et les Tilera TILE-64.

Les processeurs multi-coeurs ont typiquement une faible quantité de mémoire proche du processeur. Celle ci est généralement insuffisante pour contenir toutes les données necessaires au calcul d'une tâche. Nous étudions une classe d'applications parallèles présentant un pat-

tern régulier d'accès aux données et une grande quantité de données à traiter par un calcul uniforme. Les données doivent être acheminées depuis la mémoire principale vers la mémoire locale, traitées, puis, les résultats retournés en mémoire centrale, tout en lots. Fixer la bonne granularité des données acheminées en mémoire locale est un problème d'optimisation. Nous formalisons ce problème et proposons un moyen de déterminer la granularité de transfert optimale en fonction des caractéristiques de l'application et de la plate-forme matérielle.

En plus des problèmes d'ordonnancement et de gestion de la mémoire locale, nous étudions une partie du problème de la gestion de l'exécution des applications. Dans les systèmes embarqués modernes, les applications peuvent démarrer et s'arrêter dynamiquement. Afin d'exécuter toutes les applications de manière efficace et d'optimiser les coûts globaux tels que la consommation d'énergie, temps d'exécution, etc., les applications nécessitent d'être reconfigurées dynamiquement à l'exécution. Nous présentons une manière prévisible et composable (exécution indépendamment sans affecter les autres) de réaliser la migration des tâches conformément à la décision de reconfiguration.

Contents

List of Figures

INTRODUCTION

The first chapter introduces the embedded systems domain. It explains the multi-core scenario and technological challenges associated with it.

Multi-core processors are now an unavoidable fact of the information processing industry. Over the years, technology advancement has seen the scaling of Moore's law [119]. While the technology evolved for smaller and smaller transistor size, more amount of hardware could be fabricated on chip with same area. Increasing clock speed to gain performance improvement, a technique followed for many years, could no longer provide performance benefits, because of several issues like power consumption, current leakage, electrical interference, chip-design issues etc. [97]. In order to exploit the advantages of technology scaling, multi-core processor came into existence. It became clear that adding multiple processor cores to the same chip provides more benefits and performance rather than over-clocking a single processor [100]. This trend is illustrated in Figure 1.1, where the evolution of the processors gradually shifts towards multi-core approach.

Another motivational factor for multi-core processing was the advancements in data processing algorithms, which demanded extra computational power. Applications which perform video processing, audio-video rendering, and many other signal processing applications became computationally heavy with their evolution. In addition, such applications have stringent deadline requirements to be met on a platform having limited resources. They also provide a degree of freedom, which involve exploitation of different levels of parallelism. Multi-core approach is often a suitable solution for parallel applications. It is well-known that the speed-up due to multi-core processor is less than proportional to their number, nevertheless, it still remains attractive in power-performance trade-offs.

1.1 MULTI-CORE PROCESSOR SYSTEM ARCHITECTURE

A typical embedded system has to perform various tasks. For example, a cellphone would display images, play video/audio, send and receive messages, perform data transfer and do more. In real-time systems, the tasks have strict deadlines. For example, a cell-phone is receiving a message in background while it is decoding audio and video data and displaying it on the screen. It has timing constraints such that it should not drop frames while doing audio/video processing to ensure quality of the service. At the same time, the communication protocol to receive messages requires messages to be exchanged at definite time intervals.
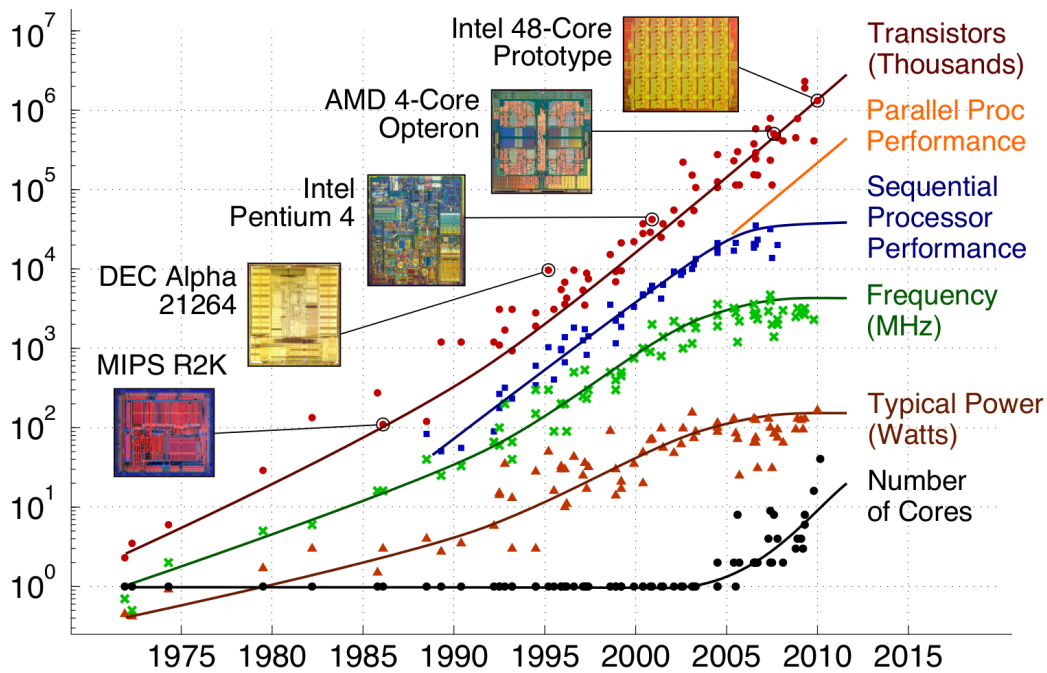
Figure 1.1 – Evolution of multi-core processors [1]

Handling such a wide variety of tasks using only one processor is difficult in practice. This will require a processor with high clock frequency, which in turn will incur higher power consumption. On mobile platforms, the resources such as energy resource are limited and must be optimally used.

In order to satisfy these constraints, designers use a piece of hardware called as *MPSoC* (MultiProcessor System on Chip) [132]. MPSoC's typically have dedicated hardware for some functions. For example, there is a dedicated IP (Intellectual Property) block, to process the input from keyboard. Suppose that instead of dedicated IP block, the *general purpose processor* (GPP) handled this task in the software, which is a possibility. This will increase the load on the processor, which will have to continuously poll for updated status from the keypad. With the IP block, this polling is offloaded to the dedicated hardware and it informs the GPP only when it has some relevant information to process. Suppose a key is pressed by the user, it is processed by the IP block and checked for its validity. And then it generates an interrupt to the GPP to process this information. Thus GPP is responsible for management of many such blocks, in addition to management of the software. Figure 1.2 shows an example MPSoC, where different functions such as Display, Camera, Keyboard, etc. are offloaded to a dedicated *reduced instruction set computing* (RISC) hardware. However, the hardware itself is managed by a GPP.

Dedicated hardware is a long-debated trade-off between performance and flexibility. Application Specific Instruction-set Processor (ASIP) can be optimized to provide high performance at low energy cost. The price to be paid is in terms of flexibility. If there is need to change in protocols, this hardware cannot be used. For example, if there is dedicated hardware for H.263 video, it cannot be used for newer specification H.264. In order to support newer algorithms and their respective upgrades, it is desirable to add more GPPs on the chip rather than to develop a dedicated hardware. Owing to this fact, there were two interesting types of processors were developed. First one is SIMD (Single Instruction Multiple Data), the chip

---

1. Taken from [12]
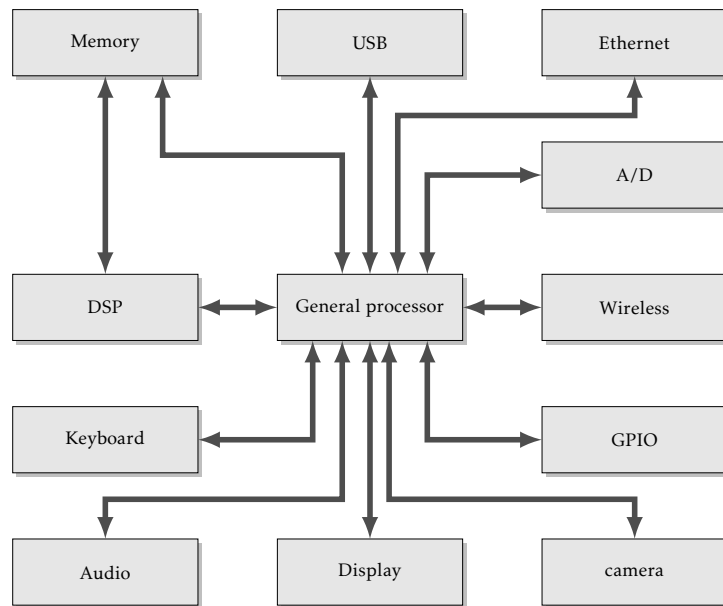
Figure 1.2 – MPSoC architecture

contains multiple processors which execute the same instruction on different data sets. It is a very useful architecture for data-parallel application like image processing. Graphics processing unit (GPU) are common example of such an architecture. The other interesting stream is MIMD (Multiple Instruction Multiple Data) or SPMD (Single Program Multiple Data). In this architecture, the processors execute independently of each other. This is the class of multi-core systems that we target for our work.

The design of multi-core processor architectures are driven mainly by the design costs pertaining to its application. Typical design and manufacturing of such systems requires investments in millions of dollars. Thus re-usability of such designs are also of primary importance. To make these systems generic and applicable in a wide range of applications typical system architectures have been followed. The dedicated IP blocks are re-used in the design, which brings lot of benefits, especially for verification of the design. Figure 1.3 shows an example of a multi-core architecture. Such multi-core systems consists of following hardware components -

1.1.1 *Host Processors*

Host Processors consist of a single or a group of general purpose processors which are used mainly for management tasks. Typically these group of processors are responsible for running operating systems or firmware and launching applications. They also handle other tasks like serving the hardware requests, running single-threaded program etc. The host processors are the masters of the systems and their main job is to control all the other pieces of hardware components and serve their requests.

1.1.2 *Peripheral Devices*

Peripheral devices typically consist of Application Specific Instruction-set Processor (ASIP) which are dedicated to a specific functionality. As seen in Figure 1.2, the peripheral devices like USB, Display etc. are the ASIP processors which are dedicated to their respective functions. They provide a minimum instruction set, which can be used to perform a restricted task. The
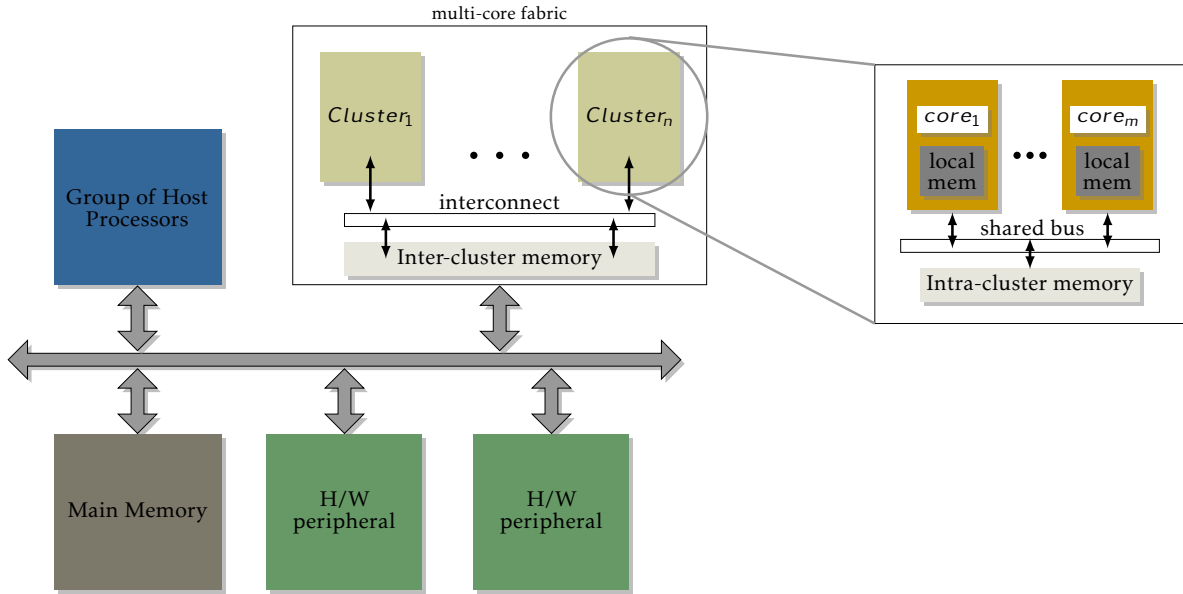
Figure 1.3 – Multi-core architecture

advantage of ASIP is that, they are optimized to run a specific function efficiently. Thus they provide high performance and reduce power consumption. However, it comes at the cost of limited flexibility and reprogramming. In contrast, GPP are less efficient compared to ASIP, but they provide high flexibility. In addition to ASIPs, some architecture also have Digital Signal Processor (DSP) which have SIMD or VLIW (Very Long Instruction Word) execution model. These processors are highly optimized to process data in parallel by using a single instruction. They are helpful in efficient execution of signal processing algorithms like filter banks, Fourier transforms, convolution etc. Application Specific Integrated Circuit (ASIC) also form a part of peripheral devices. These hardware circuits are devised to perform only specific tasks and are more efficient than ASIPs or GPPs, but at a cost of no flexibility.

1.1.3 *Multi-core Fabric*

The motivation behind introduction of multi-core accelerator is to improve the response time of the programs that have portions executing in parallel as shown in Figure 1.4. Multi-core fabric is typically used to offload the parallel computation of the program to execute it faster than on a single processor.

The multi-core fabric typically consists of multiple symmetric processors. Sometimes the symmetric cores are grouped in clusters [14, 65]. The obvious benefit of having clusters is to limit number of processors sharing common resources, resulting in less contentions and faster access. Typically, memory is a resource which is extensively used by the processor. If thousands of processors access memory, then memory will be a huge bottleneck, resulting in worse performance. Clustering helps to reduce number of processors to contend for such resources, for example by adding local memories to every cluster, which are synchronized with the main memory using either explicit transfers or synchronous hardware mechanisms (cache).

(a) Sequential execution on single processor

(b) Parallel execution
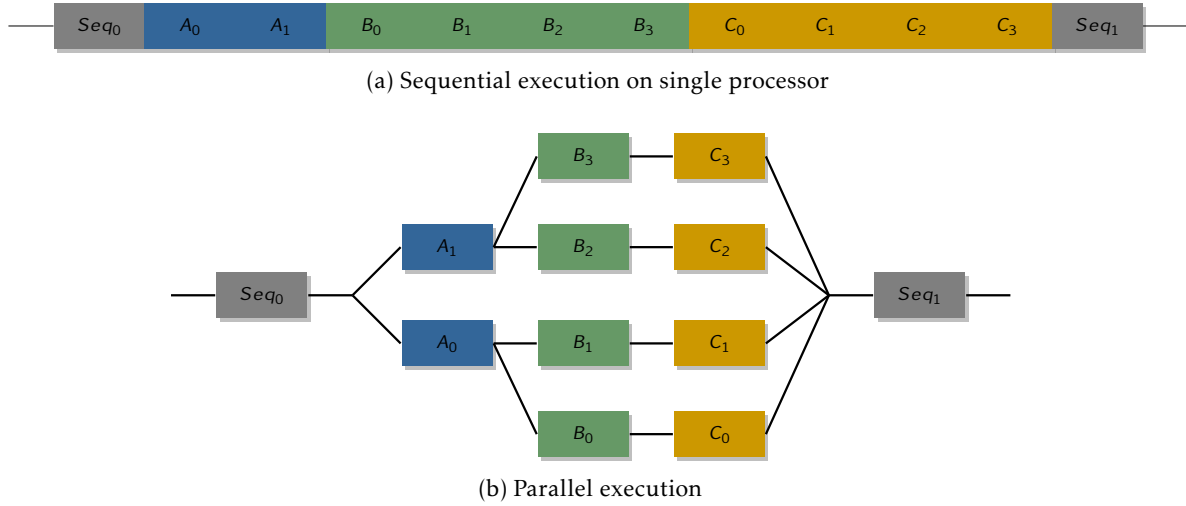
Figure 1.4 – Execution of a program

1.1.4 *Memory Organization*

The memory access in a computer system is the biggest bottleneck in the advancement of the processor technology. The gap between the the processor and memory speeds is increasingly becoming larger. Caches were added to processor systems which helped to close this gap. Multi-level caches became necessary to further boost the performance. In multi-core processors, they bring additional problems like extra area and power overhead, maintaining cache coherence (maintaining same copy of data in all the caches of the system) which limits scalability [28, 57, 69] etc.

Owing to such and many other issues, the multi-cores generally follow a Non-Uniform Memory Access (NUMA) model, where though all the memory is directly or indirectly accessible to all the cores, the access times differ according to hierarchy. Another problem with the cache in hierarchical memory, is that the access time for the cache is unpredictable (or in a wide range). If data is not found in cache, the amount of time required to bring data from main memory to cache depends on various factors like memory access latency, contentions on network, DRAM architecture and so on. Further, when processing is done in parallel, it can trigger cache coherence mechanism, maintaining the same copy of data in different caches to update the copies. This data traffic brings a further unpredictability in the software. Many a times, handling of cache coherence is offloaded to the software, making hardware design simpler. Some architectures provide scratchpad memories, which is located on-chip close to the processor. The difference is fast and predictable access as contrary to slower and with larger variation in access time for off-chip memories.

1.1.5 *Network Interconnect*

With increasing number of transistors and hardware on the chip, as discussed before, it became increasingly difficult to maintain a single clock source with small skew across the entire chip. Due to this Globally Asynchronous Locally Synchronous (GALS) approach was introduced, which involves using different clocks in different region of the chip. Traditional approaches like Point-to-Point connections, shared bus, etc. incurred number of issues to connect high number of hardware blocks. These issues include performance, dynamic power dissipation, wire delay, crosstalk, global routing congestion etc. In addition, with the increasing complexity, performance requirements, power issues, real-time requirements, Network On-

Chip (NoC) came into existence [13]. In NoC, the communication message is split into packets which are eventually transmitted on the packet-switched network. Due to their regular structure, fragmentation of wires and data multiplexing, they resolve many of the above-mentioned problems. A detailed survey of various NoC techniques and architectures are found in [3, 19].

1.2   MULTI-CORE SOFTWARE

Given that we have a multi-core processor architecture, with the applications executing on them as a software, the question is how much speed-up can be acquired compared to the single core execution and how to optimize it. In this context, given a parallel application and a parallel processing architecture, the software programmer has a very high number of design choices at various levels.

These choices can be briefly described as -

- **Software** : programming model and languages, portability, performance, re-use
- **Algorithm** : exposing the parallelism, and a set of design parameters influencing its execution, that can be chosen by the programmer.
- **Models** : abstraction of hardware platform, abstraction of software details to focus on the timing properties.
- **Optimization tools and methods** : heuristics and formal methods to be used for optimization, mapping and scheduling algorithms

Given with all these choices, it is not practically possible to evaluate every combination. In many cases, these decisions are made by intuition rather than by theory. It becomes hard to analyze if the design choices made by the programmer were correct and ensure optimal utilization of the resources. With these mentioned points, we can roughly describe the theoretical and practical issues in design and implementation of software on multi-core processors.

1.2.1   *Theoretical issues*

1.2.1.1   Theoretical limit for the speedup

When an application is executed in parallel, the speedup obtained compared to its sequential execution is theoretically limited by the Amdahl's Law.

Amdahl's Law for maximum speedup in parallel execution of a program on *n* processors is given by -

$$Speedup = \frac{1}{(1-p) + \dfrac{p}{n}}$$

where, $n \in \mathbb{N}$ number of parallel threads in execution
$p \in [0,1]$, is the proportion of the program that can be made parallel

Thus we can observe that, the maximum speedup is directly dependent on the sequential and the parallel parts of the program. Even if we have infinite processors for the execution of the software, still the speedup obtained will be limited by the factor $\frac{1}{1-p}$. We should note that this is only a theoretical limit. In practice, the program execution faces various issues like cache conflicts, network contentions, synchronization and communication overheads etc. It

(a) Task parallelism

(b) Data parallelism

(c) Pipeline parallelism
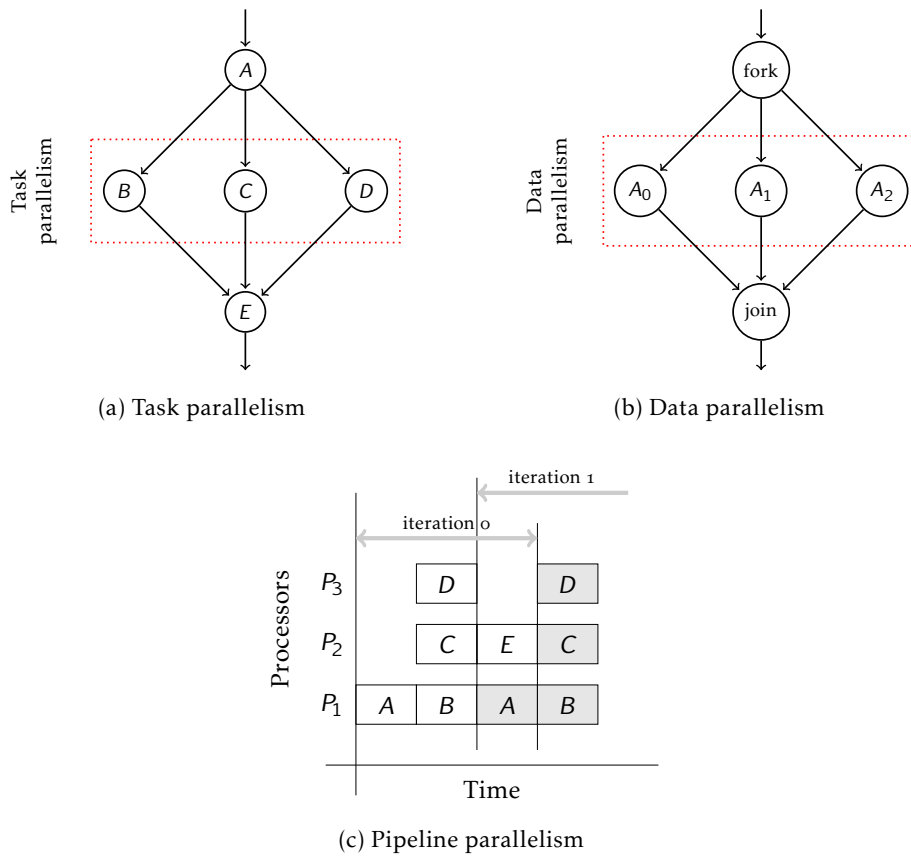
Figure 1.5 – Parallelism in a program

is difficult to take into account all these factors; however it becomes apparent that the actual speedup that is obtained will be less than the theoretical speedup.

1.2.1.2 Parallelization of the Software

Designing a parallel algorithm and expressing the parallelism efficiently is a challenging task. It refers to the concurrent execution of different parts of an algorithm. There are several types of parallelism, as discussed below.

Task Parallelism refers to different tasks in an application, that can be executed in parallel. For example, in a word processor, one task waits for input from the user and displays it on the screen, whereas another parallel task immediately processes the input in parallel and check for error in a given dictionary. These 2 tasks can run in parallel and can be represented in a task graph. An example of task parallelism is shown in Figure 1.5(a). Maximum amount of task parallelism is equal to the task graph width.

Data Parallelism is the same task executing on different data. Single Instruction Multiple Data (SIMD) and Single Program Multiple Data (SPMD) are different implementation schemes which benefit from data parallelism. The difference between them is that, in SIMD on every piece of data, each processor executes the same instruction. This is the type of input suits vector processors like GPUs. In the case of SPMD, each processor executes same software subroutine on a separate piece of data. Thus due to coarse granularity, the instructions may

differ between different data-parallel tasks (because of data dependent operations). Due to this characteristic, different data-parallel tasks can have different execution times.

For example, if we execute a blur filter on an image, in the output each pixel is replaced by the mean value of its 3x3 neighborhood. This filter can operate independently on each pixel. Figure 1.5(b) illustrates data parallelism. It looks similar to task parallelism, however with some key differences. In data parallelism, the tasks execute same piece of software on different piece of data, making the execution times of data-parallel tasks same (or nearly the same). Further the granularity and the number of tasks can be an option for the programmer, which is less the case for task parallelism. For example we can join tasks A_0 and A_1 into one task of higher granularity.

Pipeline Parallelism is the possibility of executing another instance of the entire task graph, before the completion of previous instance. Figure 1.5(c) shows an example of such execution of task graph shown in Figure 1.5(a). We call execution of the graph once as an *iteration* of the graph. We observe that the execution of iteration 0 finishes with execution of task E, however, task A of the next iteration starts before task E of the first iteration finishes. This increases the throughput of the application, which is the number of graph instances executing per unit time. It brings extra efficiency to execute such pipelined schedule, however the analysis of such schedule is more complex. Further, programming such schedule also has to ensure the data communication between tasks. The pipelined parallelism can be partly modeled by task and data parallelism if we concatenate multiple instances of the task graph into one more complex instance.

Instruction-level Parallelism (ILP) is different from that of SIMD data parallelism operating at the instruction level. In ILP, different instructions of a sequential program can be executed in parallel. For example, if the program has two instructions without dependency between them, they can execute in parallel in hardware by using hardware pipeline. Superscalar processors perform pipeline dependency analysis in hardware for parallel execution. To write such code explicitly would be a complex process which results in unportable code, specific to the processor. Further the benefits gained by this optimization would be minimal, and therefore we don't explore this parallelism. It is taken care of by hardware and compiler.

1.2.1.3   Programming Models and Languages

Given an algorithm for performing tasks and processing data, it should be represented with a model of computation. The model abstracts the algorithm in order to hide the fine details of a program, but still represent important characteristics of the algorithm, primarily its parallelism, that can be used for optimization. For example, in Figure 1.5 the model used is task graph. It represents the precedence of the tasks and also indicates about which tasks can be executed in parallel. There are different models of computation available for specific class of problems. We discuss this issue detail in Chapter 2.

In addition to the programming models, there is need of programming languages which can express the data processing in these models in order to enable their execution on the hardware platforms. Today there is no standard way of programming parallel platforms. There have been various efforts to develop languages in the context of different target architectures and programming models. OpenMP [31] is an extension to C/C++ language, which consists of set of compiler directives and library routines to specify parallel computations for a shared memory architecture. It annotates the loops and parallelizable code in a sequential program, which is used by the OpenMP compiler and runtime to effectuate a parallely executing code on the hardware. Recent versions of OpenMP support task parallelism. However, OpenMP is

designed more for shared memory programming. There have been efforts to implement it on MPSoC architectures [26, 83, 123].

OpenCL [51] and CUDA [92], target typically the GPU architectures. They are efficient in expressing and optimizing the SIMD kind of operations. In addition they support the typical GPU hardware which has different levels and types of memories. There have been recent work [81] on using OpenCL for multi-core platform with certain restrictions. There are many other works like MPI [93], Cilk [20], PGAS [24], ZPL [114] etc.

Overall the programming languages and models should provide the following

- Abstraction of the application algorithm, independent of the hardware platform with a separation between algorithm and implementation.
- Data abstraction and sharing conventions, in order to utilize different levels of memory in the hardware platform effectively.
- Portability to different hardware platforms.
- Execution model transparency, which in turn will help the tools and provide the programmer with a better understanding of how the code will execute on the hardware platform, facilitating the design choices.
- Interoperability with existing code. With new programming languages, one should not have to always completely rewrite all the programs that have already been written in other languages. It should be easy to migrate the code from other languages.

1.2.1.4 Deployment of Parallel Applications

Deployment of applications refers to organizing the execution of the program on the hardware platform. It consists mainly of two steps:

- **Mapping** is the spatial allocation of processor resources to the tasks to execute in parallel. It aims at optimizing the properties related to the amount of allocated resources, like power consumption, communication cost, load-balancing etc.
- **Scheduling** is the temporal allocation of processors between different tasks. It aims at optimizing the properties that depend on task execution order like response time, communication buffer size etc.

Embedded systems typically operate under extra-functional constraints. Hence deployment of applications has to consider various requirements such as throughput, response time, power consumption, load balancing, data communication etc. All these variables are elements of a multi-criteria optimization [38] problem where the cost variables are in conflict with each other. For example, if we use two processors instead of three, shown in Figure 1.5(c), we have to execute task D on either processor P_1 or P_2. This will delay the execution of task E, thus increasing the response time of the application.

In such case, the optimal solution is not unique, but rather a set of incomparable solutions called Pareto [94] set. These solutions represent different trade-offs between the cost variables.

Exploring the design space and finding the Pareto solutions requires a model of computation as well as a model of the hardware platform. The application model captures different aspects of algorithms, such as concurrency, data communication, precedence constraints etc. In contrast, the hardware platform models capture the hardware resources such as number of processors, communication links, interconnect bandwidth, power consumption, memory capacity etc. Deployment involves finding a solution which can make efficient use of all these resources in the hardware meeting the performance goals. Both the models, while abstracting information, lose some fine level details. This is a natural trade-off between efficiency in finding solutions and accuracy, giving rise to performance gap between the predicted properties and its actual execution.
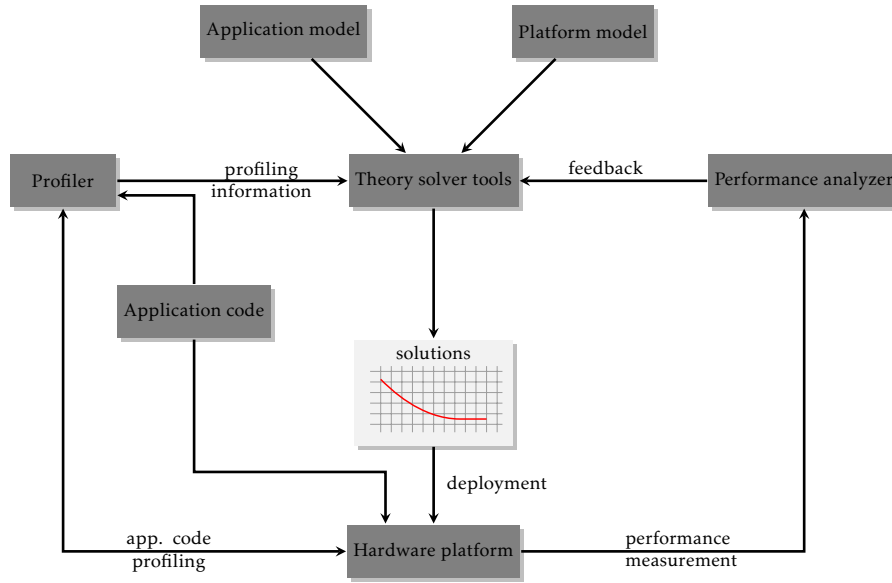
Figure 1.6 – Software design flow

1.2.2 *Practical Issues*

In addition to the theoretical issues, there are practical problems for exploitation of multi-core software -

- Lack of standard development and debugging tools
- Lack of multi-core operating systems and system software
- Lack of parallel programming models
- Non-standardized interfaces of tools, compilers, hardware from different manufacturers
- Unavailability of parallel programming expertise

Due to all these factors, the multi-core software development faced difficulties. A variety of different architectures made it impossible to standardize programming models and tools for this domain. Current software development tools for multi-core processors are far less matured than those for single core processors and even those for hardware development of multi-core processors. It is evident that the software development for multi-core processor requires highly skilled work-force.

1.3 SOFTWARE DESIGN FLOW

The software design process commonly follows the Y-chart approach [70, 117] as shown in Figure 1.6. This approach involves an application model and a hardware platform used as an input to the optimization solver. Optimization solver is particularly a set of tools which is used to determine the optimal configurations in which the application code can execute on the hardware platform satisfying different cost constraints. These solver tools need profiling information from the application code, such as task execution times, memory consumption, communication data size etc. This information varies from one hardware platform to another, and can be either calculated by the WCET (Worst Case Execution Time) analysis tools or estimated by running the code on the platform and profiling it. The performance analyzer part is optional in the design flow, and it can provide a feedback to the solver in terms of extra constraints to avoid performance problems observed during measurements.

The solver which forms the heart of this methodology, employs different solution space search algorithms to solve the mapping and scheduling problem. There exists a vast literature on algorithms for mapping and scheduling. The ultimate goal of the solver is to use such algorithms to solve this problem, and produce a solution (or set of Pareto optimal solutions).

1.4   RELATED TOOLS

1.4.1   *SDF3*

SDF3 [117] is a design flow based on synchronous dataflow (SDF) model. This design flow starts from architecture and application specification and proceeds to mapping, scheduling and performance prediction. The input to this tool is an application and architecture graph along with different constraints, like throughput, processors used etc. The tool performs static performance analysis on the input and determines if the constraints are feasible. If any constraint is violated, the bottlenecks can be analyzed. This tool is oriented to a specific predictable hardware platforms. Apart from SDF model, it also supports some other (more expressive) application models in particular SADF (Scenario Aware Data Flow).

1.4.2   *MAMPS*

MAMPS [63] extends SDF3 for code generation for an FPGA based platform. The inputs to this tool are same as SDF3 plus a template to generate hardware platform. This tool then takes input application along with its throughput requirements, generates a hardware platform and executable code which can satisfy these requirements. This hardware platform then can be programmed on an FPGA and the application can be run on the platform. This tool uses a simple Xilinx Microblaze processor and point-to-point links or a simple NoC to minimize the prediction error due to network contentions.

1.4.3   *MP-Opt*

MP-Opt [43] is a similar tool to that of SDF3/MAMPS, focussing on the throughput constraints of an application. It consists of four parts- a front-end compiler which generates SDF graphs from annotated C code. A solver, based on constraint programming, solves the allocation and scheduling problem. A back-end compiler responsible for generation of C code which can be executed on the target hardware. And finally the description languages which allows interfacing between all these components.

1.4.4   *StreamIT*

StreamIT is a programming language and a compiler which can compile the dataflow applications [49]. The language models SDF with various programming patterns like filters (same as actors) and split-join mechanism (edges). In this work, application data streams are explicitly split and merged in order to exploit the available parallelism. StreamIt can analyze and optimize sliding window extensions of SDF which are not taken into consideration in SDF3. For example it supports *peek* construct, which allows a filter to read the token without removing it from the channel.

1.4.5   *StreamRoller*

Another interesting work is StreamRoller compiler [75] which maps the StreamIT code on the Cell platform. It uses an algorithm SGMS (Stream Graph Modulo Scheduling) to

schedule the actors on the processors using ILP (Integer Linear Programming). SGMS applies a pipelining technique at coarse-grain level in order to achieve concurrent execution and hidden communication, minimizing stalls. It is done in two steps. First step involves splitting of the actor and partitioning in order to evenly balance the work among the processors. It is done by integer linear programming. Second stage is where actors are assigned to the pipeline stage in order to overlap communication and computation. It is done by greedy heuristic partitioning which assigns filters to processors.

1.4.6 *Discussion*

SDF3 tool does extensive performance analysis on SDF and other variants of the model. It also offers mapping and scheduling tools for a multiprocessor on-chip network platform with TDM scheduling. It does not support multiple core clusters accessing local shared memory without network and does not provide a run-time environment, although MAMPS as well as CompOSe operating system (see Section 3.4) offer support for SDF3. MAMPS on other hand deals with combined design space exploration of architecture and application. It has a run-time management system which performs online resource management considering the requirements of the application. MP-Opt offers optimization and runtime environment for Cell processor architecture which assumes one processor per cluster. It is designed to use the DMA communication (see Section 3.6) of the processor but does not model it explicitly. The major optimization goal of SDF3, MAMPS and MP-Opt is optimization of throughput, which requires pipelined scheduling. StreamIT is a compiler based approach to optimize the execution of streaming applications based on actor splitting and fusion. In this thesis we present our infrastructure which deals with mapping and scheduling problem on multi-core taking DMA communication into account. We consider application latency as the timing metric to optimize in our work. Our infrastructure deals with eliminating symmetrical solutions in the design space and accelerating the search for optimal solutions.

1.5   ORGANIZATION OF THESIS

In this thesis we study the problem of mapping and scheduling dataflow applications on multi-core processors. The thesis is organized as follows:

- Chapter 2 introduces the application programming model used in this work. We discuss briefly different programming models and similarity with our model that we introduce.
- Chapter 3 presents the hardware architecture model that we follow for our investigation. We describe the important parameters of the hardware platform that must be taken into account for modeling. Further we describe the working of DMA on Cell processor and introduce a model to estimate performance of the applications using it.
- Chapter 4 introduces the satisfiability solvers and briefly describe the mechanism behind them. We give a small example of how a scheduling problem can be encoded and presented to such solvers. Further, we describe the multi-criteria optimization problem, where multiple costs must be optimized simultaneously. We present algorithms to efficiently track solutions to such problems.
- Chapter 5 explains the method of deployment and evaluation of our solutions. We briefly describe the framework in which we carry out the experiments and a runtime environment which is used to execute and validate the solutions discovered by our methods.
- In Chapter 6 we introduce the symmetry in the dataflow graphs and a method to encode them in to satisfiability constraints in order to accelerate the search for solutions. We

present the constraints by which the scheduling problem can be encoded in order to optimize latency, communication buffer size and number of processors.

- Chapter 7 introduces a new multi-stage approach for the scheduling problem for the Kalray processor. We demonstrate the modeling of communication and network flow-control to orchestrate the application execution in bounded distributed memory.

- In Chapter 8 we study the parallel applications with regular access pattern, which bring the data from main memory to the limited local memory, process it and write back the results again to the main memory. We present the work on optimizing the DMA transfer size, for such applications.

- In Chapter 9 we present a runtime system which optimizes resource usage of the system by dynamically reconfiguring the applications. We present a predictable method for such reconfiguration, without affecting other running applications.

- Chapter 10 concludes the thesis and presents the future work.

Programming Model

This chapter introduces the role of programming model and split-join graphs which we use to model streaming class of applications.

The description of a multi-core mapping and scheduling problem starts with a model for applications. Models are encoded and presented to the optimization solvers to find correct and efficient parallel schedules.

While studying an algorithm specification, one encounters numerous parameters, details available to the designer consideration. It is very important to select few of them which make the most significant impact towards the solution of the problem. If the model captures too fine-grain details, then it becomes difficult for the optimization solvers and any other search strategies to find optimal solutions. This makes the model of computation or the programming model, a very important aspect of embedded system design. It gives a structured view of how a given computation will execute and is annotated by important characteristics of the program. The model helps us to study the behavior of the entire system depending on the behavior of individual components. By study of such model using timed system formalisms, the designer can estimate the required resources. It is very helpful in order to gain an insight of the algorithms and performance of the machines on which applications represented by such models execute.

A programming model, in general, should provide the following information:

- Structure of the program
- Amount of computation of different elements
- Interaction among different elements of the program

Depending on the target class of applications and modeling objectives, there are different types of models to choose from. Timed Automata, Petri Nets, acyclic task graphs, Process Networks are examples of models. They are developed for different goals e.g. analyze deadlocks, ensure safe operations etc., and hence different formalisms are applied on the models to achieve these goals.

A class of applications called *streaming applications*, process a continuous stream of data for indefinite time. The input data arrives at a given rate, and is processed by algorithms defining such applications. JPEG decoder, MPEG decoder, H-263 encoder, filter banks etc. are examples of streaming applications. These applications perform a predetermined set of operations on input stream of data and can be easily represented with the type of models referred to as *dataflow graphs*.

In this chapter we describe a few relevant dataflow graph models which are typically used in signal processing applications. We focus on synchronous dataflow graph (SDF) model in detail. We present a model namely *split-join graph* which can be regarded as a sub-class of SDF model. Further we describe the semantics and behavior of this model. We conclude the chapter by defining method to convert a split-join graph to an acyclic task graph which is a commonly used model for mapping and scheduling tasks with dependencies.

2.1 DATAFLOW GRAPHS

Dataflow graph is a popular class of model of computation related to Petri Nets, which describes a computational process with evolving availability of the data. Dataflow graphs have nodes called actors, which represent the computation performed on a data. The edges between the actors carry tokens represents the communication of the data, which is processed by the actors. It emphasizes only on the dependency between execution of different actors. Dataflow graph does not describe any timing notions explicitly, however there are common extension of time for actors and/or edges.
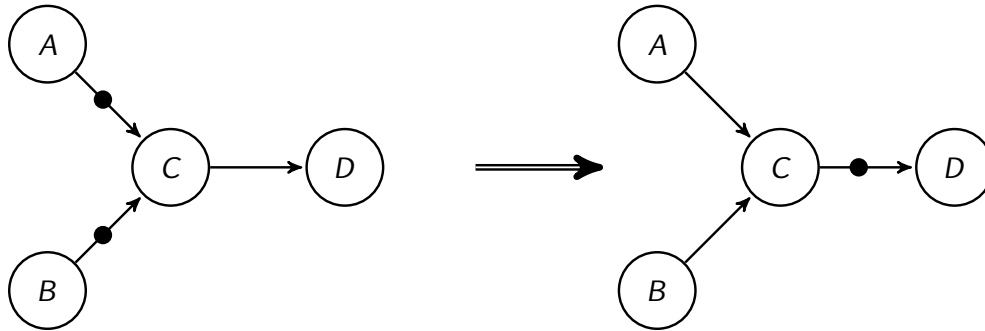
Figure 2.1 – Basic dataflow graph

Figure 2.1 shows an example of a dataflow graph. It has four computation actors namely *A*, *B*, *C*, *D*. Actors *A* and *B* produce data tokens which are taken by actor *C*. Actor *C* processes this data and produces an other data token used by actor *D*. This is represented by the edges, which exhibit the data dependency. The data tokens are depicted as black bullets in the graph. When the actors execute they consume the tokens present at their input edges and produce tokens on their output edges. In Figure 2.1, *C* consumes the tokens produced by *A* and *B* from their outputs and produces the tokens on the input edge of actor *D*.

In the dataflow terminology, the nodes called as actors produce or consume an amount of data on the edges called as *rate*. Different data flow graph models are distinguished by the rules that determine the number of tokens produced and consumed by actors. Dataflow graphs can be classified into static and dynamic depending on whether the rate of production and consumption of tokens is statically known. We discuss some of the dataflow models below.

2.1.1 *Static Dataflow*

These are very simple models which assume that the amount of data is known a priori to the execution. These models have completely predictable execution times and various other parameters of the model. This is a trade-off that has to be made in order to achieve simplicity and analyzability of the model.

Homogeneous synchronous data flow (HSDF) are the simplest dataflow graph similar to shown in Figure 2.1, where rate of every edge is equal to 1. Thus every actor produces a token

on outgoing edge while it consumes one token on an incoming edge. They can be seen as an extension to the acyclic task graph by cyclic paths and data tokens.

Synchronous Dataflow (SDF) graphs is a restricted version of dataflow in which the production and consumption rates (can be non-unity) are known at the compile time. Each actor fires or executes by consuming and producing a predefined number of tokens at its input and output respectively. Each actor may have different rates, however to an individual actor the actual rates are invariant. This restrictive property makes the model analyzable and easy to predict and produce a static schedule which can be repeatedly executed in bounded memory. Marked graphs [29], a class of Petri nets, are similar to HSDF graphs, while SDF can be regarded as equivalent to weighted marked graphs [125].

Computation graphs [67], another type of static dataflow model, are similar to the SDF, which are represented by a finite set of nodes connected with directed queues. In addition to token rates, it adds a threshold value, which is a minimum number of tokens that must be present at the input queue. SDF can be regarded as special case of computation graphs, where this threshold is equal to the consumption rate. Threshold property of this model is helpful in modeling sliding window algorithms, where the nodes need to read the data tokens without consuming them. These graphs find applications in DSP systems, where it is common to operate on a continuous stream of data, e.g. FIR filters, FFT algorithms etc.

There are various extension of SDF. Cyclo Static dataflow model [18, 40] enhances the SDF model by allowing periodically changing token production and consumption rates in contrast to the static rates of SDF. Multidimensional Data Flow (MDSDF) [89] supports the multi-dimensional tokens, such that an actor fires only if the tokens/space in all the dimensions are available. Such models are useful in applications like image processing where the data is represented in two or multiple dimensions. Windowed Synchronous Data Flow (WSDF) [68] is an extension of MDSDF, which supports sliding window algorithms.

2.1.2 *Dynamic Dataflow*

With the development of new applications and algorithms, the static models are not able to accommodate the conditional execution of the actors or varying data rates [17]. Dynamic dataflow are the models in which a set of parameters like the production and consumption rates are not completely known at compile time. This allows flexibility in modeling modern applications, however at the cost of analyzability of such models.

An example of dynamic dataflow is *Process Networks*. Kahn Process Networks (KPN or simply PN) [64] is model of concurrent computations which has a set of deterministic processes which communicate by unbounded FIFO (first-in first-out) queues. The processes block only when trying to read an empty queue, but the queues grow indefinitely when writing processes add data to them. Termination of a PN program is undecidable in finite time, as is boundedness of the queues. This property of queues makes an actual implementation infeasible in limited amount of memory. There are various algorithms to execute a process network in bounded memory, one such is described in section 4.2 of [95]. As compared to SDF, KPN is more expressive but difficult to make static analysis [46].

Boolean dataflow model (BDF) [23] is a extension of SDF model supporting conditional execution of an actor. It has two special actors called as *switch* and *select*. The switch actor has two output and one input, while select actor has two input and one output ports respectively. The former determines to which output port the tokens are produced, while the latter determines from which input port the tokens are consumed. The selection in both the cases is done with the help of a control port. This property makes the model difficult for compile time analysis. For example it is difficult to check boundedness of memory, absence of deadlock, compute a timed schedule, and the model in general is Turing complete.

Scenario Aware Dataflow (SADF) [116] is a class of extensions of SDF model, introducing the concept of scenarios. A predefined set of scenarios can be seen as different modes of operation of the model, in which the resource requirements, like communication rates and structure, differ considerably. While some properties of these graphs like deadlock and throughput in many cases are analyzable at design time, in practice this analysis can be computationally expensive.

There are many parametric extensions of SDF which allow updating of the parameters of the the dataflow graphs at run-time. Parameterized synchronous dataflow (PSDF) [16], Variable rate dataflow (VRDF) [131], Schedulable parametric data-flow (SPDF) [42] are a few examples of such extensions.

In this thesis, we use static dataflow models, typically SDF. Since our model of computation is closely related with SDF, we discuss it first in brief and then describe the split-join graphs which can be considered as a restriction to the SDF model.

2.2 SYNCHRONOUS DATAFLOW

SDF is one of the static dataflow models for computation. This model, introduced by Lee and Messerschmitt [78], provides a compact representation of applications which communicate data in regular fashion. The graph consists of actors which are connected by edges. An actor corresponds to a piece of code which has input and/or output. It performs some computation on the input tokens and as a result produces output tokens. The edges are marked with input and output rates. The actors communicate using buffers which are of limited size. An actor can fire when its input buffers have enough tokens available for firing and output buffers have enough space for tokens.
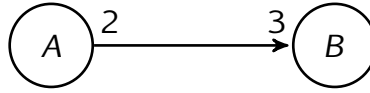
Figure 2.2 – Simple SDF with two actors

Figure 2.2 shows a very basic SDF. It has two actors namely *A* and *B*. The numbers marked on the edge connecting *A* to *B* denote the rates. Thus for this edge, the production rate is 2 and consumption rate is 3. It implies that when actor *A* executes, it will produce two tokens on the edge, while when actor *B* executes, it will consume 3 tokens on this edge.

An *iteration* of a graph is execution of all the actors of SDF for a fixed number of times greater than or equal to one. In an iteration, *consistency* property [115] of an SDF states that, for all edges in SDF, the amount of data produced on the edge of an SDF is equal to the data consumed on that edge. In short, the graph should return to the initial state after an iteration. Thus in the above example, initially there are zero tokens on the edge. If actor *A* executes 3 times, it will produce 6 tokens on its output edge, and if actor *B* executes for 2 times, then it consumes all the 6 tokens. Thus an iteration can be defined for this example for *A* executing three and *B* two times. Note that they can also execute for multiple of these values, for example 6 and 4 times respectively, but we always refer to the minimal values. An array which gives minimal number of times for execution of actor for graph to be consistent is also called *repetition vector*.

DEFINITION 1 (Repetition Vector) – *Repetition Vector is an array of length equal to number of actors in SDF, such that if each actor is invoked for the number of times equal to its entry, the number of tokens on each edge of SDF remains unchanged.*

Any SDF graph which is not consistent requires unbounded memory to execute or dead-locks [115]. When an SDF graph deadlocks, no actor is able to fire, which is due to an insufficient number of tokens in a cycle of the graph. Any SDF graph which is inconsistent or deadlocks is not useful in practice.
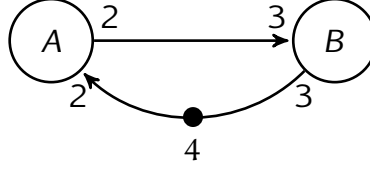
Figure 2.3 – SDF with backward edge

The edge between actor *A* and *B* represents a FIFO which carries data from the writer of the FIFO to the respective reader. In the example, if actor *A* fires continuously, the size of the FIFO will grow continuously and the graph will require unbounded memory for execution. SDF graph models bounded memory with the help of backward edges with initial tokens (similar to algorithm in [95]) as shown in Figure 2.3. The initial tokens represent the free space available in the forward edge. For every execution of actor *A*, it produces 2 (data) tokens on the forward edge, while it consumes 2 (space) tokens from the backward edge. Similarly actor *B* consumes 3 data tokens and produces 3 space tokens. The model itself does not differentiate between these tokens, and gives an opportunity to represent the communication using bounded memory.
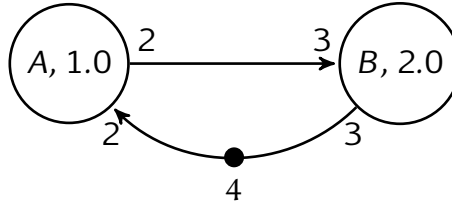
Figure 2.4 – Timed Synchronous DataFlow Graph

SDF graphs without timing notions are used to check correctness of the system (e.g. consistency and deadlock) or for property analysis (e.g. determining buffer-size). For performance analysis, timed variants are introduced (shown in Figure 2.4). The execution time of each actor is denoted with a number in the actor. Thus according to Figure 2.4, actor *A* takes 1.0 time unit, while actor *B* takes 2.0 time units for execution. However, the actors in this case has a fixed execution time and any variation in the execution time is not modeled, which makes it statically analyzable.

When scheduling the SDF graphs, fully static approach uses fixed actor execution times and schedules the actors according to strict timing and data dependency. Since dataflow applications continuously process a stream of data, the execution is periodic in nature. The periodic scheduling of the SDF can be classified into two types : (*i*) different iterations of graphs don't overlap (*non-pipelined*) (*ii*) different iterations execute in overlapping fashion (*pipelined*).

Figure 2.5 shows an example of non-pipelined and pipeline schedules. In the case of pipelined scheduling, there are two phases execution of SDF graph. The transition system consists of a finite sequence of states and transitions, called the *prologue* (or transient phase), followed by a sequence of states and transitions which is repeated infinitely often and is called the *periodic* phase. The periodic phase is very useful in analyzing the properties of the

(a) Non-pipelined   (b) Pipelined
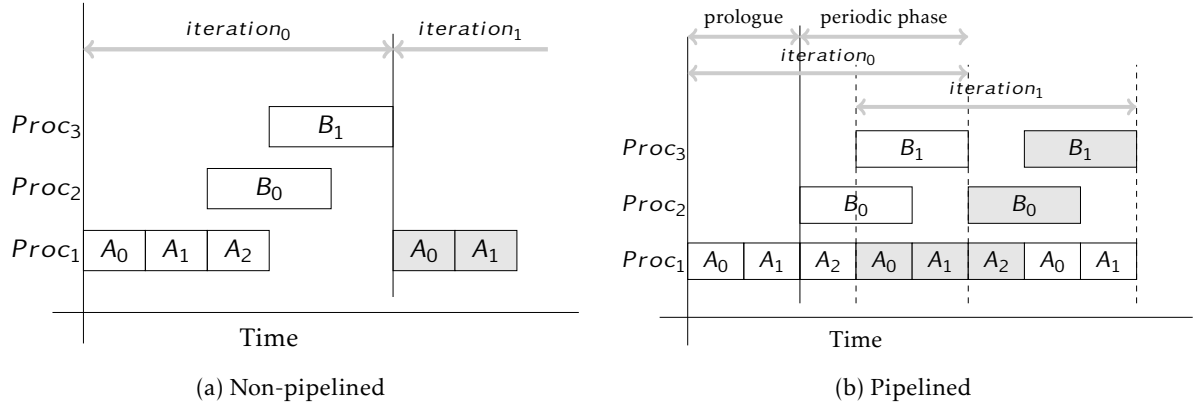
Figure 2.5 – Example schedule of a Synchronous DataFlow graph

application like throughput. It is difficult to predict and control the length of the prologue phase even for small SDF graphs. However, in the case of strictly periodic schedule one can indirectly enforce it, at the cost of some optimality loss [120]. In the case of non-pipelined schedule, the prologue is absent, while the schedule can be executed infinitely with bounded resources. For more details, the reader is referred to [115].

Pipelined scheduling is a complex task. From real-time scheduling point of view it incurs satisfaction of three cost/objective constraints at the same time: latency, processor count and throughput. Handling these three constraints at the same time is rarely studied in the literature of SDF and task graphs. Usually only two types of the above constraints are considered. In this thesis we also restrict to two of them – processor count and latency, sometimes adding buffer storage cost as an extra. Multi-criteria optimized scheduling is often done by using different generic solution search methods such as constraint programming [21], linear programming [50, 74], as well as model checking [47], genetic programming [112] etc. In this thesis we focus on non-pipelined scheduling, while pipelined scheduling is a future work. An interested reader can refer to [120] for our preliminary results on pipelined scheduling with throughput, latency and processor count criteria, using SMT solvers.

2.3 SPLIT-JOIN GRAPHS

SDF is a general programming model which can accommodate a wide range of streaming applications, and supports features like stateful actors, odd production and consumption rates etc. However in this work, we would like to focus on regular data-parallel applications. We would like to simplify this model in order to facilitate analysis while still covering most of the SDF applications. Thus, we define the split-join graphs, which can be thought as a sub-class of SDF graphs.

DEFINITION 2 (Split-Join and Task Graphs) – *A split-join graph S is defined by $S = (V, E, d, \alpha, \omega)$ where (V, E) is a directed acyclic graph (DAG), that is, a set V of actors, a set $E \subseteq V \times V$ of edges. The function $d : V \to \mathbb{R}_+$ defines the node execution time, $\alpha : E \to \{a, 1/a \mid a \in \mathbb{N}_+\}$ assigns a parallelization factor to every edge. An edge e is a split, join or neutral edge depending on whether $\alpha(e) > 1, < 1$ or $= 1$. $\omega(e)$ denotes the size of data tokens sent to each spawned task in the case of split, or received from each joined task in the case of join, or just sent and received once per execution if the edge is neutral. A split-join graph with $\alpha(e) = 1$ for every e is called an acyclic task-graph and is denoted by $T = (U, \mathcal{E}, \delta, \omega)$, where the four elements in the tuple correspond to V, E, d, and ω.*
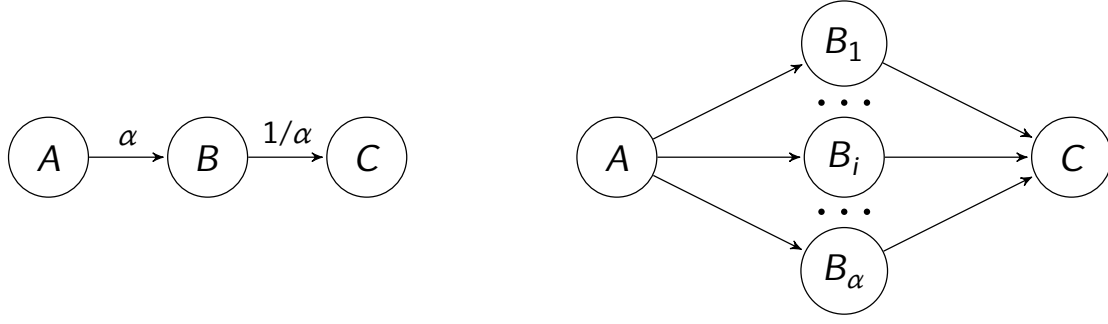
Figure 2.6 – A simple split-join graph and its expanded task graph. Actor B has α instances.

The split-join graph is a generalization of the acyclic task graph by data parallelism, explicitly represented by parallelization factors α. This is illustrated by the example in Figure 2.6. The decomposability of a task into parallelizable sub-tasks is expressed as a numerical label (parallelization factor) on a precedence edge leading to it. A label α on the edge from A to B means that every executed instance of task A spawns α instances of task B. Likewise, a $1/\alpha$ label on the edge from B to C means that all those instances of B should terminate and their outputs be joined before executing C (see Figure 2.6). An acyclic task graph can thus be viewed as derived from the split-join graph by making data parallelism *explicit* .

We call the nodes of the split-join graphs *actors* and those of the acyclic task graph *tasks*. The edges of a split-join graph $e \in E$ are called *channels*, and those in an acyclic task graph $\epsilon \in \mathcal{E}$ are called *dependency arcs* or just dependencies.

2.3.1 *The Semantics of Split-join Graphs*

In split-join graphs, the tasks (i.e. instances of an actor) can execute in parallel unless there are dependencies between them. In Figure 2.6 actor A spawns α instances of actor B, which can execute in parallel. Still, for convenience, we explain the functional behavior from sequential-execution point of view.

In sequential execution, channel $e = (v, v')$ can be seen as a FIFO (first-in-first-out) buffer. Let the task instances of each actor v execute in a fixed order, which determines their index: v_0, v_1, v_2 etc. Let $\alpha^{\uparrow}(e)$ be the amount of tokens (also called *production rate*) that are produced by actor v on channel e. The instances v_q of the writer actor of channel (v, v') produce $\alpha^{\uparrow}(v, v')$ *tokens* each in the FIFO channel; in the derived task graph $\alpha^{\uparrow}$ also corresponds to the number of outgoing dependency arcs of v_q. Similarly, $\alpha^{\downarrow}(v, v')$ denotes the number of tokens consumed (called *consumption rate*) and the number of incoming dependencies of instances v'_r of the channel reader actor. Mathematically these rates can be described as:

$$\alpha^{\uparrow}(e) = \begin{cases} \alpha(e) & \alpha(e) \geq 1 \\ 1 & \alpha(e) < 1 \end{cases} ; \quad \alpha^{\downarrow}(e) = \begin{cases} 1 & \alpha(e) \geq 1 \\ \alpha(e)^{-1} & \alpha(e) < 1 \end{cases}$$

Split-join graphs also follow the consistency property which was defined for SDF graphs in Section 2.2. Let $c(v)$ denote the *repetition count* of an actor v obtained from the repetition vector of the graph. The equations that express the consistency requirement are known as balance equations; are given for an edge(v, v') as:

$$\bigwedge_{(v,v') \in E} \alpha^{\uparrow}(v, v') \cdot c(v) = \alpha^{\downarrow}(v, v') \cdot c(v') \tag{2.1}$$

Note that if Equation 2.1 has a solution $c(v)$ then $k \cdot c(v), \forall k \in \mathbb{N}_+$ is a solution as well. However, we assume the *minimal* positive integer solution and use the notation $c(v)$ for it. Executing each actor $c(v)$ number of times is defined graph *iteration*.

The amount of data that is communicated in an iteration, on an edge can be then easily quantified. The tokens have size ω bytes, and the amount of data produced by an instance of v and consumed by an instance of v', for an edge e is:

$$w^\uparrow(e) = \alpha^\uparrow(e) \cdot \omega(e); \quad w^\downarrow(e) = \alpha^\downarrow(e) \cdot \omega(e)$$

where $w^\uparrow(e)$ and $w^\downarrow(e)$ is total amount of data produced and consumed on edge e respectively.

Given this description of the split-join graph, it can be easily converted into another graph namely *task graph*. In Chapter 6, we introduce *well formedness*, a strong property for split-join graphs (can be seen as a restriction on SDF graphs), where we define a strict nested structure. Split-join graph is comparable to SDF in terms of decidability and complexity of analysis. However it presents structured view of regular nested loops that appear in programs and hence it is easier for implementation [128]. Further with the well-formedness restriction we apply on split-join graphs, it is relatively easier to present the theory of symmetry elimination on identical task instances derived from the same actor.

2.3.2 *Derived Task Graph*

The task graph derived from a split-join graph models one graph iteration. [1] For each actor v the task graph contains $c(v)$ tasks $\{v_0, v_1, \ldots v_{c(v)-1}\}$, i.e. the *instances* of actor v.

Let us define the edges of the derived task graph. For a split-join channel (v, v') with $\alpha(v, v') = a/b$ let us consider the sequence of $a \cdot c(v)$ tokens produced in the FIFO buffer in one iteration. Let us number these tokens by index i in the order they are produced by the instances of actor $v : v_0, v_1, \ldots$. Obviously, token i is produced by task v_q where $q = \lfloor i/a \rfloor$. The actor instances consume the tokens in the same order as they are produced (the FIFO order). Therefore, the first b tokens will be consumed by task v'_0, then the next b tokens by v'_1, etc. In general, the token i is consumed by task v'_r where $r = \lfloor i/b \rfloor$. To model this *producer-consumer* dependency of token production and consumption, the task graph should contain edge (v_q, v'_r). The derived task graph then can be defined formally as:

DEFINITION 3 (Derived Task Graph) – *From a consistent marked split-join graph $S = (V, E, d, \alpha, \omega)$ we derive the task graph $T = (U, \mathcal{E}, \delta, \omega)$ as follows:*

$$U = \{v_h \mid v \in V, \ 0 \le h < c(v)\}$$

$$\mathcal{E} = \{(v_h, v'_{h'}) \mid (v, v') \in E \wedge \epsilon(v, v', h, h')\}$$

where ϵ is predicate defined by:

$$\epsilon(v, v', h, h') : \exists \, i \in \mathbb{N} : \ h = \lfloor i/\alpha^\uparrow(v, v') \rfloor,$$

$$h' = \lfloor i/\alpha^\downarrow(v, v') \rfloor, \ v_h, v'_{h'} \in U$$

and:

$$\forall (v_h, v'_{h'}) \in \mathcal{E} \quad \omega(v_h, v'_{h'}) = \omega(v, v'),$$
$$\forall v_h \in U \quad \delta(v_h) = d(v)$$

2.3.3 *Marked Split-join Graphs*

We have seen two different extension of acyclic task graphs : graph with initial tokens and split-join graphs. Here we combine them into one. In order to model the graph in bounded

---

1. In SDF terminology, deriving a task graph is equivalent to deriving a homogeneous SDF graph.

buffer, marked graphs introduce notion of channel marking, an equivalent of SDF with initial tokens.

DEFINITION 4 (Marked graph) – *The marked (split-join) graph S can be defined as a split-join graph extended by allowing cyclic paths and introducing an extra edge parameter – the marking: $m : E \rightarrow \mathbb{N}_{\geq 0}$. The extended tuple for a marked graph is thus: $\Sigma = (V, E, d, \alpha, \omega, m)$, where m represents the* marking *on the edges. We assume that any split-join graph is a marked graph with zero marking.*

The conversion of marked split-join graph is similar to that of *un-marked* graph. A non-zero marking $m' = m(v, v')$ has the semantics of initial availability of m' tokens in the channel. Therefore, for the case of a marked graph in the above example we have to calculate r as $r = \lfloor (i + m')/b \rfloor$. Without initial tokens the consumer of the token i produced on the channel (v, v') is task v'_r where $r = \lfloor i/b \rfloor$. However, now this consumption can be seen as shifted by value m. The derivation of the task graph then can be updated as follows.

DEFINITION 5 (Derived Task Graph) – *From a consistent marked split-join graph $S = (V, E, d, \alpha, \omega, m)$ we derive the task graph $T = (U, \mathcal{E}, \delta, \omega)$ as follows:*

$$U = \{v_h \mid v \in V,\ 0 \leq h < c(v)\}$$

$$\mathcal{E} = \{(v_h, v'_{h'}) \mid (v, v') \in E \wedge \epsilon(v, v', h, h')\}$$

where ϵ is predicate defined by:

$$\epsilon(v, v', h, h') \ : \ \exists\, i \in \mathbb{N} \ : \ h = \lfloor i/\alpha^{\uparrow}(v, v') \rfloor,$$

$$h' = \lfloor (i + m(v, v'))/\alpha^{\downarrow}(v, v') \rfloor,\ v_h, v'_h \in U$$

and:

$$\forall (v_h, v'_{h'}) \in \mathcal{E} \quad \omega(v_h, v'_{h'}) = \omega(v, v'),$$
$$\forall\, v_h \in U \quad \delta(v_h) = d(v)$$

2.4 SPLIT-JOIN GRAPH APPLICATION EXAMPLE : JPEG DECODER
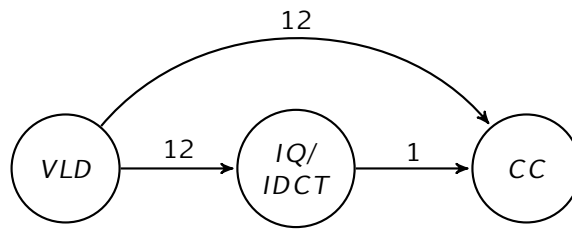
Figure 2.7 – JPEG decoder

Figure 2.7 shows a JPEG decoder expressed as Split-Join graph. It has three main actors: variable length decoding (*VLD*), inverse quantization combined with inverse discrete cosine transform (*IQ/IDCT*) combined and color conversion (*CC*). The VLD actor is responsible for decoding the JPEG parameters which are added to the image as header. After the header is decoded, it performs variable length decoding on the image data which is then converted into blocks. These image blocks are then passed to the IQ/IDCT actor which first performs inverse quantization and then performs inverse discrete cosine transform on these blocks. Finally the color actor performs color conversion which then finishes the decoding from JPEG image to Bitmap image format.

The edges in this application are used to communicate JPEG image parameters which are defined in the header and the image data. The rates are proportional to the size of image that is being decoded 32×24 pixels in our case.

2.5 CONCLUSION

In this chapter we discuss various aspects of programming model and how streaming application can be expressed in our model, namely split-join graph. Split-join graph model is similar to the fork-join model used in [9]. Both typically represent a well-structured nested loop computation which is typically observed in certain class of applications. In fork-join graphs, the tasks are divided in stages and segments. This model has a stricter requirement that tasks in the same stage can execute concurrently, while tasks in preceding stages must complete before. In split-join graph, we have a general expression of parallelism in which precedence constraints are expressed only via connected actors. Further we annotate the edges with the amount of data transfer to model the communication.

We also discussed other existing models, which can also be used for expressing such applications. However, in order to perform formal analysis, in further chapters, we need a simple model. Our model can be regarded as a subset of SDF graphs, thus enabling us to readily use some of the SDF benchmarks, or many of them with little or no modifications.

In next chapter we discuss characteristics of various hardware platforms and important parameters to model them. We further show how these parameters are effectively used in solving the mapping and scheduling problem on these hardware platforms using satisfiability solvers.

Architecture Model

After having considered the application programming model, in this chapter we introduce the hardware architecture model, which is used to model the multi-core hardware platforms.

Many-core / Multi-core processors exploit a collection of complex mechanisms and sub-systems, designed for specific purpose to accelerate certain functions. For example, SIMD (Single Instruction Multiple Data) instruction accelerates execution of instructions by executing same operations concurrently on multiple data. Such mechanism is typically used in signal processing manipulations, like matrix multiplication, where the same operation can be performed on multiple elements in parallel. DMA (Direct Memory Access) is another example, where the DMA engine facilitates the overlapping of computation and communication by performing asynchronous data transfer without intervention of a processor. Multi-core processors operate at a different level of granularity than SIMD, and they can be used to accelerate data-parallel applications. In the previous chapter we observed different programming models and how they can be used to express parallelism in certain applications. Applications, represented by such parallel models, can be accelerated on multi-core processors by executing concurrently.

With multi-core platforms the space of design-parameters is huge. They include the number of processors used, the amount of memory used, power / energy consumption, communication costs, latency, throughput and many more. If all these parameters were considered together with hardware platform low-level details such as instruction-level details (like instruction-set in simulation), the combinatorial problem of mapping and scheduling would be unmanageable. Thus with the large design space, the decisions for executing an application on these platforms must be taken at higher level of abstraction. Scheduling is an old problem and different formalisms and approaches have been developed in order to solve it. In order to study scheduling, the tasks in the application are annotated with the timing parameters, typically worst case execution time of the task. Thus the instruction-level details of the tasks are abstracted away, which simplifies the decision-making process for application execution. If an application is to be deployed efficiently on a platform, then we need accurate models of the platform as well. A similar approach is applied to multi-core processors, where the minute details of the platform are abstracted away retaining only important parameters. A decision-making theory then can be applied on the combined models of application and platform in order to find an efficient solution to the scheduling problem specific to the platform.

In this chapter, we discuss the multi-core and many-core processor architectures in general and we give details of some particular architectures. Then we describe the way we model these

architectures.

3.1   MULTI-CORE AND MANY-CORE PROCESSORS

When multi-core processor came into existence, they typically consisted of few processors with large caches located near the processor in shared global address space. These processors were rich with instruction-level acceleration mechanisms such as multi-stage pipeline, branch prediction, cache coherency etc. Further scaling of such sophisticated cores is difficult owing to various issues such as power consumption, design complexity, physical layout etc. which became a strong motivation for development of many-core processors.

In a many-core platform, multiple multi-core clusters are networked on a chip. The platform has a powerful general-purpose processor which can be located either on-chip or off-chip. This processor has full capabilities such as cache and cache coherency, coprocessor support, floating point engine etc. This processor is also called a *host CPU*. The *accelerator fabric*, the many-core system itself, contains simplified processor cores which can accelerate computation by executing application code in parallel. The processor cores in the accelerator fabric are grouped in clusters. The processors (in the cluster) share a finite amount of resources like local memory and can function independent of each other. Figure 1.3 shows a functional diagram of such a processor.

Such processor architectures, although complex, can be efficiently utilized for various applications. One important usage scenario is when a host CPU runs all the usual software stack and general-purpose tasks, whereas computationally expensive highly parallel kernels are forwarded to the many-core fabric. We discuss the components and terminology of these processors below.

3.1.1   *Clusters*

In order to facilitate the development of both software and hardware, multiple cores are grouped together as clusters which share finite resources like DMA engines, local or intra-cluster memory etc. The processors inside the cluster typically are light-weight processors (with limited capabilities in terms of pipeline stages, cache mechanism, or virtual addressing). Power consumption and design challenges are the fundamental reason behind such architecture. These processors are designed to perform computations independent of each other and communicate efficiently. Within a cluster, the processors can communicate using various mechanisms such as shared memory, common registers, etc. The communication outside the cluster is managed with help of asynchronous mechanisms such as DMA (discussed further). Typically, communication and synchronization inside a cluster is faster than between clusters.

3.1.2   *Shared Memory*

A limited amount of memory is generally made accessible to all the processors inside the cluster. Memory is usually multi-bank in order to provide good performance scalability by preventing memory conflicts due to usage of different banks. It is also called *intra-cluster memory* or local memory. This local memory ranges typically from some kilobytes to a few megabytes and has access latency usually less than 10 clock cycles. The main memory has access latency of around hundreds to a few thousand clock cycles and is of several order of megabytes or gigabytes. Typically the program is first loaded in the main memory and the program execution is started. The clusters must fetch the program data into intra-cluster memory, and subsequently perform computations on it. The processors don't have direct access to the main memory, but can fetch data from main memory to local memory. The data movement

between them is facilitated by asynchronous mechanisms namely DMA, explained further. This hierarchical organization of memory provides the programmer benefit of performing computations on a part of local memory concurrently with, asynchronous data transfer between local and main memory or different local memories.

3.1.3 *Network-On-Chip*

The multi-core / many-core processors contain network on chip (NoC) in order to facilitate data movement between clusters or between cluster and other hardware peripherals like I/O devices. A processor which is connected to the NoC via a network interface initiates a data transfer. Data is pushed on the network in form of packets. The size of these packets depends on the NoC architecture. It may also contain other information like the route, the destination address etc. The role of the NoC is also to provide arbitration between packets with conflicting routes and to ensure correct delivery of data from the source to destination. There are various arbitration policies such as round-robin, priority-based, etc. which can be used to resolve the conflicts.

3.1.4 *DMA*

A DMA controller is a piece of hardware used to move data between isolated memories, possibly asynchronously with respect to the core program execution. If data transfer is synchronous, the processor remains blocked as long as data transfer is in progress. The main memory access latency can reach few thousands of clock cycles. If the processor had access only to that memory, it will spend most of its time in blocking for memory access rather than performing useful computations. This scenario can be improved by having a local memory in the cluster such that data is fetched from the main memory to the local memory and then the processors perform the computation in the local memory. When the computation is finished, the results are written back to the main memory. This data movement is performed by the DMA controller. In order to efficiently overlap computation and communication, the DMA controller, after being setup, can function without any intervention from the processor to move the data. Different parameters such as amount of data to be transferred, the stride, the source and destination address are specified by the programmer in the setup of a DMA controller. Stride is an optional argument which is explained in later section. The DMA controller can signal back to the main processor upon completion of the data transfer by mainly two methods: *polling* and *interrupt*. In polling method the processor can check for a flag for completion of data transfer, using an interrupt mechanism the processor is interrupted of its current task and a service routine is executed to process the transfer completion.

We discuss in detail different platforms that admit such mechanisms.

3.2 TILERA TILE64

Tilera Tile64 Pro [129] processor architecture consists of 64 symmetrical processors (shown in Figure 3.1(a)), running at 862.5 MHz. Each processor (also called tile) is connected to a switch engine which can route communication packets on six independent networks in a mesh topology (seen in Figure 3.1(b)). Each processor core contains three-way VLIW processor with three instructions per bundle. It contains three computing pipelines P0, P1 and P2.

- **P0**: executes arithmetic and logical operations.
- **P1**: executes arithmetic and logical operations, branching instructions, read/write SPR (special-purpose registers).
- **P2**: executes memory load/store, test/set operations.
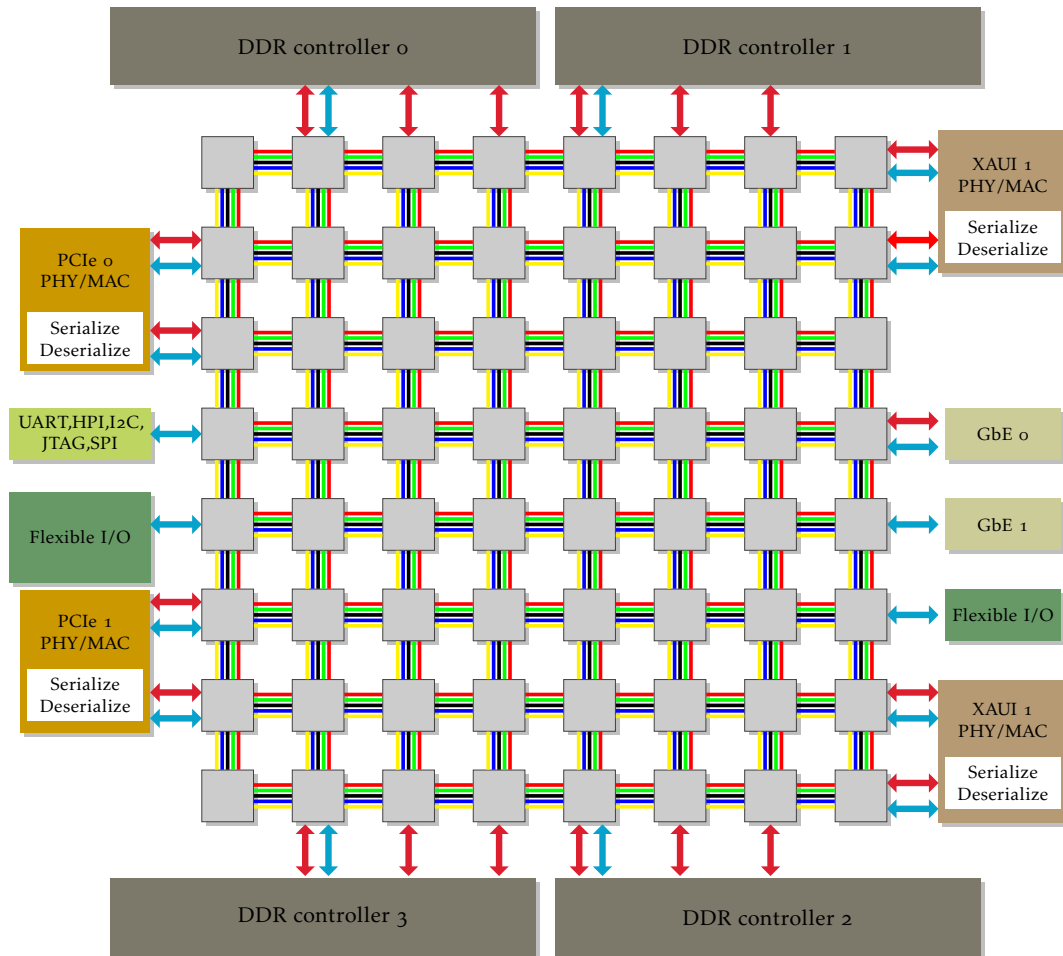
(a) Tile-64 processor architecture
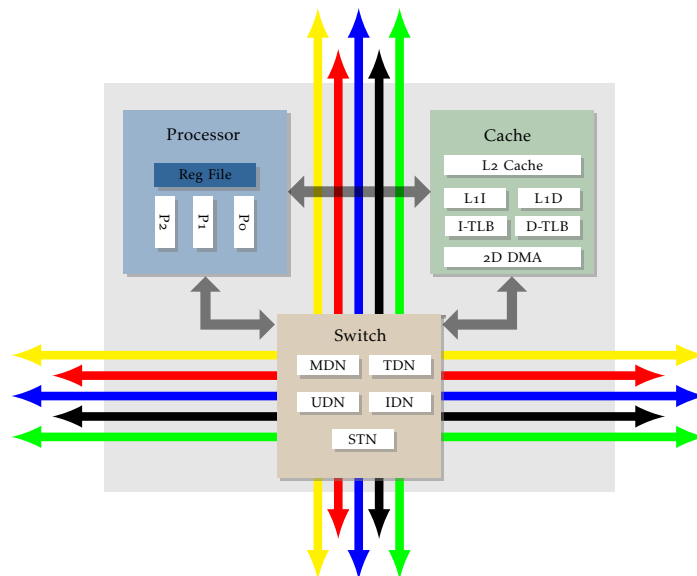
(b) Processor core

Figure 3.1 – Tilera Tile-64 processor

The Cache engine of this platform contains Translation Look-aside Buffers (TLB), which are used to support virtual memory, required for the Linux platform. The platform contains a multi-level cache. Each tile contains 16KB of Level-1 (L1) Instruction cache, 8KB of L1 data Cache and 64 KB of Level-2 (L2) cache. This accounts for total of 5.5MB on-chip cache. Each tile also has a 2D DMA engine for data transfer between the tiles and main memory, however it is currently unsupported by the software.

Every tile is also connected to different networks mentioned below -

- **User Dynamic Network (UDN)**: It is the only visible network to the user which is exposed by C library routines for streaming the data between tiles.
- **Memory Dynamic Network (MDN)**: It is interfaced with the cache engine to carry the memory traffic between tiles (such as cache misses, DMA) and between tile and main memory.
- **Coherence Dynamic Network (CDN)**: It carries the cache protocol messages to maintain the hardware cache coherence and is invisible to the programmer.

In addition there are other three networks which support traffic for I/O devices and other purposes, but are not relevant to our work. The network channels include hardware flow control and buffering mechanisms to enable asynchronous data transfers. The dynamic networks are packet-switched mesh networks which employ "XY-routing" mechanism. The packets from source to destination are switched along x-axis first and then to y-axis. Thus the hardware is robust and efficient in maintaining coherent data across all the processors and our software can use such mechanisms in order to perform communication transparently.

In addition it contains SIMD instructions for sub-word parallelism and instructions like saturating arithmetic for acceleration of DSP algorithms. We assume that the compiler is responsible for utilizing the available instruction-level parallelism.

We use this architecture to demonstrate the constraint-based solving of scheduling problem using SMT solvers. We produce solutions with different resource usages and validate them by executing on this platform. The details of the experimentation are given in Chapter 6.

3.3 KALRAY MPPA-256

Kalray MPPA-256 [65] platform is shown in Figure 3.2. This platform consists of 256 symmetrical processors which are grouped together, 16 processors in groups called *compute clusters*. These compute clusters are connected by a network of 2D toroidal topology. Apart from compute clusters, the platform also has four I/O subsystems, which are group of four cores each, called *I/O clusters*. There are two NoCs on the chip; one is for data transfer (data NoC), optimized for bulk transfers, while the other is for control messages (control NoC), mainly optimized for low latency. A cycle accurate timer is also integrated inside a cluster, accessible by any processor in the cluster and used for time measurement. The Kalray platform has a host processor, an Intel CPU, which is connected to the multi-core chip via PCI bus.

Each compute cluster contains 17 VLIW cores with a 16-bank shared memory of 2 MB capacity. Out of 17 cores, one core is called Resource Manager(RM), and is dedicated to operating system functions. The remaining 16 cores, called *Processing Elements* (PE), can be used for executing application threads based on the pthread model of execution. Every core implements a 32-bit 5-issue Very Long Instruction Word (VLIW) architecture with a 7-stage instruction pipeline. It includes two arithmetic and logic units, a multiply-accumulate/floating-point unit, a load/store unit, and a branch and control unit. These five execution units are connected through a shared register file of 64 32-bit general-purpose registers (GPRs). It also has independent 8KB instruction cache and 8KB data cache to hide the access latency to the
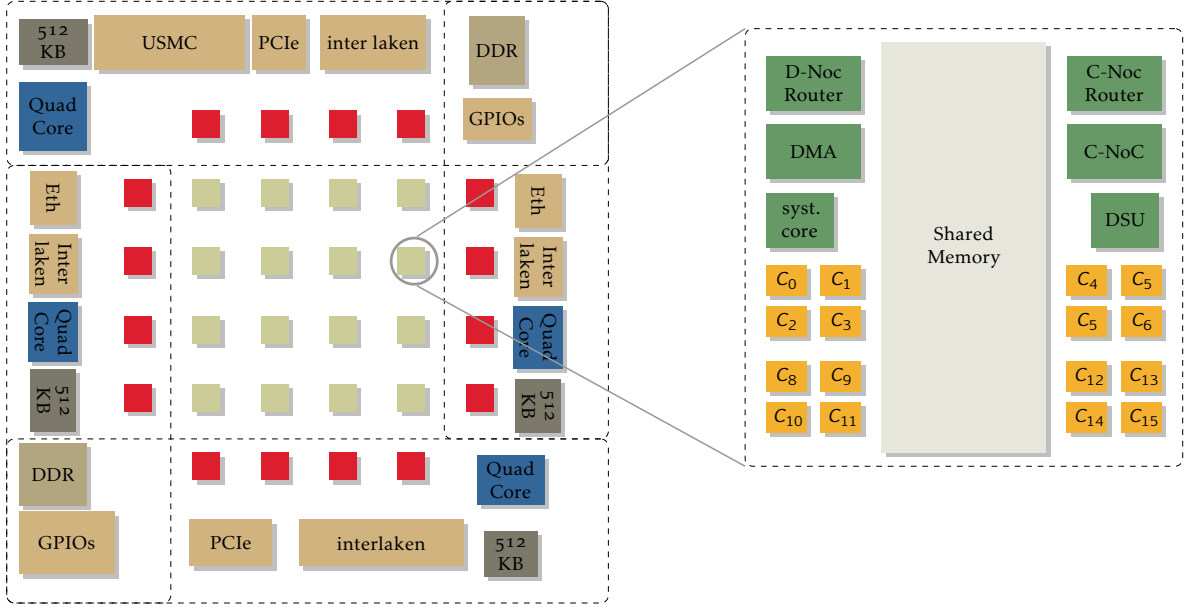
Figure 3.2 – Kalray MPPA-256 Platform (with zoom into a compute cluster)

2MB shared memory. There is no hardware cache coherency in compute clusters, and it has to be explicitly managed by the software.

Kalray MPPA software library provides an abstraction layer which provides efficient communication and synchronization primitives by hiding low-level hardware details. It provides the API (Application Programmer Interface) for setting up the DMA, for communication between compute clusters, IO clusters and host. The RM core is a privileged core and is responsible for executing the kernel routines. The main task on RM core is waiting on events from PE which are system calls to execute kernel routines. The PE is blocked till the system call is being processed, and after finished is unblocked by an event from RM core. The RM core being privileged is responsible for initiating the data transfers by setting up the DMA engines and can be accessed with MPPA library API. It also handles the hardware interrupts.

The host is connected to the IO cluster subsystem via PCI/e connection. The MPPA library also provides API which facilitates synchronization and data transfer to and from IO subsystem via PCI. More details about the architecture and its software library are given in [36].

In Tilera architecture, the data movement between the processors was managed by the hardware using cache coherence mechanism. However such a mechanism does not exist on Kalray architecture and data movement is managed explicitly by the software. Moreover the cost of moving the data is non-negligible and has to be modeled to predict the software performance accurately. We explicitly model the inter-cluster communication and orchestrate it along with the scheduling of the computation tasks. We execute a number of benchmarks on this platform to validate our results. Further details are presented in Chapter 7.

3.4 COMPSOC PLATFORM

The CompSOC platform [5] (see Figure 3.3) , is a research platform developed by the Eindhoven University of Technology. It is a tile-based architecture in which a set of processing and memory tiles are connected to each other via the Æthereal network-on-chip [48]. Each processor tile contains a Microblaze processor running the CompOSe real-time operating sys-
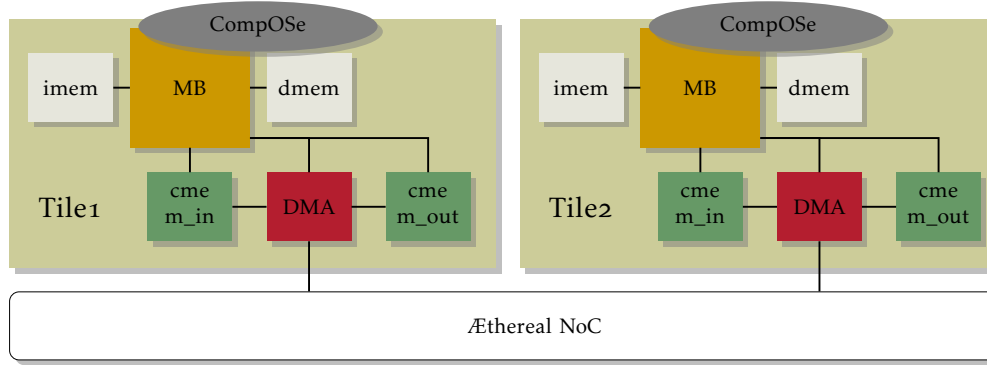
Figure 3.3 – CompSOC platform

tem [54]. CompOSe provides composable and predictable application scheduling. Composable means starting or stopping of a certain application doesn't affect timing behavior of other applications running on the same processor.

A processor tile contains also a non-shared local memory for instructions and data, as well as communication memories which are used by a DMA for communication with remote tiles. Memory tiles contain a memory sub-system that can be accessed from the processing tiles. The tiles that communicate with memory and other processing tiles using Æthereal NoC. This NoC has slots which are reserved by applications, guarantees a fixed bandwidth.

The CompSOC platform provides a predictable and composable timing behavior to applications running on the platform [4]. In order to provide composability, it uses a composable scheduling strategy such as time-division multiplexing (TDM), where the presence or absence of requests from one application cannot affect the scheduling decisions for other applications. In addition, it uses preemption after a fixed time to prevent one application from starving another. Furthermore, it delays scheduling of another task till the end of the time slice to avoid that the early completion of one request will cause subsequent requests to be scheduled earlier. In order to provide predictability, it requires that all data needed for a request must be locally available and that there should be enough local storage space to store the response of this request. In combination with the use of predictable resources with bounded worst-case execution times and the use of a predictable TDM scheduler, it is possible to compute a worst-case response time for any task running on a resource. The TDM scheduler can be thought as a time wheel with fixed number of time-slices. Each application has a dedicated number of slices for execution, and is pre-empted on completion of its slices. This guarantees the amount of time an application will have in the time wheel. The sequence of these time slices is then repeatedly executed until the platform is stopped explicitly. Thus the worst-case response time can easily be calculated, and depends on the number of slices allocated to the application, total number of slices, and worst case execution time of the application. Complete details about this platform can be found in [4, 5].

In Chapter 9 we discuss how we can perform reconfiguration of the applications for global optimization of the resources. For such a reconfiguration, the system must be able to predictably migrate the tasks of an application, without affecting execution of other applications. This can be achieved with a support from hardware. For more details the reader is referred to Chapter 9.

3.5 IBM CELL BE PROCESSOR

The Cell Broadband Engine Architecture [52, 53] is a 9-core heterogeneous multi-core architecture, consisting of a Power Processor Element (PPE) linked to 8 Synergistic Process-

ing Elements (SPE) acting as coprocessors, connected through internal high speed Element Interconnect Bus (EIB) as shown in Figure 3.4.
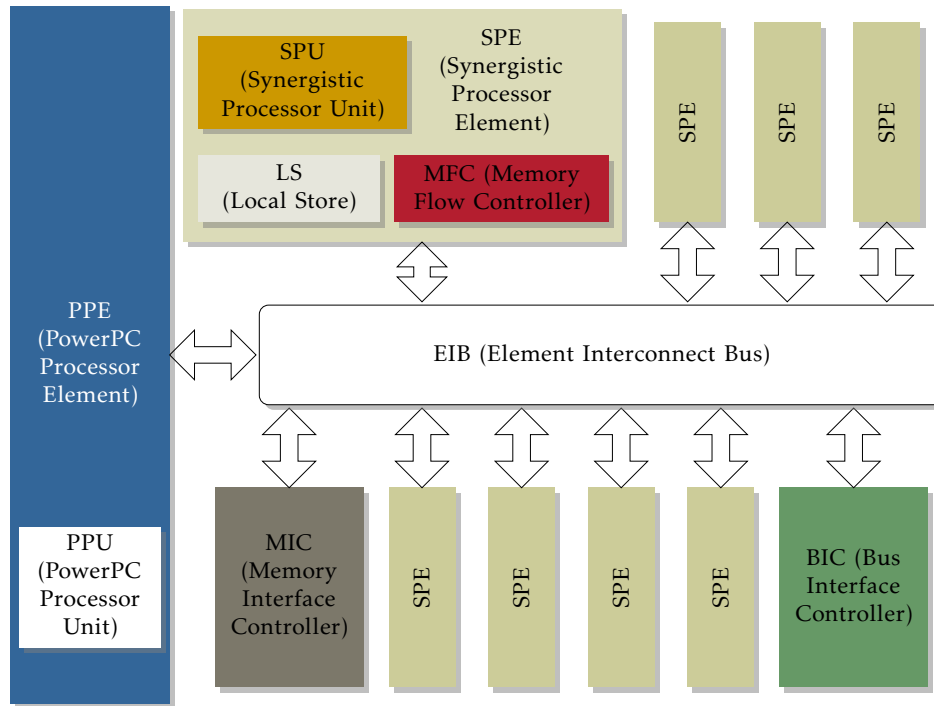
Figure 3.4 – IBM Cell BE processor

The PPE is composed of a general-purpose 64-bit RISC processor called PowerPC Processor Unit (PPU) integrated with cache mechanism and bus interface. Each SPE is composed of a Synergistic Processing Unit (SPU) which is a vector processing unit, a SRAM local store (LS) of size 256 kbytes shared between instructions and data, and a Memory Flow Controller (MFC) to manage DMA data transfers. The PPE has a single shared address space across SPEs and the memory translation units in MFC handle the required address translation. An SPE can access the external DRAM and the local store of other SPEs only by issuing DMA commands. The PPU does not have its own MFC but can initiate DMA on behalf of an SPU, by exclusively accessing its MFC. An MFC supports aligned DMA transfers of 1, 2, 4, 8, 16 or a multiple of 16 bytes, the maximum size of one DMA transfer request being 16K. To transfer more than 16K, DMA lists are supported.

The cell processor operates in virtual memory environment. The MFC of each SPE is composed of a Direct Memory Access Controller (DMAC) to process DMA commands queued in the MFC and of a Memory Management Unit (MMU) to handle the translation of DMA generated addresses. The MMU has a translation look-aside buffer (TLB) for caching recently translated addresses, which is explained in detail ahead. After the address translation, the DMAC splits the DMA command into smaller bus transfers and peak performance is achievable when both the source and destination address are 128-byte aligned and the block size is multiple of 128 bytes [71]. We can observe reduction in performance when this is not the case.

Processors use a technique called virtual memory, where an application executing on them is presented is presented with a contiguous virtual address space representing the main memory and secondary storage. This memory is divided in a unit called *pages*. A virtual page is mapped to a physical page when accessed and the translation information is marked in a *page table*. Page size is a design-parameter and has its own tradeoffs. For example, if we increase

the page size, it will cause lesser number of page table entries, but will require larger area in physical memory a part of which might remain unutilized [56].

When an application accesses an address, a virtual address translated to physical one referring to the page table. For every memory access, a corresponding entry in the page table is fetched and its physical address is calculated. Thus one virtual memory access requires two physical memory accesses (one for page table entry and other for data). Since memory is generally much slower than the processor, a special cache is added to the hardware called *Translation look-aside buffer* (TLB), which contains recently accessed page table entries. On a TLB miss the hardware has to locate the required entry in the page table. The application has view of all the memory available, however not all the physical pages are allocated at the start of the page execution to avoid the wastage of the physical memory in case if it remains unused. If the accessed physical page corresponding to the virtual address is unallocated, a new physical page is allocated by invoking complex kernel memory allocation routines. The page table entry is updated and then loaded in TLB. This allocation and translation costs many thousands of cycles on the Cell processor and should be avoided if possible [71]. In order to avoid this overhead we allocate *huge* pages, where the page size is 16MB. Note that one page has one entry in TLB, thus for this entire 16MB of data there will be only one entry in TLB. We allocate the application data strictly in this page. In our experiments (see Section 8.5) a warm-up run is performed where we access this page, loading the entry in the TLB of SPE. Next time the address of application data is accessed, it takes only a few cycles for address translation.

In the Cell Simulator that we use [59] the PPE and SPE processors run at 3.2GHz clock frequency and the interconnect is clocked with half the frequency of the processors. The Cell-simulator gives performance predictions close to the actual processor. The cell-simulator approximates the memory performance by using a DDR2 memory model. Further the TLB replacement policies in the SPE uses a Replacement Management Table (RMT) for replacement of TLB entries. The simulator does not support RMT. However in our measurements, we subside the effect of TLB using large pages and a warm-up run thus does not affect our model. Further details of the architecture can be obtained at [58].

We study data-parallel applications with regular access patterns and how they perform uniform computation by fetching the data from main memory to the local memory and writing back the results again to the main memory. The entire application data doesn't fit in the local memory and hence must be brought in the local memory in batches. The transfer granularity influences the performance of the application and is an optimization problem. We study this problem for the Cell architecture. More details are available in Chapter 8.

3.6 DMA CONTROLLER IN CELL PROCESSOR

In this section, we present the details of the DMA controller of the Cell processor and how it can be modeled. The model can be trivially adapted to the Kalray processor, which also supports DMA transfers.

Figure 3.5 shows a basic flow of a DMA command inside the SPE of the Cell processor. The SPU initiates a DMA transfer command through the channel interface. Then the command is enqueued in the DMA controller (DMAC) which can serve at maximum 16 pending requests. The DMA controller then selects a data transfer command following some complex set of rules. If the command is a *write* then it will access the local store to fetch the data to be written to a remote memory. The TLB unit provides the address translation for the memory requests. The DMA command is split into chunks of 128 bytes which is the size of the data transfer on the network, which is then enqueued into bus interface unit (BIU). The BIU then pushes the
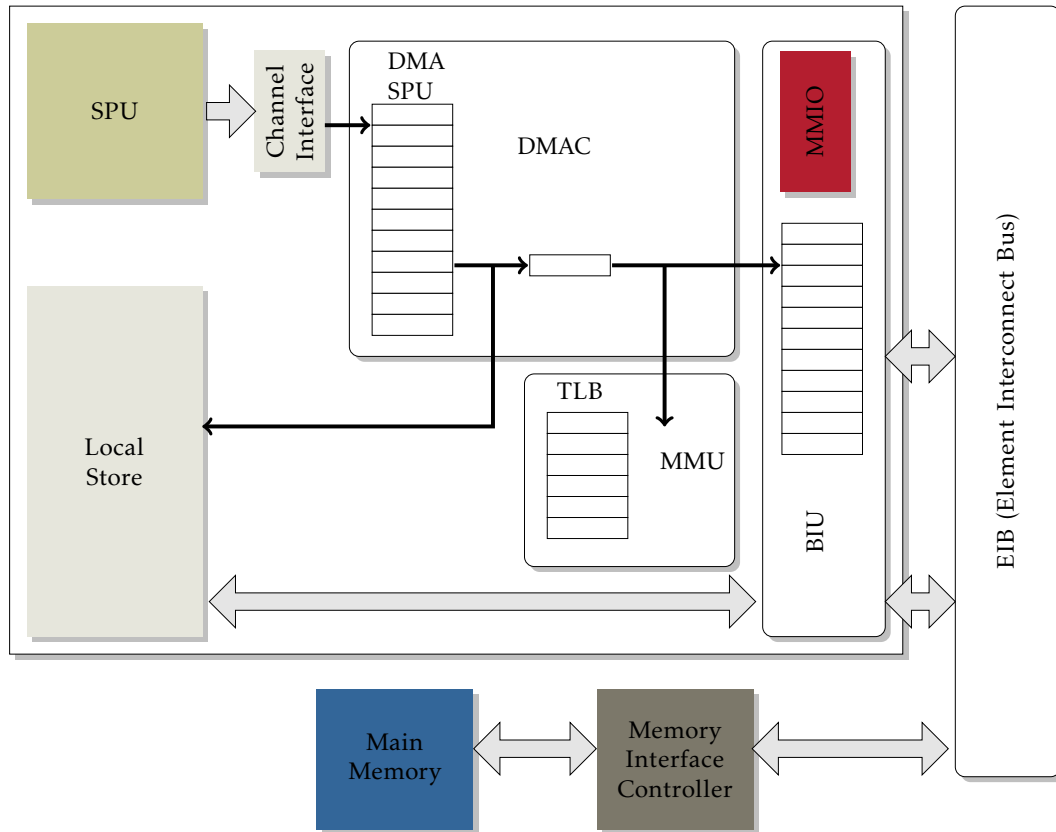
Figure 3.5 – IBM Cell processor DMA controller

data on the interconnect performing handshake with the memory interface controller. For read requests similar steps are performed, except that the direction of data is reversed and local store is accessed by BIU to write when fetched from a remote memory. For more details please refer to [71].

The DMA controller is a complex piece of hardware and it is difficult indeed to model it accurately. However we make simplified version of the model by observing the amount of time the controller takes to read or write the data. A DMA transfer constitutes of two main phases:

- *Initialization Phase*: This phase includes time to write the command in the DMA controller. Typically during this time the processor is blocked and cannot be used for any computation. The time required for this phase is independent of the amount of data to be transferred.
- *Data Transfer Phase*: This phase is where the command is ready and the data is actually transferred on the network. The duration of this phase is naturally proportional to the amount of data that is being transferred.

It is more efficient to transfer few large chunks of data than many small ones, as the controller can amortize the initialization phase costs.

3.6.1 *Strided DMA*

Remember that memory is made up of contiguous locations which are accessed by specifying addresses. For example a 1kbytes of memory can be accessed by address 0 to 1023. When the software allocates an array in the memory, it will allocate it in a contiguous address space. In image processing applications a raw image will therefore be stored incrementally

Address	Memory contents
0x0	row_0, col_0
0x1	row_0, col_1
0x2	row_0, col_2
0x3	row_0, col_3
	$\cdots$
0xFD	
0xFE	row_7, col_6
0xFF	row_7, col_7

(a) Pixels of an image in row and columns   (b) Image stored in memory
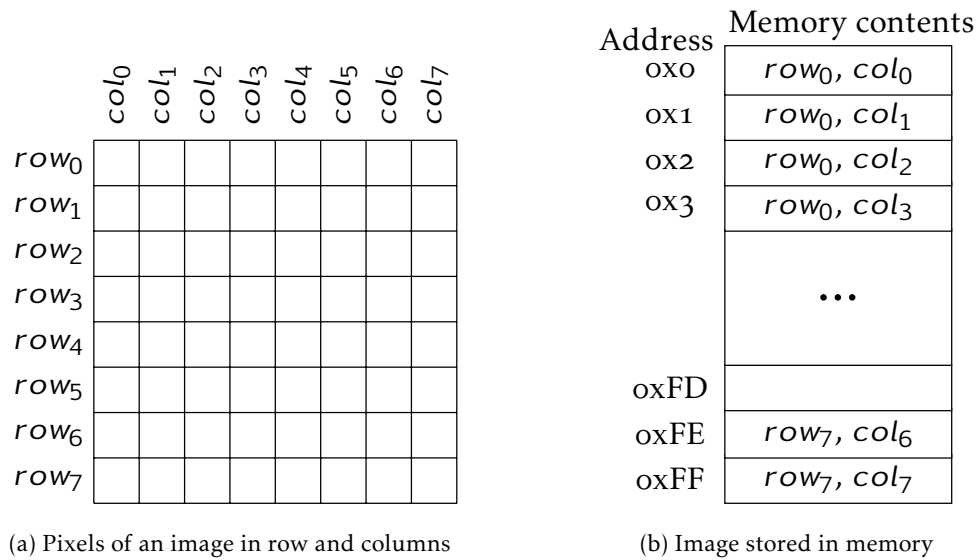
Figure 3.6 – Image stored in memory

as row 0, row 1 and so on, where each row will be composed of column 0, column 1 etc. as shown in Figure 3.6. Suppose we had an image of 8 rows and 8 columns stored contiguously in the memory, and an algorithm divides it into four blocks equally each of four rows and four columns. Suppose we want to transfer only one block (lets say top-left), we need data from row 0 to row 3 and column 0 to column 3. The hardware transferring this block must transfer first for row 0, column 0 to column 3 and skip column 4 to column 7. Then it must continue for row 1, row 2 and row 3 similarly. This skipping of data at regular offsets is called strided DMA and is shown in Figure 3.7(b) (assuming every gray part consists of four pixels in a row).

A DMA controller transfers contiguous as well as non-contiguous (strided) data blocks between the local store and main memory, as shown in Figure 3.7. In the case of a contiguous (or non-strided) transfer the data blocks residing in a contiguous address space are transferred from source to destination. In the case of strided DMA shown in Figure 3.7(b), the data can be located in uniformly spaced non-contiguous locations. Typically, the DMA controller is responsible to manage the stride by splitting the strided data into multiple requests which are contiguous in nature. The distance between two non-contiguous locations is referred to as stride as shown in Figure 3.7(b). A point to note is that in general the stride can be supported either at source or destination of the DMA transfer or both, depending on the hardware architecture. For the strided DMA, the DMA controller has some extra overhead, in order to appropriately decode command and issue a DMA transfer incrementally. Thus the transfer delay will certainly depend not only on the amount of data being transferred but also on the number of non-contiguous blocks being transferred.

3.6.2 *DMA list*

The strided DMA command is supported by Cell architecture in a more generic way using *DMA list*. DMA list contains upto 2048 elements where each element determines the size and remote memory address. The local store address is not specified in the list and passed explicitly in setup. A DMA list transfer can move data between a contiguous area in an local store and possibly a non-contiguous area in the remote memory. Cell architecture has a limitation that the SPU issuing the strided DMA does not support stride in its local store either for fetching or

(a) Non-Strided DMA   (b) Strided DMA with stride at source
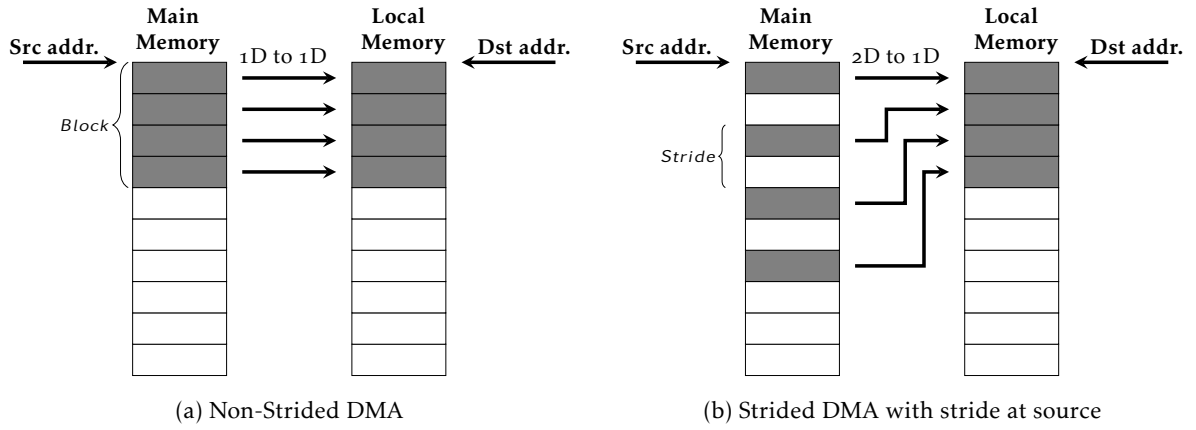
Figure 3.7 – DMA transfers

putting data into a remote memory. Thus stride is available only for a remote address. DMA list is placed in the local store of the SPE and is passed to MFC for processing. MFC fetches the list element by element and transfers the data. The local address is incremented by size in the element for every transfer. The DMA transfer is performed until the entire list is processed. The software is responsible for allocating the DMA list and initializing its parameters. The waiting on the list transfer can be done using a single routine call. More details about DMA lists are available in [62, 102]. DMA transfer is provided with an identifier which can be later used to poll for its completion.

3.7 MODELING DMA CONTROLLER

The DMA transfer time for a contiguous block of size s, denoted by $T(s)$, consists of two parts, a fixed initialization cost I and data transfer delay proportional to the data to be transferred. This delay is given by $g \cdot s$, where g refers to transfer delay in time units per byte, s refers amount of data transferred in bytes. The DMA communication delay is summarized in Equation 3.1.

$$T(s) = I + g \cdot s \tag{3.1}$$

It is important to note that during the time I both the compute core and the DMA channel are busy for setting up the transfer, whereas during the remaining time $G = g \cdot s$ only the DMA channel is still busy, pushing the data into the NoC, while the core can proceed to other tasks in parallel. At any time later, the processor may decide to wait for the completion of the transfer. DMA transfer completion status can be polled from any core inside the cluster. This operation is blocking until time $T(s)$ has elapsed plus it may take some additional timing delay χ to return the control back to the thread waiting for the transfer completion

In the case of strided transfer, the data transferred consists of multiple contiguous blocks. Let the transfer consists of s_1 blocks each of size s_2 bytes. The DMA set up call incurs a fixed initialization cost for the transfer given by I_0. The hardware transfers s_1 blocks successively. After transmission of each s_2 block, the source/destination address is incremented with an amount equal to the stride of the transfer. This incurs a fixed cost which is proportional to number of s_1 blocks and given by $I_1 \cdot s_1$. The time required for entire data to be sent through the network is proportional to the cost per byte, and is given by $g \cdot (s_1 \cdot s_2)$. The DMA transfer delay for strided DMA is shown in Equation 3.2.

$$T(s_1, s_2) = l_0 + l_1 \cdot s_1 + g \cdot (s_1 \cdot s_2) \tag{3.2}$$

Obviously from this formula, it is always costlier to perform the strided DMA rather than a contiguous DMA for the same amount of data. But it is always more beneficial to perform a strided DMA than to perform s_1 DMA transfers of contiguous blocks. The observations above can be summarized by the following inequality:

$$T(1, s_1 \cdot s_2) \ \leq \ T(s_1, s_2) \ \leq \ s_1 \times T(s_2)$$

3.8 PLATFORM MODEL

We consider the hardware platform as an interconnection of different clusters. Each cluster consists of some number of processors. The processors within the cluster are connected to a shared memory and communicate with each other without any extra overhead or cost. Clusters can have caches in them, in order to provide a faster access to frequently accessed data. However, to avoid modeling complex and difficult to predict cache functionality, we do not model execution time variations due to cache misses but assume average-case execution times instead. The communication between the clusters is facilitated by a mechanism called DMA (Direct Memory Access). Further we assume a fixed routing between global interconnect terminals with distances known statically.

Given the notions above, we can formally define the hardware architecture model as :

DEFINITION 6 (Architecture Model) – *An architecture model $A = (X, \phi, M, D, l, g, \chi)$ is a tuple, X is a set of clusters, $\phi \subseteq X \times X$ is a set of communication paths between pairs of clusters, whereas $\phi_{x,x'} \in \mathbb{N}_0$ gives the distance between any two clusters $(x, x') \in X \times X$. M and D are the number of cores and DMA channels per cluster respectively. $l \in \mathbb{R}_{\geq 0}$ is the DMA initialization time and $g \in \mathbb{R}_0^+$ is the cost per byte for a transfer. $\chi \in \mathbb{R}_{\geq 0}$ refers to the constant time required to check if the DMA transfer is complete.*

Table 3.1 provides us the summary of parameters that has been used in this work for the respective architectures.

Parameter	Symbol	Unit	IBM Cell	Tilera TILE64	Kalray MPPA
Number Of Clusters	$|X|$	Number	8	1	16
Proc. Per cluster	M	Number	1	64	16
On-chip memory per cluster		Bytes	256 kB	5.5 MB	2 MB
DMA Per Cluster	D	Number	16	0	8
DMA Init. Time	l	Cycles	400	0	6000
DMA Cost per byte	g	Cycles per byte	0.22	0	0.2818
DMA Completion time	χ	Cycles per transfer	0	0	100

Table 3.1 – Summary of hardware platform parameters

3.9 CONCLUSION

In this chapter we discussed multi-core and many-core processor architectures encountered in our work and how they can be modeled. We abstract from minute details of the hardware platforms and while retaining the important parameters which are useful for mapping and scheduling. The architecture model presented in this chapter, has a meaningful application to the architectures that we used during this work. In the next chapter we discuss formalisms

that ensure application and architecture models can be used to map and schedule application efficiently on such complex platforms.

Satisfiability solvers and multi-criteria Optimization

This chapter introduces the notions of multi-criteria optimization problem and SMT solver which we use to solve multi-criteria scheduling problems.

Given a many-core hardware platform and a parallel application, to find an optimal mapping and scheduling solution is a hard combinatorial problem. Scheduling tasks on a processor is a well-known research topic, and there are various theories and techniques developed to solve this problem. Scheduling can be viewed as a constrained optimization problem. The constraints are typically precedences (coming from the precedences in the task graph) and resource constraints (e.g. no two tasks can use the same processor simultaneously). Other constraints can concern memory usage, schedule latency, the number of processors used etc. In fact, any performance measure of the system can sometimes be viewed as a constraint and sometimes as an optimization objective. In our work, we use SMT (satisfiability modulo theory) solvers, which collect techniques to solve the problem presented in the form of linear constraints.

Solution to the problem described above with multiple costs is not unique, rather it is a set of incomparable points called *Pareto* points. Such problems are called *multi-criteria optimization* problems. In these problems, the costs generally conflict with each other, such that reducing one cost may require increase in another one and vice-versa.

In this chapter we present some basic facts on SMT solvers which we apply to optimize mapping and scheduling for many-core processors. We explain briefly the theory behind them. Then we introduce multi-criteria optimization, for which we would like to apply the SMT solvers. Multi-criteria problems need specialized algorithms in order to track their solutions. We conclude the chapter by discussing such algorithms.

4.1 SATISFIABILITY SOLVERS

The scheduling problem can be approached with different forms of expressing the problem and corresponding solution methods. A technique called Linear programming [111] is applicable when the problem is expressed in terms of conjunction of linear inequalities, and the set of feasible solutions is a convex polyhedron. Applicable for unbounded resources, this theory can produce quick solutions. However, scheduling under resource constraints is a *non-convex*

problem. For example the mutual exclusion constraint, when a pair of tasks share a processor so that they should not overlap in time, can be expressed only as a disjunction of inequalities. For example, for a pair of tasks *A* and *B* allocated to the same processor, task *A* can start either after task *B* has finished or vice-versa. Due to such constraint, the solution space of the problem consists of a number of disjoint convex sets. A branch of linear programming called *Mixed Integer Linear Programming* (MILP), also known as disjunctive linear programming, is used for scheduling problems in operations research area [8]. However the MILP-based methods can solve scheduling problems typically upto a few tens of tasks [33, 99]. Thus, the applicability of disjunctive linear programming is severely limited.

However there are techniques which we believe to be better scalable specialized in solving Boolean combinations of constraints, popularly known as constraint logic programming (CLP) [60], which explore the search space for a solution and employ techniques like constraint propagation to prune the search space. A CLP problem is composed of atoms. An *atom* is a linear inequality of a Boolean variable. A *literal* is either an atom or its negation. The constraint logic program contains different clauses (or constraints) which are disjunctions of literals that impose rules on the values that can be assigned to the variables. The solver performs search over the domains of the variables in order to find an assignment which satisfies all the constraints. For example if X is defined as a variable in the constraint logic program, $X \neq false$ will be a constraint, due to which X will be always assigned with a true value. The solvers which solve problems expressed in a Boolean CNF (conjunctive normal form) form are called *satisfiability* (SAT) solvers. In the course of its development, the SAT theory was generalized to non-Boolean variables arranged in predicates "theory constraints as more complex atoms". For example $x - y > c$, a linear inequality, can be expressed in these solvers. Such kind of solvers are called *satisfiability modulo theories* (SMT) solvers [11]. These solvers can take advantage of existing SAT solver search engine either by converting a SMT problem to a Boolean SAT problem (*eager approach*) or by using theories which can perform constant propagation and conflict analysis in order to solve the problem (*lazy approach*).

An algorithm called DPLL [34], which involves backtracking-based search procedures for deciding the satisfiability, enhanced the performance of these solvers. Further improvements were made to the solvers, like for example CHAFF SAT solver [86] incorporated carefully engineered data structures and algorithms to accelerate the search. GRASP (Generic seaRch Algorithm for the Satisfiability Problem) [84] is another example of a novel algorithm which unified several search-space pruning methods and improved back-tracking techniques. Modern SMT solvers such as Yices [37], Z3 [87] and many more employ such efficient techniques and are widely used in the verification community.

The SAT solver problem context typically consists of Boolean variables, their negations and disjunctions. For SMT, in addition to Boolean variables, there are other variable types such as integer, real, arrays, interpreted functions etc. The solver checks whether there is an assignment of values to the variables which satisfies all the mentioned constraints. If solver is unable to find such an assignment (due to conflicting clauses), the problem is said to be *unsatisfiable*. SAT or SMT solvers use backtracking algorithm which incrementally builds the solution to the problem. In case if an assignment of a value to a variable creates a conflict (unsatisfiable solution), this assignment can be retracted and checking continues with other assignments. This procedure is carried out recursively. DPLL algorithm provides rules for backtracking. We explain them briefly for satisfiability solvers.

- **Unit Propagation** is done when a clause is a unit clause containing only one literal. In such cases, the variables are assigned values to make the problem satisfiable, and the opposite value for these variables is not evaluated.
- **Clause Elimination** is done when a variable is assigned such a value, that clauses containing this variable become true. These clauses do not constrain the search and can

be safely eliminated.

Advanced backtracking techniques have enhanced the performance of SMT solvers so that they can solve large problems using multiple theories. When the problem is presented in form of variables and clauses to the solver, it uses different theories depending on the nature of the problem and produces a solution. Within a limited time budget, the SAT solver can return an answer as **sat** (satisfiable), **unsat** (unsatisfiable) or **timeout** (unable to decide). If the answer returned is **sat**, then the solver builds up a model which describes the values which are assigned to the variables. If the answer is **unsat**, it implies that the problem contains conflicting clauses and cannot be satisfied. In case the result is **timeout**, it means that with the time budget the solver could not decide upon its satisfiability if the problem is **sat** or **unsat**. This suggest that we should either increase the time budget, or the problem is too difficult to solve.

4.2 AN EXAMPLE OF SMT CONSTRAINTS

We present an example of SMT constraints for scheduling a split-join graph shown in Figure 2.6 for $\alpha = 3$. The derived task graph, $T = (U, \mathcal{E}, \delta, \omega)$, will have tasks A_0, B_0, B_1, B_2 and C_0. Let $s(u)$ and $e(u)$ denote the start and end time of a task u, and $\mu(u)$ be the processor on which it will execute.

The constraints which express the task graph for scheduling to the solver are given as:

- **Non-negative start times**: indicating that there are no negative start times of the task.

$$\bigwedge_{u \in U} s(u) \geq 0$$

  Note that this kind of constraints are implicit as we will see later they follow directly from definition of variable domains, like $s(u) \in \mathbb{R}_{\geq 0}$ in this case. In sequel we do not explicitly specify such constraints.

- **End times**: The task always finishes when the time equal to its execution time has elapsed after the starting time.

$$\bigwedge_{u \in U} e(u) = s(u) + \delta(u)$$

- **Precedence relations**: No task can start before the finish of its predecessors.

$$\bigwedge_{(u,u') \in \mathcal{E}} e(u) \leq s(u')$$

- **Non-overlapped execution on a processor**: Tasks running on same processor should not overlap in time.

$$\bigwedge_{u \neq u' \in U} (\mu(u) = \mu(u')) \Rightarrow (e(u) \leq s(u') \vee (e(u') \leq s(u))$$

- **Processor cost**: The tasks should use only M processors or less.

$$\bigwedge_{u \in U} \mu(u) \geq 0 \wedge \mu(u) < M$$

Thus the scheduling of task graph can be encoded as an SMT problem with linear arithmetic theory. A more detailed encoding of the problem is presented in Section 6.2.

4.2.1 *Non-retractable and retractable constraints*

A constraint solver can be used for optimization by adding to the constraint satisfaction problem a condition of form $x \leq C$ that can be used for cost optimization, where x is a tight upper bound on the cost and C is a constant. In order to find an optimal cost, the solver
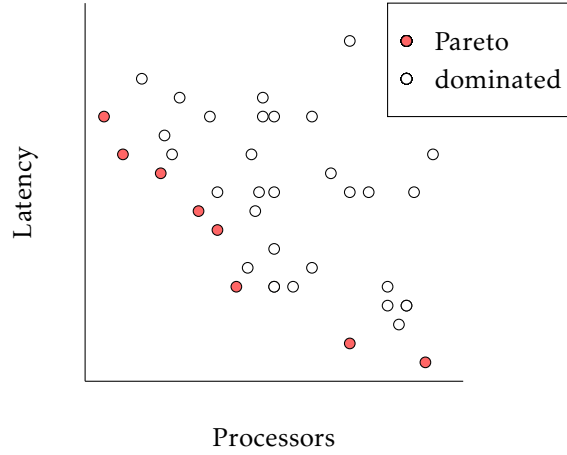
Figure 4.1 – Pareto points

must be queried for different values of the cost to detect feasible (satisfiable) and unfeasible (unsatisfiable) cost points. In such an exploration, the set of constraints remains the same, except for the value of constant C. To recall, the problem context of an SMT solver consists of variables and constraints defined on these variables. We take advantage of two specific types of constraint typically supported by the SMT solver tools: *non-retractable* constraints are the one which cannot be removed from the problem context, and *retractable* constraints which can be removed from the problem context in the order reverse from their addition order, which leads to incremental update of the problem definition without redefining the problem from scratch. This can save some solver computation time across multiple queries. In the above example, we can put all the constraints as non-retractable and the processor cost as retractable. For every query, the retractable constraint must be updated with a new value of constant M.

As explained above, if a problem has only one cost, the optimal cost can be searched using successive queries with different values of the cost (binary search) using a retractable constraint. However when the problem admits multiple costs we need a more elaborate exploration of the cost space, because in such problems there is typically no unique single solution, but rather a set of incomparable solutions (points in the cost space). Such a problem is called *multi-criteria problem*, which is discussed further in detail.

4.3 MULTI-CRITERIA PROBLEM

In optimization problems, a search is performed to reduce the cost to a feasible minimum value satisfying the constraints. For example, in binary search, a solution with higher cost is discarded, when a solution with lower cost is found, narrowing down the space of explored costs. With respect to a given point, a solution point in the cost space with higher cost is called *dominated point* and while the point with a lower cost is called *dominating point*. If we consider a problem with multiple costs, then the dominated point should have a distinct cost that is no better than reference point in all dimensions. Formally, dominated points can be defined as follows.

DEFINITION 7 (Dominated Points) – *Any two points c and c' belonging to same cost space, point c dominates c', if for every dimension $i \in [1..d]$, $c_i \leq c_i'$ and $c \neq c'$.*

If we eliminate dominated by other points from set of solutions for such multi-dimensional problem, the resulting set consists of *alternative* solutions which represent the best trade-offs
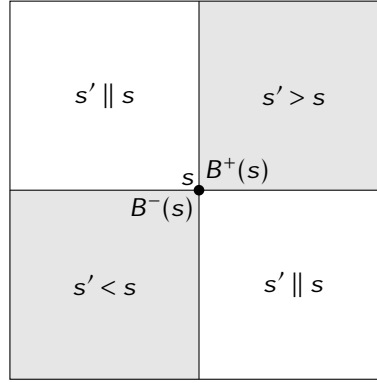
Figure 4.2 – Forward and backward cones

between different costs. Such set of solutions or points are called *Pareto* points. Figure 4.1 shows such Pareto points (marked in red) which dominate the other points (marked in white). The set of Pareto points are said to be ones, which are non-dominated by any other point.

A typical example in scheduling for such multi-criteria problem will be optimizing the latency cost of a schedule and the number of processors used in it which is proportional to amount of static power consumption. Given that there is enough parallelism available in the application, if we increase number of processors, the application can run concurrently, which will decrease the latency of execution. Similarly if we decrease number of processors, then a larger portion of application will execute sequentially thus increasing the latency of execution. It is desirable to reduce both latency and number of processors used. However, if we try to reduce one cost the other cost increases and vice-versa. Such problems are called *multi-criteria optimization problems* [38], and are being studied since a long time [38, 41, 44]. Figure 4.1 shows a sample exploration of design space for two costs - *number of processors* used and *latency* of the schedule (we ignored the communication cost).

Some of these multi-criteria problems exhibit a monotonic behavior. For example, in processors and latency trade-off, if a point $C_1 = (2, 5)$ is feasible, then the point $C_2 = (3, 9)$ will also be feasible. This can be thought as if a schedule satisfies cost of 2 processors and latency cost of 5 units, then another point with cost of 3 processors and loose latency of 9 units will also be feasible, owing to the fact that some processors and/or some time intervals might remain unused. Similarly, if the point $C_2 = (3, 9)$ is infeasible, then point $C_1 = (2, 5)$ will be infeasible because if problem cannot be scheduled with 3 processors, it cannot be satisfied with 2 processors and a tighter latency.

For such monotonic problems, we can assume that all points above a feasible cost are feasible and those below an infeasible cost are infeasible. More formally, as shown in Figure 4.2, if point s is a **sat** point, then the forward cone represented by $B^+(s)$ contains only feasible points. Similarly if point s is **unsat** point, then the backward cone represented by $B^-(s)$ contains only infeasible points. The remaining $s' \parallel s$ are incomparable points with respect to s. Also if s is a **sat**, then the algorithm must explore the $B^-(s)$ and $s' \parallel s$ for Pareto solutions. If it explores the forward cone $B^+(s)$, then it is waste of effort, since the solver will always return **sat** or **timeout**. Similarly for **unsat** point, the algorithm must explore $B^+(s)$ and $s' \parallel s$.

4.4 COST-SPACE EXPLORATION

The optimal solution for a multi-criteria optimization problem is the set of Pareto points. To find or approximate this set, we need to explore the cost space. Typically the cost space
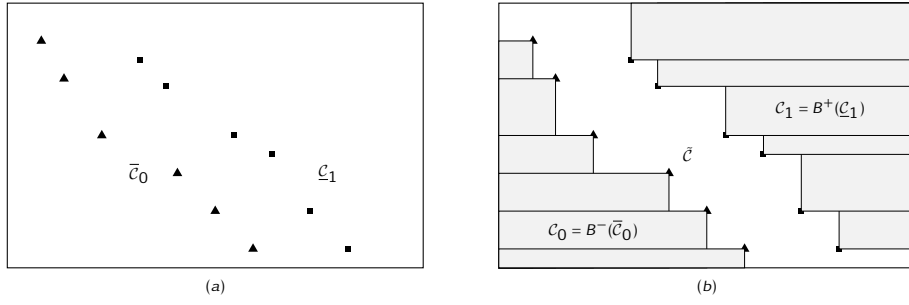
Figure 4.3 – (a) Sets $\mathcal{C}_0$ (**unsat**) and $\mathcal{C}_1$ (**sat**) represented by their extremal points $\overline{\mathcal{C}}_0$ and $\underline{\mathcal{C}}_1$; (b) The state of our knowledge at this point as captured by $\mathcal{C}_0$ (infeasible costs) $\mathcal{C}_1$ (feasible costs) and $\tilde{\mathcal{C}}$ (unknown). The actual Pareto front is contained in the closure of $\tilde{\mathcal{C}}$.

is huge and intelligent algorithms are needed to obtain quickly a good approximation of the Pareto front. There are various approaches and algorithms for this exploration. One approach is to consider the cost of a problem as a weighted sum of multiple costs and solving it for one-dimension using binary search and repeating the process with different weight vectors. Certain other popular approaches depend on heuristic search techniques to find solutions. Examples of such approaches are evolutionary / genetic algorithms [35, 73]. In [80], a distance based algorithm was proposed which approximates the Pareto front by aiming at reducing the distance between the **sat** and **unsat** points. It involves a complex algorithm to explore the cost space which depends on the current state of the exploration and can be viewed as a multi-dimensional form of binary search. Instead we use a simple grid-based exploration algorithm which relies on successive refinement of cost space in the grid. First we briefly describe below the problem formally and then we discuss the grid based algorithm.

Let $Q(c)$ be a shorthand for the satisfiability query $\exists \mu \exists s$ s.t. $\varphi(\mu, s, c)$, which asks whether there is a feasible deployment (mapping μ, schedule s), whose cost vector is less than or equal to c. If the solver answers affirmatively with some cost c we have a solution and may also conclude any cost in *forward cone* of c that $B^+(c) = \{c' \mid c' \geq c\}$ is feasible, which follows directly from the cost constraints. If the answer is negative we can conclude that any cost in the *backward cone* $B^-(c) = \{c' \mid c' \leq c\}$ is infeasible. After submitting a number of queries with different values of c we face a situation illustrated in Figure 4.3. The sets $\overline{\mathcal{C}}_0$ and $\underline{\mathcal{C}}_1$ are, respectively, the maximal costs known to be infeasible (**unsat**) and minimal feasible costs found (**sat**). Sets $\mathcal{C}_0$ and $\mathcal{C}_1$ are defined as the sets of all points known to be **unsat** and **sat**, they are equal to the union of forward/backward cones of the extremal points. The feasibility of costs which are outside $\mathcal{C}_0 \cup \mathcal{C}_1$ is unknown. The set $\underline{\mathcal{C}}_1$ constitutes an approximation of the Pareto front and its quality, defined as a kind of Hausdorff distance to the actual front, is bounded by its distance to the boundary of the backward cone of $\overline{\mathcal{C}}_0$.

4.5 DISTANCE BASED EXPLORATION

The algorithm presented in [80] calculates the Hausdorff distance between the **sat** and **unsat** sets $\overline{\mathcal{C}}_0$ and $\underline{\mathcal{C}}_1$, and tries to reduce it by successively querying the solver for different costs. The exploration algorithm maintains a list of currently explored points and updates the distance between the two sets with newly added points. The distance is then reduced by repeatedly querying at mid-way between the two points which give a maximum distance. The query returning **sat** are added to $\overline{\mathcal{C}}_0$, while those returning **unsat** are added to $\underline{\mathcal{C}}_1$. Every query that is asked to the SMT solver has a time limit (query time out). Since there will possibly be
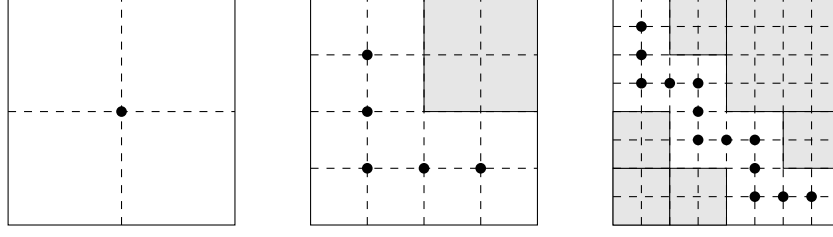
Figure 4.4 – Exploring the cost space via grid refinement. The dark points indicate the new candidates for exploration after each refinement.

a large number of Pareto points, to restrict the runtime of the algorithm a global time out is applied, after which the algorithm stops at the current state. For more details, the reader is referred to [80].

4.6 GRID BASED EXPLORATION

Instead of the distance-based algorithm in this thesis, we use here a simpler exploration algorithm based on grid refinement. In the multi-dimensional cost space, a grid is placed initially at each dimension in midway(see Figure 4.4). We perform the queries on the intersecting points of the grid. At every stage of the algorithm we refine the grid defined on the cost space, making it denser and ask $Q(c)$-queries with c ranging over the newly-added grid points which are outside $\mathcal{C}_0 \cup \mathcal{C}_1$. Note that not all these new points will necessarily be queried because each query increases the size of $\mathcal{C}_0 \cup \mathcal{C}_1$ so as to include some of these points. The description so far was based on the assumption that all queries give a definite answer (**sat** or **unsat**). However it is known [79] that as c gets closer to the boundary between **sat** and **unsat**, the computation time may grow prohibitively and the solver can get stuck. To tackle this problem we bound the time budget per query and when this bound is reached we mark the given point as **timeout** Choosing the appropriate value for the time budget is a matter of trial and error.

We use a parameter ϵ, which determines the current relative granularity of the Pareto exploration in all dimensions. In every dimension we perform linear search. In particular, in dimension p we make $1/\epsilon$ steps of equal size from $C_p^{\min}$ to $C_p^{\max}$, where $C_p^{\min}$ and $C_p^{\max}$ are the lower and upper bound respectively for exploration in dimension p.

The top level procedure of our algorithm is illustrated in Algorithm 1. The whole exploration has a global timeout, *globalTimeOut*. To accumulate the current knowledge on the cost space, the algorithm maintains two set variables, $\mathcal{C}_0$ and $\mathcal{C}_1$ (in the actual implementation we just keep track of the extremal points $\overline{\mathcal{C}}_0$ and $\underline{\mathcal{C}}_1$ along with the timeout points $\widetilde{\mathcal{C}}_t$). The initial exploration granularity is set to $\epsilon = 1/2$, and we recursively reduce it by a factor $1/2$ after the entire exploration at the given step is finished. Thus with more iterations, the algorithm explores the cost space with finer granularity.

Each iteration executes a nested loop, one level of nesting per dimension. Procedure *explore*, shown in Algorithm 2, executes one level of loop nesting and calls itself recursively to explore in the next dimension. This procedure lets the current dimension of vector c assume all linear search values, whereas the other lower dimensions remain constant, initially set to $C^{\max}$. The lower dimensions assume their currently explored value, whereas the current dimension p is explored between C_p^{min} and C_p^{max} bounds.

There can be three possible results for a SMT query: **sat**, **unsat**, and **timeout**, the latter occurring when the satisfiability conclusion was not reached within the timeout, and in this case marked as timeout. If, on the contrary, satisfiability has been computed, then one of the

---

Algorithm 1 Pareto Exploration Algorithm

$\mathcal{C}_0 := \emptyset$ ▷ unsat points
$\mathcal{C}_1 := \{C^{\max}\}$ ▷ sat points
$\widetilde{\mathcal{C}_t} = \emptyset$ ▷ timeout points
$\epsilon := 1/2$
while $algoRunTime() \leq globalTimeOut$ **do**
    $p := firstDimension()$
    $c := C^{\max}$
    $explore(\epsilon, p, c, \mathcal{C}_0, \mathcal{C}_1)$
    $\epsilon := \epsilon \, / \, 2$
end while
$\underline{\mathcal{C}_1} := minimalPoints(\mathcal{C}_1)$
return $\underline{\mathcal{C}_1}$

Algorithm 2 *explore*(**step** ϵ, **dimension** p, **cost** c, **cost set** $\mathcal{C}_0, \mathcal{C}_1$)

▷ Explore cost dimension p and higher
for $\delta := 0$ **to** 1 **step** ϵ **do**
    $c_p := round(\, (1 - \delta) \cdot C_p^{\min} + \delta \cdot C_p^{\max} \,)$
    if $c \notin \mathcal{C}_0 \wedge c \notin \mathcal{C}_1 \wedge c \notin \widetilde{\mathcal{C}_t}$ **then**
        $result := satQuery(Q(c), satTimeout)$
        if $result = $ **sat then**
            $\mathcal{C}_1 := \mathcal{C}_1 \cup B^+(c)$
        else if $result = $ **unsat then**
            $\mathcal{C}_0 := \mathcal{C}_0 \cup B^-(c)$
        else
            $\widetilde{\mathcal{C}_t} := \widetilde{\mathcal{C}_t} \cup c$
        end if
    end if
    if $c \in \mathcal{C}_1 \wedge \neg lastDimension(p)$ **then**
        $explore(\epsilon, nextDimension(p), c, \mathcal{C}_0, \mathcal{C}_1)$
    end if
end for

---

two sets are extended by point c as a minimal/maximal point. For a solver query, we impose a per query timeout *satTimeout*, which is much smaller than *globalTimeOut* and determined by manual calibration of this setup.

Note that procedure *explore* calls the solver directly only until the first **sat** answer or a $\mathcal{C}_1$ point has been reached. After this, all the remaining values to be explored for c_p automatically fall into the 'known **sat**' area, $\mathcal{C}_1$ and thus the solver is called only for the deeper recursion levels, with lower cost values in the higher dimensions. In order to speed up the algorithm, instead of linear search, a binary search method is used in the actual implementation.

4.7 CONCLUSIONS

We briefly discussed the SMT solvers and the theory behind them. We also discussed the multi-criteria optimization problems. These problems have multiple conflicting costs to be optimized, and the solution to such problems is not unique, but rather consists of multiple incomparable solutions also called *Pareto* solutions. The exploration of these problems in

their respective cost-space require specialized algorithms to track the Pareto solutions. We presented the grid-based exploration algorithm which is used in our work to explore the multi-dimensional cost space.

In further chapters we talk about how to encode the deployment problem as SMT constraints and efficiently find solutions using such exploration algorithms. We also discuss how the solutions discovered by the SMT solver can be deployed on a multi-core system.

Deployment and Evaluation Methodology

This chapter describes our framework to explore, deploy and evaluate the solutions on a multi-core platform.

The SMT solver is responsible to analyze and produce solutions to the scheduling problem, according to different costs and platform constraints that are specified in the optimization problem. In order to produce these solutions the solver needs inputs in the form of constraints which represent the structure of application including timing information of tasks, token sizes, channel parallelization factors etc. The problem expressed in the form of constraints is then explored by the cost-space exploration algorithm which is described in the previous chapter. The result is in the form of Pareto solutions, which expresses the configuration of the hardware platform on which the application must execute to produce expected results.

Given such a situation, we need a software infrastructure which can perform all the mentioned tasks. We implement a software tool in Java which is used to convert the split-join graph information to constraints which can be presented to the solver. We call this tool *StreamExplorer*. StreamExplorer also incorporates the cost-space exploration algorithm described previously. Further on the hardware platforms we implement a run-time system which deploys the solution produced by the solver. The run-time software is equipped with profiler functionality which can provide accurate timing information of the application to StreamExplorer.

In this chapter we briefly describe StreamExplorer and the run-time system. The run-time system consists of different sub-systems which initialize the application as per the specified configuration, perform different measurements and report statistical data back to StreamExplorer. The first task of the run-time is to profile the actor execution times. StreamExplorer feeds the measured times to the split-join graph. It then performs cost-space exploration to find multiple solutions to the scheduling problem. The approximate Pareto solutions are then verified on the hardware platform using run-time system. We describe the infrastructure of StreamExplorer and run-time system in this chapter.
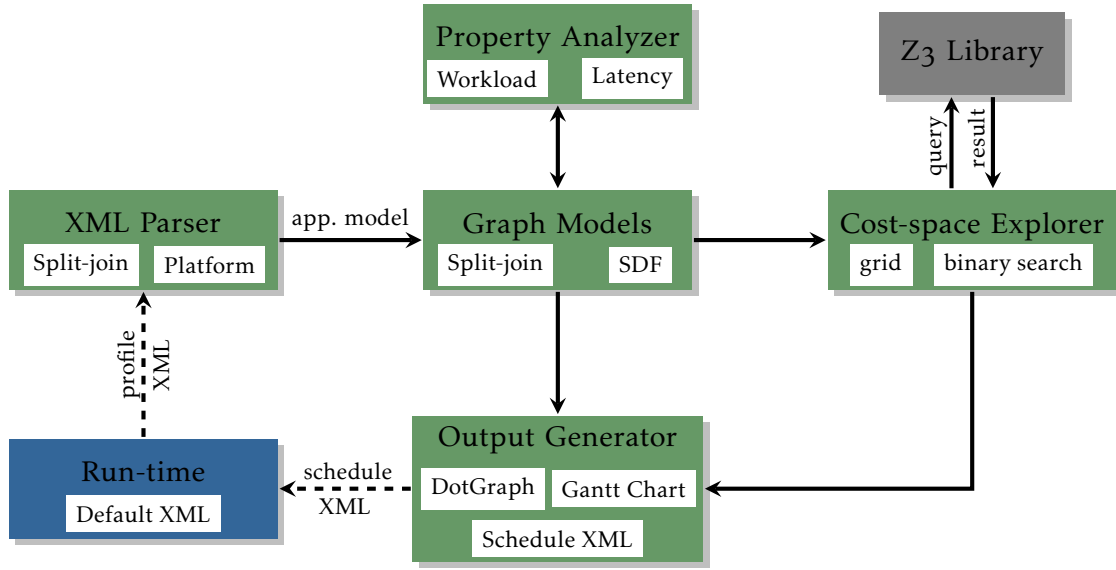
Figure 5.1 – Structure of StreamExplorer

5.1 THE TOOL

StreamExplorer is a collection of different algorithms which operate on split-join graphs. The brief structure of the tool is shown in Figure 5.1. It consists of following parts:

- **Split-Join Graph Core**: This part forms the core of the StreamExplorer, which contains the code to represent different components of the split-join graph model, like actors, channels, ports etc. It defines a graph object which is a collection of all these elements.
- **Property Analyzer**: This part performs different analyses on the split-join graph. For example it performs deadlock analysis and checks for the consistency of the graph. In addition it can provide different properties like total execution time in the graph, longest path etc. It also provides support functions to determine the connected actors / tasks, get split join factors etc.
- **Input Parsers**: The parsers form a vital part of the StreamExplorer which parses the input XML file, containing the information about the split-join graph model (rates, number of actors, interconnections etc.). It also contains a parser to parse the specification of hardware platform defined in Chapter 3.
- **Solver Interface**: This part of the StreamExplorer performs the conversion of the split-join graph and platform model to constraints which are presented to the SMT solver. We have integrated Z3 SMT solver tightly with the StreamExplorer. The StreamExplorer creates variables and constraints for non-pipelined scheduling, from the split-join graph model and platform parameters.
- **Cost-space Explorer**: This part of the StreamExplorer consists of the cost-space exploration algorithms (grid-based, binary search etc.). The algorithms in this part are generic and can be easily configured for different exploration like latency vs buffer size, latency vs processors etc.
- **Output Generators**: The output generators produce the results of different analyses and optimizations performed. The most commonly used for analysis of inputs and results, are the dot graph generation for graphical representation of split-join graphs, Gantt chart generation for a schedule, schedule XML generation for run-time system.

StreamExplorer consists of total 25K+ lines of code written in Java and 15K+ lines of code written in C++ for runtime system. It is interfaced with the SMT solver using Z3 library. Z3 solver in the tool, can be easily replaced by another solver by performing minimal changes. The link between the StreamExplorer and the run-time system is provided with bash scripts which automates the process.

5.2 PROFILING THE APPLICATION

The profiling of the application is done mainly in order to measure the execution time of different actors. The execution times of actors are used for scheduling in the solver. We use the XML format similar to that in SDF3 tool [118] to present the split-join application graphs in the form of an equivalent SDF graph. The profiling of application is done on the platform, which uses the run-time system for measurement and produces a XML file which contains the timing information of the application. In order to run the profiled application, the run-time system must be initialized (for data-structure, FIFOs etc.) and hence must be provided with a schedule XML. Thus it is a cyclic requirement, where for initial profiling we need a schedule XML, but also an XML is generated by the run-time based on the profiling. It is possible to create a front-end parser, which can detect the parameters of split-join application graph (for example MpOpt [43]) automatically. However it is an extra effort and out of scope of this thesis. In order to satisfy this cyclic dependency, we provide a default XML configuration file, which is able to configure the split-join application graph to execute on a single processor with an arbitrary valid sequential schedule and the required buffer size for that schedule.

This default configuration is used to profile the application graph in the run-time. In its sequential schedule neither shared memory contention on the bus nor network contentions are present. Thus we measure the timing which is unaffected by these two factors. In profiling, the application is executed for 100 graph iterations and a statistical data is produced on the execution times. It contains minimum, maximum, average values for every actor present in the application. Thus the profiling reports worst case and average case value of the actors and either of them can be chosen for the optimization purpose.

5.3 RUN-TIME ENVIRONMENT

The run-time system is a piece of software written in C++, which requires an input schedule XML file that describes the split-join application graph and is linked with the functional code of the actors. It is used to deploy the scheduling solutions on the platform and consists of three parts:

- Initialization : allocate data structures for execution.
- Execution : execute different schedules on the processors.
- Release : collect results and deallocate data structures.

5.3.1 *Initialization of the application*

The first phase performs the initialization of the application graph on the platform. It first reads the buffer size of the all the channels allocated in the schedule, and it allocates the memory and initializes the data structures for them. Further it initializes every processor that is supposed to execute tasks and provides it with the static order of tasks according to the schedule. The information of the static schedule described in the schedule XML file is discarded, however the ordering between the actors is retained. The name of actors in the schedule XML file are resolved to the functions which execute as functional code for the actors. Every application defines a standard translation function which associates the name of the

actor to the function that can be executed on behalf of it. The run-time is thus able to associate a piece of code to every actor. When this resolution is finished, the run-time system then calls application specific initialization where the pre-execution functions are performed such as memory allocations specific to the application.

From programmers perspective, an actor is a function which will be called by the runtime to execute it on the platform. It will be provided with the data structures where it can access the tokens in input and output FIFO.

5.3.2 *Execution of the application*

After the initialization is finished on the hardware platform, the run-time enters *Execution* phase. In this phase, the run-time spawns a thread on every processor to which a schedule is allocated. The remaining processors remain idle. Every processor executes its respective static-order schedule in a self-timed way for a fixed number of iterations. In later subsection, we explain how due to our FIFO design, the precedence between the reader and writer are respected. We enforce a barrier synchronization between iterations, since we assume a non-pipelined execution. The run-time measures the time required for execution of the actor instance which is also used for debugging purpose. The latency is measured for every graph iteration on every processor. The maximum of the values measured on all the processors, gives the latency of the graph iteration.

5.3.3 *Release of the resources*

After the execution has finished, the run-time calls the application specific exit calls for application to release any allocated resources and transfer of the application output data (e.g. decoded JPEG image). Then it deallocates all the data structures and gathers the measurements that were done on the platform. Finally it performs statistical analysis on the measurements and generates the output XML. The output XML generated by the run-time system contains statistical information about actor execution as well as latency. The latter is compared with the model predicted latency.

5.3.4 *Hardware Specific Implementation*

Depending on the hardware platform, the run-time system is adapted accordingly. We discuss the hardware-specific implementation briefly.

5.3.4.1 Kalray MPPA-256

On the Kalray architecture, the host processor and the MPPA platform are connected with a PCI bridge. The run-time system implements a data transfer part, which allows communication of data over the PCI to the MPPA fabric. The data is first uploaded to the IO clusters and then from there it is distributed to the compute clusters. In the initialization phase this data consists of schedule information for the involved compute clusters. At the release phase, collected measurement and the application output is fetched from the compute clusters to the host via I/O clusters.

The barrier synchronization on this platform is performed in two phases. Inter-cluster synchronization is done using IO cluster which communicated with 0th processor of every compute cluster. After all the compute clusters are synchronized, the synchronization inside a cluster (intra-cluster synchronization) is performed using a pthread library barrier.

The I/O cluster reads the schedule XML file for the application and starts the compute clusters accordingly. It keeps the unallocated clusters inactive.

5.3.4.2   Tilera Tile Pro64

The Tilera processor architecture has support for hardware cache coherency. Thus the run-time execution system can execute as if it had just one cluster. It does not need any explicit data transfer mechanisms between processor cores, as it is taken care by coherency mechanism.

In order to execute plain application code without any interruption from the operating system, the Tilera software supports *bare-metal* environment. The processors configured this way execute only the application code in the pthread parallel programming library. We configure only one processor for execution of operating system and another one for handling I/O devices. The remaining 62 processors execute the application tasks.

The processor running the operating system starts a thread on every allocated processor. Each thread then executes the allocated schedule for a fixed number of iterations. We use the low-level hardware primitives provided by the Tilera software library which implements locks, barriers, atomic operations etc. These primitives significantly speed-up the software execution compared to the pthread primitives.

5.4   COMMUNICATION BUFFERS

In the split-join graph application model, the actors communicate via channels. The writer actor of the channel generates the tokens on the channel, while the reader actor of the channel removes them.

Following the model semantics, the split-join graph channels should use a bounded memory. To suit this requirements, the communication memory should be re-used efficiently. Moreover, to exploit the data parallelism of the split-join channels, it should be possible to execute different concurrent task instances of channel writer and reader on different cores. The requirements of the communication buffers can be listed as follows :

- Correct token delivery from the writer of the token to its reader.
- Reuse of the communication buffers (i.e. of the token space positions inside).
- Arbitration between writers and readers of the same token position.
- Support for concurrent instances of writer and reader that access the same reused token position.
- In the case of inter-cluster channel, the delivery of data and its status between writer and reader.

We created a software FIFO buffer implementation that satisfies these requirements. To support concurrent writers and readers we exploit the fact that in a split-join graph each writer/reader instance writes/reads statically known amounts of data at statically known offsets. Therefore, each token data is extended with status record that can be updated independently, indicating whether the given token is ready to be written or read. For the reuse of bounded token memory, we use standard circular buffer with a wrap-around mechanism.

It is very important to note the following. The term FIFO here indicates a certain ordering restriction between the concurrent instance of writer and reader that write and read to the same reused token position in the buffer. The task instances with a smaller task index should execute earlier than those with a larger index. This restriction on the schedule coincides with the notion of task symmetry introduced in Chapter 6, where we also prove that restricting the schedule like this does not lead to sub-optimality.

The *FIFO* buffer can be regarded as a collection of a memory allocated for a data structure that includes token data and token status. In the *FIFO* buffer implementation, the status record for every token consists of 2 parts. The first part is the *index* of the token, while the second part consists of a state of the token which can be *{'e'-empty, 'b'-busy, 'd'-contains data}*. Every writer task of the FIFO buffer can acquire token only when the token is empty, similarly, a reader task

(a) FIFO structure in memory, st_i is status   (b) FIFO token and status on executions
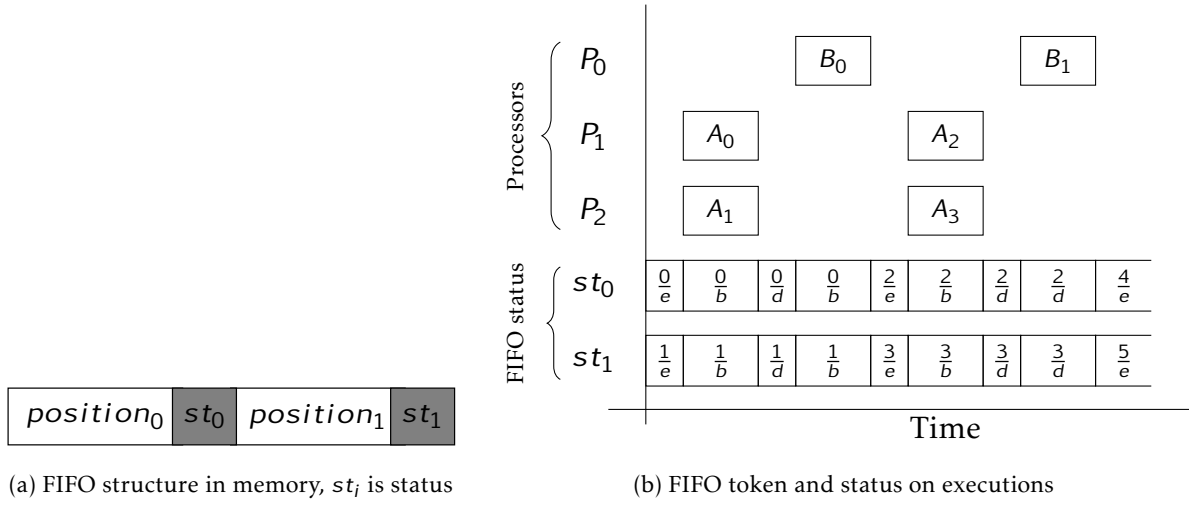
Figure 5.2 – FIFO token example for actors A and B with buffer size = 2

of the FIFO buffer can acquire token only when the token contains data. The reader and writer both, when acquiring a token change the state to *busy*. When writer task finishes, it updates the state part to from busy to *contains data*. Similarly when reader task finishes, it updates the state part from busy to *empty*. In addition to state, the index part identifies to which instance the token belongs. Before execution, the tokens are assigned index 0,1,... The instance 0 of the writer can access only the tokens in index range $[0, \alpha^\uparrow)$, instance 1 can access $[\alpha^\uparrow, 2 \cdot \alpha^\uparrow)$ and so on. In general any instance i of a writer can access tokens in range $[i \cdot \alpha^\uparrow, (i+1) \cdot \alpha^\uparrow)$. Similarly, any instance j of reader can access tokens having index $[j \cdot \alpha^\downarrow, (j+1) \cdot \alpha^\downarrow)$. In a bounded buffer with size b tokens, a token space at position k is first used for token k then reused for tokens $k+b$, $k+2b$ etc. Therefore, the value of index is monotonically increasing and the reader instance increases it by the size of the FIFO buffer upon its completion. The index value also implies the position in the FIFO buffer. The writer and reader of the FIFO buffer, both access statically known positions in the FIFO buffer, which can be easily calculated from their task index. Thus, a combination of index and state gives an accurate information about which task has a right over the token position in the FIFO buffer.

An important point to note is, the FIFO design does not make any assumption about execution time of its producers and consumers. The FIFO is robust against the variation in the execution time.

5.4.1 *FIFO buffer example*

Suppose in a split-join graph (A, B) is a channel with parameters $\alpha = 1/2$, $c(A) = 4$, $c(B) = 2$ $\alpha_{AB}^\uparrow = 1$, $\alpha_{AB}^\downarrow = 2$, and buffer size $b_{AB} = 2$. Figure 5.2(a) shows how the FIFO buffer is allocated in the local memory of the processor and Figure 5.2(b) shows an execution of this example. We use the notation $\frac{index}{state}$ for the status record. The execution happens as follows:

- Initially $position_0$ and $position_1$ are marked with status $\frac{0}{e}$ and $\frac{1}{e}$, indicating two empty tokens with index 0 and 1 respectively.
- writer instance A_0 acquires $token_0$ and marks it as busy. Similarly A_1 marks $token_1$ as busy.
- After A_0 finishes execution, $token_0$ contains data which changes its status to $\frac{0}{d}$ indicating it contains data. Similarly A_1 changes the status of $token_1$.

- reader instance B_0, who should read from both $token_0$ and $token_1$, waits till both of them contain data. At the beginning of its execution it marks both them as busy.
- After B_0 finishes execution and increments the index of $token_0$ and $token_1$ to 2 and 3 respectively. Also it marks them as empty.
- A_2 and A_3 waited for token status $\frac{2}{e}$ and $\frac{3}{e}$ respectively. Now they can continue execution and the rest follows similarly.

Table 5.1 shows the token status before and after execution of each instance. From this table, we can see that each actor instance awaits and modified token status records at pre-known locations with unique index and status. Thus we can conclude that irrespective if the reader and writer instances are executing in parallel, they will wait for each other to maintain the correct order of production and consumption of tokens. This logic guarantees that the tokens are delivered correctly.

Table 5.1 – FIFO buffer token status before and after execution of tasks

instance	$token_0$		$token_1$	
	before	after	before	after
A_0	$\frac{0}{e}$	$\frac{0}{d}$	-	-
A_1	-	-	$\frac{1}{e}$	$\frac{1}{d}$
A_2	$\frac{2}{e}$	$\frac{2}{d}$	-	-
A_3	-	-	$\frac{3}{e}$	$\frac{3}{d}$
B_0	$\frac{0}{d}$	$\frac{2}{e}$	$\frac{1}{d}$	$\frac{3}{e}$
B_1	$\frac{2}{d}$	$\frac{4}{e}$	$\frac{3}{d}$	$\frac{5}{e}$

FIFO buffer used by Erbium compiler [85] to map streaming applications on shared memory processors use similar notions.

5.4.2 *Inter-cluster FIFO buffer in Kalray*

In Kalray MPPA, we have a possibility of allocating communicating actors on different clusters if memory or processor resources of a single cluster are insufficient. In such a case, the FIFO buffer must be adapted to transport the tokens from one cluster to another. The run-time checks the cluster assignment of writer and reader to detect inter-cluster FIFO. Such FIFO buffer has the writer actor and reader actor allocated on different clusters. In the FIFO buffer implementation, the same amount of memory is allocated at the source and destination cluster. Also in addition to the token status at the writer side, it also contains remote token status which is updated by the reader side when consuming of the token is finished. At the end of its execution, the writer starts a DMA transfer from the source cluster to destination cluster. It can start writing only if the remote token status indicates a free token at the same position. Similarly, when the reader finishes execution, it starts a DMA transfer to update the token status at the positions that it has finished reading, back to the writer side. Such a communication is referred to as flow control. It allows the reader and writer of the FIFO buffer to be in sync with each other. More details on inter-cluster FIFO buffer will follow in Chapter 7.

5.5 CONCLUSIONS

In this chapter we described the tool StreamExplorer, that we have developed, which performs a task of converting a split-join application graph, represented in form of XML, to SMT constraints and provides a run-time environment to deploy the SMT solutions on the platform. We also describe how StreamExplorer performs profiling of the application code via run-time system. The profiling information is used by the StreamExplorer to annotate the split-join application with actor execution times to be used in SMT solver scheduling constraints. The problem is explored by the StreamExplorer to find solutions, which are are implemented and verified for accuracy by measurements in the run-time system. The run-time system produces statistics for every solution, which is used to evaluate the accuracy of solutions produced by the solver. Such infrastructure allows us to experiment with various split-join applications represented by their respective XML files in SDF3 format. In the next chapter we will discuss how the split-join graph can be represented using SMT solver constraints in order to solve the scheduling problem on a hardware platform, first restricting ourselves to one shared memory cluster to extend later to multiple clusters. We will also discuss the techniques enabling the SMT solver to accelerate the discovery of the solutions.

Scheduling in shared memory

This chapter deals with the scheduling problem in shared memory multi-processors using SMT solvers and a technique to eliminate the symmetry.

We encode the multi-core deployment for split-join graphs as a quantifier-free SMT problem, defined by a combination of linear constraints. The major computational obstacle is the intractability of the mapping and scheduling problems aggravated by the exponential blow-up while expanding the graph from symbolic (actors) to explicit (tasks) form (refer to Section 2.3). We tackle this problem by introducing "symmetry breaking" constraints among identical processors and identical tasks. For the latter we prove a theorem concerning the optimality of schedules where instances of the same actor are executed according to a fixed *lexicographical order*.

In this chapter we discuss the symmetry breaking constraints and the theorem which underlies them. We present the experimental results of exploration with and without the symmetry constraints. We conclude this chapter by presenting the results with real JPEG decoder and Video decoder applications.

6.1 symmetry in split-join graphs

The data-parallel tasks encoded in the split-join graph model are symmetrical in nature. They represent equal computations which can be executed in parallel. A permutation of instances of an actor in a schedule of a split-join graph can produce an equivalent schedule, such that the properties of the schedule (latency, number of processors used etc.) remain the same. Such equivalent schedules do not give any additional benefit to the programmer, rather they prove to be a hurdle in satisfiability proving of the problem. The amount of permutations grow exponentially with the number of data-parallel tasks. Thus these equivalent schedules must be excluded, without losing optimal solutions in the cost space. We discuss the symmetry in the split-join graphs further in detail.

Recall that in a split-join graph the decomposability of a task into parallelizable sub-tasks is expressed as a numerical label (parallelization factor) on a split edge leading to it. In Figure 2.6 an integer label α on the edge from *A* to *B* means that every executed instance of task *A* spawns α instances of task *B*. Likewise, a $1/\alpha$ label on the edge from *B* to *C* means that all those instances of *B* should terminate and their outputs be joined before executing *C*. The derived task graph can thus be viewed as obtained from the split-join graph by making data parallelism

explicit (see Figure 2.6). To distinguish between these two types of graphs we call the nodes of the split-join graphs *actors* (task types) and those of the task graph *tasks*.

In Chapter 2 we have defined the notions of consistency of split-join graphs and showed how a task graph can be derived from a consistent split-join graph. In this section we formally define a sub-class of consistent split-join graphs called well-formed split-join graphs. Most of practical split-join graphs are well-formed. For these graphs we prove a theorem leading to symmetry breaking for identical data-parallel tasks.

The DAG structure of a split-join graph naturally induces a partial-order relation $\angle$ over the actors such that $v \angle v'$ if there is a path form v to v'. The set of *minimal* elements with respect to $\angle$ is $V_\bullet \subseteq V$ consisting of nodes with no incoming edges. Likewise, the *maximal* elements $V^\bullet$ are those without outgoing edges. An *initialized path* in a DAG is directed path $\pi = v_1 \cdot v_2 \cdots v_k$ starting from some $v_1 \in V_\bullet$. Such a path is *complete* if $v_k \in V^\bullet$. With any such path we associate the *multiplicity signature*

$$\xi(\pi) = (v_1, \alpha_1) \cdot (v_2, \alpha_2) \cdots (v_{k-1}, \alpha_{k-1})$$

where $\alpha_i = r((v_i, v_{i+1}))$. We will also abuse ξ to denote the projection of the signature on the multiplication factors, that is $\xi(\pi) = \alpha_1 \cdot \alpha_2 \cdots \alpha_{k-1}$.

To ensure that different instances of the same actor communicate with the matching instances of other actors and that such instances are joined together properly, we need an *indexing scheme* similar to indices of multi-dimensional arrays accessed inside nested loops. Because an actor may have several ancestral paths, we need to ensure that its indices via different paths agree. This will be guaranteed by a well-formedness condition that we impose on the multiplicity signatures along paths.

DEFINITION 8 (Parenthesis Alphabet) *– Let $\Sigma = \{1\} \cup \Sigma_l \cup \Sigma_j$ be any set of symbols consisting of a special symbol 1 and two finite sets Σ_l and Σ_j admitting a bijection which maps every $\alpha \in \Sigma_l$ to $\alpha' \in \Sigma_j$, for $\alpha > 1, \alpha \in \mathbb{N}$.*

Intuitively α and α' correspond to a matching pair consisting of a split α and its inverse join $1/\alpha$. These can be viewed also as a pair of (typed) left and right *parentheses*.

DEFINITION 9 (Canonical Form) *– The canonical form of a sequence ξ over a parentheses alphabet Σ is the sequence $\bar{\xi}$ obtained from ξ by erasing occurrences of the neutral element 1 as well as matching pairs of the form $\alpha \cdot \alpha'$.*

For example, the canonical form of $\xi = 5 \cdot 1 \cdot 3 \cdot 1 \cdot 1 \cdot 1/3 \cdot 1 \cdot 2$ is $\bar{\xi} = 5 \cdot 2$. Note that the (arithmetic) products of the factors of ξ and of $\bar{\xi}$ are equal and we denote this value by $c(\xi)$ and let $c(\epsilon) = 1$. A sequence ξ is *well-parenthesized* if $\bar{\xi} \in \Sigma_l^*$, namely its canonical form consists only of left parentheses. Note that this notion refers also to signature *prefixes* that can be *extended* to well-balanced sequences, namely, sequences with no violation of being well-parenthesized by a join not compatible with the *last* open split.

DEFINITION 10 (Well Formedness) *– A split-join graph is well formed if:*
1. *Any complete path π satisfies $c(\xi(\pi)) = 1$;*
2. *The signatures of all initialized paths are well parenthesized.*

In fact, the first condition implies SDF consistency. It ensures that the graph is meaningful (all splits are joined) and that the multiplicity signatures of any two paths leading to the same actor v satisfy $c(\xi) = c(\xi')$. We can thus associate unambiguously this number with the actor itself and denote it by $c(v)$. Note that it coincides with the repetition count defined in Section 2.2. The *repetition count* is the number of instances of actor v that should be executed.

The second condition forbids, for example, sequences of the form $2 \cdot 3 \cdot 1/2 \cdot 1/3$. It implies an additional property: every two initialized paths π and π' leading to the same actor satisfy $\bar{\xi}(\pi) = \bar{\xi}(\pi')$. Otherwise, if two paths would reach the same actor with different canonical signatures, there will be no way to close their parentheses by the same path suffix. Although consistent split-join graphs *not* satisfying Condition 2 can make sense for certain computations, they require more complicated mappings between tasks and they will not be considered here. As for the restrictions that we imposed on the split-join graph compared to more general SDF graphs admitting non-divisible token production and consumption rate, let us first remark that the main result of this chapter, Theorem 6.1, can be extended, somewhat less elegantly, to an acyclic SDF, as we do in [120]. Moreover, the extensive study of StreaMIT benchmarks found in [128] reports that most actors in most applications, fall into the category of well-formed split-join graphs that we treat [1]. For well-formed graphs, a *unique canonical signature*, denoted by $\bar{\xi}(v)$, is associated with every actor.

DEFINITION 11 (Indexing Alphabet and Order) – *An actor v with $\bar{\xi}(v) = \alpha_1 \cdots \alpha_k$ defines an indexing alphabet A_v consisting of all k-digit sequences $h = a_1 \cdots a_k$ such that $0 \le a_i \le \alpha_i - 1$. This alphabet can be mapped into $\{0, \ldots, c(v) - 1\}$ via the following recursive rule:*

$$\mathcal{N}(\varepsilon) = 0 \quad and \quad \mathcal{N}(h \cdot a_j) = \alpha_j \cdot \mathcal{N}(h) + a_j$$

We use $\ll_v$ to denote the lexicographic total order over A_v which coincides with the numerical order over $\mathcal{N}(A_v)$.

Every instance of actor v will be indexed by some $h \in A_v$ and will be denoted as v_h. We use notation h and A_v to refer both to strings and to their numerical interpretation via $\mathcal{N}$. In the latter case v_h will refer to the task in position h according to the lexicographic order $\ll_v$. See for example, tasks $B_0, B_1, \ldots$ in Figure 2.6.

Recall from Definition 3, that a derived task graph consists of tasks U which are connected by edges $\mathcal{E}$. In the context of well-formed graphs, we establish a relation between the tasks and dependency arcs on the one hand and their corresponding actors and channels on the other hand as follows:

$$U = \{v_h | v \in V, h \in A_v\}$$
$$\mathcal{E} = \{(v_h, v'_{h'}) \mid (v, v') \in E, (h \sqsubseteq h' \vee h' \sqsubseteq h)\}$$
$$\forall v, \forall h \in A_v \; \delta(v_h) = d(v)$$

Notation $h \sqsubseteq h'$ indicates that string h' is a prefix of h. To take an example, according to the definition, a split edge (v, v') is expanded to a set of dependency arcs $\{(v_h, v'_{h \cdot a}) \mid a = 0 \ldots \alpha - 1\}$, where $\alpha = r((v, v'))$. The tasks can be partitioned naturally according to their actors, letting $U = \bigcup_{v \in V} U_v$ and $U_v = \{v_h : h \in A_v\}$. A permutation $\omega : U \to U$ is *actor-preserving* if it can be written as $\omega = \bigcup_{v \in V} \omega_v$ and each ω_v is a permutation on U_v.

DEFINITION 12 (Deployment) – *A deployment for a task graph $T = (U, \mathcal{E}, \delta)$ on an execution platform with a finite set M of processors consists of a mapping function $\mu : U \to M$ and a scheduling function $s : U \to \mathbb{R}_+$ indicating the start time of each task. A deployment is called feasible if it satisfies precedence and mutual exclusion constraints, namely, for each pair of tasks we have:*

Precedence: $\quad (u, u') \in \mathcal{E} \to s(u') - s(u) \ge \delta(u)$

Mutual exclusion: $\quad \mu(u) = \mu(u') \to [(s(u') - s(u) \ge \delta(u)) \vee (s(u) - s(u') \ge \delta(u'))]$

---

1. From a total of 65 application benchmarks, it is observed that 63% of the actors in a program have same repetition count. An average of 72% of the actors have a repetition count of 1. Non-divisible communication rates, for example in CD-DAT benchmark, are rare.

Note that $\mu(u)$ and $s(u)$ are decision variables while $\delta(u)$ is a constant. The *latency* of the deployment is the termination time of the last task, $\max_{u \in U}(s(u) + \delta(u))$. The problem of optimal scheduling of a task-graph is already NP-hard due to the non-convex mutual exclusion constraints. This situation is aggravated by the fact that the task-graph will typically be exponential in the size of the split-join graph. On the other hand, it admits many tasks which are identical in their duration and isomorphic in their precedence constraints. In the sequel we exploit this symmetry by showing that all tasks that correspond to the same actor can be executed according to a *lexicographic order* without compromising latency.

DEFINITION 13 (Ordering Scheme) – *An ordering scheme for a task-graph $T = (U, \mathcal{E}, \delta)$ derived from a split-join graph $G = (V, E, r, d)$ is a relation $\prec = \bigcup_{v \in V} \prec_v$ where each $\prec_v$ is a total order relation on U_v.*

In the lexicographic ordering scheme $\ll$, the tasks $v_h \in U_v$ are ordered in the lexicographic order $\ll_v$ of their indices 'h'. We say that a schedule s is *compatible* with an ordering scheme $\prec$ if $v_h \prec v_{h'}$ implies $s(v_h) \leq s(v_{h'})$. We denote such an ordering scheme by $\prec^s$ and use notation $v[h]$ for the task occupying position h in $\prec_v^s$.

LEMMA 1 – *Let s be a feasible schedule and let v and v' be two actors such that $(v, v') \in E$. Then*
  1. *If $r(v, v') = \alpha \geq 1$, then for every $h \in [0, c(v) - 1]$ and every $a \in [0, \alpha - 1]$ we have*
$$s(v'[\alpha h + a]) - s(v[h]) \geq d(v).$$
  2. *If $r(v, v') = 1/\alpha$ then for every $h \in [0, c(v) - 1]$ and every $a \in [0, \alpha - 1]$ we have*
$$s(v'[h]) - s(v[\alpha h + a]) \geq d(v).$$

Proof. The precedence constraints for Case 1 are in fact $s(v'_{\alpha h + a}) - s(v_h) \geq d(v)$, and we have to prove that in this expression the lexicographic index v_h can be replaced by schedule-compatible index $v[h]$. Let $j = h\alpha + a$ and $j' = j + 1$. Since each instance of v is a predecessor of exactly α instances of v', the execution of $v'[j]$ must occur after the completion of *at least* $\lceil j'/\alpha \rceil$ instances of v. By construction, this is not earlier than the termination of the *first* $\lceil j'/\alpha \rceil$ instances of v to occur in schedule s. In our notation this can be written as:
$$s(v'[j]) \geq s(v[\lceil j'/\alpha \rceil - 1]) + d(v)$$
Substituting j and j' into the above formula we obtain our hypothesis. A similar argument holds for Case 2. $\square$

THEOREM 6.1 (Lexicographic Ordering) – *Every feasible schedule s can be transformed into a latency-equivalent schedule s' compatible with the lexicographic order $\ll$.*

Proof. Let ω_s be an actor-preserving permutation on U defined as $\omega_s(v_h) = v[h]$. In other words, ω_s maps the task in position h according to $\ll$ to the task occupying that position according to $\prec^s$. The new deployment is defined as
$$\mu'(v_h) = \mu(\omega_s(v_h)) \quad \text{and} \quad s'(v_h) = s(\omega_s(v_h)).$$
Permuting tasks of the same execution time does not influence latency nor the satisfaction of resource constraints. All that remains to show is that s' satisfies precedence constraints. Each v_h is mapped into $v[h]$ and each of its v' sons (respectively parents) is mapped into $v'[\alpha h + a]$, $0 \leq a \leq \alpha - 1$. Hence a precedence constraint for s' of the form
$$s'(v_{h \cdot a}) - s'(v_h) \geq d(v)$$
is equivalent to
$$s(v[\alpha h + a]) - s(v[h]) \geq d(v)$$

(a) A task graph   (b) A schedule   (c) A lexicographic schedule
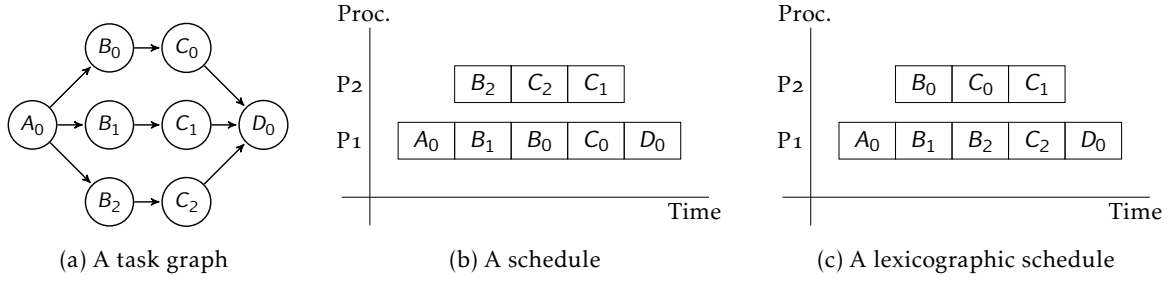
Figure 6.1 – Illustration of the lexicographic ordering theorem

which holds by virtue of Lemma 1 and the feasibility of s.   □

For example, in Figure 6.1 we illustrate a task graph, a feasible schedule and the same schedule transformed into a lexicographic-compatible schedule by a permutation of the task indices. The implication of this result is that we can introduce additional constraints to the formulation of the scheduling problem requiring lexicographic compatibility of the schedule without losing optimality. These constraints enforce one ordering of equivalent tasks in time out of many possible, thus significantly reducing the search space.

6.2 SMT CONSTRAINTS

Expressing scheduling problems using constraint solvers is fairly standard [10, 22, 79, 134] and various formulations may differ in the assumptions they make about the application and the architecture and the aspects of the problem they choose to capture. For split-join graphs, we need to adapt these constraints and re-invent them in order to accommodate the calculation of communication buffer size. In the following section we write down the constraints that define a feasible schedule and its cost in terms of latency, number of processors and buffer size.

■ **Completion time and precedence**: $e(u)$ is the time when task u terminates and a task cannot start before the termination of its predecessors.

$$\bigwedge_{u \in U} e(u) = s(u) + \delta(u) \ \wedge \bigwedge_{(u,u') \in \mathcal{E}} e(u) \leq s(u')$$

■ **Mutual exclusion**: tasks running on same processor should not overlap in time.

$$\bigwedge_{u \neq u' \in U} (\mu(u) = \mu(u')) \Rightarrow (e(u) \leq s(u') \vee (e(u') \leq s(u)) \tag{6.1}$$

■ **Communication buffer**: these constraints compute the buffer size of every channel $(v, v') \in E$. They are based on the observation that buffer utilization is piecewise-constant over time, with jumps occurring upon initiation of writers and termination of readers. Hence the peak value of memory utilization can be found among one out of finitely-many starting points.
Recall that in Section 2.3 we define $w^{\uparrow}$ and $w^{\downarrow}$ as functions on split-join graph channel specifying the size of data in bytes produced and consumed on the channel by execution of one writer or reader instance respectively. Below we use $w^{\uparrow}_{v,v'}$ and $w^{\downarrow}_{v,v'}$ for the values of these functions for channel (v, v').
The first constraint defines $\tilde{w}^{\uparrow}(u, u_*)$, the contribution of writer $u \in U_v$ to the filling of

buffer (v, v') observed at the start of a writer $u_* \in U_v$:

$$\bigwedge_{(v,v') \in E} \bigwedge_{u \in U_v} \bigwedge_{u_* \in U_v} \begin{array}{l} (s(u) > s(u_*)) \wedge (\tilde{w}^{\uparrow}(u, u_*) = 0) \vee \\ (s(u) \leq s(u_*)) \wedge (\tilde{w}^{\uparrow}(u, u_*) = w^{\uparrow}_{v,v'}) \end{array}$$

Likewise the value $\tilde{w}^{\downarrow}(u', u_*)$ is the (negative) contribution of reader u' to buffer (v, v') observed at the start of writer u_*:

$$\bigwedge_{(v,v') \in E} \bigwedge_{u' \in U_{v'}} \bigwedge_{u_* \in U_v} \begin{array}{l} ((e(u') > s(u_*)) \wedge \tilde{w}^{\downarrow}(u', u_*) = 0) \vee \\ (e(u') \leq s(u_*)) \wedge (\tilde{w}^{\downarrow}(u', u_*) = w^{\downarrow}_{v,v'}) \end{array}$$

The total amount of data in buffer (v, v') at the start of task $u_* \in U_v$, denoted by $R_{v,v'}(u_*)$, is the sum of contributions of all readers and writers already executed:

$$\bigwedge_{(v,v') \in E} \bigwedge_{u_* \in U_v} R_{v,v'}(u_*) = \sum_{u \in U_v} \tilde{w}^{\uparrow}(u, u_*) - \sum_{u' \in U_{v'}} \tilde{w}^{\downarrow}(u', u_*)$$

The buffer size for (v, v'), denoted by $B_{v,v'}$ is the maximum over all the start times of tasks in U_v:

$$\bigwedge_{(v,v') \in E} \bigwedge_{v_* \in U_v} R_{v,v'}(u_*) \leq B_{v,v'}$$

- **Costs**: The following constraints define the cost vector associated with a given deployment, which is $C = (C_L, C_M, C_B)$, where the costs indicate, respectively, *latency* (termination of last task), number of *processors* used and total *buffer size*.

$$\bigwedge_{u \in U} e(u) \leq C_L \ \wedge \ \bigwedge_{u \in U} \mu(u) \leq C_M \ \wedge \ \sum_{(v,v') \in E} B_{v,v'} \leq C_B$$

We refer to the totality of these constraints as $\varphi(\mu, s, C)$ which are satisfied by any feasible deployment (μ, s) whose cost is C.

- **Symmetry breaking:** We add two kinds of symmetry-breaking constraints, which do not change optimal costs.

  ◇ **Task Symmetry**: We add the lexicographic task ordering constraints justified by [Theorem 6.1](#)

  $$\bigwedge_{v \in V} \bigwedge_{v_h, v_{h+1} \in U_v} s(v_h) \leq s(v_{h+1})$$

  where v_h denotes the instance of v at the hth position in the order $\ll_v$.

  ◇ **Processor Symmetry**: Secondly we add fairly standard constraints to exploit the processor symmetry: processor 1 runs task 1, processor 2 runs the lowest index task not running on processor 1, etc. Therefore, let us number all tasks arbitrarily with a unique index: u^1, u^2, etc. The processor symmetry breaking is defined by the following constraint:

  $$\mu(u^1) = 1 \ \wedge \bigwedge_{2 \leq i \leq |U|} \mu(u^i) \leq \max_{1 \leq j < i} \mu(u^j) + 1$$

It can be easily checked that the costs such as latency, number of processor used, buffer size etc. remain unchanged with such restrictions.

6.3 EXPERIMENTS

In this section we investigate the performance of the cost-space exploration algorithm using the constraints defined in the previous section. First, we assess the contribution of the symmetry reduction constraints on the execution time and the quality of solutions for a synthetic example. Then we explore the cost space for a split-join graph derived from a real

(a) 5-processor deployment

(b) 20-processor deployment
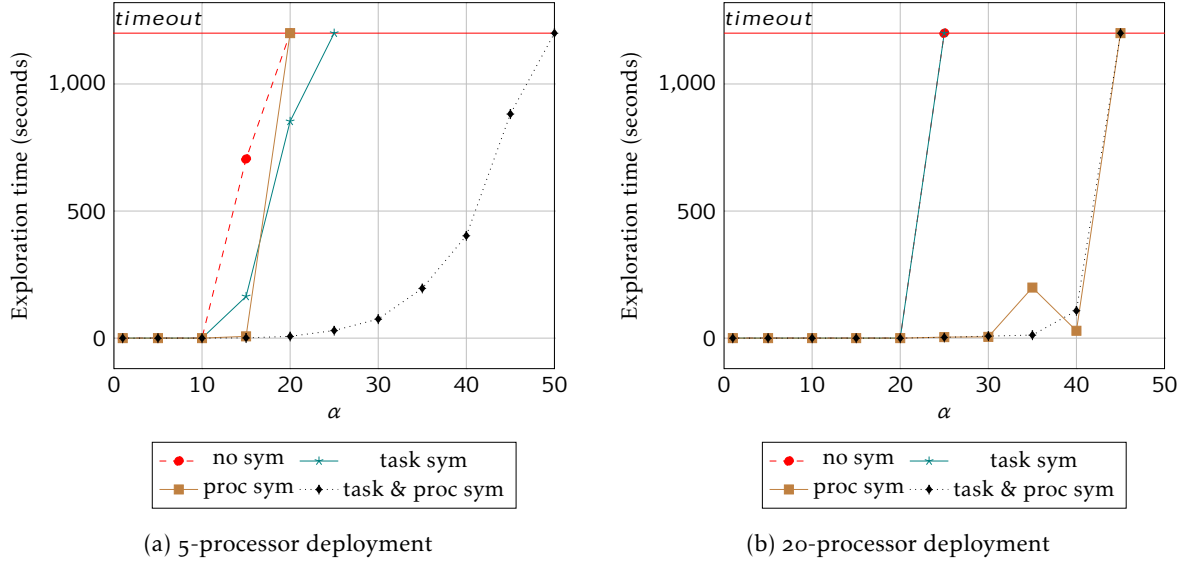
Figure 6.2 – Time to find optimal latency as a function of the number of tasks for 5 and 20 processors.

video application. These experiments use version 4.1 of the Z3 Solver [87] running on a Linux machine with *Intel Core i7* processor at 1.73 GHz with 4 GB of memory. Finally, we validate the model used to derive the solution by deploying a JPEG decoder on the Tilera platform [129] according to the derived schedule. The measured performance is very close to the predicted one.

6.3.1 Finding Optimal Latency

We use the split-join graph of Figure 2.6 with various values of α to explore the effect of the symmetry reduction constraints on the performance of the solver. We start with a single cost version of the problem and perform binary search to find the minimum latency that can be achieved for a fixed number of processors. We solve the same problem using four variations of the constraints: without symmetry reduction, with processor symmetry, with task symmetry and with both. Figure 6.2 depicts the computation times for finding the optimal latency as a function of α on platforms with 5 and 20 processors. We use time-out per query of 20 minutes, which is much larger than the one minute we typically use because we want to find the *exact* optimum in order to compare the effects of different symmetry constraints. Scheduling problems are known to be easy when the number of processors approaches the number of tasks. For the difficult case of 5 processors, task symmetry starts dominating beyond 10 tasks and the combination of both gives the best results. It increases the size of graphs whose optimal latency can be found (with no query executing more than 20 minutes) from $\alpha = 12$ to $\alpha = 48$. Not surprisingly, for 20 processors, the relative importance of processor symmetry grows. In Figure 6.2(b) we see no advantage from the task symmetry, as the problem suffers mainly from processor symmetry.

We perform a similar experiment but now we explore not only for different α but also for different number of processors. In this experiment, we fix the number of processors and α. Again we perform binary search in order to find the minimal latency that can be achieved. This time we put a time-limit of three minutes on each of the query. We do not change the time-out settings. We perform this experiment for all four settings of symmetry. The results

(a) Without symmetry constraints

(b) With task symmetry constraints

(c) With processor symmetry constraints

(d) With task and processor symmetry constraints
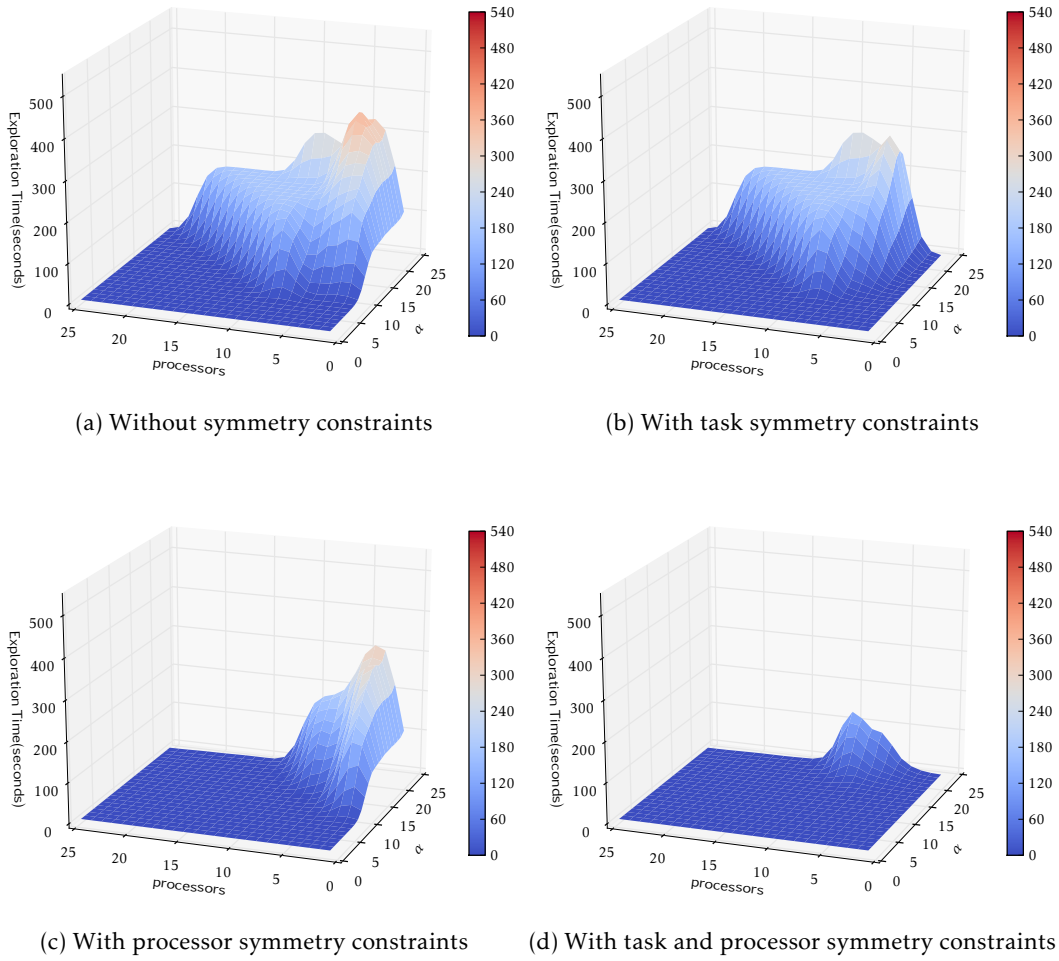
Figure 6.3 – Exploration time to find optimal latency as a function of the number of tasks and processors.

of this experiment are shown in Figure 6.3. We observe in this experiment the peak levels where the solver takes maximum time when no symmetry breaking is applied to the problem. When the task symmetry constraints are applied to the exploration, we see the reduction in exploration time for larger α. However the symmetry due to large number of processors still prevails. When we apply just processor symmetry constraints, we lose the benefit of the task symmetry, but the solver time for larger number of processors is reduced significantly (seen in Figure 6.3(c)). When both task and processor symmetry are applied to the problem, we get a significant reduction, for both α and large number of processors, as observed in Figure 6.3(d). Thus we observe the benefits of task symmetry and processor symmetry constraints for the problem.

6.3.2   *Processor-Latency Trade-offs*

To demonstrate the effect of symmetry reductions on the Pareto front exploration we fix $\alpha = 30$ and seek trade-offs between latency and the number of processors. We use a time budget of one minute per query and a total time budget to 60 minutes, but is never reached. Figure 6.4 depicts the results obtained with and without symmetry breaking constraints.
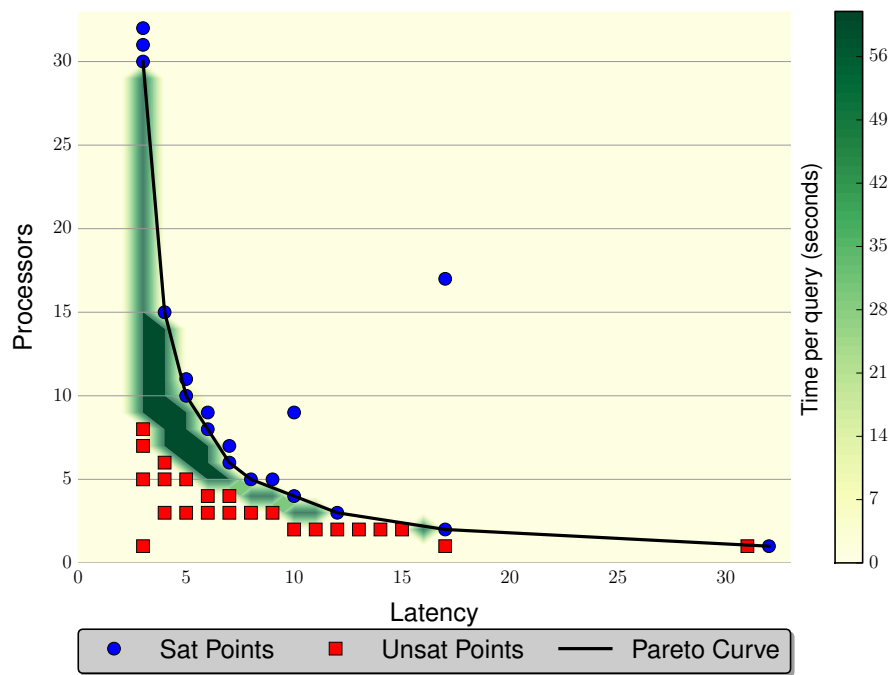
(a) Without symmetry constraints
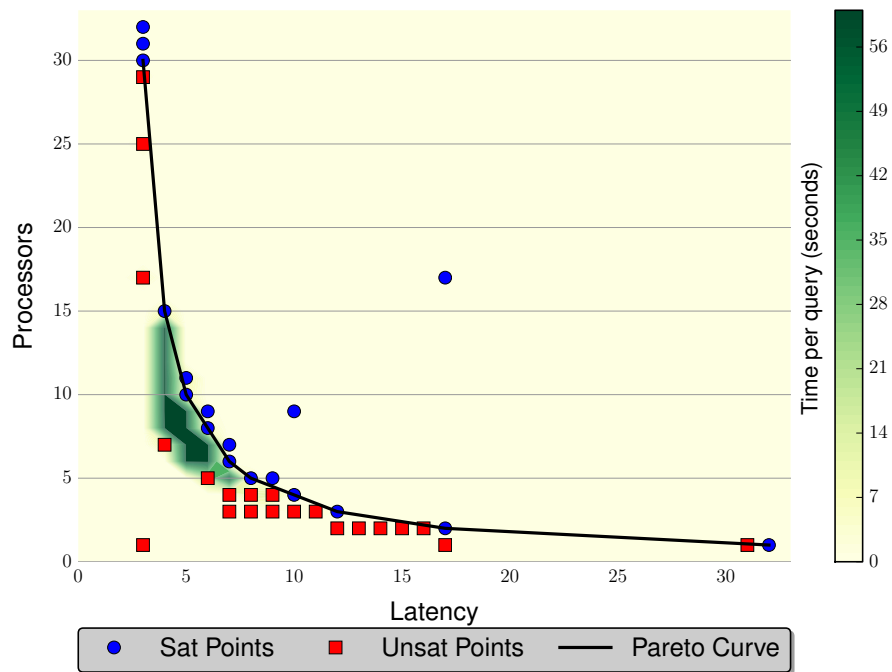
(b) With task and proc. symmetry constraints

Figure 6.4 – Result of Pareto exploration for $\alpha = 30$
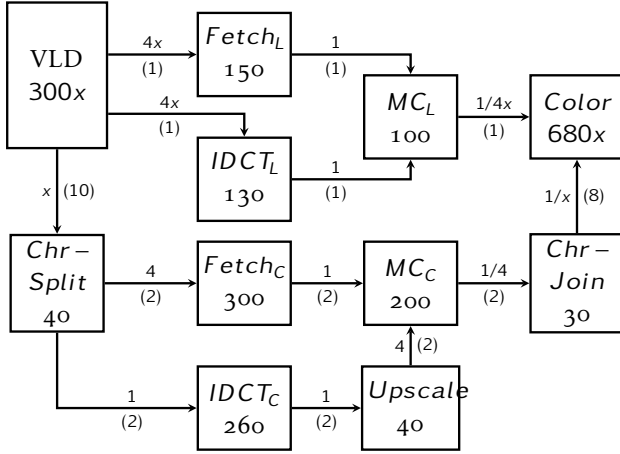
Figure 6.5 – Video decoder example

Actor	Execution Time
VLD	$300 \cdot x$
Fetch$_L$	150
IDCT$_L$	130
Mc$_L$	100
Chr − Split	40
Fetch$_C$	300
IDCT$_C$	260
Upscale	40
Mc$_C$	200
Chr − Join	30
Color	$680 \cdot x$

Table 6.1 – Execution times

The square points show the **unsat** points whereas the circle are the **sat** points. The black curve is the approximation of the Pareto front, connecting all the minimal **sat** points. Points whose queries took long time to answer are surrounded by a dark halo whose intensity is proportional to the time (the darkest areas are around the **timeout** points). As one can see from the figure, symmetry constraints reduce significantly the number of time-outs. From additional experiments we noticed that processor symmetry reduces the upper-left part of the curve while task symmetry is useful around the middle. The total exploration time observed is 42 minutes without symmetry, and 16 minutes with both types of symmetry constraints.

6.3.3 *A Video decoder*

Next we perform a 3-dimensional cost exploration for a model of a video decoder taken from [42]. In our video decoder programming model, a complete video frame can be processed by a repeated execution of the split-join graph shown in Figure 6.5. Every edge is marked with a parallelization factor α and data token size in blocks (in parentheses) for buffer size computations. The *VLD* actor parses the input bitstream, extracting the subsequent macroblocks. The parameter *x* selects the number of macroblocks processed per one graph iteration. The larger *x* the more macroblocks can be processed in parallel, but the more difficult it is to generate an optimal schedule. The actors indexed by 'L' process four luminance blocks per macroblock and the actors indexed by 'C' process two chrominance blocks. The *Color* actor converts the frame into the RGB format. The *Fetch* actors fetch the reference blocks from the previous frame for motion compensation, done by *MC* actors. Actor *Upscale* scales 2 chrominance blocks into 8. The weights *w* of the channels (in blocks) are depicted in parentheses in the figure. We perform the exploration for $x = 5$ which yields a task-graph with 122 tasks.

Without any symmetry constraints the solver quickly times out for most queries of interest. Symmetry constraints do not completely eliminate time-outs but reduce them significantly and therefore the quality of the Pareto front approximation is much better, as shown in Figure 6.6. Note that for a sequential implementation ($C_M = 1$) the constraints improve the best buffer size found from 276 to 182 and for the most parallel deployment ($C_M = 122$) they reduce the best latency from 10 K to 7 K and the buffer size from 333 to 229. The Pareto point $(14, 333, 62)$ found without symmetry constraints is strongly dominated by the point $(10, 229, 31)$ found with symmetry breaking. This solution improves the latency and buffer usage by roughly a third

while using half of the processors. We believe it is a promising indication of the applicability of our approach and of the potential performance gains in treating the optimization problem globally.
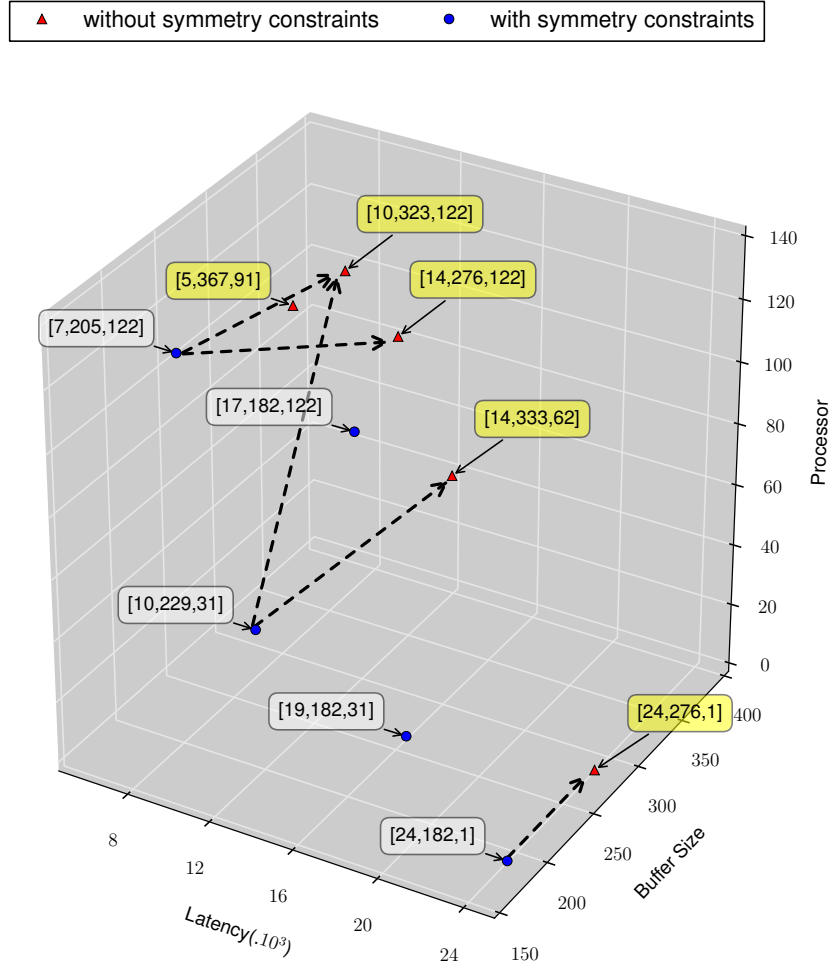
Figure 6.6 – Video decoder exploration result

6.3.4 *JPEG decoder*

Finally we validate our model by deploying a JPEG decoder, see Figure 2.7, on the Tilera platform [129] which is a 64-core symmetric multi-core platform running at 862.5 MHz (described in Section 3.2). Unlike the real execution on the platform, theoretical scheduling problem that we solve is *deterministic* where task execution times are assumed to be precisely known. The obtained schedule is time-triggered, given in terms of the *exact* start time function s. In reality, there are variations in execution times and in our implementation we use static order schedules, preserving only the *order* of task execution on each processor (as described in Section 5.3). This is a common way to generate schedules for task graphs and SDF, see for example [115]. When task execution times agree with the nominal values used in the optimization problem, this scheduling policy for a non-lazy schedule coincides with s. Unlike the traditional work on dataflow mapping, we support mappings where the writers and readers of the same buffer storage can be spread over more than two different processors. Our experience confirms that this scheduling policy can be implemented with a reasonable amount of additional synchronization between the cores. Note also that when the schedule

is compatible with lexicographical task order (justified by Theorem 6.1), the accesses to the channels automatically become FIFO and this facilitates the implementation of cyclic buffers (described in Section 5.4).

The JPEG decoder has three main actors: variable length decoding (*VLD*), inverse quantization and inverse discrete cosine transform (*IQ/IDCT*) combined and color conversion (*Color*). To measure execution times we run the decoder several times on a single processor and measure the execution time of each actor. To mitigate cache effects, we consider the average execution time rather than worst case, which occurs only in the first execution due to cache misses. We use these average execution times in the model we submit to the solver. We then deploy the decoder on the platform and run it 100 times (again to dampen cache effects). The relation between the average latency (in *microseconds*) observed experimentally and the Pareto points computed by the SMT solver is depicted below and the deviation is typically smaller than 15%.

number of processors		1	3	4	6	8	12
Latency (μs)	predicted	506	314	278	261	243	226
	measured	461	309	299	307	300	351

We observe in these results, that the predicted latency decreases with the increasing number of processors. However, in practice, we see the minimum latency with four processors and then an increase with the increasing number of processors. This effect is observed due to the fact that we do not model communication in this architecture. When multiple processors use the same FIFO buffers, it causes frequent data movement between them resulting from the cache coherency protocol. This movement of data is difficult to model and causes prediction errors in our experiment. From this we conclude that for better accuracy the communication should be modeled explicitly.

6.4 CONCLUSIONS

In this chapter we discussed the effect of symmetry constraints on the scheduling of split-join graphs. We presented results that prove that symmetry breaking constraints accelerates the exploration process for the SMT solver. We proved strength of these constraints first using synthetic benchmarks, and then with real-life applications such as JPEG and Video decoder (with 122 tasks) by a 3-dimensional cost-space exploration.

Various approaches to facilitate the task of the solver by additional symmetry breaking constraints have been tried, for example [101] for graph coloring or an automated method for discovering graph automorphism [32] which can lead to significant improvements [39]. However, our deployment problem does not require complex detection of isomorphic sub-graphs. Instead we exploit the knowledge about the structure of the task graphs coming from the original split-join graph and not relying in any way on the graph automorphism. We gave a simple proof of the lexicographic for arbitrary nesting, Theorem 6.1, thus improving the results of [39], which was restricted to one level of nesting as it was relying on isomorphic sub-graphs.

SMT solvers have been used to solve the scheduling problem previously [79]. There have been attempts to use constraint programming techniques to solve this problem [22, 134]. In [133] a quantitative model checking engine is developed using a variant of timed automata for combined scheduling and buffer storage optimization of SDF graphs.

In the next chapter, we solve the same scheduling problem for Kalray processor architecture. We observe that the inherently complex nature of the platform, restricts us from solving the problem as a whole. In addition, the explicit communication must be modeled in order to

predict the performance of the application correctly. Thus, in addition to scheduling problem, which in that case becomes more complex, we employ the solver for additional optimization problems that concern with planning the communication on the platform.

Multi-stage scheduling for distributed memory processors

This chapter introduces the multi-stage approach that we use for scheduling applications on the Kalray processor architecture.

In previous chapters we observed that on the Tilera platform, with help of the symmetry reduction constraints, the SMT solver was able to handle a scheduling problem with 122 tasks. Tilera is a, programming-wise, simple shared-memory platform with hardware support of single consistent memory space visible from all cores. Due to this support, the data transfer between the processor cores was managed by the hardware, and transparent to the software. Hence the number of decision variables in the optimization problem was relatively small. In contrast to the Tilera platform, the Kalray platform has a richer and more complex set of mechanisms which are exposed to the user. The processors are grouped into clusters so that processors on the same cluster communicate rapidly via shared memory while tasks that run on different clusters communicate via more expensive means (DMA channels over the NoC) which are under explicit control of the programmer. A modeling and optimization framework that does not reflect communication costs will not be useful for this architecture.

Thus the optimization problem on Kalray has many more decision variables that affect costs such as load-balancing, communication, number of used clusters etc. If all these decision variables as a monolithic optimization problem are presented to the SMT solver, it would result in a very complex set of constraints, practically solvable only for a small number of tasks. Thus in order to be able to schedule larger applications on this architecture, the problem must be split into a few sub-problems. After splitting, the top-level decisions about load-balancing and cluster allocation are made first and later the scheduling inside the clusters is done, combined with mapping to cores and buffer size allocation. This makes the problem solving better tractable.

Our design flow consists of three stages. First we partition the actors into software clusters, then we map them into the clusters of the platform and finally scheduling (of execution and communication) for each cluster. The inter-cluster communication, performed using DMA, must be modeled correctly to obtain good latency prediction. We discuss the constraints in detail and the modeling of communication by inserting new actors and edges representing the data transfer and flow control, to support program execution using limited resources. Finally
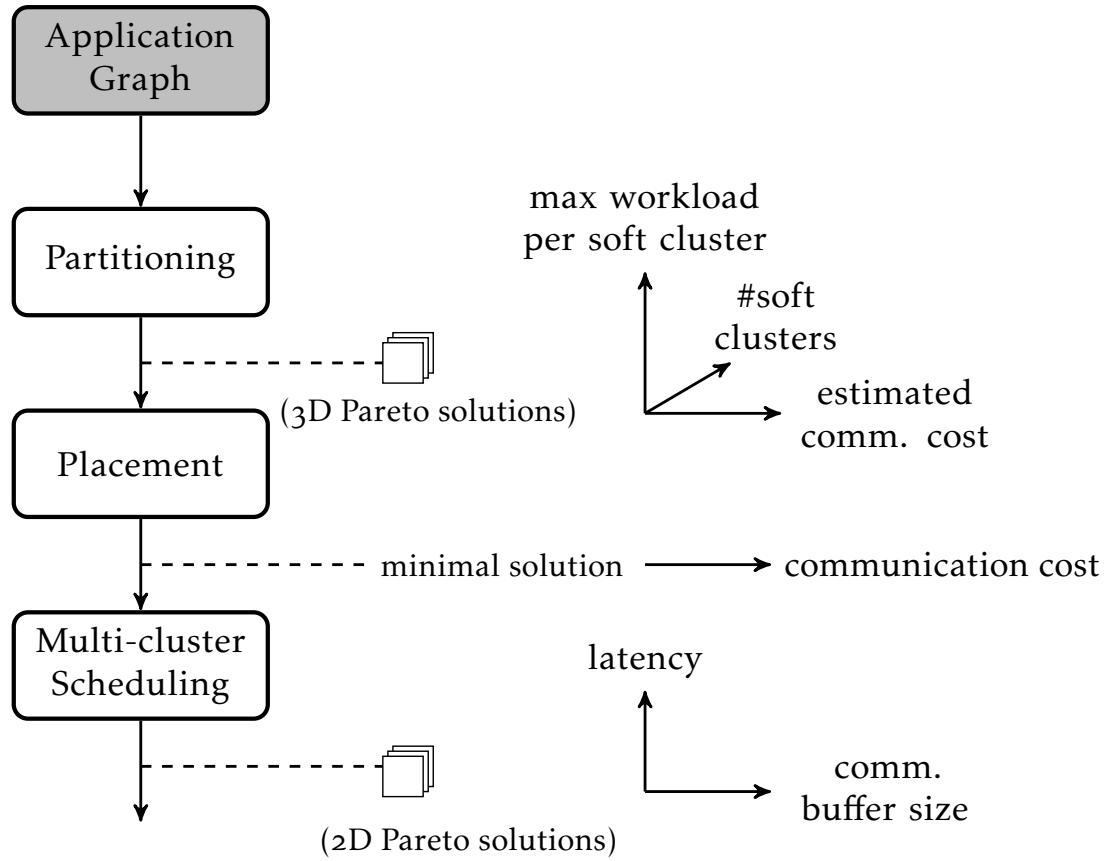
Figure 7.1 – Multi-stage design flow

we present the results of experiments that were performed on the Kalray processor using a number of streaming applications.

7.1 DESIGN FLOW

Kalray platform has explicit communication mechanisms which have costs that cannot be ignored and must be taken into account in order to perform a faithful communication modeling. We explain in Section 7.2 how inter-task communication is implemented depending on whether or not the two communicating tasks reside on the same cluster. As we shall see, when they are not on the same cluster, we need to augment the split-join/task graphs with additional *communication tasks* which are based on the DMA cost model introduced in Section 3.7. Moreover, in case of communication channels between two different clusters we need to model synchronization as well. As a result we obtain models that yield constrained optimization problems too large to be solved monolithically and we need to split (as suggested in [66]) the design flow into stages described below.

- **Software partitioning**: We partition the actors into groups (software clusters) with the intention that each software cluster is executed on one (hardware) cluster of the platform. Solutions are evaluated according to three criteria: the number of soft clusters (which determines the number of hardware clusters that will be used for the application), the maximal computational workload over the soft clusters, and the amount of communication between tasks belonging to different clusters. These are

conflicting criteria and we provide a set consisting of the best trade-offs provided by our cost exploration procedure.

- **Mapping software to hardware clusters**: Due to the toroidal architecture of the interconnect, the distance between pairs of clusters is not uniform and hence, while mapping soft clusters to hard ones we attempt to optimize inter-cluster communication multiplied by the distance. This is a standard optimization problem applied to each of the solution obtained in the previous stage.
- **Mapping and scheduling**: In this stage we map the tasks into the cores of their respective clusters and perform scheduling on shared resources: cores that run numerous tasks and DMA channels that serve many software channels. To this end the original split-join and task graphs are modified to reflect tasks associated with DMA transfers as well as synchronization mechanism.

The outcome of the process is a set of solutions in terms of schedules, representing different trade-offs between latency, amount of memory used in a cluster and number of clusters. Our run-time environment executes these multi-cluster schedules produced by the SMT solver, where we measure the latency and compare with the value predicted by the model. Below we provide a detailed description of those steps.

7.1.1 *Software partitioning*

In the first step, we partition the actors of the application graph. We follow the partitioning technique of [30], adapted here to our application model. We experimentally observed (not reported here) that combined partitioning and placement is a hard problem, which significantly limits the size of problems that can be solved. Thus we decide to keep them as two different sub-problems.

In this step, all the actors of the applications are grouped so that each actor is assigned to a unique group called *soft cluster*. The workload assigned to a soft cluster is then equal to the product of the execution time of the actor and its repetition count. We calculate the total computation workload allocated to each soft cluster and try to minimize the maximal one.

Every two actors that communicate via a split-join channel incur zero communication cost if they are allocated to the same soft cluster. Otherwise the channel contributes to the communication cost proportionally to the total amount of data communicated through the channel in one graph iteration.

The maximal workload per soft cluster and communication cost are conflicting criteria. If we try to minimize the workload per soft cluster (equally distribute the workload among the soft clusters), we increase the communication cost (communicating actors allocated to different soft clusters). Similarly if we try to reduce the communication cost by assigning more actors to the same soft cluster, the maximum workload in a soft cluster increases. We apply our grid based design-space algorithm to this multi-criteria optimization problem to estimate the Pareto front.

Problem Inputs:

- Application Graph $S^M = (V^M, E^M, d, \alpha, \omega)$.
- Hardware Architecture Model $A = (X, \phi, M, D, I, g, \chi)$

Output:

- A partition of at most $|X|$ soft clusters represented by a mapping $z : V \rightarrow \mathcal{Z}$, where $\mathcal{Z} = \{1, 2, 3..., |X|\}$ is a set of soft clusters identified by unique numbers.

Costs and bounds:

Symbol	Cost Description	Bounds	
		Lower	Upper
C_τ	Max. computation workload per soft cluster	$min \bigwedge\limits_{v \in V^M} d(v) \cdot c(v)$	$\sum\limits_{v \in V^M} d(v) \cdot c(v)$
C_η	Estimated communication cost	o	$\sum\limits_{(v,v') \in E^M} c(v) \cdot w^\uparrow(v,v')$
$C_\mathcal{Z}$	Number of soft clusters	1	$\lvert X \rvert$

Constraints:

Values $z(v)$ for all actors are decision variables that have to be computed by the solver. The partitioning variables have to satisfy the following constraints:

- **Workload Calculation**: Let variables $\tau(a)$ signify the total workload assigned to soft cluster a. The workload of an actor can be easily calculated by the execution time times its repetition count. The workload allocated to a soft cluster is the sum of workload of actors allocated to it.

$$\bigwedge_{a \in \mathcal{Z}} \tau(a) = \sum_{v \in V^M \,:\, z(v)=a} d(v) \cdot c(v)$$

- **Estimated Communication Cost for soft cluster**: Let $\eta(v,v')$ be the estimated communication cost associated with channel (v,v'). The channel will incur communication cost only if the reader and writer of the channel are allocated to different soft clusters. The estimated communication cost is equal to the amount of data that is being produced (or consumed) on the channel.

$$\bigwedge_{(v,v') \in E^M \,:\, z(v) \neq z(v')} \eta(v,v') = c(v) \cdot w^\uparrow_{v,v'} = c(v') \cdot w^\downarrow_{v,v'}$$

If the reader v and the writer v' are in the same soft cluster, then the communication cost for them is zero.

$$\bigwedge_{(v,v') \in E^M \,:\, z(v) = z(v')} \eta(v,v') = 0$$

The last three constraints simply define the three cost criteria.

- **Partition Cost**: This constraint defines the partitioning cost of the solution in terms of number of soft clusters. The number of soft clusters to which the actors are assigned should be less than the cost $C_\mathcal{Z}$.

$$\bigwedge_{v \in V^M} z(v) \leq C_\mathcal{Z}$$

- **Total Estimated Communication Cost**: The total communication cost is subject to cost constraint and is equal to the sum of communication cost of all the channels.

$$\sum_{(v,v') \in E^M} \eta(v,v') \leq C_\eta$$

- **Workload Cost**: C_τ is the maximum workload.

$$\bigwedge_{a \in \mathcal{Z}} \tau(a) \leq C_\tau$$

The solution expresses the assignment of actors to their respective soft clusters. The communication cost and the workload incurred by each soft cluster can be trivially derived from it.

Our cost exploration algorithm produces an approximation of a three-dimensional Pareto front of solutions, and each of the discovered solutions is then subject to subsequent steps of

placement and scheduling.

7.1.2 *Mapping software to hardware cluster*

The placement step allocates a unique physical cluster to every soft cluster. In this step, the communication cost is based on the distance between the clusters. We do a one-dimensional minimization of the total communication cost by a binary search method.

Problem Inputs:

- Application Graph S^M.
- Hardware Architecture Model A.
- Partitioning scheme z.

Output:

- $x : \mathcal{Z} \rightarrow X$: represents assignment of a soft cluster to platform cluster where we assume that clusters are identified by positive numbers: X = { 1, 2, ..., |X|}

Costs and bounds:

Symbol	Cost Description	Bounds	
		Lower	**Upper**
C_θ	Total communication cost	o	$\displaystyle\sum_{(v,v')\in E^M\ :\ z(v)\neq z(v')} c(v)\cdot w_{v,v'}^{\uparrow}\cdot\phi_{max}$

Constraints:

The values for decision variables $\{x(a) : a \in \mathcal{Z}\}$ for soft clusters computed by the solver at this step should satisfy the following constraints.

- **Allocation of soft cluster to platform cluster**: We assign the soft cluster a to a cluster from set X of clusters available on the platform.

$$\bigwedge_{a\in\mathcal{Z}} x(a) \leq |X|$$

- **Unique cluster for every soft cluster**: No two soft clusters should be placed on the same cluster:

$$\bigwedge_{a,b\in\mathcal{Z}\ :\ a\neq b} x(a) \neq x(b)$$

- **Communication costs between soft clusters**: The distance aware communication cost $\theta(a, b)$ between two soft clusters a,b is the weighted sum of the communication between respective actors with the weight given by the distance between the clusters.

$$\bigwedge_{a,b\in\mathcal{Z}} \theta(a,b) = \sum_{\substack{(v,v')\in E^M:\\ z(v)=a\\ z(v')=b}} \eta(v,v')\cdot\phi_{x(a),x(b)}$$

- **Total communication cost**: The total communication cost is the sum of communication costs of all the soft clusters. It is a cost constraint which is subject to minimization.

$$\sum_{a,b\in\mathcal{Z}:a\neq b} \theta(a,b) \leq C_\theta$$

The solution of the placement step expresses a mapping of each soft cluster to a unique cluster on the platform.
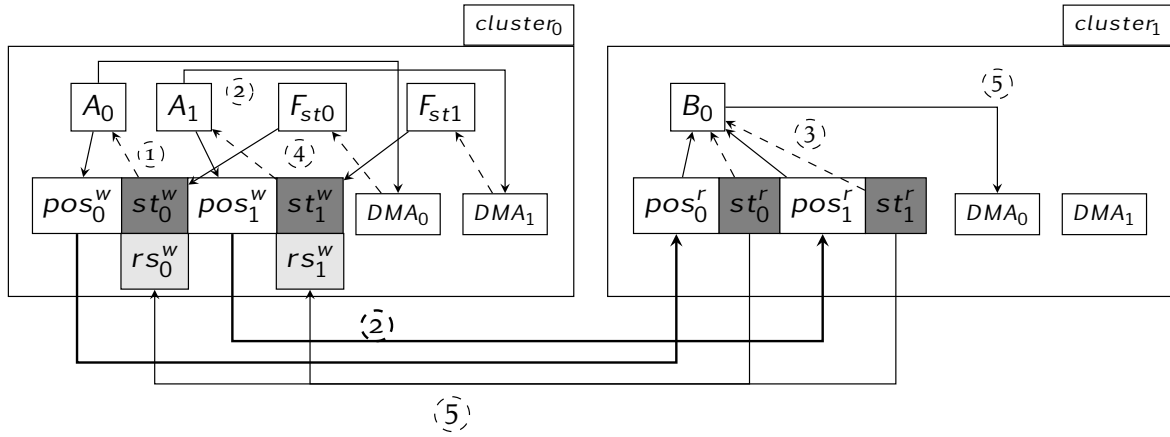
Figure 7.2 – Working of inter-cluster FIFO

7.2 INTER-CLUSTER FIFO

Before we further describe the design flow, we need to explain how we implement communication between tasks that reside in different clusters. The basic ideas have been already introduced in Section 5.4, whereas here we give more implementation details by means of an illustrative example. We implement an inter-cluster FIFO which transparently handles the data transfer and synchronization between the readers and writers of the FIFO. We follow the same example as in subsection 5.4.1 where we assumed a simpler, intra-cluster case. Suppose in a split-join graph, where A and B are connected actors with parameters $\alpha = 1/2$, $c(A) = 4$, $c(B) = 2$ $\alpha_{AB}^{\uparrow} = 1$, $\alpha_{AB}^{\downarrow} = 2$, and $m_{AB} = 2$. Also suppose that $z(A) \neq z(B)$, so they have to communicate using a DMA channel. Let us recall that the FIFO buffer is split to writer and reader sub-buffers. Figure 7.2 demonstrates such a scenario, where A_0, A_1, F_{st0} etc. are the tasks running on the processors of the writer cluster ($cluster_0$). Here F_{st} represents a special task which is needed to detect the completion of DMA transfer in the writer cluster. It is explained in detail later. The token data and token status entries in the FIFO buffer located in the local shared memory of the writer cluster is shown as pos_0^w, st_0^w etc. The corresponding data structures at the reader cluster ($cluster_1$) are shown as pos_0^r, st_0^r. The writer cluster has an additional status denoted by rs_0^w which indicates FIFO status at reader cluster for this position. The following steps illustrate how the inter-cluster communication takes place using the FIFO.

- **Step 1**: Task A_0 checks for the status st_0^w in the FIFO if the token status at position zero (pos_0^w) indicates an empty token. If the token status is 'e'-empty, then it marks the token as busy and starts to produce data in this position of the FIFO. Similarly task A_1 checks for status st_1^w and starts to produce data in position 1 of the FIFO. A point to note here is that A_0 and A_1 are not necessarily synchronized. They can execute on the same or different processors. Flags rs_0^w and rs_1^w are initialized empty before the execution of the schedules indicating free space for tokens on $cluster_1$.

- **Step 2**: After task A_0 finishes, the position 0 of the FIFO contains data and updated status, 'd' - contains data, the task A_0 starts a DMA transfer of token data and its status together, but not before the remote status rs_0^w indicates an empty token space at remote cluster, to move the data in position 0 and its status (pos_0^w and st_0^w in Figure 7.2) to another cluster (where tasks of actor B are located). Similarly task A_1 also starts another DMA to move the data and status in position 1. Or otherwise the tasks wait till the
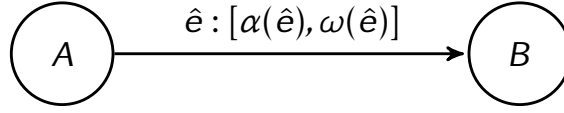
Figure 7.3 – Communicating tasks
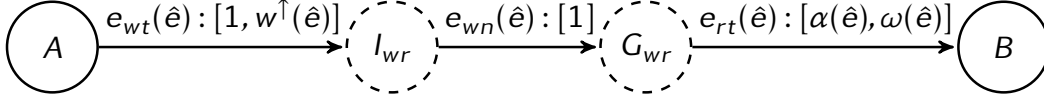
Figure 7.4 – Partition aware communicating tasks

remote status indicates free space.

- **Step 3**: The DMA transfer copies the data from pos_0^w and st_0^w to the reader cluster pos_0^r and st_0^r respectively. When the DMA transfers initiated by tasks A_0 and A_1 are finished, task B_0 which is continuously polling on the status at position 0 (st_0^r) and position 1 (st_1^r), can start its execution owing to the availability of the data.

- **Step 4**: Task F_{st0}, executing in the same cluster as task A_0, is continuously polling on the status of DMA transfer started by task A_0. Once it detects completion of this transfer, it marks the status of the position 0 (st_0^w) as empty and available for re-use. Similarly task F_{st1} marks st_1^w when it is available. Note that step 3 and step 4 may execute concurrently.

- **Step 5**: After task B_0 finishes execution, its marks the local status st_0^r and st_1^r of token as free. Task B_0 initiates a DMA transfer explicitly, which copies st_0^r to rs_0^w and st_1^r to rs_1^w.

After these steps, similarly tasks A_2, A_3 and B_1 can execute on same or different processors, repeating the same sequence of communication and synchronization operations as A_0, A_1 and B_0 respectively. The advantage of using such FIFO is that it decouples the two clusters, meaning that the writer cluster can immediately re-use the FIFO buffers when the DMA transfer has finished transferring the data to the reader cluster without waiting for reader to finish.

7.3 MODELING COMMUNICATION

With the background of inter-cluster FIFO we continue discussion of design flow steps. In the scheduling step we perform a joint scheduling of the computation tasks (the tasks of the application graph) and the communication tasks (the DMA transfers). Our approach to define the scheduling problem is to first describe the *model* that reflects both the computation and the communication in a many-core system and then to explain how this model is *encoded* in terms of constraints. The model is obtained from a series of *graph transformations*, which gradually changes the application graph into a final *schedule graph* which models all the deployment decisions.

In this section we focus on modeling the communication, and assume that we are given a ready *partitioning* solution (from the partitioning step) and the *buffer allocation*, which is part of the combined solution computed at the scheduling step (described later).

7.3.1   *Partition-Aware Graph*

For defining the graph transformations, in particular for adding new actors, it is convenient to introduce notation $v : [d(v), z(v)]$, which represents an actor v with delay $d(v)$ and soft cluster number $z(v)$. Similarly, $e : [\alpha(e), \omega(e), m(e)]$ represents an edge e with parallelization factor $\alpha(e)$, token size $\omega(e)$ and marking $m(e)$. When the latter two parameters are omitted, the default values are considered zero. Note that if ω is zero, the given edge models an *execution order constraint* and does not have a memory buffer for communication.

We assume to have a ready *partitioning solution* $z(v)$, calculated earlier in the design flow. In order to model the communication delay, we need to introduce additional actors in the split-join graph, representing the DMA transfers.

Recall from Equation 3.1 that a DMA transfer consists of two phases: DMA transfer initialization and network communication. We model these two phases of DMA transfer by two actors : I_{wr} and G_{wr}. Figure 7.3 shows an application graph channel and Figure 7.4 shows its partition-aware graph for the case where actors A and B are assigned to different soft clusters. In this case the channel (A, B) is split into writer and reader sub-buffers. The FIFO buffers that are allocated at the writer and reader side are modeled by e_{wt} and e_{rt} respectively. The working of inter-cluster FIFO was explained in detail in the previous section.

Edge $e_{wt}(\hat{e})$ models the writer sub-buffer; $\alpha = 1$ indicating that one writer instance is followed by exactly one data transfer (where ω and $w^{\uparrow}$ represents the data size (in bytes) of one token and the total data size of tokens produced by one writer instance respectively. Refer to Section 2.3.), whereas $\omega(e_{wt}) = w^{\uparrow}(\hat{e})$ indicates that the $\alpha^{\uparrow}$ tokens produced by an instance of the writer actor is encapsulated into one token. Edge $e_{wn}(\hat{e})$ reflects the sequential order between the two DMA phases. Channel $e_{rt}(\hat{e})$ corresponds to the reader sub-buffer. It has the same parameters as the original edge $\hat{e}$.

Let $E^M_{\Delta Z}$ denote the set of inter-cluster channels crossing the partition boundaries and $E^M_{\overline{\Delta Z}}$ its complement. Formally,

$$E^M_{\Delta Z} \quad = \quad \{(v, v') \in E^M \mid z(v) \neq z(v')\}$$

$$E^M_{\overline{\Delta Z}} \quad = \quad E^M \setminus E^M_{\Delta Z}$$

DEFINITION 14 (Partition-Aware Graph) – *A partition-aware graph $S^P = (V^P, E^P, d, \alpha, \omega)$ is a split-join graph obtained by replacement of application graph edges $E^M_{\Delta Z}$ by a sub-graph with new actors and edges as defined below.*

The delay for the newly added actors is the DMA initialization time I and the network sending time $g \cdot w^{\uparrow}(\hat{e})$.

$$V^P = V^M \cup \{I_{wr}(\hat{e}), \ G_{wr}(\hat{e}) \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

Actors	Parameters	Predecessor	Successor
$I_{wr}(\hat{e})$	$[I, z(v)]$	v	$G_{wr}(\hat{e})$
$G_{wr}(\hat{e})$	$[g \cdot w^{\uparrow}(\hat{e}), z(v)]$	$I_{wr}(\hat{e})$	v'

The edges of partition-aware are given by-

$$E^P = E^M_{\overline{\Delta Z}} \cup \{e_{wt}(\hat{e}), \ e_{wn}(\hat{e}), \ e_{rt}(\hat{e}) \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

Edge	Parameters	Edge-writer	Edge-reader
$e_{wt}(\hat{e})$	$[1, w^{\uparrow}(\hat{e})]$	v	$I_{wr}(\hat{e})$
$e_{wn}(\hat{e})$	$[1]$	$I_{wr}(\hat{e})$	$G_{wr}(\hat{e})$
$e_{rt}(\hat{e})$	$[\alpha(\hat{e}), \omega(\hat{e})]$	$G_{wr}(\hat{e})$	v'
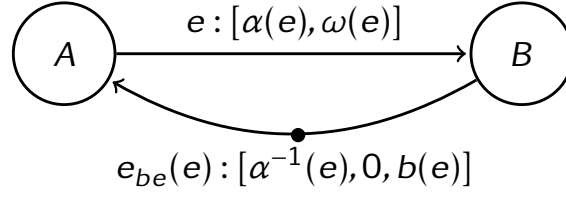
Figure 7.5 – Buffer aware graph model for a channel without DMA
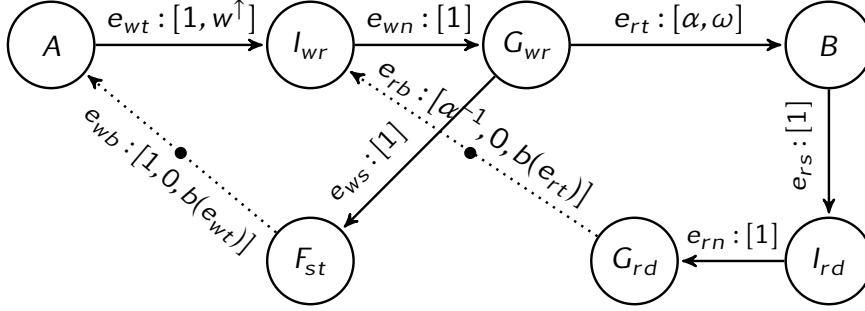
Figure 7.6 – Buffer aware graph model for a channel with DMA

The partition-aware graph implicitly models the application execution on the platform with unbounded resources, such as buffer memory, cores and DMA channels. The remaining graph transformations model the synchronization on bounded resources.

7.3.2 *Buffer Aware Graph*

The next transformation after obtaining a partition-aware graph has for the purpose to model the bounded buffer capacity allocated to the channels. Note that we do not allocate buffers for the channels e_{wn}, as they model the ordering of the actors. Let $E^{P'}$ denote the the subset of all edges in E^P except for e_{wn} edges.

DEFINITION 15 (Buffer Allocation) – $b : E^{P'} \rightarrow \mathbb{N}_+$ *defines the bounded capacity assigned to the channels of the partition-aware graph. It is part of the solution of the combined scheduling and buffer allocation problem.*

DEFINITION 16 (Buffer Aware Graph) – *is a cyclic marked split-join graph $\Sigma^B = (V^B, E^B, d, \alpha, \omega, m)$ obtained from the partition-aware graph by adding marked channels that model the allocated buffer capacity and new actors that model the DMA polling and flow control.*

In this graph, for each channel $\hat{e} \in E^{P'}$ in we add a new *backward channel* – in the opposite direction, marked with the buffer allocation $b(\hat{e})$, signaling the free space availability in the channel. If $\hat{e}$ is an intra-cluster channel then the backward channel is the inversion of $\hat{e}$, with inversely proportional parallelization factor, see Figure 7.5.

Recall that the marking represents the initial number of tokens in the channel. In the backward channel the marking $b(\hat{e})$ indicates the free space that is available initially i.e. the total buffer space. At every execution, the writer takes $\alpha^\uparrow$ tokens of space and produces $\alpha^\uparrow$ tokens of data, and the reader takes $\alpha^\downarrow$ tokens of data and produces $\alpha^\downarrow$ tokens of space.
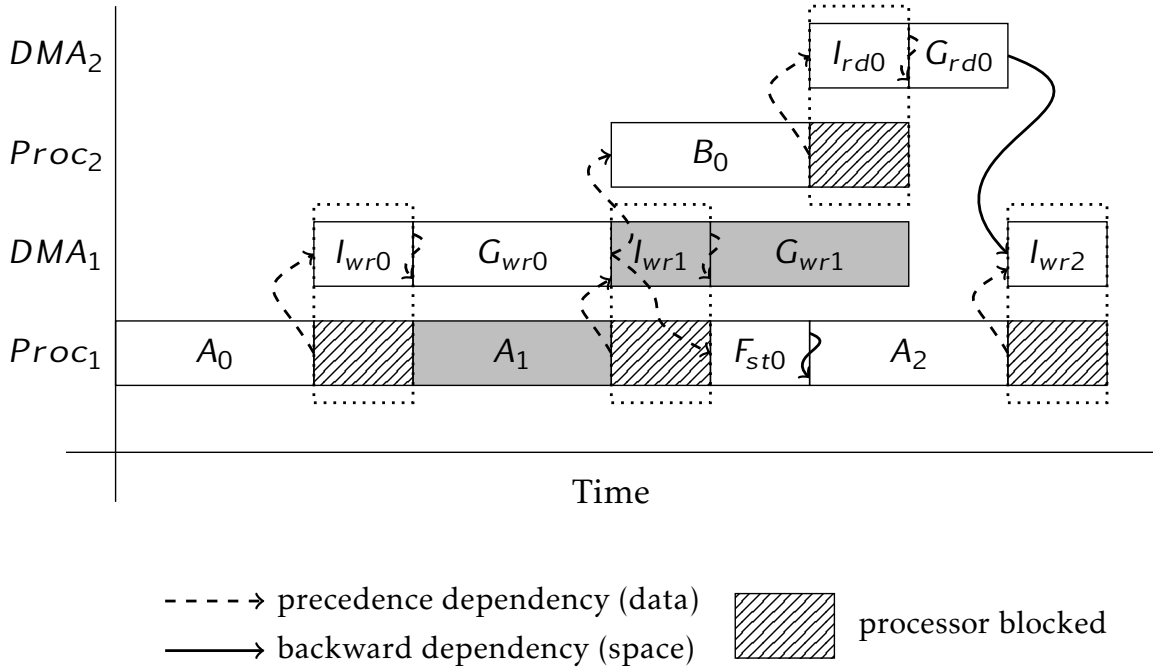
Figure 7.7 – Double buffering example schedule

If the channel is an inter-cluster channel, involving DMA, then we model the free space of the writer sub-buffer, e_{wt} and the reader sub-buffer, e_{rt} by separate backward channels. To model the communication we also add additional actors, see Figure 7.6.

Consider the writer sub-buffer e_{wt}. The direct reader of this sub-buffer is the DMA transfer actor. The space in this sub-buffer becomes available when the DMA transfer is complete. Detecting the transfer completion takes non-negligible processor time denoted by χ, modeled by a new actor F_{st}. This actor produces the space tokens on the backward channel of e_{wt}. The reader sub-buffer e_{rt} is written by the DMA transfer actor, which copies the data from e_{wt} to e_{rt}. Recall that for flow control purposes, the status of the tokens of the reader sub-buffer have to be communicated back to the writer. The DMA transfer required for the flow control is modeled by I_{rd} and G_{rd}, and it is actually the flow control itself that is represented by the backward channel of e_{rt}. The number of tokens on this channel represents the number of empty rs_i^w status records (see Figure 7.2) which indicate the empty (i.e. free) state of the token positions.

Let us consider an example to illustrate the working of our model for inter-cluster FIFO. We assume that two actors A and B are connected with a neutral channel ($\alpha = 1$) and assigned to different clusters. Suppose that we use well-known double-buffering approach, such that $b(e_{wt}(A,B)) = b(e_{rt}(A,B)) = 2$ tokens. Initially, the available space in both buffers is 2. Let us simulate the execution of the model in Figure 7.7 unfolded for three actor instances: A_0, A_1, A_2.

- A_0 executes and consumes one space token.
- At the end, A_0 triggers an initialization of DMA transfer, modeled by I_{wr0}
- Data is sent to the network, which is represented by G_{wr0}. In parallel, A_1 picks the second space token and triggers another DMA transfer: I_{wr1} and G_{wr1}.
- When G_{wr0} finishes, B_0 receives the input data tokens. At the end it releases the space occupied by these tokens and triggers a DMA transfer to update the writer accordingly.

This transfer is modeled by I_{rd0} and G_{rd0}.

- Before A_2 can execute, the task F_{st0} must finish, which releases the necessary free space tokens in the backward channel.

This concludes a complete flow-control cycle between the writer and reader.

We formalize addition of the new actors and channels as:

$$V^B = V^P \cup \{F_{st}(\hat{e}),\ I_{rd}(\hat{e}),\ G_{rd}(\hat{e}) \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

Actors	Parameters	Predecessor	Successor
$F_{st}(\hat{e})$	$[\chi, z(v)]$	$G_{wr}(\hat{e})$	v
$I_{rd}(\hat{e})$	$[l, z(v')]$	v'	$G_{rd}(\hat{e})$
$G_{rd}(\hat{e})$	$[\alpha^{\downarrow}(\hat{e}) \cdot \omega_0 \cdot g, z(v')]$	$I_{rd}(\hat{e})$	$I_{wr}(\hat{e})$

The edges of partition-aware are given by- The delay of network sending node $\alpha^{\downarrow} \cdot \omega_0 \cdot g$ corresponds to the delay of sending ω_0 bytes of status record for all $\alpha^{\downarrow}$ tokens read in the channel, where ω_0 depends on the implementation.

$$E^B = E^P \cup \{e_{ws}(\hat{e}), e_{wb}(\hat{e}), e_{rs}(\hat{e}), e_{rn}(\hat{e}), e_{rb}(\hat{e}), e_{be}(\hat{e}) \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}\ \cup$$

$$\{e_{be}(\hat{e}) : [\alpha^{-1}(\hat{e}), 0, b(\hat{e})] \mid \hat{e} = (v, v') \in E^M_{\Delta Z}\}$$

Edge	Parameters	Edge-writer	Edge-reader
$e_{ws}(\hat{e})$	$[1]$	$I_{wr}(\hat{e})$	$F_{st}(\hat{e})$
$e_{wb}(\hat{e})$	$[1, 0, b(e_{wt}(\hat{e}))]$	$F_{st}(\hat{e})$	v
$e_{rs}(\hat{e})$	$[1]$	v'	$I_{rd}(\hat{e})$
$e_{rn}(\hat{e})$	$[1]$	$I_{rd}(\hat{e})$	$G_{rd}(\hat{e})$
$e_{rb}(\hat{e})$	$[\alpha^{-1}(\hat{e}), 0, b(e_{rt}(\hat{e}))]$	$G_{rd}(\hat{e})$	$I_{wr}(\hat{e})$

The important point to remember is that the buffer allocation b for channels is decided by the SMT solver and the corresponding solver constraints are described in subsection 7.4.2.

7.3.3 Communication Aware Graph

In our implementation, the compute core remains busy until the completion of the DMA initialization tasks (I_{wr} or I_{rd}), started after the completion of the corresponding computation actor (writer or reader). Moreover, if the computation actor instance starts multiple DMA transfers (as a writer/reader of multiple channels), then none of these DMA transfers can start until the initialization of the previous transfer has finished. This restriction can be modeled by adding extra edges between the actors modeling the DMA initialization phase, to enforce a sequential execution order in the schedule.

DEFINITION 17 (Communication Aware Graph) – *Communication Aware Graph $S^K = (V^K, E^K, d, \alpha, \omega, m)$ is obtained from the buffer aware graph S^B by adding the edges to model the ordering of the DMA transfers with regard of their transfer initialization phase.*

The set of actors of the communication aware graph is same as in the buffer aware graph $V^K = V^B$. To define the new edges, we first introduce function $I(v)$, representing an ordered set of DMA transfer initialization actors (I_{wr} or I_{rd}) connected to the output of given actor v, i.e.

$$I(v) = \{v' \mid (v, v') \in E^B \wedge \exists\ \hat{e} \in E^M : v' = I_{wr}(\hat{e}) \vee v' = I_{rd}(\hat{e})\}$$

The ordering $(I_1, I_2, \ldots \mid I_k \in I(v))$ is selected arbitrarily, but it must be the same in the model and the implementation. Then, we have:

$$E^K = E^B \cup \{(I_i, I_{i+1}) : [1] \mid I_i, I_{i+1} \in I(v),\ v \in V^B\}$$

The final transformation to model the communication is the derivation of the task graph $T^K = (U^K, \mathcal{E}^K, \delta, \omega)$ from the communication aware split-join graph S^K. This is done according to Definition 5, of deriving the task graph from a split-join graph. In addition we require that the task graph inherit the partitioning from the split-join graph: $\forall v_h \in U.z(v_h) = z(v)$. This derived task graph is called *Preliminary schedule graph*.

Having shown the model for communication, we proceed to the scheduling model in the next section.

7.4   SCHEDULING

The schedule graph represents a final solution which can be deployed on the platform. It consists of processor allocation and ordering scheme for tasks, buffer sizes for different channels, and the start times of the tasks. It is described below.

7.4.1   *Schedule Graph*

The schedule graph $T^S = (U^S, \mathcal{E}^S, \delta, \omega)$ is obtained from preliminary schedule graph T^K by adding the *mutual exclusion edges*, according to the given *schedule* s and *intra-cluster mapping* μ, where:

- $\mu : U^S \to \mathbb{N}_{\geq 0}$ maps every task to a core (for computation tasks) or a DMA channel (for transfer tasks);
- $s : U^S \to \mathbb{R}_+$ associates each task with a start time.

Like buffer allocation, b, the schedule and the mapping should be computed by the solver during the scheduling step of the design flow, as discussed later.

Let us recall and introduce some notations.

- $U^S = U^S_T \cup U^S_C$ is the task partitioning into:
  ◇ *transfer tasks* U^S_T, derived from DMA transfer actors: I_{wr}, G_{wr} I_{rd}, and G_{rd}.
  ◇ *computation tasks* U^S_C
- $\mathcal{I}(u)$ are the transfer tasks connected to the output of task u, by analogy to $I(v)$ of the computation-aware graph.
- U^S_{C+} is the set of computation tasks connected to DMA transfers: $U^S_{C+} = \{u \in U^S_C \mid \mathcal{I}(u) \neq \emptyset\}$
- $U^S_{C\emptyset}$ is the set of the remaining computation tasks, i.e. those with no DMA transfer tasks at the output.

Note that not all tasks introduced for communication are transfer tasks, as the tasks F_{st} (see Figure 7.6) are computation tasks, as they are executed on the compute cores.

Due to a limited number of the compute cores and DMA channels, multiple tasks are mapped to the same core or channel, and their execution intervals should not overlap in time. This requirement is modeled by adding mutual exclusion edges. The edges of the schedule graph are obtained from:

$$\mathcal{E}^S = \mathcal{E}^K \cup \mathcal{E}^\mu_T \cup \mathcal{E}^\mu_{C\emptyset} \cup \mathcal{E}^\mu_{C+}$$

$\mathcal{E}^\mu_T$ are the mutual exclusion edges for the transfer tasks mapped at the same DMA channel.

$$\mathcal{E}^\mu_T = \{(u, u') : [1] \mid u, u' \in U^S_T, z(u) = z(u'), \mu(u) = \mu(u'), \ s(u') \geq s(u)\}$$

Similarly, we insert an edge for the computation tasks without any DMA task at the output, allocated on the same core.

$$\mathcal{E}^\mu_{C\emptyset} = \{(u, u') : [1] \mid u \in U^S_{C\emptyset}, u' \in U^S_C, \ z(u) = z(u'), \mu(u) = \mu(u'), \ s(u') \geq s(u)\}$$

Finally, when the earlier task u starts some DMA transfers upon its completion, let $\mathcal{I}_{\max}(u)$ represent the last transfer in the ordered set $(\mathcal{I}_1(u), \mathcal{I}_2(u), \dots)$. The compute core becomes

available to a later task u' after the last transfer has finished:

$$\mathcal{E}^\mu_{C+} = \{(\mathcal{I}_{\max}(u), u') : [1] \mid u \in U^S_{C+}, u' \in U^S_C, z(u) = z(u'), \mu(u) = \mu(u'), \; s(u') \geq s(u)\}$$

7.4.2 *Mapping and scheduling using SMT*

In the previous sections, we have described a sequence of rules to derive a schedule graph, which can model the effect of a joint scheduling/buffering solution to be calculated by the SMT solver. The solver constraints for this problem are therefore directly generated from the schedule graph. Below we present the corresponding definition of the optimization problem solved at the scheduling step of the design flow.

Problem Inputs:
- partition-aware graph S^P
- hardware architecture model A
- partitioning solution z

Output:
- (T^S, b, μ, s), where $T^S = (U^S, \mathcal{E}^S, \delta, \omega)$ is the schedule graph obtained from S^P, the partition-aware graph by adding extra actors and edges based on buffering b, mapping μ and scheduling s. According to the calculated buffer allocation b, mapping μ, and schedule s, which are also part of the solution.
    - $\mu : U^S \to \mathbb{N}_{\geq 0}$ maps every task to a processor or DMA channel inside its soft cluster
    - $s : U^S \to \mathbb{R}_{\geq 0}$ associates each task with a start time
    - $b : E^{P'} \to \mathbb{N}_+$ associates each channel in the partition-aware graph to a buffer size measured in tokens.

Costs and bounds:

Symbol	Cost Description	Bounds	
		Lower	Upper
C_B	max. communication memory per cluster	$\min_z \sum\limits_{\substack{(v,v')\in E^{P'} \\ : z(v')=z}} w^\uparrow_{v,v'} + w^\downarrow_{v,v'} - gcd(w^\uparrow_{v,v'}, w^\downarrow_{v,v'})$	$\max_z \sum\limits_{\substack{(v,v')\in E^{P'} \\ : z(v')=z}} c(v) \cdot w^\uparrow_{v,v'}$
C_L	schedule latency	*longest path delay in task graph derived from S^P*	$\sum\limits_{v\in V^P} d(v) \cdot c(v)$

Constraints:
- **Application and Schedule**: The schedule should respect all the dependencies, including the application dependencies, bounded buffer space, DMA transfer ordering, and mutual exclusion, all represented by edges in the schedule graph:

$$\bigwedge_{(u,u')\in \; \mathcal{E}^S(b,s,\mu)} s(u') \geq s(u) + \delta(u)$$

As we explicitly indicate here, the set of schedule dependencies is a function of the problem solution. However, the SMT solvers require a static set of constraints. Therefore, observing that the set U^S is static we rewrite the constraints as:

$$\bigwedge_{u,u'\in U^S} \epsilon^S(u, u', \mu, s, b) \implies s(u') \geq s(u) + \delta(u)$$

where ϵ^S is a predicate that determines whether $(u, u') \in \mathcal{E}$.

For the mutual exclusion edges, this predicate can be trivially obtained from the definition of sets $\mathcal{E}^\mu$, and, in the case of no DMA (i.e. single cluster) the result is equivalent to Equation 6.1.

- **Communication buffer**: The constraints for modeling the communication buffer are included in the above constraints through the backward edges. Predicate for these edges can be obtained from predicate $\epsilon(v, v', h, h')$ in Definition 5, by substituting the buffer allocation to marking m. However, in practice we do not include these constraints, instead reuse the buffer constraints explained in Section 6.2. For the intra-cluster channels, we use exactly the same constraints as there. In the case of inter-cluster channels we take into account that for the writer sub-buffer the free space is produced by F_{st} actor. Therefore, it is this actor, instead of the direct reader of the channel, I_{wr}, who plays the reader role in the buffer constraints for the edge that models the writer sub-buffer. Similarly for reader sub-buffer, the data is produced by G_{wr} actor and free space is produced by G_{rd} actor instead of the intra-cluster channel reader..
- **Resource constraints**: These constraints express bounded number of DMA channels and cores per cluster.

$$\bigwedge_{u \in U_T^S} \mu(u) < |D| \quad \wedge \quad \bigwedge_{u \in U_C^S} \mu(u) < |M|$$

  Bounded buffer memory per cluster:

$$\bigwedge_{a \in \mathcal{Z}} \quad \sum_{e=(v,v') \in E^{P'} \,:\, z(v')=a} b(e) \cdot \omega(e) \ \leq\ C_B$$

- **Latency constraint**: The schedule must observe the latency cost constraint.

$$\bigwedge_{u \in U^S} s(u) + \delta(u) \leq C_L$$

- **Extra constraints**: In our implementation we require the writer and reader sub-buffers to have equal buffer memory:

$$\bigwedge_{\hat{e} \in E_{\Delta Z}^M} b(e_{wt}(\hat{e})) \cdot \omega(e_{wt}(\hat{e})) = b(e_{rt}(\hat{e})) \cdot \omega(e_{rt}(\hat{e}))$$

  We also require that the initialization and network phases of DMA transfers follow immediately one after the other and on the same DMA channel:

$$\bigwedge_{\hat{e}=(v,v') \in E_{\Delta Z}^M} \ \bigwedge_{0 \leq h < c(v)} s(I_{wr\,h}(\hat{e})) + I = s(G_{wr\,h}(\hat{e}))) \ \wedge\ \mu(I_{wr\,h}(\hat{e})) = \mu(G_{wr\,h}(\hat{e})))$$

$$\bigwedge_{\hat{e}=(v,v') \in E_{\Delta Z}^M} \ \bigwedge_{0 \leq h' < c(v')} s(I_{rd\,h'}(\hat{e})) + I = s(G_{rd\,h'}(\hat{e}))) \ \wedge\ \mu(I_{rd\,h'}(\hat{e})) = \mu(G_{rd\,h'}(\hat{e})))$$

  where h and h' correspond to the index of task instance of channel writer and channel reader.
- **Symmetry Constraints**: Last but not the least, we add *task and processor symmetry breaking* constraints (explained in Section 6.2), which improve the performance of the constraint solvers.

For feasible costs and tractable problem sizes the solver produces the scheduling problem solutions, which include the mapping of tasks to the compute cores and DMA channels, as well as task start times and channel buffer allocations.

7.5 SCHEDULE IMPROVEMENT

The schedule generated by the solver is not necessarily optimal in terms of latency and processor usage. For example, even if a task is enabled (i.e. it has all input data and output buffer space and the core at its disposal), the solver might schedule it for later execution.

Similarly, even if a processor is idle when the task should start, the solver might allocate a new processor to run a task. This happens because when the exploration algorithm makes queries at the points lying away from the Pareto front, the solver might produce a schedule with under-utilization of resources like processors or time without violating the cost constraints. Such a schedule must be improved in terms of latency and number of processors used, to efficiently utilize the resources. If such improvement constraints are included in our query, it complicates the problem and makes it harder for the solver to produce a solution. Hence this improvement must be done separately.

To summarize, the solutions obtained by the solver are correct but often not tight, in terms of latency and processing resources. In two steps: (1) we *improve the latency* and (2) we *improve the mapping*.

7.5.1 *Improvement of latency*

According to task graph scheduling theory, the schedule constraints imply that each task can be scheduled in $[ASAP(u), ALAP(u)]$ interval (see e.g. [55]), where $ASAP$ and $ALAP$ stand for as soon as possible and as late as possible start time for the given mapping and resource access ordering. They are obtained by longest-delay path algorithms in the schedule graph. The solver typically computes start times $s(u)$ somewhere inside this interval, whereas using $ASAP(u)$ would allow tightening the latency constraint and it would directly correspond to our 'non-lazy' (self-timed) online scheduling policy.

To realize this improvement, upon a "satisfiable" response from the SMT solver query we extract the solver-computed mapping, the task execution order per core and DMA channel and the buffer allocation per edge and calculate the corresponding schedule graph. In the schedule graph, we set all start times according to $s(u) = ASAP(u)$ and update the latency cost C_L attributed to the obtained solution to the completion time of the latest task.

7.5.2 *Processor Optimal Schedule*

When we acquire a non-lazy schedule from the above procedure, we fix the task start-times, and apply the *left edge algorithm* [1] to recompute the mapping $\mu(u)$. As a result, we obtain an improved scheduling solution with a compact schedule and mapping.

The tasks are assigned to processors in non-decreasing start time order. The processor usage is optimized such that processors are assigned an index in increasing order inside a cluster starting from zero. The processor with higher index is assigned only when the processor of lower index has an overlapping task with the current task which is to be assigned.

This optimization of schedule helps the cost-space exploration process, as the loose points in the solution provided by the solver are tightened with the above mentioned process. As tighter solutions are obtained, it reduces number of queries in the exploration.

7.6 EXPERIMENTS

In this section we give an empiric evaluation of the validity of our many-core scheduling approach, using a set of application benchmarks. We run our design flow in order to approximate the Pareto points of feasible schedules. We execute every solution obtained on the real hardware of the MPPA platform and compare its real performance to the one predicted by our scheduling solution. Our application benchmarks consist of the JPEG Image Decoder, and a subset of StreamIt benchmarks [127]. The characteristics of the benchmarks are summarized in Table 7.1. The exploration experiments use version 4.1 of the Z3 Solver [87] running on a

---

1. A classical algorithm used in design automation for this purpose [76].

Table 7.1 – Application benchmark characteristics

BenchMark	#Actors	#Channels	#Tasks	Total Exec. Time (cycles)	Total Comm. Data (bytes)
JPEG Decoder	3	2	25	934288	12384
Beam Former	8	7	53	342816	944
Insertion Sort	6	5	6	40033	320
Merge Sort	12	11	31	102347	704
Radix Sort	13	12	13	85464	768
Dct1	4	3	4	127496	768
Dct2	7	6	21	215525	1536
Dct3	5	4	12	129105	1024
Dct4	7	6	21	183890	1536
Dct5	7	6	21	216079	1536
Dct6	8	7	36	258304	1792
Dct7	8	7	29	218577	1792
Dct8	10	9	38	272514	2304
Dct Coarse	3	2	3	74401	512
Dct Fine	6	5	20	163708	1280
Comparison Count	5	5	20	141397	1280
Matrix multiplication	11	11	79	1087840	10656
Fft	13	12	96	640109	6144

Table 7.2 – Jpeg decoder : Partitioning step solutions

Solution	Allocated soft cluster			Exploration Cost		
	VLD	IQ	Color	C_τ	C_η	$C_\mathcal{Z}$
P_{s0}	0	1	2	424012	12384	3
P_{s1}	0	0	1	758116	2736	2
P_{s2}	0	0	0	934288	0	1
P_{s3}	0	1	1	510276	9648	2

Linux machine with Intel Core i7 processor at 1.73 GHz with 4 GB of memory. We use the JPEG decoder as an example to illustrate the experiments done for each benchmark in detail.

For the partitioning step, the exploration is done in a 3-dimensional cost space: $(C_\tau, C_\eta, C_\mathcal{Z})$ i.e. the maximal workload per soft cluster, communication cost estimate and number of soft clusters. There is a trade-off between these costs, and our grid-based exploration strategy, finds four Pareto points as shown in Table 7.2.

At the placement step, the soft clusters are mapped to neighbor clusters. We perform a binary search in order to find the minimal feasible total communication cost.

In the scheduling step, the exploration is done for each partitioning solution in a 2-dimensional space (C_L, C_B), i.e. latency and maximal buffer size per soft cluster. We plot all four Pareto fronts in Figure 7.8. We observe some Pareto fronts cross because they differ in the number of available soft clusters, which leads to the following effect. On the top-left side of the cost diagram a large buffer memory per soft cluster is allowed. Therefore, even a single

soft cluster solution has enough buffer memory to minimize the latency, and it dominates the solutions with more soft clusters because it does not use DMA transfers. On the bottom-right side, the parallelism available to the solutions with less soft cluster count is restricted by small amount of buffer memory, whereas adding more soft clusters results in a larger total buffer memory, allowing to run more tasks in parallel and get a better latency. Therefore, a larger number of clusters may yield a solution with a smaller latency, even despite the larger DMA cost. The combination of these individual Pareto fronts can be pruned, retaining only the overall Pareto solutions. These solutions will be a mix of multi-cluster and single-cluster ones, showing interesting deployment trade-offs. This proves that it is not a trivial problem to select the number of clusters, and we observe the true multi-criteria nature of the problem.
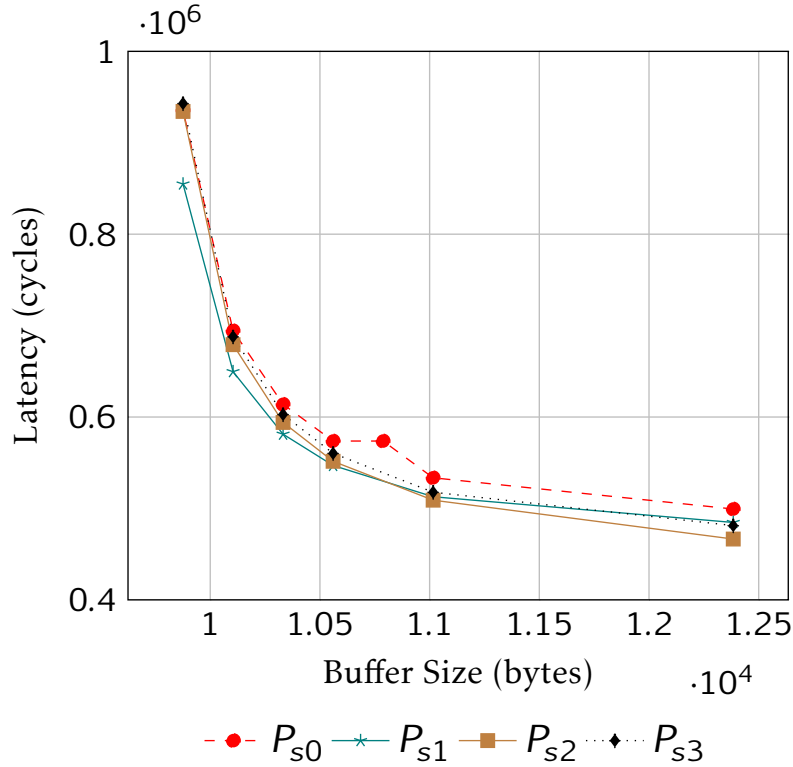
Figure 7.8 – 25 scheduling solutions for 4 partitioning solutions

Figure 7.9 shows the results obtained for the scheduling solutions when executed on the Kalray platform. We plot the minimum and maximum observed latency on the platform as well as the predicted one. The maximum error that was observed in this configuration, as well as for the entire JPEG experiment, was 8%.

Figure 7.10 shows the summary of results obtained for all benchmarks. Firstly, we have plotted the total number of scheduling solutions obtained for an application benchmark, adding up those obtained for different partitioning solutions. Note that this solution number depends on the amount of parallelism available in the application as well as on the structure of the application graph. For example, for an application graph as shown in Figure 2.6 there is no latency-buffer size trade-off irrespective of the parallelism factor α, the buffer size required is α tokens for each channel, the application cannot execute with less and will not improve with more.

Secondly, we plot the the maximum error of the predicted latency vs the one measured on the platform for 100 iterations. The overall maximum error (27%) was observed in the
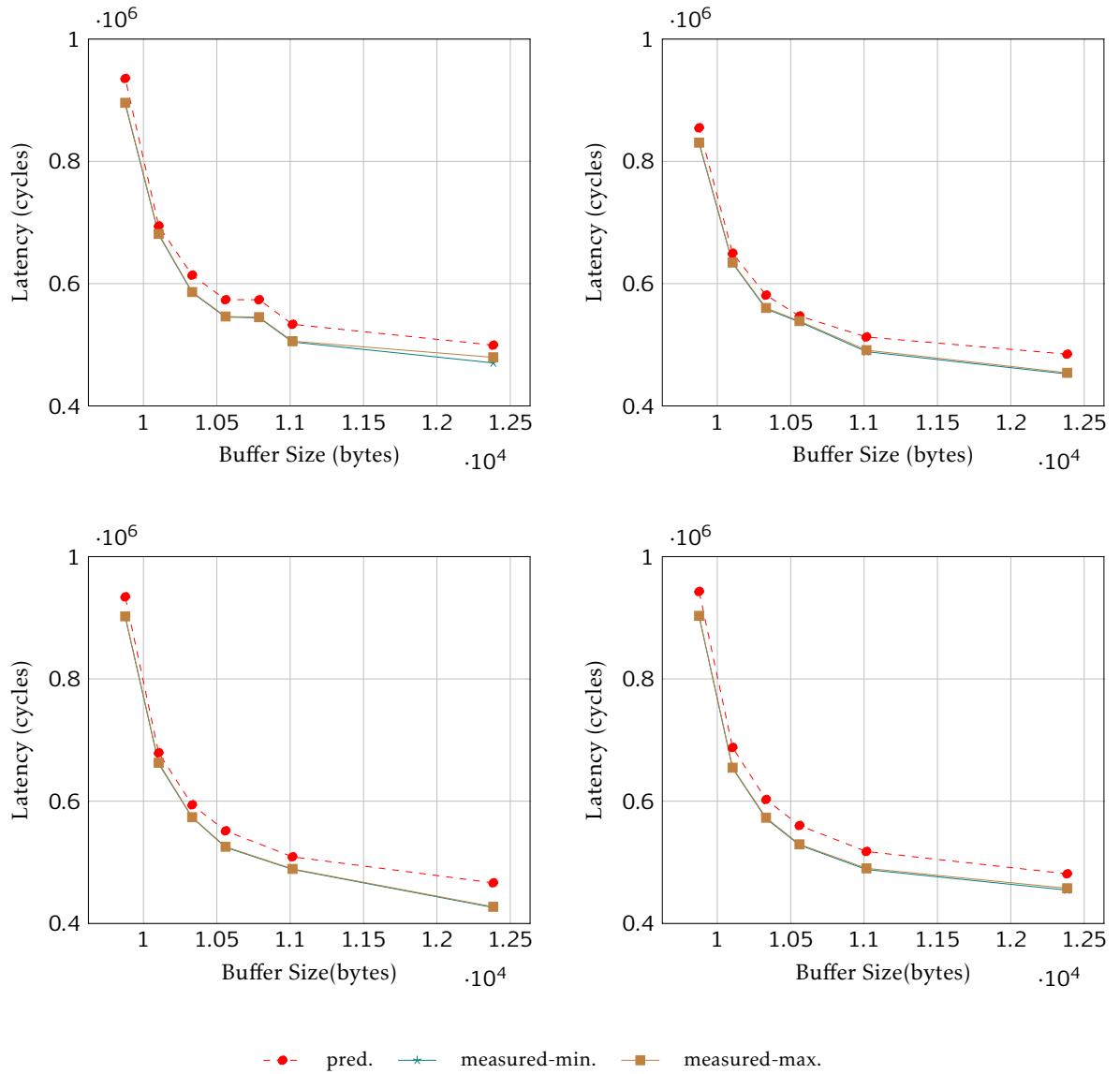
Figure 7.9 – Jpeg decoder solutions measured on Kalray platform

'Comparison Count' benchmark. In the schedule that resulted in this error we observed that there were 8 simultaneous DMA transfers between a pair of clusters, which contributed to network contention and hence a less accurate latency prediction. However, we believe that another factor contributing to inaccuracy is the contention on the shared memory bus inside the clusters. We also believe that the error could be larger if we ran with caches enabled.

7.7 conclusions

In this chapter, we presented a multi-stage approach which was used to schedule split-join application models on the Kalray processor architecture. The major contribution in this work is an accurate modeling of network communication with buffering that supports multiple parallel writers and readers per channel, as well as network DMA transfers and flow control. We validate the framework using a dozen of real applications from the well-known StreamIt

Figure 7.10 – Application benchmarks: summary of results

set of benchmarks for signal-processing applications. For the optimal schedules generated for benchmarks, we performed latency measurements on real many-core hardware. Despite the fact that we ignored the network contention in scheduling, the maximal error of scheduler's timing estimation was only 27% and typically in the range of 10-15%. We 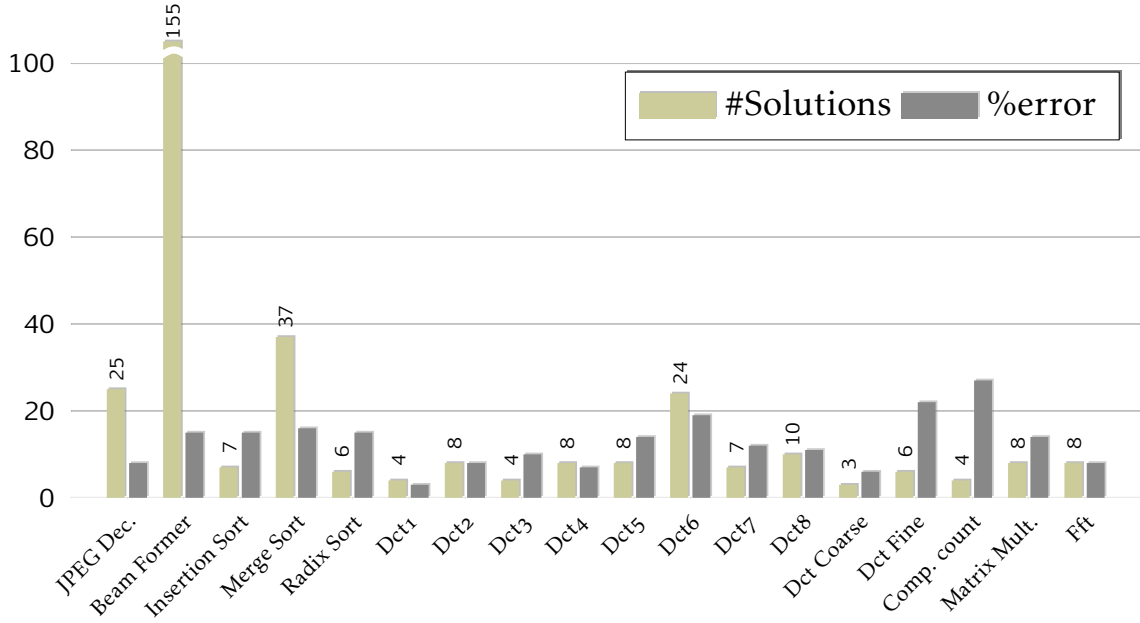obtain this level of precision by exploiting non-cached shared local memory system at the cluster level. The maximal benchmark size we could handle had 96 tasks, where the solver's performance started to saturate.

This chapter borrows some ideas from previous work. Implementation-aware graphs, a formalism similar to our schedule graphs were proposed in [98] and we extend this work by a more realistic modeling which consider processor blocking and verify it on a real platform. A DMA model similar to ours is considered in [130] who also use an SMT solver to compute schedules, but only for uni-processor systems. The work of [135] considers the network routing and pipelined scheduling (which is outside the scope of this thesis) but they do not combine their the network communication model with parallel scheduling in shared-memory clusters.

Data-parallel applications apply uniform computations on large amount of data. Due to large size, the input and output data cannot be placed in relatively small local memory. Thus the data must be processed in small parts. The amount of the data that should be fetched in the local memory has an impact on performance of the applications and is an optimization problem that we discuss in the next chapter.

Optimizing the DMA communication

This chapter introduces the DMA transfer granularity optimization for data parallel computations with regular memory access patterns.
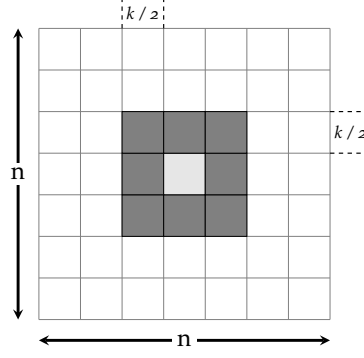
Data parallel applications, like filtering in image processing, apply the same computation to different blocks of data. Such type of application benefit from available data parallelism when executed concurrently on multi-core platforms. The processors connected to main memory with DMA have to explicitly fetch the data into local memory from the main memory, on which the computation should be performed and finally the results written back in main memory. However the choice of the amount of data to be brought to the local memory in each DMA transfer, which is obviously limited by the size of local memory, is a design choice to be made by the programmer.

In this chapter we discuss influence of DMA transfer granularity and access pattern by DMA on performance of data-parallel applications. We start by discussing the data parallelism in applications and how software pipelining helps to achieve the overlap between data transfer and computation. With this software pipelining into consideration, the non-trivial choice of granularity of DMA transfer is described in detail. Further, different mechanisms to exchange the shared data between different processors and its effect on the optimal granularity are discussed. We conclude this chapter with experiments performed on the Cell processor architecture.

8.1 data-parallel applications

There are different types of parallelism available in parallel applications, as discussed in Section 1.2. A class of applications known as *embarrassingly parallel*, performs uniform computation on a large amount of data blocks. In this chapter, we focus on such applications, which have a regular memory access pattern that can be modeled for static optimization. Our application structure consists of computation $Y = f(X)$, where X and Y are *large* input and output arrays respectively. For simplicity, we assume that both have the same dimensions. The computation $f()$ is applied uniformly to every element of this array.

If the input (and the output) array is one-dimensional, then each element of this array can be computed as $Y(i) = f(X(i))$, where i represents the position of an element in the array. However,

Figure 8.1 – Neighborhood pattern of size k

in some applications like filtering in image processing, the input and output data arrays are two dimensional instead of one. Such a computation can be represented as $Y(i_1, i_2) = f(X(i_1, i_2))$, where both the input and output arrays are two-dimensional. In such cases, the position of element in the array must be determined by two indices i_1 and i_2. Applications compute data for more than two dimensions, however we restrict ourselves to two dimensions for simplicity.

Given such a scenario, the computations can be further classified as:

- *independent computations*: This type of computations does not require any surrounding data, but the one which is being processed. $Y(i) = f(X(i))$ and $Y(i_1, i_2) = f(X(i_1, i_2))$ are examples of independent computations.
- *overlapped data computations*: In this case, the computation requires surrounding block of data to produce output. For example, in the case of two dimensional data, $Y(i_1, i_2) = f(V(i_1, i_2))$ is an example of overlapped computation, where $V(i_1, i_2)$ is a $(k+1) \times (k+1)$ matrix [1],

$$V(i_1, i_2) = \left\{ \ X(j_1, j_2) : \begin{array}{c} (i_1 - k/2 \ \leq \ j_1 \leq i_1 + k/2) \\ (i_2 - k/2 \ \leq \ j_2 \leq i_2 + k/2) \end{array} \right\} \tag{8.1}$$

We refer to V as the shared data required for the computation. We assume a symmetric window of size k around the data block as shown in Figure 8.1.

In practice it is possible that algorithms require shared data only from one side around the data block and we consider it later.

8.1.1 *Buffering schemes*

Semantically, a one-dimensional data-parallel algorithm can be specified by the following sequential program.

Algorithm 3 Sequential Data Processing

---
    for i:=1 to n **do**
        Y[i]:=f(X[i])
    end for

However, for the processors with hierarchical memory, the data must be brought from the main memory, into a memory closer to the processor. Typically the entire input and output array does not fit into the local memory and one needs to split the data into parts for processing.

---

1. for convenience k is assumed an even integer.

In such case, the processor will have to prefetch the data (*dma_get*), then process it and write back (*dma_put*) the results to the output, as shown in Algorithm 4. The meaning of e_{in} and e_{out} variables used is that there is an identifier to check completion of dma_get or dma_put.

Algorithm 4 Single Buffering

---

for i:=0 to m-1 **do**
    dma_get (in_buf,X[$i \cdot s + 1$ .. $(i+1) \cdot s$],e_{in});   ▹ fetch in-buffer
    dma_wait (e_{in});
    for j:=1 to s **do**   ▹ compute
        out_buf[j]= f(in_buf[j]);
    end for
    dma_put (out_buf,Y[$i \cdot s + 1$ .. $(i+1) \cdot s$], e_{out});   ▹ write out-buffer
    dma_wait (e_{out});
end for

However if we implement this single buffering algorithm, the processor remains idle when the input and output arrays are being transferred. To improve this situation double buffering is employed, where the input and the output buffer are split into two parts. While the processor performs computation on first part of the buffer, the transfer is carried on the other part and vice-versa. This is done to overlap the computation and communication.

Algorithm 5 Double Buffering

---

d:=0;
dma_get (in_buf[0],X[1..s],e_{in}[0]);   ▹ first read
for i:=0 to m-1 **do**
    if $i < m - 1$ **then**
        dma_get (in_buf[d⊕1],X[$(i+1) \cdot s + 1..(i+2) \cdot s$],$e_{in}$[d⊕1]);   ▹ fetch next
    end if
    dma_wait (e_{in}[d]);   ▹ wait for current input
    for j:=1 to s **do**   ▹ process current
        out_buf[d][j]= f(in_buf[d][j]);
    end for
    if $i > 0$ **then**
        dma_wait (e_{out}[d⊕ 1]);   ▹ wait for previous write
    end if
    dma_put (out_buf[d], Y[i]$\cdot s + 1..(i+1) \cdot s$], e_{out}[d]);   ▹ write current
    d:=(d⊕ 1);   ▹ toggle buffer
end for
dma_wait (e_{out}[d⊕1]);   ▹ wait for last write to complete

Algorithm 5 defines a software pipeline with 3 stages: input transfer, computation and output transfer. In this algorithm, the processor issues a fetch for a block of data and proceeds to the computation of previously fetched data. After finishing the computation it issues a write of the computed results and completes the iteration. Fetching the first block and waiting for the write of the last block are respectively the beginning (*prologue*) and the end of the pipeline (*epilogue*).
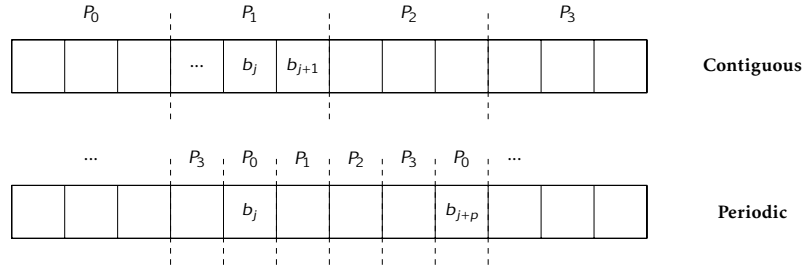
Figure 8.2 – Contiguous vs periodic allocation of data

8.1.2 *Data distribution, block Shape and Granularity*

The input data to be processed must be distributed among the available processors for parallel processing. Depending on the number of dimensions, the data can be split in various ways. We discuss the splitting and distribution of one-dimensional and two-dimensional data below.

8.1.2.1 One-dimensional data

In the case of one dimensional data without any shared data computations, it is simple to cut the input array uniformly into parts equal to the number of processors and allocate one part per processor. This allocation can be done in two types : (a) *contiguous* : where the adjacent blocks j and $j + 1$ are allocated to the same processor or (b) *periodic* where the blocks j and $j + p$ are allocated to the same processor, where p is the number of processors. Figure 8.2, shows both schemes for four processors P_0 to P_3. These solutions are equivalent in the amount of distribution of data.

In the case of one dimensional data, when there is need for shared data for computation, *periodic* allocation can be beneficial as the processors can exchange the data amongst themselves. In the case of contiguous allocation the processor has to fetch the computation data as well as shared data into its local memory to perform the processing, thus increasing the granularity of data transfer. Contiguous allocation is favorable when processor stores the shared data locally at the end of current iteration and reuses it for the next. We discuss this further in Section 8.3.

8.1.2.2 Two-dimensional data

Two dimensional data can be divided equally into various shapes as illustrated in Figure 8.3. Again such solutions are equivalent in amount of data that is being transferred from the main memory. However, depending on the shape, the data maybe contiguously transferred in one DMA transfer or by using expensive strided DMA (additional delay due to stride is described in Section 3.7).

In the case of shared data for two-dimensional data, the geometry of the data partitioning scheme determines the amount of data that will be shared between the processors. For example, as shown in Figure 8.4 we see that for completely horizontal or vertical blocks of the same area, the amount of shared data, is greater than for square shaped blocks.

8.2 OPTIMAL GRANULARITY FOR DATA TRANSFERS

For one-dimensional or two-dimensional data, finding the optimal granularity for the data transfer, that minimizes the total execution time of the application is a non-trivial problem. In order to derive the optimal granularity, the performance of the pipelined execution of the
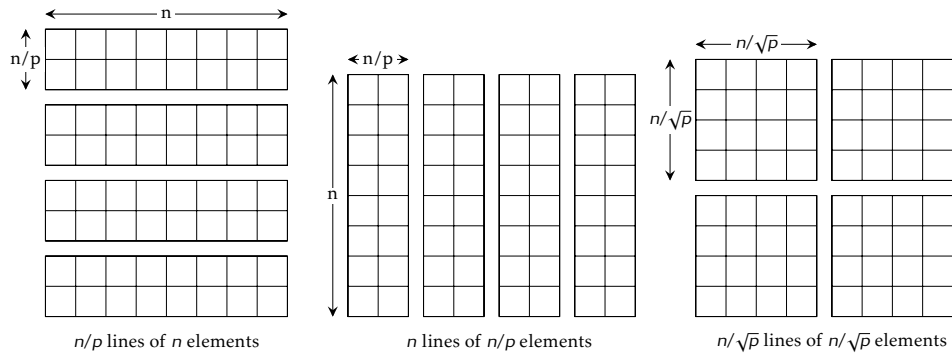
Figure 8.3 – Distribution of 2D data of same size, but different shapes
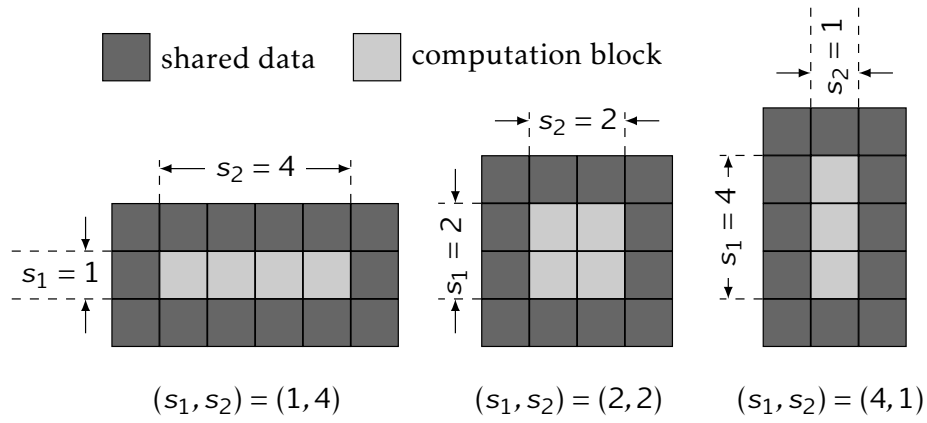

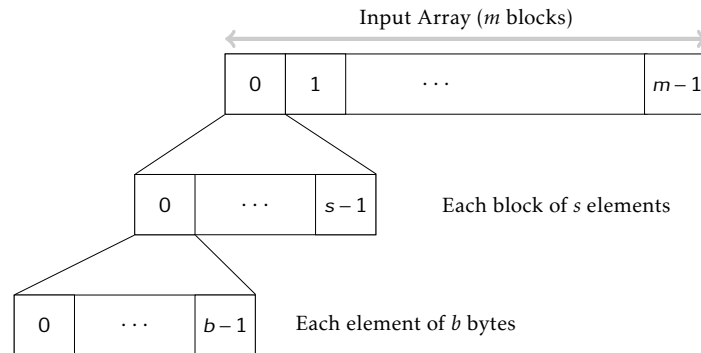Figure 8.4 – Influence of block shape on the amount of shared data

Figure 8.5 – Decomposition of one dimensional input array

double buffering algorithm must be analyzed. In this section we first analyze the performance of this algorithm for different size and shapes for independent computations, without any shared data.

For one-dimensional data we assume that the application performs a computation on a block of data s consisting of elements, where each element is of b bytes of data. Figure 8.5 shows an input array, composed of m blocks. We assume that the computation time to perform f on one element is ω time units. Since we already have assumed data independent computation time, if the deviation in this time is not significant, then we can assume an average value of ω which gives a good approximation of reality. Therefore the computation time of s elements can be given by $C(s) = \omega \cdot s$.

For two-dimensional data, we assume that the computation time depends only on the area of the rectangle is given by:

$$C(s_1, s_2) = \omega \cdot s_1 \cdot s_2$$

In practice, $C(s_1, s_2)$ has a component which depends on number of lines s_1, due to expensive loop overheads for inner and the outer loop on certain architectures. We assume that this overhead is negligible and discuss it further in the experiments section.

For transfer time, we use the formula in Equation 3.1 for a non-strided DMA:

$$T(s) = l + g \cdot s \cdot b$$

and from Equation 3.2 for strided DMA:

$$T(s_1, s_2) = l_0 + l_1 \cdot s_1 + g \cdot (s_1 \cdot s_2) \cdot b$$

As discussed before, the software pipeline has 3 stages, namely, input data transfer, computation and output data transfer. Depending on the ratio between computation time for one block C, and transfer time per block T for input and output blocks, the execution pipeline can be classified into two types :

- *Computation Regime*: In this case, the computation time dominates the transfer time $C > T$, shown in Figure 8.6(a).
- *Transfer Regime*: Here the transfer time dominates the computation time $C \leq T$, shown in Figure 8.6(b).

As mentioned earlier, the execution of the pipeline stage has an epilogue and prologue stage which equals to the transfer time of a block. Now, we can observe in Figure 8.6(b) a transfer regime, in which the total execution time of the pipeline is dominated by the transfer time. Whereas in computation regime shown in Figure 8.6(a), the execution time is dominated by the computation time. We can see that apart from the computation time, the prologue and the epilogue also contribute to the execution time. Let m be number of blocks which are grouped for the the pipeline stage. The total execution time ν, of the pipeline can easily be estimated with the help of transfer time and computation time and given by -

$$\nu = \begin{cases} m \cdot C + 2T & \text{in the computation regime} \\ (m+1) \cdot T + C & \text{in the transfer regime} \end{cases} \tag{8.2}$$

The ratio between computation time of a block C and its transfer time T is not fixed but varies with the block size and shape. We can therefore control it to some extent in order to optimize performance, but which relation is preferred? The answer depends on which resource is more stressed by the application, which is either computation or communication, and this characterized by the parameter ψ,

$$\psi = \omega - g \cdot b \tag{8.3}$$

(a) Computation Regime

(b) Transfer Regime
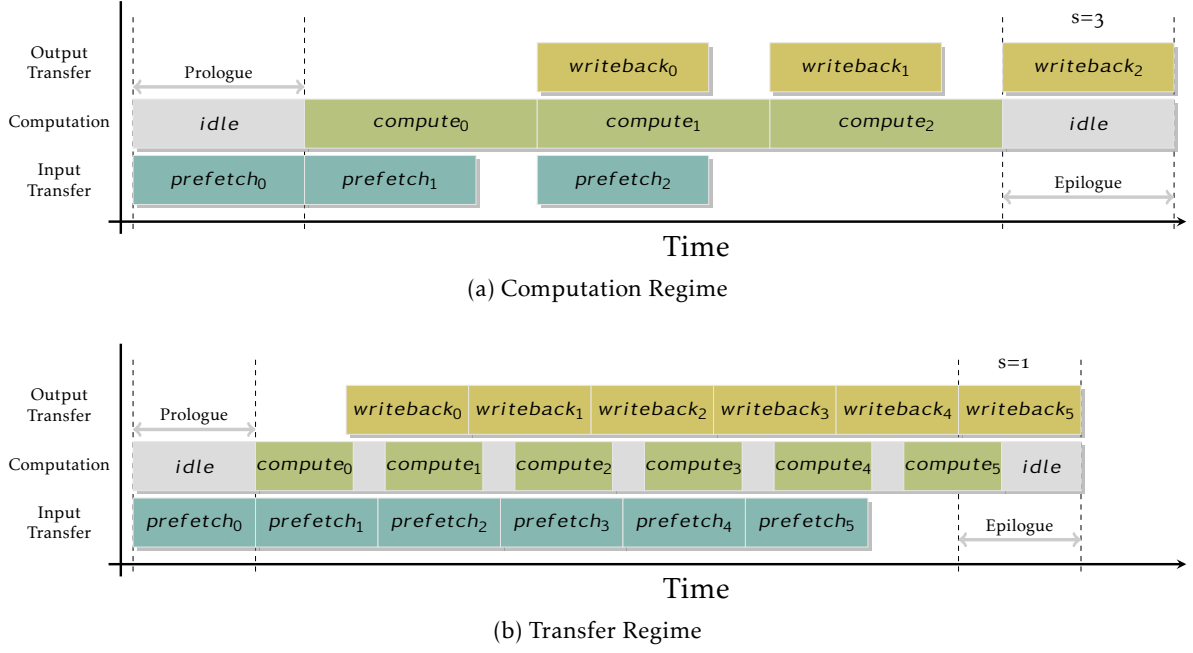
Figure 8.6 – Pipelined execution using double buffering on one processor

where g is the cost per byte for DMA transfer (refer to Section 3.7). In fact ψ gives the difference in cost per element for computation and communication.

If $\psi \leq 0$, then it signifies that the pipeline would always execute in the transfer regime where transfer time is dominating the computation time. The optimal transfer granularity for this case is easy to calculate and should be equal to the maximum size limited by the local store. This will improve the performance of the application by reducing the idle time between computations. Similarly if $\psi > 0$, then the pipeline is in computation regime, and we consider this case for further analysis.

As we see in Figure 8.7(a) if s is small we are still in the transfer regime, because the transfer time $T(s)$ is dominated by DMA initialization time. However, when $\psi > 0$ the computation grows faster for larger s than the transfer time and for a certain block size s^* we enter the computation regime. To see the impact of s on the total pipeline time (Figure 8.7(b)) we have to substitute $m = \left\lceil \frac{n}{s} \right\rceil$ into Equation 8.2, where n is the size of the array. For small enough s, where the transfer regime is dominated by DMA initialization time, the total pipeline time mainly depends on m giving the number of DMA initialization, and the latter decreases inverse proportional to s.

However for larger s the component proportional to s starts to dominate. In the computation regime the total computation time $m \cdot C$ can be seen as constant as the total computation time for n elements is roughly $\omega \cdot n$. In this regime, prologue and epilogue times $2T$ result in growth of pipeline time with s. Because in computation regime the pipeline time thus always grows with s, our strategy is to make s small enough so that we get into the transfer regime. In this regime we have to find s yielding minimal $T(s)$.

We also have a constraint that the block granularity is between one element and the full input array size, provided that its size does not exceed the maximum local buffer size B imposed by the local store limited capacity. This leads to following constrained optimization problems for one or two-dimensional data:

(a) $C(s)$ and $T(s)$ per block

(b) Pipeline execution time
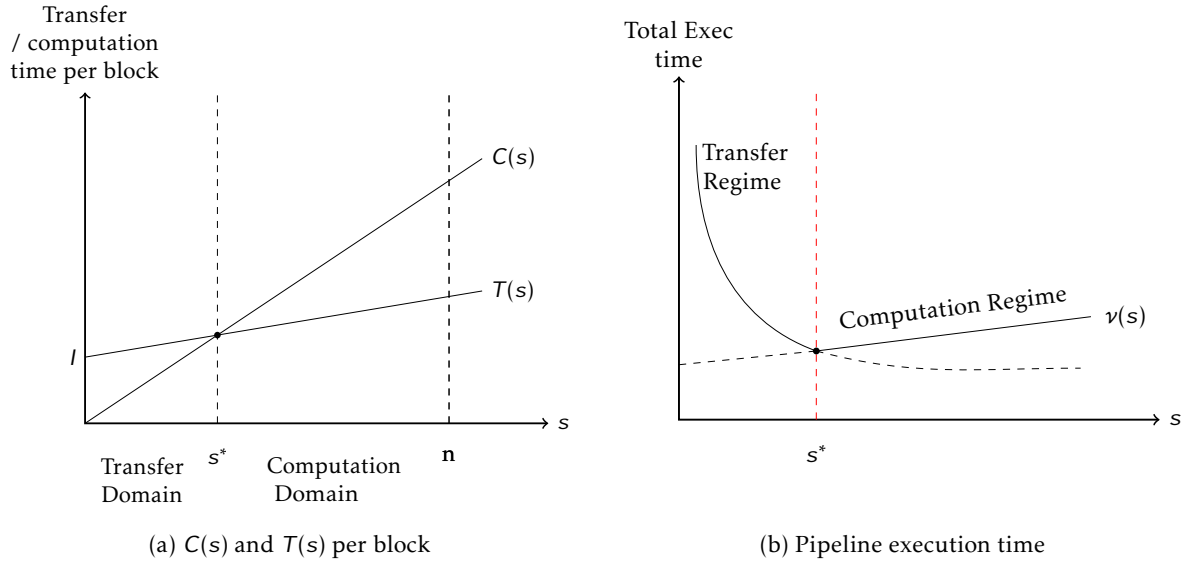
Figure 8.7 – Regimes depending on the block size

$$\min\ T(s)\ \text{s.t.}$$
$$T(s) \leq C(s)$$
$$s \in [1..n]$$
$$b \cdot s \leq B$$

$$\min\ T(s_1, s_2)\ \text{s.t.}$$
$$T(s_1, s_2) \leq C(s_1, s_2)$$
$$s_1, s_2 \in [1..n_1] \times [1..n_2]$$
$$b \cdot s_1 \cdot s_2 \leq B$$

For this problem, we analyze the case with single and multiple processors for one-dimensional and two-dimensional data separately.

8.2.1 *Single Processor*

8.2.1.1 One-dimensional data

We plot the computation time and the transfer time in Figure 8.7(a). The ratio between the computation time and the DMA transfer time of the block splits the domain of solutions into *computation domain* and *transfer domain*.

The intersection of functions $T(s)$ and $C(s)$ for one-dimensional data blocks where the computation domain corresponds to the interval $[s^*, n]$, and the overall execution switches from a transfer regime to a computation regime for granularity s^*, as illustrated in Figure 8.7(b).

The total execution time for a large n is then approximated by:

$$v(s) = \begin{cases} 2 \cdot T(s) + m \cdot C(s) \simeq 2 \cdot g \cdot s \cdot b + n \cdot \omega & for\ s > s^* \\ \left(\left\lceil \frac{n}{s} \right\rceil + 1 \right) \cdot T(s) + C(s) \simeq (n \cdot l)/s + (n \cdot g \cdot b) & for\ s \leq s^* \end{cases} \tag{8.4}$$

Recall that according to our strategy we search for optimal values of s by minimizing $T(s)$ for $s \leq s^*$. From Figure 8.7 we see that $v(s)$ is a decreasing function for $s \leq s^*$. Therefore the optimal value is s^* given by:

$$s^* = l/(\omega - gb) \tag{8.5}$$
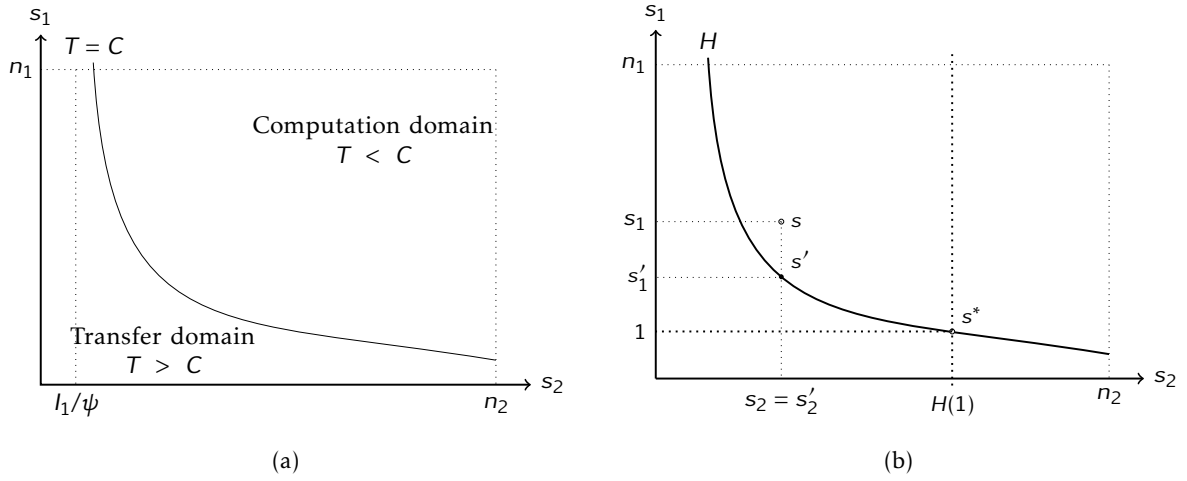
Figure 8.8 – Rectangular blocks: (a) computation and transfer domains, (b) optimal granularity candidates and optimal granularity.
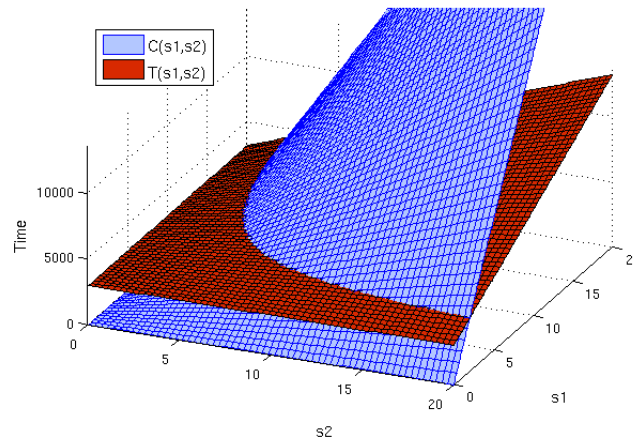
Figure 8.9 – The dependence of computation C and transfer T on the granularity (s_1, s_2).

If for any granularity s the execution is always in the computation regime (due to high ω), the optimal unit of transfer is a block, that is $s^* = 1$, which guarantees minimal prologue and epilogue.

8.2.1.2 Two-dimensional data

For two-dimensional data, we should find an optimal rectangular block which should give minimal execution time for the pipeline. For rectangular blocks, the dependence of $T(s_1, s_2)$ and $C(s_1, s_2)$ on their arguments is illustrated in Figure 8.9 (assuming $\psi > l_1$). Similarly, the intersection of these two surfaces separates the domain of (s_1, s_2) into two sub-domains, the computation domain where $T < C$ and the transfer domain where $T > C$, see Figure 8.8.

Comparing the transfer time of the different shapes is not trivial since the computation domain forms a partially ordered set where possibly two different shapes s and s' are such that $s_1 < s_1'$ and $s_2 > s_2'$. This is due to the fact that if these shapes have the same area, then the
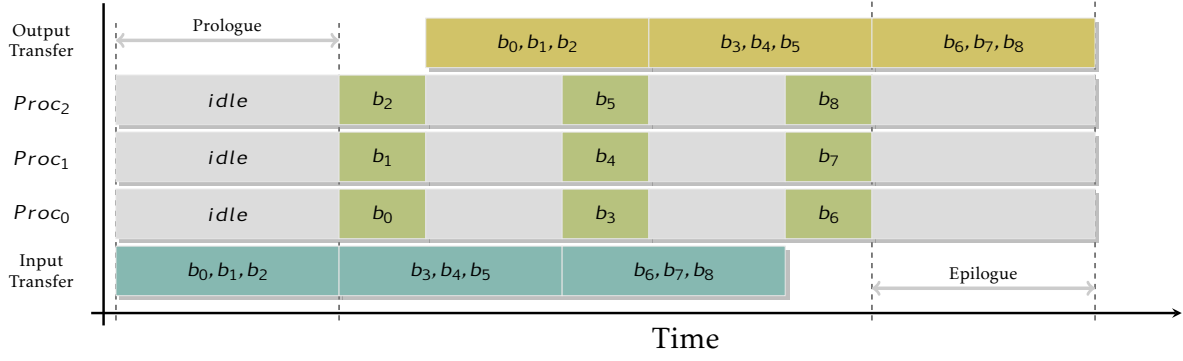
Figure 8.10 – Pipelined execution in the transfer regime using multiple processors

shape with smaller number of lines has a smaller transfer overhead, that is $T(s_1, s_2) < T(s_1', s_2')$.

Observe that the computation domain is convex where for any point s inside the domain, we can always find another point s' on the boundary such that $s_2' = s_2$ and $s_1' < s_1$ and hence with a smaller transfer time. Therefore the candidates for optimality are restricted, as for the one-dimensional case, to the intersection $T(s_1, s_2) = C(s_1, s_2)$. These points are of the form $(s_1, H(s_1))$ where,

$$H(s_1) = (I_0 + I_1/s_1)/\psi$$

and their transfer time is expressed as a function of the number of clustered horizontal blocks s_1:

$$T(s_1, H(s_1)) = c \cdot (I_0 + I_1 s_1)$$

where c is the constant $1 + (gb/\psi)$. This function is linear and monotone in s_1, meaning that as we move upwards in the hyperbola H the transfer time increases, and hence the optimal shape is,

$$(s_1^*, s_1^*) = (s_1, H(s_1^*)) = (1, H(1))$$

which constitutes a contiguous block of one line of the physical data array. This is not surprising as the asymmetry between dimensions in memory access prefers *flat* blocks with $s_1 = 1$. Without data sharing and memory size constraints the problem becomes similar to the one-dimensional case where it is only the size of the block that needs to be optimized.

8.2.2 *Multiple Processors*

Given p identical processors having the same processing speed and the same local store capacity, the input array is partitioned into p chunks of data distributed among the processors to execute in parallel. We assume that processors have enough memory to that each of it can implement double buffering algorithm. We extend our granularity analysis to this case assuming all processors implement the same granularity.

Data transfers in this setting may happen in parallel, leading to network contentions and we model the corresponding increase in transfer time as proportional to the number of processors. It also reduces the workload per processor as computation happens in parallel. We assume that each processor has its own DMA so that it can issue request irrespective of the other processors and all the processors start at the same time in a synchronized manner. Thus the DMA initialization phase is overlapped in time. Figure 8.10 illustrates a pipelined execution using several processors where p concurrent transfer requests arrive simultaneously to the
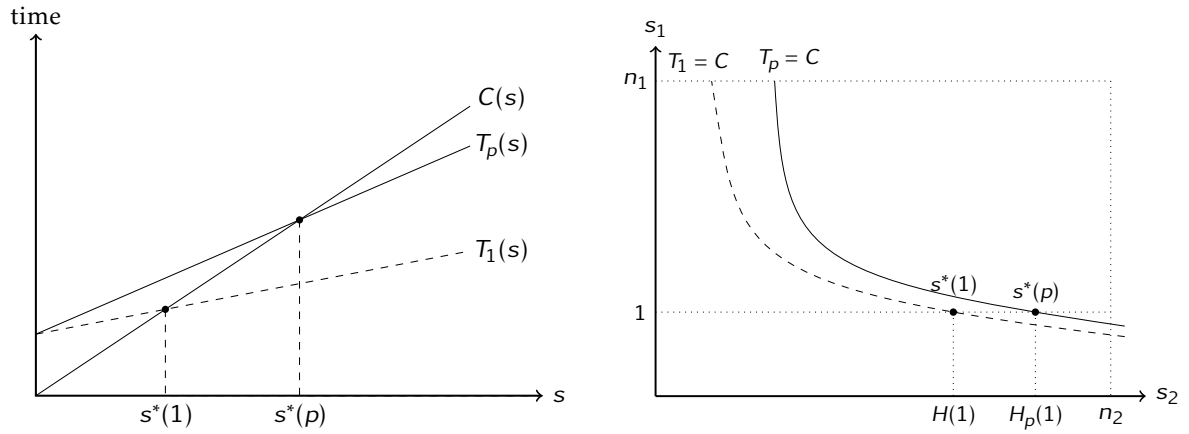
Figure 8.11 – Evolution of the computation domain and optimal granularity as we increase the number of processors

(a) One Dimensional   (b) Two Dimensional
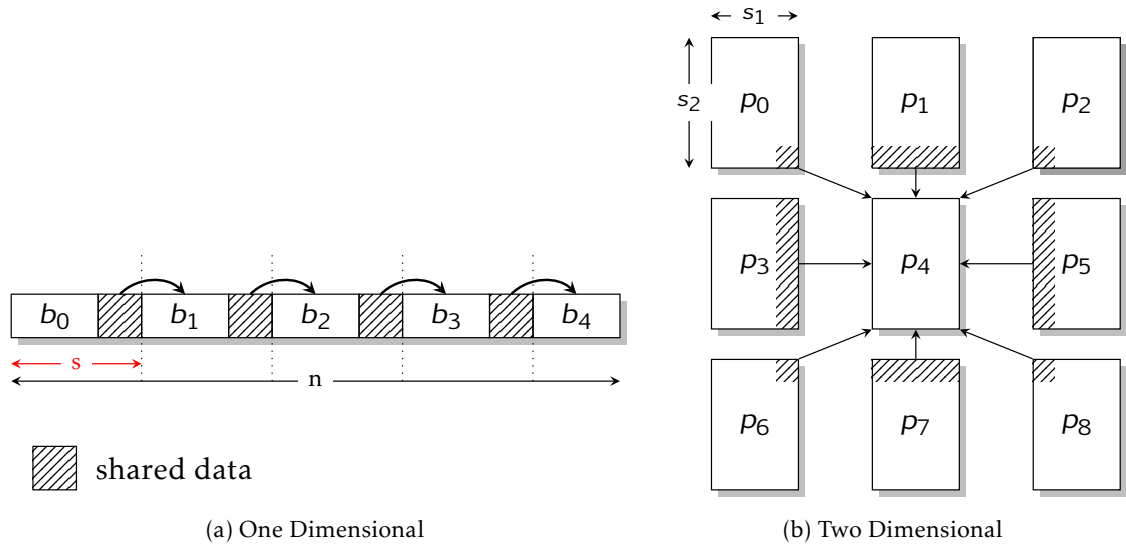
Figure 8.12 – Shared data communication

shared interconnect. Arbitration of these requests is left to the hardware which performs the data transfers at a low granularity (packet based) in round robin fashion. We assume that the time difference between processors receiving their blocks is negligible.

We model by parameterizing the transfer cost per byte g with the number of active processors such that g_p that increases linearly with p. Obviously this changes the ratio between the computation time and the transfer time of a block and consequently the optimal granularity. Figure 8.11 shows the evolution of the computation domains and optimal granularity for one-dimensional and two-dimensional data as we increase the number of processors. The reasoning is similar to the previous one, where functions T and H become T_p and H_p, yielding an optimal data granularity for each p. For more details the reader is referred to [104, 105].

8.3   SHARED DATA TRANSFERS

A shared data computation is a data-parallel loop in which the computation for each block needs additional data from the neighboring blocks. In the one-dimensional case the processing of the input block is assumed to be:

$$Y_i = f(X[i], V[i])$$

where $V[i]$ is additional data taken from the input left neighbor $X[i-1]$. For two dimensional case, the assumed data sharing $V(i_1, i_2)$ was defined in Section 8.1. For both one and two dimensional case the shared data is illustrated in Figure 8.12. This shared data, can be transferred between the processors in various ways. We consider three strategies/mechanisms for transferring shared data that mainly differ in the component of the architecture which carries the burden of the transfer,

1. Replication: transfer via the global interconnect from main to local memory. In this method, the processor issues the DMA get for input and shared data for both X and V.
2. Inter-processor communication: transfer via the global interconnect between the cores. In this method, the processors synchronize at the beginning of the iteration and transfer the shared data using DMA, and then proceed to computations.
3. Local buffering: transfer by the core processors themselves. In this method, the processor after computation of current iteration copies the shared data from current iteration to a new position in the local store to be used in the next iteration.

We can observe that in the second case, we need to do periodic allocation of data, while for the third case we have to do contiguous allocation of data as shown in Figure 8.2. These strategies can be compared with respect to the overhead for communication time or replication time in the third case.

In general we observe that replication performs better in the computation regime because even if the data transfer time has extra overhead for V which is overlapped with the computation time and hence compensated. In transfer regime, the execution time depends also on other factors like amount of shared data k, cost for locally copying the data etc. A detailed analysis of how to select the best strategy is given in [104, 105], and we omit it here for brevity.

8.4   DMA PERFORMANCE OF THE CELL PROCESSOR

We measure the DMA transfer time and accordingly the cost per byte as we increase the block size in one transfer, as shown in Figure 8.13. The DMA cost per byte is reduced significantly for block size larger than 128 bytes. On Cell architecture if the DMA is not multiple of 128-bytes or unaligned at source and/or destination to 128-byte boundary, the interconnect performs poorly [71]. We observe this effect in Figure 8.13(b), where below 128 bytes the cost per byte is significantly high and is consistent for higher sizes. For large block sizes the ratio converges to cost per byte g_p proportional to the number of processors. As we increase the number of processors and synchronize concurrent DMA transfers, we can observe that transfer time is not highly affected for a small granularity because the initial phase of the transfer is done in parallel in each processor's MFC, whereas for large granularity the transfer time is proportional to the number of processors due to the contentions of concurrent requests on the bus and bottleneck at the Memory Interface Controller (MIC) as explained in [71].

The initialization phase time for one-dimensional blocks l is about 400 cycles and the DMA transfer cost per byte to read from main memory g_1 is about 0.22 cycles per byte, and it increases proportionately to the number of processors to reach $p \cdot g_1$ (for high granularity transfers).

(a) DMA Transfer Time

(b) Cost Per Byte
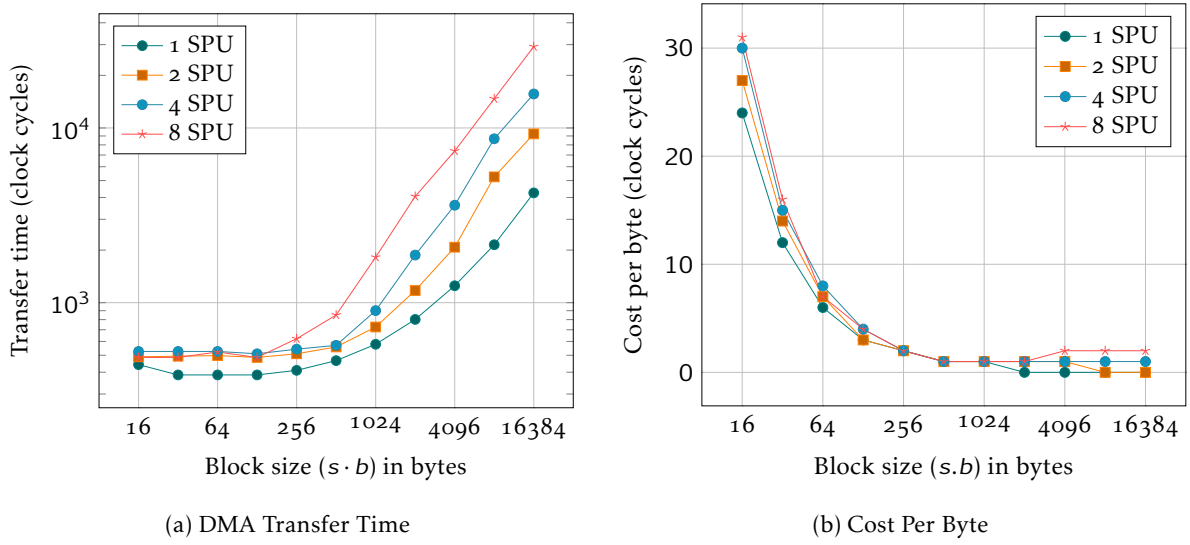
Figure 8.13 – DMA performance for contiguous blocks

p	g_p		
	min	max	avg
1	1.13	14.00	2.57
2	1.78	29.98	4.13
4	3.97	47.23	11.07
8	5.43	87.86	18.82

Table 8.1 – DMA performance for contiguous blocks

For two-dimensional data blocks DMA transfers are implemented using DMA lists (explained in Section 3.6), for which we derive the DMA parameter values based on profiling information. As modeled in Section 3.7, these parameters consist of a fixed initialization cost $l_0 = 108$ cycles and an initialization cost per line $l_1 = 50$ cycles which corresponds to the cost of the processing of each list element. The transfer cost per byte g is subject to variations that are more visible and amplified for data block transfers with increasing block height s_1. These variations are due to several factors such as concurrent reading and writing requests of the same processor, packet-level arbitration between requests of different processors as well as the effect of stridden accesses of main memory. The minimal, maximal and average values of g_p measured for two-dimensional data are shown in Table 8.1 and we use the average value in our model used to calculate the optimal granularity.

Note that due to the characteristics of the Cell B.E. not all block size and shape combinations are possible. The limit for number of elements in a DMA list is 2K, where each element is a contiguous block transfer of upto 16KBytes. However, the Cell B.E. has a strict alignment requirements on 16-byte boundary for both DMA transfers and SPU vector instructions for which the processor is optimized. If this is not taken care of, the DMA engine aligns the data by itself causing data transfers to local memory at shifted offsets. To consider the shifting in application code would become too complex, and hence the scenario is avoided altogether by restricting to aligned transfers.
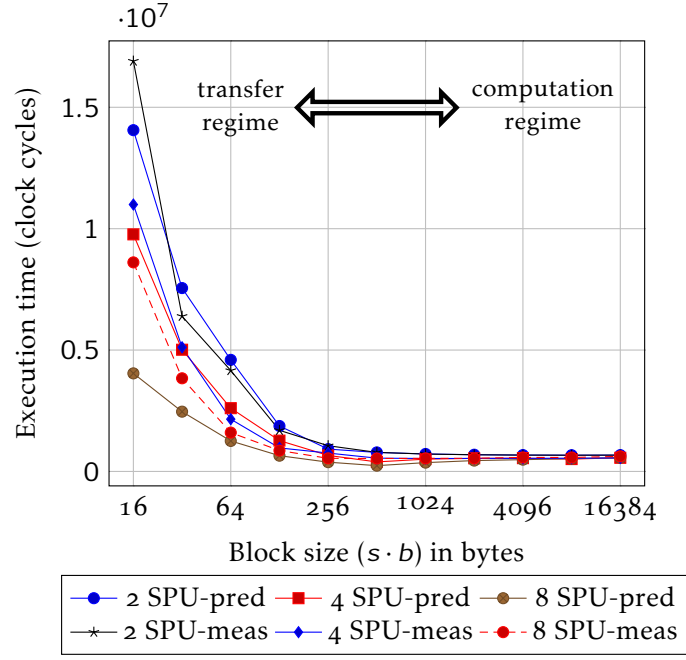
Figure 8.14 – Independent data computations

8.5 EXPERIMENTAL RESULTS

In the experiments, we implement a double buffering algorithm for different benchmarks, some of them are applications where computations are completely independent and others have shared data. For each benchmark, we vary the block size and shape, along with the number of processors. We compare the performance (and the optimal solution) obtained in practice with those predicted by our analytical model. Our benchmarks consist of, first synthetic algorithms of (independent/shared) computations where f is a synthetic function for which we vary the computation workload per byte ω and the size of shared data k.

We then implement a convolution algorithm (a FIR filter) that computes an output signal on as a convolution of an input signal and an impulse response signal. These signals are encoded as one-dimensional data arrays and the size of the impulse response signal determines the size of the data window required to compute one output item. We vary the size of the impulse signal to vary the size of the shared data.

Our last benchmark is a mean filtering algorithm working on a bitmap image, encoded as a two-dimensional data array. This algorithm computes the output for each pixel as the average of the value of its neighborhood.

8.5.1 *Independent Data Computations*

To validate our analytical results for independent computations, we use synthetic algorithms and focus only on one-dimensional data, since, as explained previously, optimizing data granularity for independent two-dimensional data is similar to the one-dimensional case where it is about optimizing the granularity (and not the shape) of data.

For the synthetic algorithms, we fix the size b of an element to 16 bytes and the size n of input data array to 64K elements, the total size of the array being 1Mbytes. The local storage memory capacity 256K and two double buffers allocated then limit the possible clustered elements in one transfer to B=4K elements, not taking into account the memory space allocated

(a) shared data size 128 bytes

(b) shared data size 1024 bytes

Figure 8.15 – Measurements for shared data in computation regime

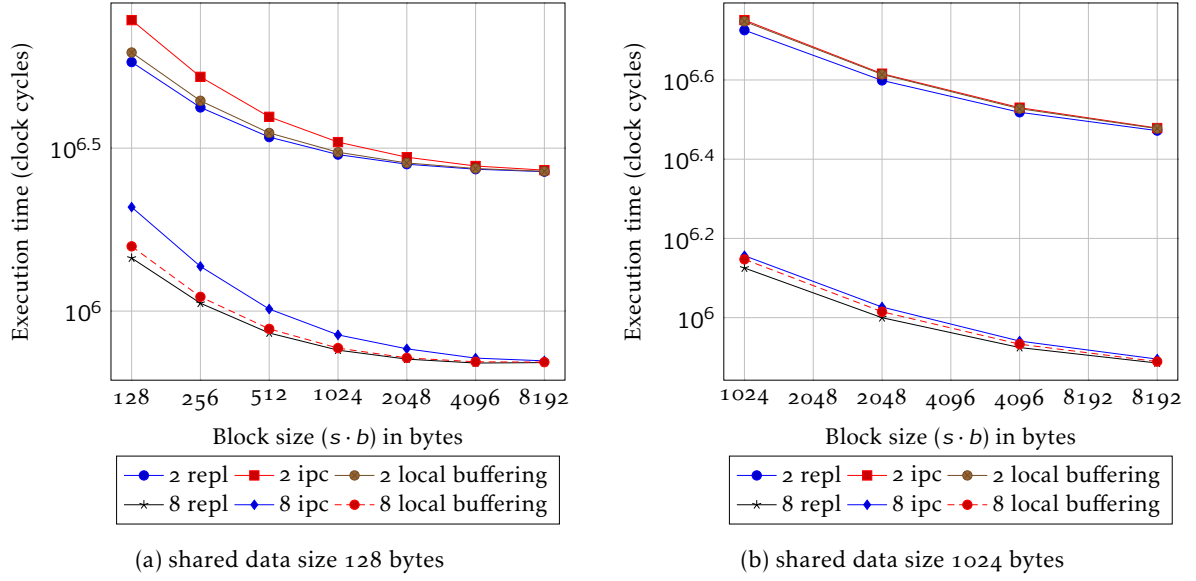for code and the memory required for other data structures. We vary the size of block s and the number of processors. We compare both predicted and measured optimal granularity, and the total execution time for both transfer and computation regimes.

Figure 8.14 shows the predicted and measured values for 2, 4 and 8 processors. We can observe that the values are very close to each other. The predicted optimal points are not exactly the measured ones but they give very good accuracy. Performance prediction in the computation regime is better than in the transfer regime, the dominant part of the total execution time is due to the processors which have more predictable performance than the interconnect. Besides, as mentioned in [107] in this regime we hide delays due to the interconnect latency and bandwidth.

8.5.2 *Shared Data Computations*

For shared data computations we perform the experiments for two shared data sizes k=128, 1024 bytes for 2 and 8 processors. Observe that as expected in the computation regime the replication strategy performs always better than local buffering and IPC as shown in Figure 8.15. Further we also observe that IPC performs worse than local buffering owing to the high synchronization overheads between the processor for the inter-processor data transfer. The synchronization cost using barrier is between 800 and 2400 cycles at each iteration, which is comparatively larger than overhead replicating data in the computation regime.

In the transfer regime, performance varies according to the value of *k* and number of processors. We can observe in Figure 8.16 that the costs of local buffering and replication are nearly the same, and that replication performs even better for a transfer of blocks size between 512 bytes and 2K. This demonstrates that using DMA for transferring additional data can perform sometimes better than local buffering even for a small value of *k*, and that keeping shared data in the local store may have a non-negligible cost. Therefore, even when considering contiguous partitioning of data, shared data redundant fetching using a replication strategy can be as efficient, if not more efficient than keeping shared data in the local store. However, the cost of transferring shared data using replication becomes higher than other strategies when the number of processors increase because of the contentions even for small values of *k*.

(a) shared data size 128 bytes   (b) shared data size 1024 bytes
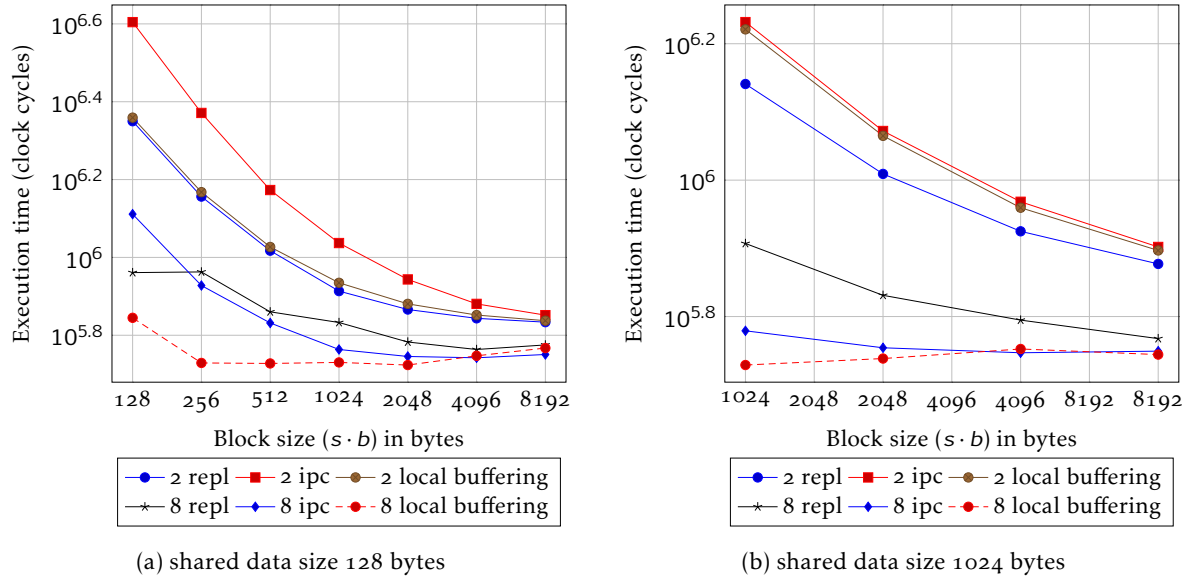
Figure 8.16 – Measurements for shared data in transfer regime

(a) 2 Processors   (b) 8 Processors
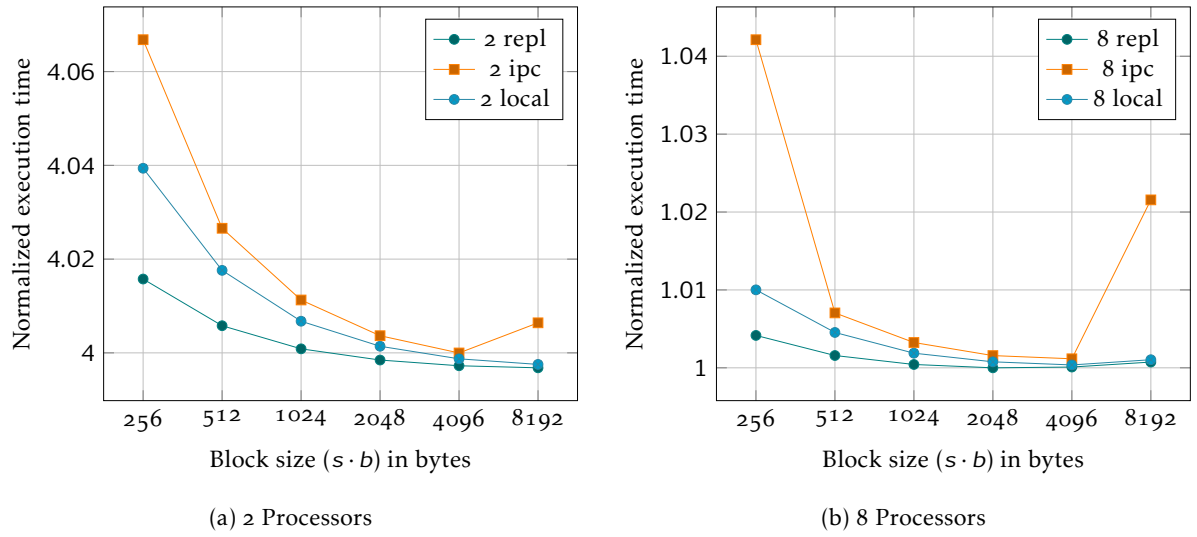
Figure 8.17 – Convolution algorithm

8.5.3 *Convolution Benchmark (FIR filter)*

Convolution is an algorithm [91] benchmark (FIR filter) used commonly in signal processing applications. In a FIR (finite impulse response) filter, convolution is done between a given finite impulse response signal of size r and the the input signal. The algorithm assumes 1D input array share $r - 1$ data elements between the subsequent iterations. The equation to calculate the output Y for an input arrays X and C is given as:

$$Y[i] = \sum_{j=0}^{m} X[i - j] \cdot C[j]$$

In our experiments, the size of input array X is chosen to be 1Mbytes of data, so it cannot fit
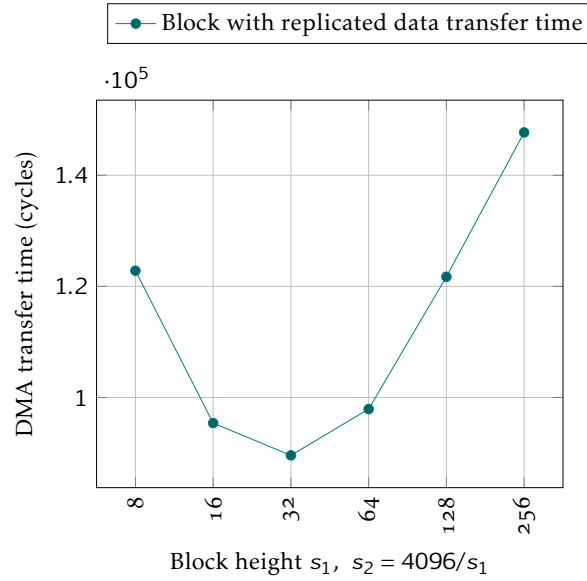
Figure 8.18 – Predicted transfer time for different block shapes with shared data

in the scratchpad memory, whereas C, the filter coefficients are small enough to be permanently stored in the local store of each SPU. Hence double buffering is implemented to transfer data blocks of array X (respectively Y). The measured computation cost per element ω is about 53 cycles, owing to the SIMD (single instruction multiple data) floating point operations performed on *double* data type. Note that for this algorithm, despite an optimized implementation using vector operations the computation cost per byte ω is much higher than the transfer cost per byte with maximum contentions g_8, resulting from the use of maximum number of available cores. Therefore, the overall execution is always in a *computation regime* for all strategies.

Figure 8.17 summarizes performance results for size of $b = 8$ bytes and $r = 32$ samples, using 2 and 8 processors. In the computation regime, replication strategy outperforms local buffering and IPC strategies since it avoids the computational overhead at each iteration of copying shared data locally or exchanging data between neighboring processors using synchronous DMA calls. This overhead is proportional to the number of iterations and therefore decreases with higher granularities to be eventually negligible which leads all strategies to perform with nearly the same efficiency for large granularities.

8.5.4 *Mean Filter Algorithm*

We use a mean filter algorithm which works on a bitmap image of 512×512 pixels. Each pixel is characterized by its intensity ranging over $[0 - 255]$. The output for a pixel is the average of the values in its neighborhood defined as a square (mask) centered around it. We have experimented with different mask sizes and focus on the presentation of the results for a 9 x 9 mask, that is, $k = 8$. Note that the Cell architecture does not support SIMD instructions for *char* (1-byte) datatype, and non-SIMD operations are highly inefficient on SPU. In order to use SIMD operations for efficient execution of the filter code, we encode a pixel as an integer ($b = 4$ bytes). Based on profiling information, the computation workload per element obtained, is roughly $\omega = 62$ cycles.

Figure 8.18 illustrates the influence of the shape of the block (and its implied replicated area) on the transfer time calculated using formula mentioned in [104]. We consider in this plot different feasible combinations of (s_1, s_2) so that $s_1 \times s_2 = 4096$. A shape (s_1, s_2) yields a

(a) $s_1 \times s_2 = 1024$

(b) $s_1 \times s_2 = 2048$
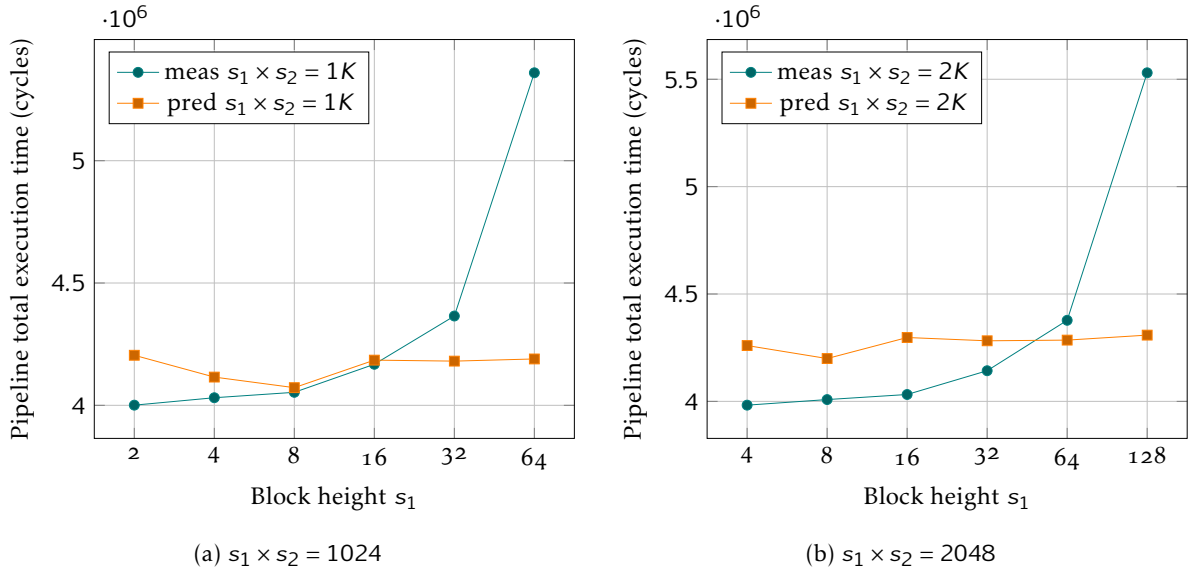
Figure 8.19 – Predicted and measured values for different combinations of $s_1 \times s_2$

block of $s_1 + 8$ lines, each line corresponding to a contiguous transfer of $b \cdot (s_1 + 8)$ bytes. The optimal transfer time is obtained neither for square $(64, 64)$ nor the flattest possible $(8, 512)$ blocks and the best trade-off in this case is $(s_1, s_2) = (32, 128)$.

Finally we evaluate the effect of the granularity and shape of the blocks and the total execution time of the pipeline for different numbers of processors. The distance between the predicted and measured values is rather small except for large values of s_1. The major reason for the discrepancy between the model and the reality is that $C(s_1, s_2)$ has non negligible component that depends on s_1 for two reasons. The first is due to the overhead at each computation iteration related to the setting required between the outer loop and the inner loop like adjustment of the pointers for every row, pre-calculation of sums of borders etc. Secondly, the creation of DMA list elements equal to s_1, occupies the processor as discussed in Section 3.6 and this overhead is also added to the overall execution time.

We observed that overall predicted results are close to the measured ones. However, some error in prediction results from -

- Variation in the DMA cost per byte parameter g, with respect to contentions in the global interconnect, which is difficult to model accurately.
- Cell SPU are vector processors, which have high overheads for non-vector instructions like branching, scalar calculations etc. Owing to this fact, the code which performs initialization of the data structures, adjustments of pointers etc. requires considerably large amount of time.
- In the case of strided DMA the overhead of formulating a DMA list is significantly higher, which results in poor prediction for rectangular blocks with higher s_1 value.

A further detailed explanation of the experiments is given in [104, 105] and skipped here to keep the text concise.

8.6 CONCLUSIONS

In this chapter, we presented a way to model the DMA transfer granularity problem for data-parallel applications, under simplified assumptions. The selection of such parameters is

an additional burden to the application programmer and in general a non-trivial issue, owing to various factors that must be considered such as dimensionality, amount of shared data etc., combined with the hardware architecture parameters. We captured the essential parameters of the applications with or without shared data and presented a method to calculate the optimal data-transfer granularity for double-buffering algorithms. We presented real-life application benchmarks in the experiments section to prove validity of our results.

Pre-fetching the data to the local processor memory is an old technique to close the gap between the speed of the processor and the main memory. Caches used for this reason faced the problem of area and speed overhead in supporting cache coherency [82]. Scratchpad Memories (SPMs) is an alternative solution to caches where data (and sometimes code) transfers through the memory hierarchy explicitly by the software. In architectures supporting scratchpad memories, buffering techniques have been suggested previously to hide the access latencies [108]. In [27], the author performs parametric analysis for calculating optimal buffer size for data independent parallel loops for a cache based architecture. However the work doesn't consider multi-processor scenario or data sharing. Effect on multiple processors performing DMA concurrently is studied in [71]. Our work on one-dimensional data is very close to one in [136], where the author presents a formal analysis on the optimal buffer size on a single processor system. We extend the problem with multiple processors accessing the resources. Work has been done in context of parallel loop partitioning [2, 7, 77] for effective distribution of data on multiple processors. However, we deal with DMA-based architectures instead of cache-based, and study the problem formally. This is a joint work and is also published in [103].

Despite the fact that for each application the SMT solver can generate different Pareto configurations which can optimally execute an application maintaining the given costs, this is not always sufficient in practice due to fundamental problem that the exact set of applications executing on the on the platform at the same time is often unknown at compile time, as it is dynamically decided at run time. In order to make an efficient execution of all the applications, decisions must be made at system-level at run-time to optimize the utilization of the resources by a dynamically changing set of applications. In next chapter we describe such a run-time system which manages applications dynamically.

Run-time application management and reconfiguration

This chapter introduces a run-time management system to handle the applications dynamically starting at unknown time in the on resource-constrained multi-core processors.

In contemporary MPSoCs, the number of applications that run concurrently is increasing rapidly. It is often the case that the designer has only a predefined set of applications, which can be launched by the user. However, launching of an application depends on the user requirements. For example, if user is playing a video on a mobile device, the audio related to it should also be decoded in parallel. In background, the device may update its messages or emails, which is happening transparently to the video application. Thus even if the applications are known at compile-time, the set of simultaneously running applications varies. In addition each application has its own Pareto optimal configurations which defines the trade-offs between various system parameters like resource usage, energy consumption etc. Due to this situation there is a need for having a runtime manager which will launch or reconfigure all the applications in order to globally optimize the resources of the device, while satisfying the requirements of the applications. This launching or reconfiguration should also be done transparently without affecting running applications like an MPEG decoder, which have strict deadlines. In short, the application launching and reconfiguration should be predictable (should finish in a definite amount of time) and composable (should not affect the running applications).

In this chapter we describe a run-time manager which guarantees predictable and composable behavior. We describe how the system-level resource manager manages applications and provides configuration decisions to the application-level resource-manager. The application resource manager then, depending on the configuration provided, reconfigures the application(s) in a predictable and composable way. We demonstrate this run-time manager on a real hardware platform. We perform a case-study of JPEG decoder application on this platform and demonstrate its performance.
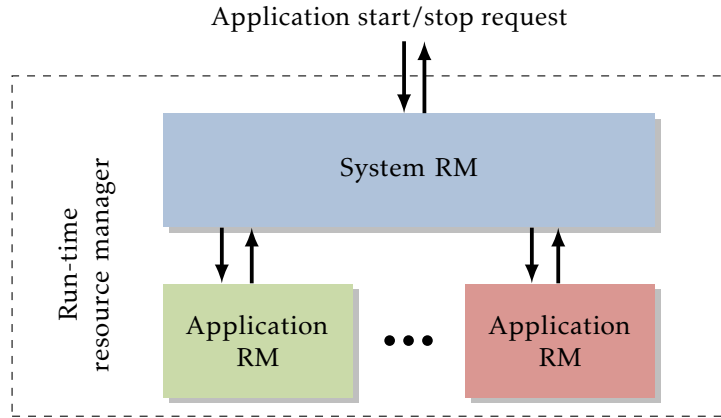
Application start/stop request

Figure 9.1 – Run-time resource manager (conceptual view).

9.1   RUNTIME RESOURCE MANAGER

At design time, the design flow generates multiple configurations for a given application which represent the available trade-offs between resource usage and quality. Configurations could for example provide a trade-off between the use of different resources, such as different processors, or trade-offs between processing and memory usage, or through DVFS, between energy consumption and resource utilization. The configurations determined at design time are Pareto optimal trade-offs between the various quantities considered at run-time (i.e., resource usage, energy usage and quality). The configurations are determined at design time on a per-application basis. The run-time resource manager must combine these configurations at run-time to investigate system-wide trade-offs. After combining the trade-off spaces of the individual applications, the resource manager can decide on the optimal system-wide configuration of all running applications and subsequently it can implement this decision by reconfiguring the individual running applications. As mentioned before, all of these steps need to be performed by the run-time resource manager while providing timing guarantees to the running applications. A resource manager which fulfills these constraints is introduced here. The resource manager uses the predictable and composable *multi-dimensional multiple-choice knapsack* (MMKP) heuristic presented in [113] to select an optimal configuration for all applications running on the platform. The configuration selected by this heuristic may trigger a reconfiguration of these applications. Our proposed run-time mechanism ensures that this reconfiguration process is completed within a bounded amount of time (i.e, the reconfiguration is predictable). Moreover, the timing behavior of applications of which the resource assignment is not changed will not be affected (i.e., the reconfiguration is composable).

9.1.1   *System-level resource manager*

In order to perform a predictable and composable migration, we need a hardware platform which supports such features. In composable execution, starting and stopping of an application does not affect other applications running on the platform. Depending on the allocated resources a guaranteed worst-case response time for an application is called a predictable execution. The CompSOC platform discussed in Section 3.4 which runs CompOSe operating system provides an infrastructure for predictable and composable execution of applications.

The platform is equipped with techniques such as TDM based task and network scheduling etc. as discussed in Section 3.4. Every processor in the platform has a fixed number of time slots which can be allocated to different applications.

The system-level and application-level resource managers will be running on top of CompOSe. The system-level resource manager is implemented as a separate application running on the platform. Because of the composability offered by CompSOC, it is guaranteed that the manager itself will not interfere with the timing behavior of other applications. The system-level resource manager can optimize the resource budget distribution using an algorithm such as the MMKP heuristic presented in [113].

9.1.1.1 MMKP Algorithm

This algorithm assumes that the set of applications that execute on the hardware platform are known at the compile-time. A Pareto set of solutions can be calculated with methods such as cost space exploration discussed in Chapter 4 for each application. This algorithm decides on the optimal configuration of every application in a bounded amount of time (i.e., it is predictable) while considering multiple costs, with the goal of globally efficient execution of all applications satisfying the resource constraints.

The inputs to this algorithms are the Pareto points for each application, where every point indicates the amount of resources used and a value associated with the point which indicates its optimality. Suppose an application is already running on the platform. A maximum-value configuration can be selected for optimized execution of this application. When a new application starts on the platform this algorithm combines the Pareto configurations of the new application and the currently running application(s) to form a new set of configurations. In order to speed up the computation of new set, the multi-dimensional cost space is transformed to a two-dimensional space by aggregating the resource usage. In this transformed space the dominated points for every application are eliminated. Every Pareto point of newly started application is combined with every Pareto point in the current set such that it satisfies the resource constraints. If the combination of the configurations does not satisfy the platform resource constraints the point is eliminated. The dominated points in the combined set of solutions obtained are again eliminated. Thus the resultant set consists of combined configurations of already running applications and the newly starting application annotated with total resource usage and optimality value. A tunable parameter L provides a bound on how many total combined solutions are retained. If the above combination generates more Pareto points than the parameter L, it eliminates some of them heuristically. A Pareto point with maximum-value is then chosen which provides a configuration for every application running on the platform. Given an MMKP instance I with N newly starting applications, the bounds on the execution time can be given by:

$$T(I) \leq C \cdot N \cdot L^2$$

where C is a processor dependent constant.

In this work, we use the MMKP heuristic in our system-level resource manager. For further details on the algorithm the reader is referred to [113]. It is possible to replace the MMKP algorithm with any other one that would to decide reasonably optimal configuration for all the applications in predictable amount of time. The configuration of applications represented by the chosen Pareto point by system-level resource manager can call for reconfiguration of application(s) running on the hardware platform. We focus on the application-level resource manager which implements the resource allocation decisions taken by the system-level resource manager within a bounded amount of time.

Figure 9.2 – Run-time resource manager (deployment view).

9.2 IMPLEMENTATION

In this section, we describe the mechanism by which the application-level resource manager handles the application reconfiguration.

9.2.1 *Application-level resource manager*

The implementation of a reconfiguration decision, which is handled by the application-level resource manager, may require that actors from an application are migrated from one processing tile to another. This process is triggered from inside the application which is being reconfigured. This is because from the perspective of the CompOSe operating system, our application-level resource manager is part of the application. The application programmer does not have to code for the application manager, but only add the application manager task in its schedule.

Our application-level resource manager uses a master-slave configuration (see Figure 9.2). The master stores the active resource configuration of the application. Whenever the MMKP algorithm decides to reconfigure the application, the master will send a set of reconfiguration commands to the slaves. For this purpose, a pair of dedicated FIFOs is used between the master processor and each slave in the platform.

Since the whole reconfiguration process takes place only during the TDM time slices allocated to the application, it is guaranteed that the other applications will not be affected by the reconfiguration (i.e., the run-time resource manager is composable). In order to arrive at a predictable run-time mechanism it is however also important that the reconfiguration process itself can be completed within a bounded and known amount of time. Otherwise, the application which is being reconfigured may miss its own timing deadlines.

9.2.2 *Migration Point*

To reduce the migration overhead, a reconfiguration can only be performed at specific moments during the execution of the application. In the SDF MoC (model of computation), actors have no state that needs to be preserved across firings. Any state (data) that needs to

be stored between firings should be stored explicitly as a token that circulates on a special self-source self-sink edge (self-edge) of an actor. Allowing actor migrations only when an actor is not executing (firing) ensures that no state (other than the initial token on the self-edge) needs to be transferred. This implies, no processor context needs to be migrated, which reduces the migration overhead considerably.

Whenever an actor is migrated to a different processor, the edges connected to this actor must also be reconfigured. Tokens that are present in these edges must then be migrated from the old edges to the newly created edges. Throughout the execution of the SDF graph, the number of tokens in the edges may vary. Hence, the amount of tokens that needs to be transferred may vary depending on the moment when an actor would be migrated. As a result, the time needed to migrate an actor and its connected edges might depend strongly on the number of tokens in these edges. In order to provide timing guarantees, design-time analysis techniques must take the worst-case situation into account when analyzing whether a particular reconfiguration would be feasible given the timing constraints of the application. When actor migrations are allowed to occur at any moment when an actor is inactive, this could lead to very pessimistic estimates on the number of tokens that needs to be transferred. As a result, design-time analysis techniques would most probably indicate that many reconfiguration options cannot be performed within the given timing constraints. To address this issue, we use a special property of SDF. Since the production and consumption rates of the actors on the edges are constant in an SDF graph, there exists an execution interval called graph iteration (see Section 2.2), where each actor has been fired for a certain number of times, the token distribution in an SDF graph returns to its original state. At this moment, the number of tokens in all edges connected to an actor are equal to the number of initial tokens. Since this is a well defined amount, it can be easily taken into account in the timing analysis performed at design-time. Therefore, our application-level resource manager will only migrate an actor and its connected edges when the actor and all actors connected to the edges of this actor have completed an iteration. Note that this condition is not sufficient. The actors that communicate with migrating actor should also be halted at the start of an iteration. Otherwise during the migration, tokens may be added or removed from these edges unequal to the number of initial tokens and taking this into account it too complex.

9.2.3   *Actor and FIFO Migration on CompOSe*

When migrating an actor, the schedule on the processor on which this actor was originally running as well as the schedule on the processor to which the actor is moved must be changed. In CompOSe, actors from the same SDF graph are scheduled using a static-order schedule. To modify these schedules, CompOSe requires that the complete schedule of all actors that belong to the same SDF is removed from a processor and subsequently a new schedule can be loaded. This implies that the application (SDF) should be stopped on the processor from which the actor is migrated away as well as the processor to which the actor is moved. When migrating an actor, the channels (FIFOs) connected to the actor must also be migrated. This can be done through CompOSe by removing the old FIFOs and adding the FIFOs to the new migrated actor. If initial tokens are present in the old FIFO, they are retained and copied to the new FIFO. CompOSe requires that the application whose set of FIFOs is modified on a processor is not running during this reconfiguration. Hence, the application must be stopped on the processor from which migrated actor departs, the processor receiving the migrated actor and all processors that run actors which have at least one FIFO channel connected to migrated actor. In many practical situations, this will often imply that the application must be halted on all processors. Considering this aspect and in order to reduce the number of messages exchanged between the master and slave components of our application-level resource manager, the resource manager
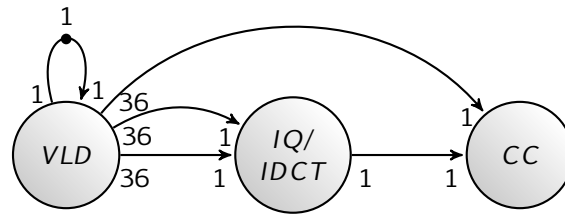
Figure 9.3 – JPEG decoder SDF graph

will halt the application on all processors. Next, it will perform the reconfiguration. When this process is completed, the application will be resumed on all processors. Note that during this whole process, the other applications running on these processors are not interrupted which ensures that our run-time reconfiguration is composable.

The next section presents in detail the messages that are exchanged between the master and slave components in our application-level resource manager when reconfiguring an application. As explained in that section, each individual operation during the reconfiguration process (e.g., reconfiguration decision, actor migration, FIFO creation, etc.) can be performed in a bounded amount of time. Since the complete set of operations that needs to be performed in order to migrate an actor is known at design-time, it is possible to provide a timing guarantee on the completion of the whole reconfiguration. Hence, the run-time resource manager offers a predictable reconfiguration mechanism.

9.3 APPLICATION-LEVEL MANAGER

We demonstrate our resource manager with a running example of a JPEG decoder where actors migrated from one processor to another. The application SDF graph shown in Figure 9.3, consists of three actors, namely Variable Length Decoder (VLD), Inverse Quantizer combined with Inverse Discrete Cosine Transform (IQ/IDCT) and Color Conversion (CC). The VLD actor has a state that needs to be preserved across actor firings. In between firings, this state is stored as an initial token on the self-edge connected to this actor. One pair of edges from the VLD to IQ/IDCT respectively CC actor is used to communicate JPEG header parameters (e.g., image size) between the actors. The remaining edge from the VLD to the IQ/IDCT actor and the edge from the IQ/IDCT to the CC actor are used to communicate the actual image data.

9.3.1 *Run-Time Actor and FIFO Migration*

Figure 9.4 illustrates the actor migration process. Initially the VLD and IQ/IDCT actors are running on the second tile and the CC actor is running on the first tile. Next to these actors, the second tile is also running the system-level resource manager as well as the master application-level resource manager for this application. The other two tiles are running a slave application-level resource manager for our JPEG decoder application. On each tile, if the application does not execute, a small time slice is allocated to application level resource manager which checks for any migration decisions. It facilitates to migration of applications on the tile where the application is not executing previously. In this example, no other applications are active on the platform. At some point in time, the system-level resource manager decides to migrate the CC actor from tile 1 to tile 3. (This decision would normally be triggered when a new application is started on the platform, but for simplicity we assume in this example that the system-level resource manager takes this decision without any new application entering the system). Once the system-level resource manager has taken the decision to reconfigure the
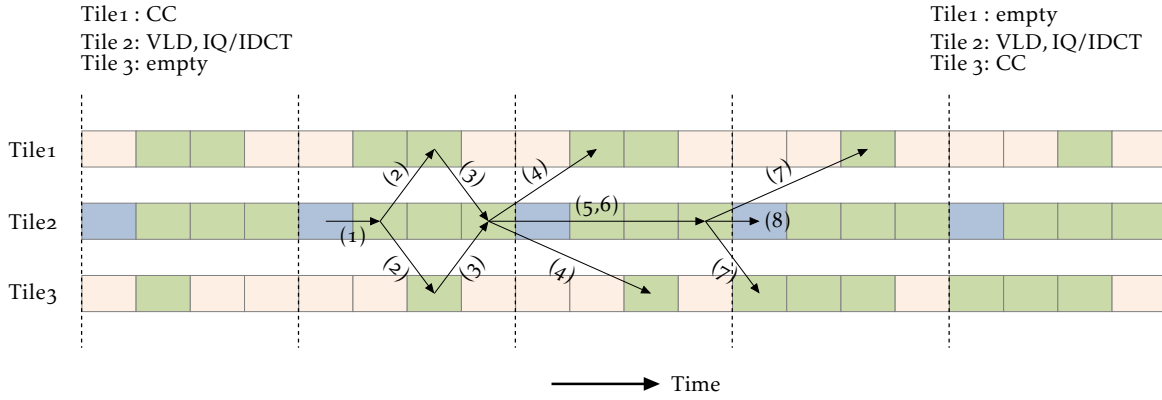
Figure 9.4 – Actor migration (Steps performed by resource manager. For steps (1), (2), (3) etc. refer to Table 9.1).

application, it informs the master application-level resource manager about this decision (step 1 in Figure 9.4). Since the application-level resource manager is part of the JPEG application running on tile 2, it will be periodically scheduled on this tile. The first time it gets scheduled after the system-level resource manager took the decision to reconfigure, it will start the actual reconfiguration process. This process starts by sending a command to all tiles to halt the application (step 2 in Figure 9.4). This is done by sending a message through the dedicated FIFOs between the master and slave application-level resource managers. Whenever an application-level resource manager is scheduled on a processor, it will check this FIFO to verify whether new commands are available in this FIFO. If so, these commands will be processed. Once the command to halt the application on the tile has been completed, the slave application-level resource manager will inform the master application-level resource manager by sending an acknowledgment (step 3 in Figure 9.4). When all slaves has confirmed that the application has been halted and the application has also been halted on the tile running the master application-level resource manager, the resource manager continues with the next step of the reconfiguration process. Since after reconfiguration there will be no actors of this application running on tile 1, the number of time slices allocated to the application can be reduced on tile 1 to just one (which is needed to periodically execute the slave resource manager). Furthermore, the number of TDM slices allocated on tile 3 may have to be increased. This is done as step 4 in Figure 9.4. In this step, the master resource manager instructs the slave resource managers to make this change in the TDM allocation. Step 5 involves the removal of the old FIFOs from tile 2 to tile 1 and in step 6 new FIFOs are created between tile 2 and 3. These FIFOs are immediately connected to the VLD actor on tile 2 and a new instance of the CC actor on tile 3. (Note that we assume that the instruction code of all actors is available on all tiles. Hence, no code migration is required). Next, the static-order schedule on tile 3 is updated (i.e., the CC actor is added to it). Subsequently (step 7), a message is sent to all tiles to resume the execution of the application. At this moment, the reconfiguration process of the application has ended and the master application-level resource manager confirms this to the system-level resource manager (step 8). This completes the complete reconfiguration process of the application.

As mentioned before, in order to ensure a predictable reconfiguration process, it is important that each of the steps described above can be completed within a bounded amount of time. Our implementation ensures that this constraint is met. Table 9.1 lists the number of clock cycles needed to perform the various steps in the reconfiguration process. These times

Table 9.1 – Actor migration overhead (in clock cycles).

| Step	Description	Master	Slave
(1)	Instruct application RM to reconfigure	50	n/a
(2)	Request removal of application from TDM	140	760
(3)	Remove application from TDM and ack.	App. dependent	
(4)	Resize TDM allocation	300	850
(5/6)	Add/remove FIFO	Table 9.2	Table 9.2
(7)	Add application to TDM	570	900
(8)	Inform system RM about completion	50	n/a

depend on the tile which ultimately needs to perform the operation. When the operation (e.g., removal of application from TDM) needs to be performed on a slave tile, then we need to consider the overhead of sending a message from the resource manager running on the master tile to the resource manager on a slave tile. The time required to remove an application from the TDM schedule (step 3) depends on the application. As explained before, an application may only be stopped on a tile when the application has completed a full iteration of the SDF graph. In the worst-case, the application may have just started a new iteration on all tiles when a request to remove the application from the TDM schedule is received. In that case, a complete iteration of the SDF graph must be finished before the request can be executed. Hence, the worst-case time needed to complete step 3 is bounded by the worst-case time needed to complete one iteration of the SDF graph on the platform. Since all resources in the platform are predictable and since we assume that the worst-case execution time of all actors are known, we can compute the worst-case time needed to complete step 3 at design-time. Table 9.2 shows the time required to reconfigure a FIFO. Depending on the tiles to which a FIFO is connected the time required to add or remove a FIFO varies. If the source and destination of a FIFO are both on the master tile, then there is minimal overhead to remove or add the FIFO. In this case, the overhead is limited to the allocating/freeing the data structure associated with the FIFO. When a FIFO is used to communicate between different tiles, for example between a tile running the master resource manager and a tile running a slave resource manager, the overhead will also include communication overhead to send a message to add or remove a FIFO and to send an acknowledgment after adding or removing the FIFO. The highest overhead occurs when a FIFO is connected between two different slave cores. In this scenario, the master application-level resource manager has to communicate messages between two different slaves and wait for their acknowledgments.

Table 9.2 – FIFO add/remove overhead (in clock cycles).

Src	Master	Master	Slave	Slave-1	Slave-1
Destination	Master	Slave	Master	Slave-2	Slave-1
Time to remove FIFO	570	1235	1235	1735	975
Time to add FIFO	1280	4925	5050	8000	4400
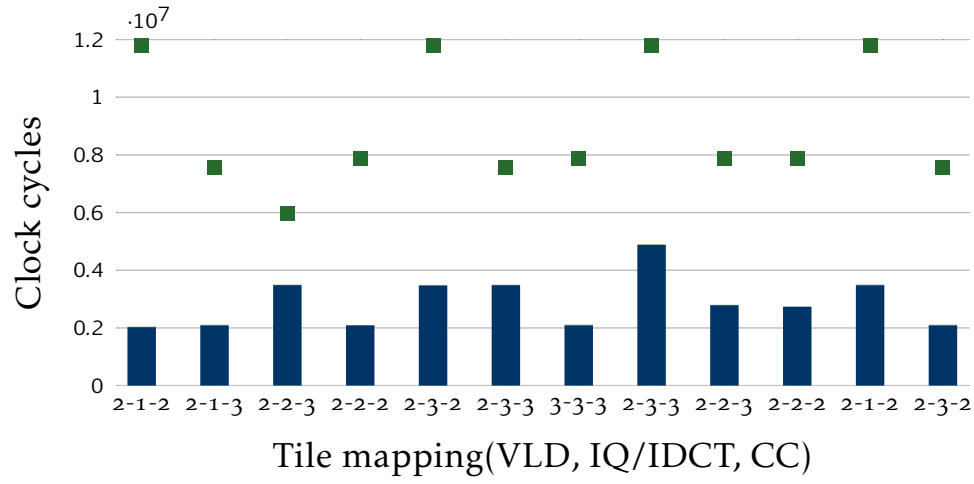
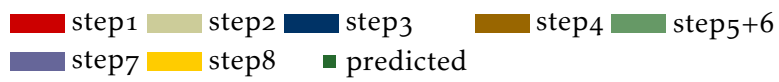(a) Measurements with step 3 included
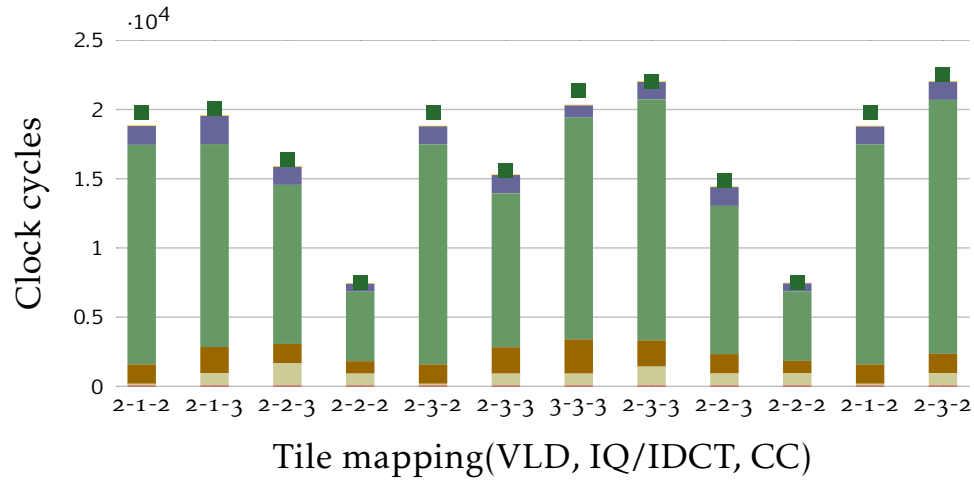
(b) Measurements without step 3 included

Figure 9.5 – Measured and predicted reconfiguration times

9.3.2 *Results*

It follows from Table 9.1 and Table 9.2 that the amount of time taken to complete an actor migration depends on the old and new configuration of the application on the system. We have performed an experiment in which the JPEG decoder is reconfigured several times. Initially all three actors are mapped to tile 2 i.e., configuration 2,2,2 in Figure 9.5 which indicates that the three actors VLD, IQ/IDCT and CC are running on tile 2. As a first reconfiguration, the IQ/IDCT actor is migrated to tile 1. The left-most bar in Figure 9.5 shows the run-time (in clock-cycles) needed to perform this reconfiguration. This run-time is measured using an actual implementation of our run-time reconfiguration mechanism on the CompSOC platform. The squared box above the bar indicates the worst-case reconfiguration time as computed using the run-times listed in Table 9.1 and Table 9.2. The next bar in Figure 9.5 shows the time needed to migrate the CC actor to tile 3 (configuration 2,1,2 to configuration 2,1,3). The other bars indicate other reconfiguration options that have been tested. Each time the label left to the bar indicates the configuration prior to the reconfiguration. The top part of Figure 9.5 shows that the reconfiguration time is dominated by step 3. The bottom part of Figure 9.5 shows the run-time of the reconfiguration process when excluding step 3. Comparing these two parts it is clear that the actual run-time of the reconfiguration process as well as the worst-case run-time of the reconfiguration process are dominated by step 3 (i.e., the worst-case of all other steps is always very close to the measured run-time for these steps). The fact that step 3 is the bottleneck in the reconfiguration process is not unexpected. As explained in Section 9.1, the run-time resource manager must in step 3 halt the execution of the application on all resources. In the worst-case this may require the application to execute a complete iteration before it can be halted. Even in the typical situation, it will require that many actors finish their execution before this step is completed. Hence the long worst-case and measured run-times for this step. The only option to reduce the time taken by step 3 would be to relax the constraint that an application can only be reconfigured at the end of an iteration. However, as discussed in Section 9.1, this could in the worst-case lead to a large overhead in migrating tokens when reconfiguring FIFOs. An experiment with a small test application have confirmed that this overhead for most realistic applications by far exceeds the worst-case time needed to complete an iteration of the graph. From this we concluded that the choice to allow reconfigurations only at an iteration boundary is still the best option.

9.4 CONCLUSIONS

In this chapter we introduced a run-time reconfiguration mechanism which provides timing guarantees on the time needed to migrate actors and their communication channels in a MPSoC. The mechanism guarantees that the reconfiguration of one application will not affect the timing behavior of other applications running on the same resources ensuring a composable behavior enables predictable and composable MPSoC reconfiguration. The proposed run-time mechanism is demonstrated on a realistic MPSoC (called CompSOC), which is running a JPEG decoder application. In our implementation we make a design choice of migrating the application only at the end of the iteration. However when relaxing this constraint, the amount of data that will be needed to be migrated will cause a very pessimistic prediction of the actor migration.

Task migration in a multi-processor system is performed for various reasons like thermal [45, 61], load-balancing [6], fault-tolerance [109] etc. We can classify the task migration work briefly in two types depending on the underlying architecture which is shared memory or distributed memory. We consider a scalable approach for multiprocessor with private memory for each processor, where the migration is done only at pre-decided checkpoints [15]. Taking

the advantage of SDF programming model, in our case the programmer need not explicitly store the context for migration.

A comprehensive survey of the run-time resource managers and their optimization strategies is provided in [90]. It divides the run-time resource manager into two parts; one which deals with optimization decision making while the second is responsible for implementation of this decision. Casavant et. al. in [25] provides with the classification of different optimization algorithms that can be used for resource manager. We use MMKP [113], which guarantees us predictability and flexibility. Owing to the private shared memory for each core, we implement the master-slave configuration of the resource manager, where the master is responsible for the decision making process, while the implementing the decision is performed collectively by all the responsible cores. This gives us an advantage of managing the data-structures in a simple way [90]. We belong to the (type 3) adaptive applications classified in [90], where applications are allocated certain resource budgets, and is responsible to manage them efficiently.

Master-slave configuration for task migration with copy of task in all processors is presented in [1], is similar to our work. However they require sufficiently large queue size between the communicating actors to avoid the deadline misses in the application during the task migration. We migrate the application only the end of iteration, in a predictable one time slot in order to avoid any deadline misses and disturbances to other applications.

Almeida et al. in [6] propose tasks migration to provide workload balance in multiprocessor system to optimize the throughput. There is no indication of amount of time required to migrate the task and its related resources. Further, their task migration is not capable of migrating stateful actors.

In[110], the author proposes locking of caches for hard-real time tasks which can provide predictable task migration. Their work focuses on cache based techniques, which become unscalable with increasing amount of processors on chip. Whereas we focus on private memory for each core, which has own instances of tasks running.

Hardware mechanisms [72, 96, 126] are proposed in order to offload the task of scheduling and migration to the hardware in order to save from the run-time overhead. However it is essentially a trade-off between the speed, flexibility and scalability of the system. Definitely, we do not outperform the hardware mechanisms, but our methodology gives a predictability and composability without requiring specialized hardware. In AsyMOS [88], a different approach is taken, where the system management functions are handled by dedicated cores. Ours is a flexible approach where we dedicate TDM slices in our framework for the management tasks.

In the next chapter we conclude this thesis and discuss the future work.

CONCLUSIONS AND FUTURE WORK

This chapter concludes the thesis and discusses about the possible future work.

IN this thesis, we used a constraint satisfaction approach to solve mapping and scheduling problems on multi-core architectures. This thesis can be seen as an elaboration and extension of work started in [79] which utilized multi-criteria optimization exploration algorithm to find efficient mapping and scheduling solutions for a multi-processor architecture. It experimented on relatively simple architectures and small problem sizes. In this thesis, we tried to overcome the limitations of this work and provide experiments on real platforms. We established symmetry breaking techniques for data-parallel computations which significantly improved the performance of the SMT solver and increased the size of the application model that could be solved. In this experiment we observed that task symmetry and processor symmetry applied in conjunction amplify the effect symmetry reduction in SMT. Either of them applied separately has a smaller effect on the reduction.

We used SMT solver to further perform mapping and scheduling of applications on the Kalray many-core architecture. The processors were grouped into clusters and connected by network-on-chip, communicate with each other via DMA. We used the DMA model in order to characterize the data transfers between different clusters. Our framework provided an automated way of performing distribution of tasks across the clusters and generating communication-aware models. These models were used to deploy the solution on the hardware platform. We verified the performance on the hardware platform and found it to be fairly accurate. We believe that with additional modeling of network route selection and contentions, we will be able to further reduce the error in performance prediction.

We also addressed the problem of optimization of DMA data transfer granularity for data-parallel applications with regular memory access patterns. We compared different methods by which the shared data could be exchanged between the processors in an efficient way. We built a model of the DMA mechanism with which we were able to successfully demonstrate a methodology to differentiate between the transfer and computation regimes, and how they affect the performance of the application. We experimented our formalization on the Cell processor architecture and proved it to be reasonably realistic considering the assumptions we made.

Further in this thesis, we demonstrated a necessity of a runtime manager on the multi-core platforms that chooses the configurations of the applications in execution in order to optimize resource-utilization globally. We discussed how an algorithm like MMKP can decide an optimal

configuration out of available Pareto set for an application and take a decision to migrate the tasks dynamically. We demonstrate how this application reconfiguration can be performed in a predictable (i.e. amount of time known at design time) and composable (i.e. without affecting other applications running on the platform) way. For this work, we performed experiments on a platform designed for predictability. We run experiments using the JPEG decoder application and prove that a predictable migration of application can be performed, in a composable way.

We believe that, this thesis can be further extended in many possible directions, some of which are discussed below.

- **Scheduling under variation in execution time**: In our case, we always consider the worst-case execution time for the actors (or tasks). However in reality the execution time between tasks of the same actor differ due to many reasons such as data-dependent execution. Predictions made this way are good when the range of execution times is small, but their quality degrades when variations become large. It will be interesting to incorporate this variation in execution time of tasks into the formulation of the constraints and costs. As we observe in the experiments that, the SMT solver is overwhelmed with increase in the problem size (number of tasks). This variation of execution time can cause additional burden on the solver. Probably it will involve support of special formalism such as probabilistic scheduling under uncertainty.

- **Splitting of actor over multiple clusters**: In our multi-stage design-flow for Kalray processor architecture, we assign an actor to a partition in the partitioning step. This makes the granularity of assignment equal to an actor. However, if an actor has a very high repetition count (for example 100 tasks or more), we could possibly split the actor between multiple clusters. Actor splitting will introduce additional decision variables and further overhead to the solver. This will also have an impact on the communication buffer memory and must be studied in detail.

- **Network route selection and communication scheduling**: In the mapping and scheduling on the Kalray processor, the second stage places the communicating partitions as close as possible. In reality the NoC is fast enough and the distance between clusters is a factor of less importance. We made an attempt by putting them close so that the network contentions are reduced to a minimal value, as the contentions can cause an error in performance prediction. We believe that if we perform network route selection and scheduling of communication, we will be able to eliminate this source of error completely. Finding optimal routes for communication is a hard problem and will need special encoding techniques to solve it using the SMT solver.

- **Combination of DMA transfer granularity results with split-join graphs**: We model the DMA transfer size optimization problem for application which have regular memory access pattern. It is possible to model such applications using split-join graph model. If we use such application model, then the DMA transfer size can affect the split-join factor of the application. As choosing a DMA transfer size implies combining of blocks for processing. It also means combining tasks which can actually execute in parallel. It will be interesting to see how the SMT solver can decide between data parallelism and data transfer granularity. Also extending split-join graphs to multiple dimensions and shared data are other related work directions.

- **Pipelined scheduling on the platform**: Streaming application have an another important constraint in addition to latency, called *throughput*. It emphasizes the number of graph iterations that can execute in a unit time. For meeting this requirement, multiple iterations of the application execute in parallel in a *pipelined* execution. A constraint formulation for pipelined scheduling will require additional variables, modulo arith-

metic operation and special mutual exclusion constraints and these clauses make the problem solving harder. We would like to experiment with such pipelined scheduling on and see it in action on the multi-processor hardware platform.

Currently we are in process of experimenting with three different encodings of the pipelined scheduling, a problem not so simple for SMT solvers [120]. For the moment we can solve typically problem upto size of 30-50 tasks. However, the benefits of pipeline parallelism for the performance justify such a restriction.

Contributions

- Selma Saidi, Pranav Tendulkar, Thierry Lepley, and Oded Maler. « Optimizing Explicit Data Transfers for Data Parallel Applications on the Cell Architecture ». In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 37:1–37:20. ISSN: 1544-3566. DOI: 10.1145/2086696.2086716. URL: http://doi.acm.org/10.1145/2086696.2086716

- Selma Saidi, Pranav Tendulkar, Thierry Lepley, and Oded Maler. « Optimal 2D Data Partitioning for DMA Transfers on MPSoCs ». In: *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*. DSD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 584–591. ISBN: 978-0-7695-4798-5. DOI: 10.1109/DSD.2012.99. URL: http://dx.doi.org/10.1109/DSD.2012.99

- Selma Saidi, Pranav Tendulkar, Thierry Lepley, and Oded Maler. « Optimizing two-dimensional DMA transfers for scratchpad Based MPSoCs platforms ». In: *Microprocessors and Microsystems* 37.8, Part A (2013). Special Issue DSD 2012 on Reliability and Dependability in MPSoC Technologies, pp. 848 –857. ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2013.04.006. URL: http://www.sciencedirect.com/science/article/pii/S0141933113000549

- Pranav Tendulkar, Peter Poplavko, and Oded Maler. « Symmetry Breaking for Multi-criteria Mapping and Scheduling on Multicores ». In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Víctor Braberman and Laurent Fribourg. Vol. 8053. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 228–242. ISBN: 978-3-642-40228-9. DOI: 10.1007/978-3-642-40229-6_16. URL: http://dx.doi.org/10.1007/978-3-642-40229-6_16

- Pranav Tendulkar and Sander Stuijk. « A Case Study into Predictable and Composable MPSoC Reconfiguration ». In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. 2013, pp. 293–300. DOI: 10.1109/IPDPSW.2013.12

- Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. « Many-Core Scheduling of Data Parallel Applications using SMT Solvers ». In: *Proceedings of the 2014 17th Euromicro Conference on Digital System Design*. DSD '14. 2014

- Pranav Tendulkar, Peter Poplavko, and Oded Maler. *Strictly Periodic Scheduling of Acyclic Synchronous Dataflow Graphs using SMT Solvers*. Tech. rep. TR-2014-5. Verimag Research Report, 2014

Schedule XML

An example of schedule XML for JPEG decoder application, generated by StreamExplorer for a schedule shown in Figure A.1.

Listing A.1 – Schedule XML for JPEG decoder on single cluster

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
 <!-- Application Structure -->
 <!-- Actor description -->
    <graphActors size="3">
        <actor function="VLD" instances="1" name="VLD" numPorts="1"/>
        <actor function="IQ" instances="12" name="IQ" numPorts="2"/>
        <actor function="COLOR" instances="12" name="COLOR" numPorts="1"/>
    </graphActors>
 <!-- Channel description -->
    <graphChannels size="2">
        <channel channelSize="12" dstActor="COLOR" dstPort="0" dstPortRate="3"
            name="IQ2COL" srcActor="IQ" srcPort="1" srcPortRate="3" tokenSize="
            76"/>
        <channel channelSize="36" dstActor="IQ" dstPort="0" dstPortRate="3"
            name="VLD2IQ" srcActor="VLD" srcPort="0" srcPortRate="36" tokenSize
            ="268"/>
    </graphChannels>
 <!-- Schedule Description -->
    <schedule size="1">
  <!-- Cluster Allocation -->
        <cluster id="0" size="4">
  <!-- Processor Allocation -->
            <processor id="0" size="7">
                <actor instance="0" name="VLD"/>
                <actor instance="0" name="IQ"/>
                <actor instance="0" name="COLOR"/>
                <actor instance="4" name="IQ"/>
                <actor instance="4" name="COLOR"/>
                <actor instance="8" name="IQ"/>
                <actor instance="8" name="COLOR"/>
            </processor>
            <processor id="1" size="6">
                <actor instance="1" name="IQ"/>
                <actor instance="1" name="COLOR"/>
                <actor instance="5" name="IQ"/>
                <actor instance="5" name="COLOR"/>
```

```
34                    <actor instance="9" name="IQ"/>
35                    <actor instance="9" name="COLOR"/>
36               </processor>
37               <processor id="2" size="6">
38                    <actor instance="2" name="IQ"/>
39                    <actor instance="2" name="COLOR"/>
40                    <actor instance="6" name="IQ"/>
41                    <actor instance="6" name="COLOR"/>
42                    <actor instance="10" name="IQ"/>
43                    <actor instance="10" name="COLOR"/>
44               </processor>
45               <processor id="3" size="6">
46                    <actor instance="3" name="IQ"/>
47                    <actor instance="3" name="COLOR"/>
48                    <actor instance="7" name="IQ"/>
49                    <actor instance="7" name="COLOR"/>
50                    <actor instance="11" name="IQ"/>
51                    <actor instance="11" name="COLOR"/>
52               </processor>
53          </cluster>
54      </schedule>
55  <!-- FIFO Allocation -->
56      <fifoallocation size="2">
57          <fifo dstCluster="0" fifoSize="12" name="IQ2COL" srcCluster="0"/>
58          <fifo dstCluster="0" fifoSize="36" name="VLD2IQ" srcCluster="0"/>
59      </fifoallocation>
60  </mapping>
```
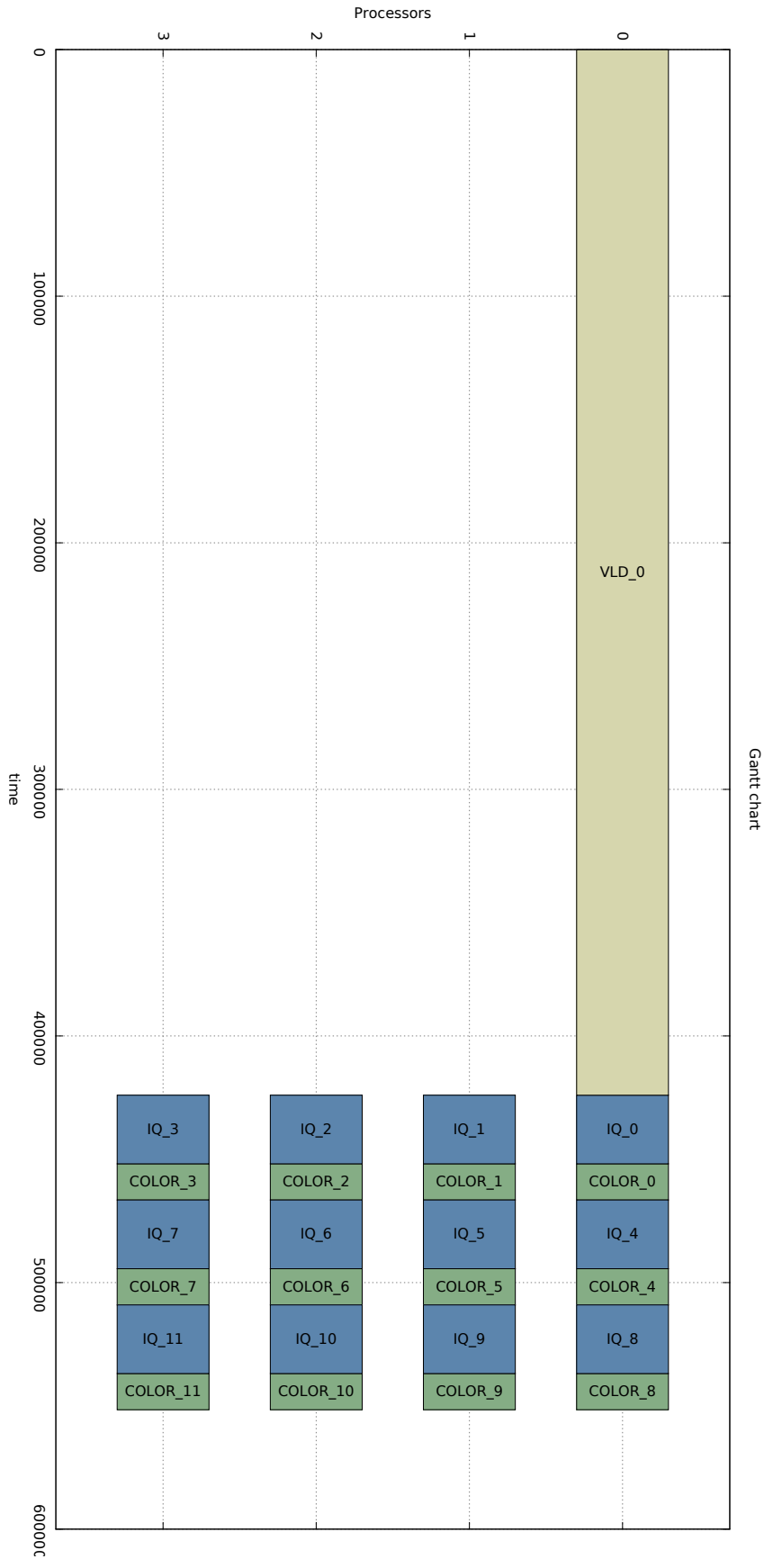
Figure A.1 – Gantt chart for schedule XML

Bibliography

[1]   Andrea Acquaviva et al. « Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications ». In: *EURASIP Journal on Embedded Systems* 2008.1 (2008), p. 518904.

[2]   A Agarwal, D.A Kranz, and V. Natarajan. « Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors ». In: *Parallel and Distributed Systems, IEEE Transactions on* 6.9 (1995), pp. 943–962. ISSN: 1045-9219. DOI: 10.1109/71.466632.

[3]   Ankur Agarwal, Cyril Iskander, and Ravi Shankar. « Survey of network on chip (noc) architectures & contributions ». In: *Journal of engineering, Computing and Architecture* 3.1 (2009), pp. 21–27.

[4]   B. Akesson et al. « Composability and Predictability for Independent Application Development, Verification, and Execution ». In: *Multiprocessor System-on-Chip*. Ed. by M. Hübner and J. Becker. Circuits & Systems. London: Springer Verlag, 2010, pp. 25–56.

[5]   B. Akesson et al. « Virtual Platforms for Mixed Time-Criticality Applications: The CoMPSoC Architecture and SDF3 Design Flow ». In: *Proceedings of workshop on Quo Vadis, Virtual Platforms? Challenges and Solutions for Today and Tomorrow*. 2012.

[6]   Gabriel Marchesan Almeida et al. « Evaluating the impact of task migration in multi-processor systems-on-chip ». In: *Proceedings of the 23rd symposium on Integrated circuits and system design*. SBCCI '10. ACM, 2010, pp. 73–78.

[7]   Turgay Altilar and Yakup Paker. « Minimum Overhead Data Partitioning Algorithms for Parallel Video Processing ». In: *Proceedings Domain Decomposition Methods Conference*. 2001, pp. 25125–8.

[8]   Alper Atamtürk and Martin WP Savelsbergh. « Integer-programming software systems ». In: *Annals of Operations Research* 140.1 (2005), pp. 67–124.

[9]   P. Axer et al. « Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints ». In: *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. 2013, pp. 215–224. DOI: 10.1109/ECRTS.2013.31.

[10] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling*. Kluwer international series in engineering and computer science. Kluwer, 2001.

[11] Clark W Barrett et al. « Satisfiability Modulo Theories. » In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[12] Christopher Batten. *ECE 5950 Complex Digital ASIC Design Course Overview*. URL: http://www.csl.cornell.edu/courses/ece5950.

[13] Luca Benini and Giovanni De Micheli. « Networks on chips: a new SoC paradigm ». In: *Computer* 35.1 (2002), pp. 70–78. ISSN: 0018-9162. DOI: 10.1109/2.976921.

[14]   Luca Benini et al. « P2012: Building an Ecosystem for a Scalable, Modular and
       High-efficiency Embedded Computing Accelerator ». In: *Proceedings of the Confer-
       ence on Design, Automation and Test in Europe*. DATE '12. Dresden, Germany: EDA
       Consortium, 2012, pp. 983–987. ISBN: 978-3-9810801-8-6. URL: http://dl.acm.org/
       citation.cfm?id=2492708.2492954.

[15]   Stefano Bertozzi et al. « Supporting task migration in multi-processor systems-on-
       chip: a feasibility study ». In: *Proceedings of the conference on Design, automation and
       test in Europe: Proceedings*. DATE '06. EDAA, 2006, pp. 15–20.

[16]   B. Bhattacharya and S.S. Bhattacharyya. « Parameterized dataflow modeling for
       DSP systems ». In: *Signal Processing, IEEE Transactions on* 49.10 (2001), pp. 2408–
       2421. ISSN: 1053-587X. DOI: 10.1109/78.950795.

[17]   Shuvra S. Bhattacharyya, EdF. Deprettere, and BartD. Theelen. « Dynamic Dataflow
       Graphs ». In: *Handbook of Signal Processing Systems*. Ed. by Shuvra S. Bhattacharyya
       et al. Springer New York, 2013, pp. 905–944. ISBN: 978-1-4614-6858-5. DOI: 10.
       1007/978-1-4614-6859-2_28. URL: http://dx.doi.org/10.1007/978-1-4614-6859-
       2_28.

[18]   G. Bilsen et al. « Cycle-static dataflow ». In: *Signal Processing, IEEE Transactions on*
       44.2 (1996), pp. 397–408. ISSN: 1053-587X. DOI: 10.1109/78.485935.

[19]   Tobias Bjerregaard and Shankar Mahadevan. « A Survey of Research and Practices
       of Network-on-chip ». In: *ACM Comput. Surv.* 38.1 (June 2006). ISSN: 0360-0300.
       DOI: 10.1145/1132952.1132953. URL: http://doi.acm.org/http://doi.acm.org/10.
       1145/1132952.1132953.

[20]   Robert D. Blumofe et al. « Cilk: An Efficient Multithreaded Runtime System ».
       In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice
       of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: ACM, 1995,
       pp. 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958. URL: http://doi.
       acm.org/10.1145/209936.209958.

[21]   Alessio Bonfietti et al. « A Constraint Based Approach to Cyclic RCPSP ». In: *Princi-
       ples and Practice of Constraint Programming – CP 2011*. Ed. by Jimmy Lee. Vol. 6876.
       Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 130–
       144. ISBN: 978-3-642-23785-0. DOI: 10.1007/978-3-642-23786-7_12. URL: http:
       //dx.doi.org/10.1007/978-3-642-23786-7_12.

[22]   Alessio Bonfietti et al. « An efficient and complete approach for throughput-maximal
       SDF allocation and scheduling on multi-core platforms ». In: *DATE*. DATE. Dres-
       den, Germany: IEEE, 2010, pp. 897–902. ISBN: 978-3-9810801-6-2. URL: http://dl.
       acm.org/citation.cfm?id=1870926.1871143.

[23]   J.T. Buck and E.A. Lee. « Scheduling dynamic dataflow graphs with bounded mem-
       ory using the token flow model ». In: *Acoustics, Speech, and Signal Processing, 1993.
       ICASSP-93., 1993 IEEE International Conference on*. Vol. 1. 1993, 429–432 vol.1. DOI:
       10.1109/ICASSP.1993.319147.

[24]   Bill Carlson et al. *Programming in the partitioned global address space model*. Novem-
       ber 2003. URL: http://upc.gwu.edu.

[25]   T.L. Casavant and J.G. Kuhl. « A taxonomy of scheduling in general-purpose dis-
       tributed computing systems ». In: *Software Engineering, IEEE Transactions on* 14.2
       (1988), pp. 141 –154.

[26]   B. Chapman et al. « Implementing OpenMP on a high performance embedded multicore MPSoC ». In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161107.

[27]   Tien-Fu Chen and Jean-Loup Baer. « A performance study of software and hardware data prefetching schemes ». In: *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*. 1994, pp. 223–232. DOI: 10.1109/ISCA.1994.288147.

[28]   Byn Choi et al. « DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism ». In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. 2011, pp. 155–166. DOI: 10.1109/PACT.2011.21.

[29]   F. Commoner et al. « Marked Directed Graphs ». In: *J. Comput. Syst. Sci.* 5.5 (Oct. 1971). DOI: 10.1016/S0022-0000(71)80013-2. URL: http://dx.doi.org/10.1016/S0022-0000(71)80013-2.

[30]   S. Cotton et al. « Multi-criteria optimization for mapping programs to multiprocessors ». In: *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. 2011, pp. 9–17. DOI: 10.1109/SIES.2011.5953650.

[31]   L. Dagum and R. Menon. « OpenMP: an industry standard API for shared-memory programming ». In: *Computational Science Engineering, IEEE* 5.1 (1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313.

[32]   Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. « Faster Symmetry Discovery Using Sparsity of Symmetries ». In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: ACM, 2008, pp. 149–154. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391509. URL: http://doi.acm.org/10.1145/1391469.1391509.

[33]   Tatjana Davidovi. *Mathematical programming-based approach to scheduling of communicating tasks*. GERAD, HEC Montréal, 2004.

[34]   Martin Davis, George Logemann, and Donald Loveland. « A machine program for theorem-proving ». In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[35]   Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Chichester, New York: John Wiley & Sons, 2001. URL: http://www.bibsonomy.org/bibtex/29f7d242bb8c221e5afca03dd32d1de4c/lcschoening.

[36]   Benoît Dupont de Dinechin et al. « A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor ». In: *Procedia Computer Science* 18.0 (2013). 2013 International Conference on Computational Science, pp. 1654–1663. ISSN: 1877-0509. DOI: 10.1016/j.procs.2013.05.333. URL: http://www.sciencedirect.com/science/article/pii/S1877050913004766.

[37]   Bruno Dutertre and Leonardo De Moura. « The yices smt solver ». In: *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2 (2006), p. 2.

[38]   Matthias Ehrgott. *Multicriteria optimization*. Springer, 2005.

[39]   C. A. J. van Eijk et al. « Identification and Exploitation of Symmetries in DSP Algorithms ». In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '99. Munich, Germany: ACM, 1999. ISBN: 1-58113-121-6. DOI: 10.1145/307418.307572. URL: http://doi.acm.org/10.1145/307418.307572.

[40]   M. Engels et al. « Cycle-static dataflow: model and implementation ». In: *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*. Vol. 1. 1994, 503–507 vol.1. DOI: 10.1109/ACSSC.1994.471504.

[41] J. Figueira, S. Greco, and M. Ehrgott. *Multiple criteria decision analysis: state of the art surveys*. Vol. 78. Springer Verlag, 2005, pp. 445–470. URL: http://books.google.com/books?hl=en&http://www.bibsonomy.org/bibtex/22952817899439e6150a557f21104e2e2/kamil205.

[42] P. Fradet, A. Girault, and P. Poplavko. « SPDF: A schedulable parametric data-flow MoC ». In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. 2012, pp. 769–774. DOI: 10.1109/DATE.2012.6176572.

[43] Alessio Franceschelli et al. « MPOpt-Cell: A High-performance Data-flow Programming Environment for the CELL BE Processor ». In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. CF '11. Ischia, Italy: ACM, 2011, 11:1–11:2. ISBN: 978-1-4503-0698-0. DOI: 10.1145/2016604.2016618. URL: http://doi.acm.org/10.1145/2016604.2016618.

[44] Rosario G. Garroppo, Stefano Giordano, and Luca Tavanti. « A Survey on Multi-constrained Optimal Path Computation: Exact and Approximate Algorithms ». In: *Comput. Netw.* 54.17 (Dec. 2010), pp. 3081–3107. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.017. URL: http://dx.doi.org/10.1016/j.comnet.2010.05.017.

[45] Yang Ge, Parth Malani, and Qinru Qiu. « Distributed task migration for thermal management in many-core systems ». In: *Proceedings of the 47th Design Automation Conference*. DAC '10. ACM, 2010, pp. 579–584.

[46] Marc Geilen and Twan Basten. « Requirements on the Execution of Kahn Process Networks ». In: *Programming Languages and Systems*. Ed. by Pierpaolo Degano. Vol. 2618. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 319–334. ISBN: 978-3-540-00886-6. DOI: 10.1007/3-540-36575-3_22. URL: http://dx.doi.org/10.1007/3-540-36575-3_22.

[47] Marc Geilen, Twan Basten, and Sander Stuijk. « Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking ». In: *Proceedings of the 42Nd Annual Design Automation Conference*. DAC '05. Anaheim, California, USA: ACM, 2005, pp. 819–824. ISBN: 1-59593-058-2. DOI: 10.1145/1065579.1065796. URL: http://doi.acm.org/10.1145/1065579.1065796.

[48] Kees Goossens and Andreas Hansson. « The aethereal network on chip after ten years: goals, evolution, lessons, and future ». In: *Proceedings of the 47th Design Automation Conference*. DAC '10. ACM, 2010, pp. 306–311.

[49] Michael I. Gordon. « Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures ». Ph.D. Thesis. Cambridge, MA: Massachusetts Institute of Technology, 2010. URL: http://groups.csail.mit.edu/commit/papers/10/mgordon-phd-thesis.pdf.

[50] R. Govindarajan, Guang R. Gao, and Palash Desai. « Minimizing Buffer Requirements Under Rate-Optimal Schedule in Regular Dataflow Networks ». In: *J. VLSI Signal Process. Syst.* 31.3 (July 2002), pp. 207–229. ISSN: 0922-5773. DOI: 10.1023/A:1015452903532. URL: http://dx.doi.org/10.1023/A:1015452903532.

[51] Khronos OpenCL Working Group et al. « The OpenCL Specification, Version 1.1, 2010 ». In: *Document Revision* 44 ().

[52] Michael Gschwind. « Chip multiprocessing and the cell broadband engine ». In: *Proceedings of the 3rd conference on Computing frontiers*. CF '06. Ischia, Italy: ACM, 2006, pp. 1–8. ISBN: 1-59593-302-6. DOI: 10.1145/1128022.1128023. URL: http://doi.acm.org/10.1145/1128022.1128023.

[53] Michael Gschwind et al. « Synergistic Processing in Cell's Multicore Architecture ». In: *IEEE Micro* 26.2 (Mar. 2006), pp. 10–24. ISSN: 0272-1732. DOI: 10.1109/MM.2006.41. URL: http://dx.doi.org/10.1109/MM.2006.41.

[54] Andreas Hansson et al. « Design and implementation of an operating system for composable processor sharing ». In: *Microprocessors and Microsystems* 35.2 (2011). Special issue on Network-on-Chip Architectures and Design Methodologies, pp. 246–260.

[55] Marcus Josephus Maria Heijligers. « The application of genetic algorithms to high-level synthesis ». In: (1996).

[56] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[57] J. Howard et al. « A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS ». In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077.

[58] IBM. *Cell SDK 3.1*. https://www.ibm.com/developerworks/power/cell/. 2008.

[59] IBM. *Cell Simulator*. http://www.alphaworks.ibm.com/tech/cellsystemsim. Version 3.1. 2009.

[60] J. Jaffar and J.-L. Lassez. « Constraint Logic Programming ». In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: ACM, 1987, pp. 111–119. ISBN: 0-89791-215-2. DOI: 10.1145/41625.41635. URL: http://doi.acm.org/10.1145/41625.41635.

[61] J. Jahn, M.A.A. Faruque, and J. Henkel. « CARAT: Context-aware runtime adaptive task migration for multi core architectures ». In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 2011, pp. 1–6.

[62] C. R. Johns and D. A. Brokenshire. « Introduction to the Cell Broadband Engine Architecture ». In: *IBM J. Res. Dev.* 51.5 (Sept. 2007), pp. 503–519. ISSN: 0018-8646. DOI: 10.1147/rd.515.0503. URL: http://dx.doi.org/10.1147/rd.515.0503.

[63] Roel Jordans et al. « An Automated Flow to Map Throughput Constrained Applications to a MPSoC ». In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Ed. by Philipp Lucas et al. Vol. 18. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 47–58. ISBN: 978-3-939897-28-6. DOI: 10.4230/OASIcs.PPES.2011.47. URL: http://drops.dagstuhl.de/opus/volltexte/2011/3081.

[64] G. Kahn. « The Semantics of a Simple Language for Parallel Programming ». In: *Information Processing '74: Proceedings of the IFIP Congress*. Ed. by J. L. Rosenfeld. New York, NY: North-Holland, 1974, pp. 471–475.

[65] Kalray. *Kalray MPPA 256*. URL: http://www.kalray.eu/.

[66] Shin-Haeng Kang et al. « Multi-objective mapping optimization via problem decomposition for many-core systems ». In: *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*. IEEE. 2012, pp. 28–37.

[67] Richard M Karp and Rayamond E Miller. « Properties of a model for parallel computations: Determinacy, termination, queueing ». In: *SIAM Journal on Applied Mathematics* 14.6 (1966), pp. 1390–1411.

[68] J. Keinert, C. Haubelt, and J. Teich. « Modeling and Analysis of Windowed Synchronous Algorithms ». In: *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*. Vol. 3. 2006, pp. III–III. DOI: 10.1109/ICASSP.2006.1660798.

[69] J.H. Kelm et al. « Cohesion: An Adaptive Hybrid Memory Model for Accelerators ». In: *Micro, IEEE* 31.1 (2011), pp. 42–55. ISSN: 0272-1732. DOI: 10.1109/MM.2011.8.

[70] Bart Kienhuis et al. « A Methodology to Design Programmable Embedded Systems ». In: *Embedded Processor Design Challenges*. Ed. by EdF. Deprettere, Jürgen Teich, and Stamatis Vassiliadis. Vol. 2268. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 18–37. ISBN: 978-3-540-43322-4. DOI: 10.1007/3-540-45874-3_2. URL: http://dx.doi.org/10.1007/3-540-45874-3_2.

[71] M. Kistler, M. Perrone, and F. Petrini. « Cell Multiprocessor Communication Network: Built for Speed ». In: *Micro, IEEE* 26.3 (May 2006), pp. 10–23. ISSN: 0272-1732. DOI: 10.1109/MM.2006.49.

[72] T. Klevin. « Get RealFast RTOS with Xilinx FPGAs ». In: ().

[73] Abdullah Konak, David W. Coit, and Alice E. Smith. « Multi-objective optimization using genetic algorithms: A tutorial. » In: *Rel. Eng. & Sys. Safety* 91.9 (2006), pp. 992–1007. URL: http://dblp.uni-trier.de/db/journals/ress/ress91.html#KonakCS06;http://dx.doi.org/10.1016/j.ress.2005.11.018;http://www.bibsonomy.org/bibtex/28a293983a13e46a6acf0b157c05e85cb/dblp.

[74] Manjunath Kudlur and Scott Mahlke. « Orchestrating the Execution of Stream Programs on Multicore Platforms ». In: *SIGPLAN Not.* 43.6 (June 2008), pp. 114–124. ISSN: 0362-1340. DOI: 10.1145/1379022.1375596. URL: http://doi.acm.org/10.1145/1379022.1375596.

[75] Manjunath V Kudlur. *Streamroller: A Unified Compilation and Synthesis System for Streaming Applications*. ProQuest, 2008.

[76] Fadi J Kurdahi and Alice C Parker. « REAL: a program for REgister ALlocation ». In: *Proceedings of the 24th ACM/IEEE Design Automation Conference*. ACM. 1987, pp. 210–215.

[77] Chi-kin Lee and Mounir Hamdi. « Parallel Image Processing Applications on a Network of Workstations ». In: *Parallel Comput.* 21.1 (Jan. 1995), pp. 137–160. ISSN: 0167-8191. DOI: 10.1016/0167-8191(94)00068-L. URL: http://dx.doi.org/10.1016/0167-8191(94)00068-L.

[78] E.A. Lee and D.G. Messerschmitt. « Synchronous data flow ». In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[79] Julien Legriel and Oded Maler. « Meeting Deadlines Cheaply ». In: *ECRTS*. IEEE, 2011, pp. 185–194.

[80] Julien Legriel et al. « Approximating the Pareto Front of Multi-criteria Optimization Problems ». In: *TACAS*. Ed. by Javier Esparza and Rupak Majumdar. Vol. 6015. LNCS. Springer, 2010, pp. 69–83. ISBN: 978-3-642-12001-5.

[81] Thierry Lepley, Pierre Paulin, and Eric Flamand. « A Novel Compilation Approach for Image Processing Graphs on a Many-Core Platform with Explicitly Managed Memory ». In: *Proceedings of the IEEE* (2013).

[82]   Mirko Loghi, Massimo Poncino, and Luca Benini. « Cache Coherence Tradeoffs in Shared-memory MPSoCs ». In: *ACM Trans. Embed. Comput. Syst.* 5.2 (May 2006), pp. 383–407. ISSN: 1539-9087. DOI: 10.1145/1151074.1151081. URL: http://doi.acm.org/10.1145/1151074.1151081.

[83]   Andrea Marongiu and Luca Benini. « Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy ». In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 809–814. ISBN: 978-3-9810801-5-5. URL: http://dl.acm.org/citation.cfm?id=1874620.1874819.

[84]   J.P. Marques-Silva and K.A. Sakallah. « GRASP: a search algorithm for propositional satisfiability ». In: *Computers, IEEE Transactions on* 48.5 (1999), pp. 506–521. ISSN: 0018-9340. DOI: 10.1109/12.769433.

[85]   Cupertino Miranda et al. « Erbium: A Deterministic, Concurrent Intermediate Representation to Map Data-flow Tasks to Scalable, Persistent Streaming Processes ». In: *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '10. Scottsdale, Arizona, USA: ACM, 2010, pp. 11–20. ISBN: 978-1-60558-903-9. DOI: 10.1145/1878921.1878924. URL: http://doi.acm.org/10.1145/1878921.1878924.

[86]   Matthew W. Moskewicz et al. « Chaff: Engineering an Efficient SAT Solver ». In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379017. URL: http://doi.acm.org/10.1145/378239.379017.

[87]   Leonardo Mendonça de Moura and Nikolaj Bjørner. « Z3: An Efficient SMT Solver. » In: *TACAS*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. URL: http://dblp.uni-trier.de/db/conf/tacas/tacas2008.html#MouraB08;http://dx.doi.org/10.1007/978-3-540-78800-3_24;http://www.bibsonomy.org/bibtex/23964b30f7dd5e5f7133e45828935ac10/kaptoxic.

[88]   S. Muir and J. Smith. « AsyMOS-an asymmetric multiprocessor operating system ». In: *Open Architectures and Network Programming, 1998 IEEE*. 1998, pp. 25 –34.

[89]   P.K. Murthy and E.A. Lee. « Multidimensional synchronous dataflow ». In: *Signal Processing, IEEE Transactions on* 50.8 (2002), pp. 2064–2079. ISSN: 1053-587X. DOI: 10.1109/TSP.2002.800830.

[90]   Vincent Nollet, Diederik Verkest, and Henk Corporaal. « A Safari Through the MPSoC Run-Time Management Jungle ». In: *J. Signal Process. Syst.* 60.2 (Aug. 2010), pp. 251–268.

[91]   Henri J. Nussbaumer. *Fast Fourier transform and convolution algorithms*. Springer-Verlag, Berlin ; New York : 1981, x, 248 p. : ISBN: 0387101594 3540101594 0387101594.

[92]   CUDA Nvidia. « Compute unified device architecture programming guide ». In: (2007).

[93]   Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997. ISBN: 9781558603394.

[94]   Vilfredo Pareto. « Cours d\'economie politique ». In: (1896).

[95]   Thomas Martyn Parks. « Bounded Scheduling of Process Networks ». UMI Order No. GAX96-21312. PhD thesis. Berkeley, CA, USA, 1995.

[96] Pierre G. Paulin et al. « Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management ». In: *Proceedings of the international conference on Hardware/Software Codesign and System Synthesis: 2004*. CODES+ISSS '04. IEEE Computer Society, 2004, pp. 48–53.

[97] Lu Peng et al. « Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study ». In: *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*. 2007, pp. 55–64. DOI: 10.1109/PCCC.2007.358879.

[98] P. Poplavko et al. « Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-chip ». In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '03. San Jose, California, USA: ACM, 2003, pp. 63–72. ISBN: 1-58113-676-5. DOI: 10.1145/951710.951721. URL: http://doi.acm.org/10.1145/951710.951721.

[99] Shiv Prakash and Alice C Parker. « SOS: Synthesis of application-specific heterogeneous multiprocessor systems ». In: *Journal of Parallel and Distributed computing* 16.4 (1992), pp. 338–351.

[100] RM Ramanathan. « Intel Multi-core Processors: Making the move to Quad-core and Beyond, white paper ». In: *Intel Corporation* ().

[101] A Ramani et al. « Breaking instance-independent symmetries in exact graph coloring ». In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. Vol. 1. 2004, 324–329 Vol.1. DOI: 10.1109/DATE.2004.1268868.

[102] IBM Redbooks. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. Vervante, 2008. ISBN: 0738485942, 9780738485942.

[103] Selma Saidi. « Optimizing DMA Data Transfers for Embedded Multi-Cores ». In: *Phd thesis*. 2012.

[104] Selma Saidi et al. « Optimal 2D Data Partitioning for DMA Transfers on MPSoCs ». In: *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*. DSD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 584–591. ISBN: 978-0-7695-4798-5. DOI: 10.1109/DSD.2012.99. URL: http://dx.doi.org/10.1109/DSD.2012.99.

[105] Selma Saidi et al. « Optimizing Explicit Data Transfers for Data Parallel Applications on the Cell Architecture ». In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 37:1–37:20. ISSN: 1544-3566. DOI: 10.1145/2086696.2086716. URL: http://doi.acm.org/10.1145/2086696.2086716.

[106] Selma Saidi et al. « Optimizing two-dimensional DMA transfers for scratchpad Based MPSoCs platforms ». In: *Microprocessors and Microsystems* 37.8, Part A (2013). Special Issue DSD 2012 on Reliability and Dependability in MPSoC Technologies, pp. 848 –857. ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2013.04.006. URL: http://www.sciencedirect.com/science/article/pii/S0141933113000549.

[107] J.C. Sancho et al. « Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications ». In: *SC 2006 Conference, Proceedings of the ACM/IEEE*. 2006, pp. 17–17. DOI: 10.1109/SC.2006.51.

[108] Jose Carlos Sancho and Darren J. Kerbyson. « Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE ». In: *Parallel and Distributed Processing Symposium, International* 0 (2008), pp. 1–12. DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2008.4536316.

[109] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. « Task migration for fault-tolerance in mixed-criticality embedded systems ». In: *SIGBED Rev.* 6.3 (Oct. 2009), 6:1–6:5.

[110] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. « Predictable task migration for locked caches in multi-core systems ». In: *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems.* LCTES '11. ACM, 2011, pp. 131–140.

[111] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[112] Tae-ho Shin, Hyunok Oh, and Soonhoi Ha. « Minimizing Buffer Requirements for Throughput Constrained Parallel Execution of Synchronous Dataflow Graph ». In: *Proceedings of the 16th Asia and South Pacific Design Automation Conference.* ASPDAC '11. Yokohama, Japan: IEEE Press, 2011, pp. 165–170. ISBN: 978-1-4244-7516-2. URL: http://dl.acm.org/citation.cfm?id=1950815.1950860.

[113] Hamid Shojaei et al. « A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management ». In: *Proceedings of the 46th Annual Design Automation Conference.* DAC '09. ACM, 2009, pp. 917–922.

[114] Lawrence Snyder. *The ZPL Programmer's Guide. Scientific and Engineering Computation.* March 1999.

[115] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Second Edition. CRC Press, 2012. ISBN: 1420048023, 9781420048025.

[116] S. Stuijk et al. « Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications ». In: *Embedded Computer Systems (SAMOS), 2011 International Conference on.* 2011, pp. 404–411. DOI: 10.1109/SAMOS.2011.6045491.

[117] Sander Stuijk. « Predictable Mapping of Streaming Applications on Multiprocessors ». In: *Phd thesis.* 2007.

[118] Sander Stuijk, Marc Geilen, and Twan Basten. « SDF³: SDF For Free ». In: *Application of Concurrency to System Design 6th International Conference ACSD 2006 Proceedings.* Turku Finland: IEEE Computer Society Press Los Alamitos CA USA, 2006, pp. 276–278. DOI: 10.1109/ACSD.2006.23. URL: http://www.es.ele.tue.nl/sdf3.

[119] Herb Sutter. « The free lunch is over: A fundamental turn toward concurrency in software ». In: *Dr. Dobb's Journal* 30.3 (2005), pp. 202–210.

[120] Pranav Tendulkar, Peter Poplavko, and Oded Maler. *Strictly Periodic Scheduling of Acyclic Synchronous Dataflow Graphs using SMT Solvers.* Tech. rep. TR-2014-5. Verimag Research Report, 2014.

[121] Pranav Tendulkar, Peter Poplavko, and Oded Maler. « Symmetry Breaking for Multi-criteria Mapping and Scheduling on Multicores ». In: *Formal Modeling and Analysis of Timed Systems.* Ed. by Víctor Braberman and Laurent Fribourg. Vol. 8053. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 228–242. ISBN: 978-3-642-40228-9. DOI: 10.1007/978-3-642-40229-6_16. URL: http://dx.doi.org/10.1007/978-3-642-40229-6_16.

[122] Pranav Tendulkar and Sander Stuijk. « A Case Study into Predictable and Composable MPSoC Reconfiguration ». In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International.* 2013, pp. 293–300. DOI: 10.1109/IPDPSW.2013.12.

[123]   Pranav Tendulkar et al. « Fine-grain OpenMP runtime support with explicit com-
        munication hardware primitives ». In: *Design*, *Automation Test in Europe Conference
        Exhibition (DATE)*, *2011*. 2011, pp. 1–4. DOI: 10.1109/DATE.2011.5763299.

[124]   Pranav Tendulkar et al. « Many-Core Scheduling of Data Parallel Applications
        using SMT Solvers ». In: *Proceedings of the 2014 17th Euromicro Conference on Digital
        System Design*. DSD '14. 2014.

[125]   E. Teruel et al. « On weighted T-systems ». In: *Application and Theory of Petri Nets
        1992*. Ed. by K. Jensen. Vol. 616. Lecture Notes in Computer Science. Springer
        Berlin Heidelberg, 1992, pp. 348–367. ISBN: 978-3-540-55676-3. DOI: 10.1007/3-
        540-55676-1_20. URL: http://dx.doi.org/10.1007/3-540-55676-1_20.

[126]   B. D. Theelen et al. « A scalable single-chip multi-processor architecture with on-
        chip RTOS kernel ». In: *J. Syst. Archit.* 49.12-15 (Dec. 2003), pp. 619–639.

[127]   William Thies. « Language and compiler support for stream programs ». PhD thesis.
        Massachusetts Institute of Technology, 2009.

[128]   William Thies and Saman Amarasinghe. « An empirical characterization of stream
        programs and its implications for language and compiler design ». In: *international
        conference on Parallel architectures and compilation techniques*. PACT '10. Vienna,
        Austria: ACM, 2010, pp. 365–376. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.
        1854319. URL: http://doi.acm.org/10.1145/1854273.1854319.

[129]   Tilera, LTD. *Tilera TILE64 processor*. URL: http://www.tilera.com/.

[130]   Saud Wasly and Rodolfo Pellizzoni. « A Dynamic Scratchpad Memory Unit for Pre-
        dictable Real-Time Embedded Systems ». In: *Proceedings of the 2013 25th Euromicro
        Conference on Real-Time Systems*. ECRTS '13. Washington, DC, USA: IEEE Com-
        puter Society, 2013, pp. 183–192. ISBN: 978-0-7695-5054-1. DOI: 10.1109/ECRTS.
        2013.28. URL: http://dx.doi.org/10.1109/ECRTS.2013.28.

[131]   M.H. Wiggers, M.J.G. Bekooij, and G. J M Smit. « Efficient Computation of Buffer
        Capacities for Cyclo-Static Dataflow Graphs ». In: *Design Automation Conference*,
        2007. DAC '07. 44th ACM/IEEE. 2007, pp. 658–663.

[132]   W. Wolf, A.A. Jerraya, and G. Martin. « Multiprocessor System-on-Chip (MPSoC)
        Technology ». In: *Computer-Aided Design of Integrated Circuits and Systems*, *IEEE
        Transactions on* 27.10 (2008), pp. 1701–1713. ISSN: 0278-0070. DOI: 10.1109/TCAD.
        2008.923415.

[133]   Yang Yang et al. « Exploring trade-offs between performance and resource require-
        ments for synchronous dataflow graphs ». In: *Embedded Systems for Real-Time Mul-
        timedia*, *2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*. 2009, pp. 96–105.
        DOI: 10.1109/ESTMED.2009.5336821.

[134]   Jun Zhu, Ingo Sander, and Axel Jantsch. « Buffer minimization of real-time stream-
        ing applications scheduling on hybrid CPU/FPGA architectures ». In: DATE. Nice,
        France: IEEE, 2009, pp. 1506–1511. ISBN: 978-3-9810801-5-5. URL: http://dl.acm.
        org/citation.cfm?id=1874620.1874980.

[135]   Jun Zhu, Ingo Sander, and Axel Jantsch. « Constrained global scheduling of stream-
        ing applications on MPSoCs ». In: *ASPDAC*. 2010.

[136]   C. Zinner and W. Kubinger. « ROS-DMA: A DMA Double Buffering Method for
        Embedded Image Processing with Resource Optimized Slicing ». In: *Real-Time and
        Embedded Technology and Applications Symposium*, *2006. Proceedings of the 12th
        IEEE*. 2006, pp. 361–372. DOI: 10.1109/RTAS.2006.38.