TECHNIQUES FOR CRAFTING CUSTOMIZABLE MPSOCS

LIANG CHEN

(B.Eng., Xi'an Jiaotong University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information that have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Liang Chen
April, 2014

Acknowledgement

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Tulika Mitra for her patience, motivation, immense knowledge and extensive supports throughout my Ph.D. candidature.

My sincere thanks to Prof. Wong Weng Fai, Prof. Liang ZhenKai and Prof. Kiyoung Choi for being my dissertation committee members. Their valuable comments and recommendations help to shape this dissertation.

I would like to thank all the teachers during my Ph.D. course works. I thank School of Computing to cover the expenses of my conference trips and the administrative staffs there for all the helps.

I am grateful to meet all my friends in Embedded System Lab and School of Computing. Many thanks go to Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Sudipta Chattopadhyay, Wang Chundong, Ding Huping, Qi Dawei, Zhong Guanwen, Tan Cheng, Yao Yuan, Huynh Phung Huynh, Pan Yu, Kaushik Mysu, Vanchinathan Venkataramani, Alok Prakash, Lu Peng, Nie Liqiang, Zhu Minhui and Wang Yuhui.

It is my fortune to meet so many cool guys in SoC basketball team including Bao Zhifeng, Beng Chin Ooi, Wu Sai, Ju Lei, Liu Chen, Zhang Zhenjie, Xue Mingqiang, Guo Long, Lin Yuting, Zhang Dongxiang, Zheng Yuxin, Lu Wei, Li Yuchen, Zhang Jingbo, Yang Yang, Fan Ju, Huang Hao, Song Zheng, Li Guangda, Zhou Lizhu, Zhong Qing, Guo qi, Yao Chang, Li Guoliang, Guan Yue, Huang Zhi and many others that have not been listed.

It is my lifetime precious to have my old friends, Luo Wenxin, Cao Chengxiu, Jiang Qunwei and Yu Zimou. Their encouragements and supports are the best things I could ever have. I would also like to take this opportunity to thank Prof. Qi Yong and Prof. Zheng Qinghua in Xi'an Jiaotong University.

My deepest gratitude to my parents, my sister and my family. They are always standing as my solid backing and encouraging me to pursue my dream. *This dissertation is dedicated to them.*

Contents

ABSTRACT

General-purpose processors offer high flexibility in terms of supporting wide range of applications. However, they hardly meet the high performance demands of computationally intensive applications. A common method for bridging the gap between flexibility and performance demands is to add customized accelerators into general-purpose processors. These customized accelerators are designed to explore the special features of different applications so that they can achieve dramatic speedups.

On the other hand, with the inevitable transition into the multi-core era, heterogeneity is emphasized to improve the overall efficiency of the application executions. Rather than integrating multiple simple cores within one chip, each of the cores could be tailored through customization techniques to meet the specific demands of the applications.

In this thesis, we propose a customized multiprocessor system-on-chip (MPSoC) architecture and the associated design automation tool-chain covering compiler supports and design space exploration techniques.

At the beginning, we first create a static heterogeneous MPSoC system by using custom functional units. The custom functional units are designed for accelerating different custom instruction sets. The limited chip area budget for customization and alternative customization choices present a challenging optimization problem for design space exploration. A dynamic programming algorithm is then designed to optimally retrieve the set of custom instructions for every task of the target application so as to have the highest speedup under the area constraint.

The rest of the thesis focuses on a reconfigurable heterogeneous MPSoC design where the customization is achieved through a reconfigurable fabric shared among the cores. We first focus on designing the appropriate reconfigurable fabric for customization. We propose a novel custom functional unit design that can be reconfigured to support most of the identified custom instructions across multiple application domains. The efficiency of the custom functional unit design is then evaluated by integrating it into the pipeline to form a just-in-time

reconfigurable processor. We then design a coarse-grained reconfigurable array using the proposed custom functional unit as the primary processing element. Finally the customizable MPSoC architecture is completed by sharing the coarse-grained reconfigurable array among multiple cores.

We then study design automation problem for the newly designed customizable MPSoC architecture, in particular, the compiler support. We formulate the problem of mapping loop kernels onto the reconfigurable fabric as a graph minor containment problem. With the formalization, we design an efficient search algorithm adopted from the graph theory domain to solve the mapping problem.

As the final step of the design automation toolchain, we develop a design space exploration technique for mapping multi-threaded applications on the customizable MPSoC with shared reconfigurable fabric. In the presence of a shared reconfigurable fabric, the complexity of the design space is dramatically increased compared to the static approach. We propose an optimal solution based on dynamic programming that not only selects the appropriate customization for each core but also the appropriate reconfiguration points along the timeline to maximize performance while satisfying the area constraints of the shared reconfigurable fabric.

List of Publications

1. Liang Chen, Tulika Mitra. Shared Reconfigurable Fabric for Multi-core Customization. *In Proceedings of the 48th Design Automation Conference*, DAC'11, pages 830-835, San Diego, California, USA, June 2011. ACM.

2. Liang Chen, Nicolas Boichat, Tulika Mitra. Customized MPSoC Synthesis for Task Sequence. *In Proceedings of the 9th Symposium on Application Specific Processors*, SASP'11, pages 16-21, San Diego, California, USA, June 2011. IEEE.

3. Liang Chen, Thomas Marconi, Tulika Mitra. Online Scheduling for Multi-Core Shared Reconfigurable Fabric, *In Proceedings of the 15th Design Automation and Test in Europe*, DATE'12, pages 582-585, Dresden, Germany, March 2012. IEEE.

4. Liang Chen, Tulika Mitra. Graph Minor Approach for Application Mapping on CGRAs. *In Proceedings of the 2012 International Conference on Field Programmable Technology*, ICFPT'12, pages 285-292, Seoul, South Korea, December 2012. IEEE.

5. Liang Chen, Joseph Tarango, Tulika Mitra, Philip Brisk. A Just-in-Time Customizable Processor. *In Proceedings of the 31st International Conference on Computer-Aided Design*, ICCAD'13, pages 524-531, San Jose, California, USA, November 2013. ACM/IEEE.

6. Liang Chen, Tulika Mitra. Graph Minor Approach for Application Mapping on CGRAs. *Transactions on Reconfigurable Technology and Systems*, TRETS'14, 2014. ACM.

List of Tables

List of Figures

Chapter 1

Introduction

Multiprocessor system-on-chips (MPSoCs) have emerged as an inevitable trend in embedded system designs. This evolution brings tremendous challenges. First of all, embedded system designs could be highly application specific. Rather than simply packing several cores into a single chip, each of the cores can be customized for the specific embedded applications to create a heterogeneous MPSoC. The customization could be done through either instruction-set extensions or much coarse-grained accelerators, both of which have been extensively studied in single core context. However, customization techniques become more challenging for MPSoC designs when customizable resources are shared among multiple cores. The design complexity becomes even higher when reconfigurability is brought in to increase the flexibility and programmability. Driven by the time-to-market constraint, the MPSoC design and optimization problems present urgent demands for design automation tools.

1.1 Processor Customization

The balance between performance and the generality or flexibility is always a challenge for computer designs. While the general-purpose processors are designed to support vast range of applications, they fail to match the increasing demands for high throughput, fast response time and scalability required by computationally intensive or time-sensitive applications such as image processing, encryption and others. To bridge the gap, especially in embedded system, a common method is to use application-specific accelerators added to the general-purpose processors. For example, a math coprocessor could be integrated with a baseline processor to perform mathematical computations, particularly floating-point operations.

Processor customizations explicitly diverge into two categories, depending on

the granularity of the code segments to be accelerated [128]. The *fine-grained processor customization* aims at accelerating very small pieces of code, while the *coarse-grained processor customization* targets for much coarser-grained loops.

1.1.1 Fine-grained processor customization

Fine-grained processor customization is realized through custom instruction identification, custom instruction selection and custom functional unit implementation. All these three phases together in essence create an application specific instruction-set processor (ASIP). Figure 1.1 shows the design and execution flow for fine-grained processor customization. The very first step is to identify the custom instructions. A **custom instruction** is formed by grouping a set of frequently executed operations together. Hot basic blocks are perfect candidates to be used for custom instruction identification analysis. A set of custom instructions will then be selected meeting the specified optimization metrics such as speedup, area, power and etc. These custom instructions replace the original pieces of code in the binary and they are implemented as hardwired datapaths (custom functional units) in the existing processor core. With the extended instruction set to support all the selected custom instructions, these custom instructions could be fetched and decoded as normal instructions, while they are dispatched and executed in the custom functional units.

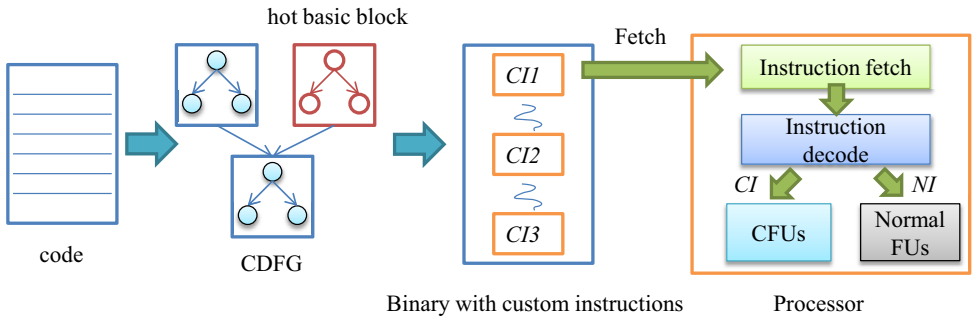


Figure 1.1: Fine-grained processor customization flow

The reasons why executing the custom instructions in custom functional units can bring speedup are straightforward. The first and widely accepted wisdom is that custom instruction can explore the instruction-level parallelism (ILP) through sufficient and dedicated hardware resources. Inside one custom instruction, multiple independent operations are free to be executed in parallel rather than blocking each other. The second reason is also straightforward but more related to hardware implementation – that is several low-latency operations can be chained together to be executed in a single cycle of the processor. In the

normal processor design, even if an operation requires less than a cycle to execute, it still needs to occupy the entire cycle. As an example, suppose the frequency is determined by the multiplier in the processor; obviously, the critical path length of one multiplier can accommodate multiple shifts, logic operations or even some simple arithmetic operations. Another advantage of using custom instructions is much more implicit. It relates to the register pressure and bypassing inside the processor pipeline. As the dependent operations are grouped together in one custom instruction and they are executed inside the custom functional unit, the communications through the register file and the bypassing network could be avoided. This reduces the register file pressure and improves power efficiency.

1.1.2 Coarse-grained processor customization

For *coarse-grained processor customization*, the acceleration candidates are usually nested-loops. These coarse-grained loops are computationally intensive kernels and contain massive data/instruction parallelism, making them ideal for accelerations. To execute large body of operations, the best way is to use a loosely coupled coprocessor rather than tightly coupled functional units considering the area/power efficiencies and communication overheads.

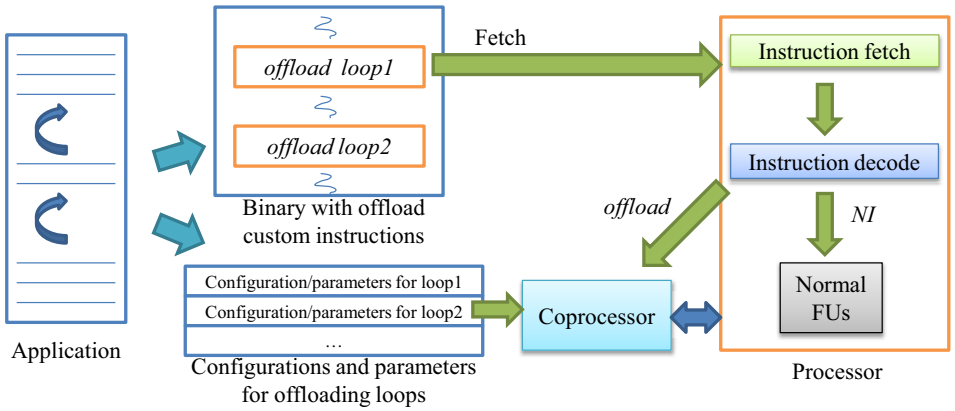


Figure 1.2: Coarse-grained processor customization flow

Figure 1.2 shows the design and execution flow for coarse-grained processor customization. The loops in the application can be identified through manual annotations or automatic design tools. A special offloading instruction is inserted into the binary executable replacing the entire code segment for the loop body with a call to the accelerator. In the decode stage, once an offloading instruction is encountered, the pipeline will be stalled and the execution will be switched to the coprocessor side through proper configuration and parameter transfers.

Once the execution in the coprocessor finishes, the results will be sent back to the main processor and the pipeline will be resumed.

Another essential factor to be considered is the communication cost. The communication cost includes the delays for setting up the coprocessor by reading the necessary configurations, transferring the parameters and sending back the results. Usually, the communication cost offsets only a fraction of the total benefits gained from parallel execution.

1.2 MPSoC Customization

The advancement of semiconductor process technology following Moore's Law has enabled the chip designers to put multiple processors into one single chip. Rather than simply including identical processors, a challenging problem is to design heterogeneous processors to fulfill the demands of specific applications. Intuitively, processor customization could be directly applied to each core to create a heterogeneous MPSoC system while minimizing the cost and energy consumptions and optimizing the performance. We call the customization for multi-core system as *multi-processor system-on-chip customization* or *MPSoC customization*.

1.2.1 MPSoC Customization Overview

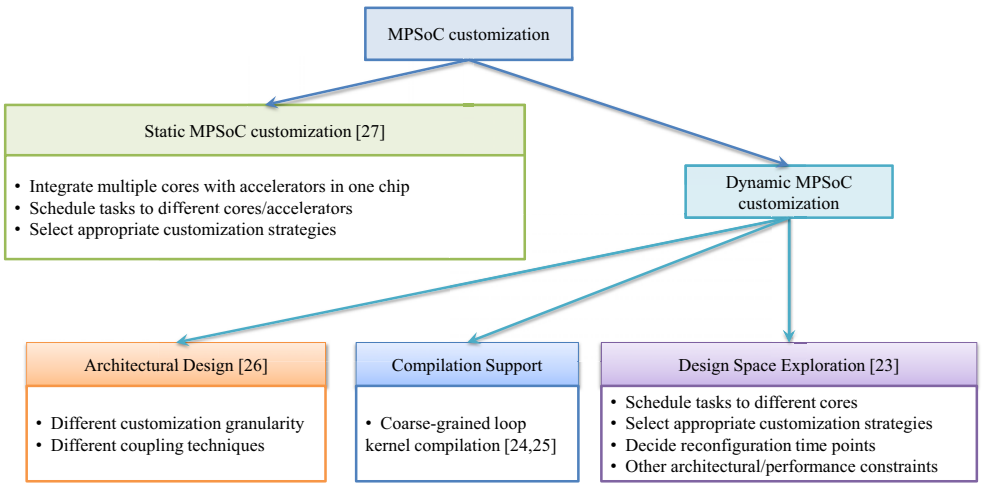


Figure 1.3: Overview of MPSoC customization techniques

Figure 1.3 provides an overview of the MPSoC customization techniques to be discussed in this thesis. Depending on whether the MPSoC system support reconfiguration or not, the MPSoC customization techniques could be divided

into *static MPSoC customization* and *Dynamic MPSoC customization*. In our static MPSoC customization work [27], multiple application specific instruction-set processors (ASIPs) are integrated within one chip. Each of the ASIPs is generated by tightly integrating custom functional units with the baseline processor. The challenges are to schedule the tasks of the target application to different ASIPs and customize each of the ASIPs specifically for the tasks mapped to it.

On the other hand, for dynamic MPSoC customization, we need to first come up with the architectural designs. The architectural designs could vary according to different customization granularity. Usually, fine-grained customization will require a reconfigurable functional unit, which is tightly coupled with the pipeline of the individual processor. One of our works [26] revisits the reconfigurable functional unit design. This is followed by the design of a coprocessor consisting of a coarse-grained reconfigurable array implemented with reconfigurable functional units as basic processing elements. The coarse-grained reconfigurable array is shared among multiple cores. We study the compilation problem for the computationally intensive loop kernels in [24, 25]. With the architectural specifications and compilation supports, the design space exploration problem for dynamically customizable MPSoC is addressed in our final work [23]. With the reconfigurability, we need to consider not only the scheduling and customization strategies but also the reconfiguration decisions during the design space exploration.

1.2.2 Static Customized MPSoC Synthesis

As all the cores are on the same die, the chip area is shared among all the cores when customizing MPSoC. Driven by the preset throughput demand or QoS (quality of service), each core has to compete with other cores for the chip area. Hence, given a performance constraint imposed by the system designer (e.g., for MPEG-2 encoder, 30 frames per second would be the minimum throughput to provide smooth viewing experience), we are interested in allocating the customization resources to each core while satisfying this performance constraint. The complexity of the problem, however, comes from task mapping and multiple alternative customization choices of the tasks allocated to each of the cores. We will illustrate the challenges in MPSoC customization through a concrete example.

An application consists of a sequence of tasks, which could be selectively combined to be mapped to the cores in the MPSoC. Each task is associated with multiple alternative customization choices or **custom extensions**. As shown in Figure 1.4(a), each task of MPEG2 application has multiple available

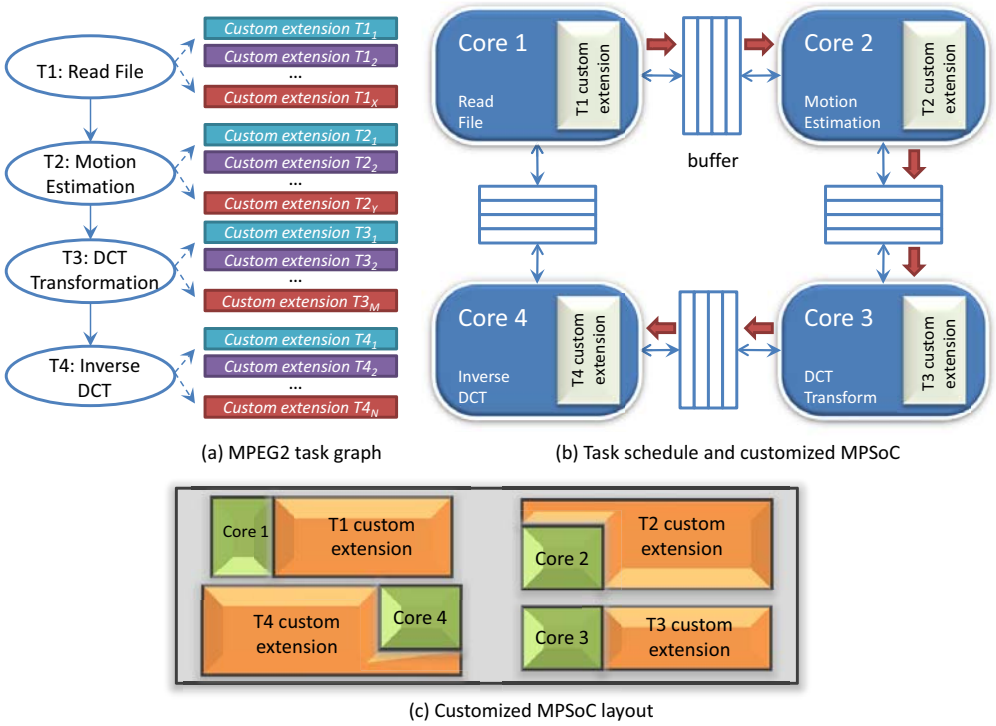


Figure 1.4: Static MPSoC customization

custom extensions. Each of the custom extensions has a different execution time and area requirement, and is used to empower the baseline processor to execute special code segments such as custom instructions or computationally intensive kernels. Note that custom instruction is introduced in fine-grained processor customizations and computationally intensive kernels are generally used in coarse-grained processor customizations. It is the job of the compiler to generate alternative custom extensions regarding to different architectural specifications. MPSoC customization technique has to be aware of the existences of the alternative designs and the area/speedup tradeoff offered by each of the alternatives. It should produce a feasible schedule by mapping tasks to the cores with appropriate selection of the custom alternatives. Figure 1.4(b) shows such a schedule by creating a one-to-one mapping from the tasks to a ring-connected four-core system. The thick arrows shown in Figure 1.4(b) represent the actual data transfer flow. The optimization problem of scheduling and selection is essentially brought by the resource competition among the cores. From the layout shown in Figure 1.4(c), we can see that, the custom extensions require different amounts of area to be instantiated. With a preset QoS constraint, the cores compete with each other struggling to satisfy the constraint by using a more powerful custom extension, which has less execution time but higher chip

area requirement. Meanwhile, the whole area consumption has to be kept under the chip area budget. The exponential complexity of the problem presents a requirement for an efficient design space exploration algorithm, which could be used to tune the processors in a synergistic manner and create an optimal system.

1.2.3 Dynamic MPSoC customization

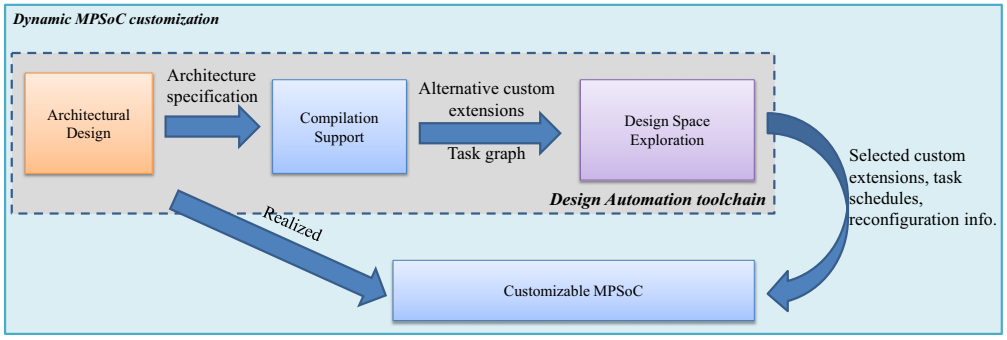


Figure 1.5: An overview of the dynamic MPSoC customization

In processor customization, it is highly desired that the processor could be reconfigured for different applications or further upgrades. To achieve this flexibility, a reconfigurable fabric could be adopted to accommodate custom instructions or computationally intensive loop kernels. In fact, the custom instructions or loop kernels implemented in the fabric can be changed even within the lifetime of an application. This is especially true when reconfigurable fabric is area constrained and reconfiguration is required for time-multiplexing resource re-usage. Here, we will discuss how the dynamic customization techniques are extended to MPSoC context focusing on three main aspects including the architectural designs, compilation supports and design space explorations. Figure 1.5 shows how these three major parts interact with each other and contribute to the final customizable MPSoC. In the architectural design, we propose a concrete MPSoC design which could support dynamic customization. Given the architectural specification and the loop kernels from the input application, the compiler generates a set of alternative custom extensions. Finally, the design space exploration makes a selection among the alternative custom extensions, and creates task schedules together with the necessary reconfiguration information. All the decisions from design space exploration are used to customize the system dynamically during the application execution. In the following, we will give more detailed illustrations for each of these three major parts.

Architecture Overview

A straightforward approach to introduce reconfigurability to multi-core for customization will be to couple each core with its own reconfigurable fabric as shown in Figure 1.6(a). A runtime reconfigurable engine would be designed for each core to control the reconfiguration and support all the communications between the core and its dedicated reconfigurable fabric. This architectural design with a fixed reconfigurable resource partition, however, is not an ideal solution for multi-threaded applications, which can have pronounced imbalances in execution time and customizable resource requirements among their threads.

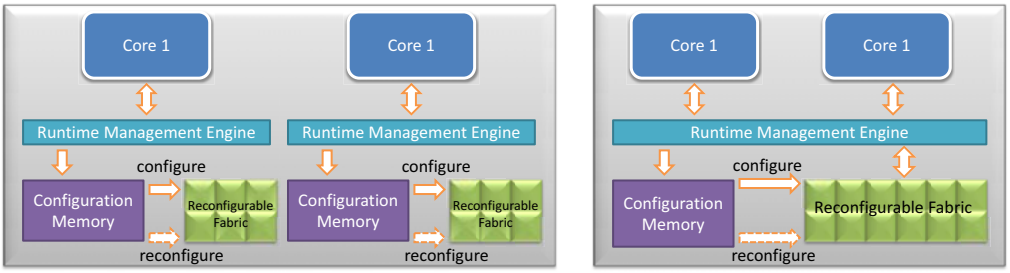


Figure 1.6: MPSoC architectures supporting dynamic customization

A much more effective architectural design is to share a reconfigurable fabric among all the extensible cores as shown in Figure 1.6(b). By sharing among multiple cores, the reconfigurable fabric can be more efficiently integrated and utilized. Obviously, by combining the reconfigurable resources from individual cores together, each core can have more resources to implement much more powerful custom extensions. This increases the chance for the multi-core system to provide much more efficient solutions. On the other hand, sharing could increase the resource utilization. When one core is using less reconfigurable resources, other cores could have more accesses to the available resources. In this sense, we might be able to reduce the size of the reconfigurable fabric through sharing without affecting the final performance, which eventually results in less area and power consumption.

The reconfigurable fabric could be tightly coupled into the processor pipelines to support fine-grained customizations. On the other hand, the reconfigurable fabric could also be used as a shared coprocessor to execute offloaded loop kernels from different cores for coarse-grained customizations. By tightly integrating the reconfigurable fabric into the processor pipelines, we essentially create a conjoined-core chip [77]. However, the tightly coupling approach cannot scale well with number of cores. On the other hand, using the reconfigurable fabric

as a shared coprocessor avoids the overheads of integration, and could be used to accelerate much larger loop kernels. Thus, we will mainly focus on the architectural design for the coarse-grained reconfigurable coprocessor in Chapter 4. More concretely, we propose a specialized functional unit design and use it as the primary processing element of the coarse-grained reconfigurable coprocessor.

Compilation Techniques

For both static and dynamic MPSoC customization, the compiler needs to generate multiple custom extensions. Figure 1.7 depicts the entire compilation flow for MPSoC customizations. The architectural specifications could be read after intermediate representation is generated between the front end and the back end of the compiler. For static MPSoC customization, the outputs of the compiler would be the synthesis results for the custom extensions and the binary. On the other hand, for dynamic MPSoC customization, the custom extensions would be generated as configurations, which are used to configure the reconfigurable fabric during the execution.

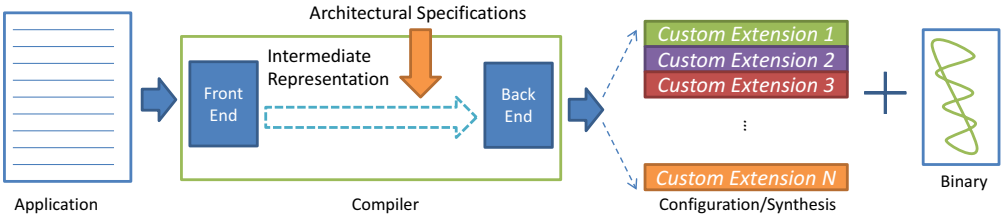


Figure 1.7: Compilation technique for MPSoC customizations

As mentioned, our target architectural is coarse grained reconfigurable coprocessor. In Chapter 5, we will revisit the compilation techniques for application mapping on coarse-grained reconfigurable arrays (CGRAs). The application mapping problem is proved to be a graph minor containment problem. Efficient algorithms are further proposed to solve the application mapping problem.

Design space Exploration

To exploit the customizable MPSoC architecture with shared reconfigurable fabric as shown in Figure 1.6(b), all the problems presented in the static MPSoC customization have to be dealt with, such as scheduling the tasks of the target multi-threaded application among all the available processors, select appropriate sets of custom extensions for the cores. In dynamic MPSoC customization, however, the optimization problem is much more tricky as reconfiguration is

introduced. The design space has been dramatically increased as custom extensions could be grouped into different configurations and realized in different time during the program execution. The configurations for the reconfigurable fabric are stored in an on-chip configuration memory, and they are loaded dynamically in different points of time to reconfigure the fabric.

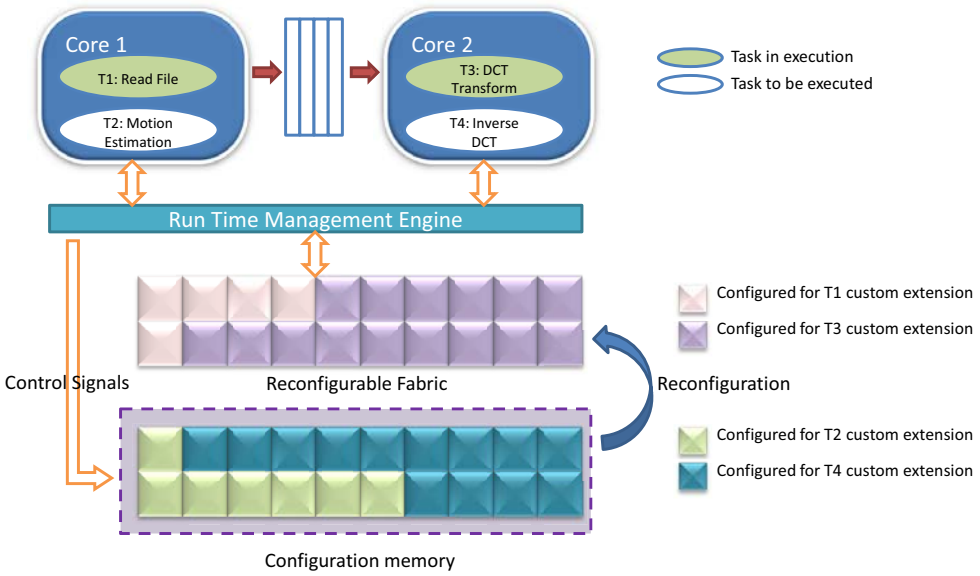


Figure 1.8: An example for dynamic MPSoC customization with MPEG-2 application

Figure 1.8 shows an example for dynamic MPSoC customization using the MPEG-2 application in Figure 1.4(a). In the example, each core executes two tasks of the MPEG-2 application. So when the application starts execution, there would be two threads, each of which is created in one of the two cores executing on different task sets. More concretely, core 1 executes T1 and T2, while core 2 executes T3 and T4. Assuming core 1 is executing T1 and core 2 is executing T3, then the reconfigurable fabric is configured for the two tasks' custom extensions. Some time later, core 1 finishes the execution of T1 and starts T2; meanwhile, core 2 could also finish T3 and start T4. A reconfiguration then occurs by loading the configuration of the custom extensions for T2 and T4 into the reconfigurable fabric. Task scheduling, appropriate customization extension selections and reconfiguration time point decisions present a complex design space exploration problem for dynamic MPSoC customization. In Chapter 6, we propose a dynamic programming algorithm, which can generating optimal solutions with all these considerations for design space exploration.

1.3 Organization of the Chapters

In this dissertation, our ultimate objective is to create a full design automation tool chain for crafting a customizable MPSoC. At the very beginning, in Chapter 3 we highlight the resource sharing problem by considering the static MPSoC customization. Each task of the target application to be executed in the MPSoC is associated with a set of custom extensions with different area and speedups. Each core could be customized by using alternative custom extensions of the tasks mapped to it. All the cores on the same die must compete for the chip area for customizations to meet certain QoS requirement of the target application. With the conflicting goals of minimizing the total area consumption and meeting the throughput requirement, we design a dynamic programming algorithm for task mapping and identifying appropriate custom extensions given a streaming application.

Observing the benefits of introducing reconfigurability in processor customization, Chapter 4 proposes a novel design for the reconfigurable coprocessor, which is used to execute computational intensive loop kernels. We first design a specialized customizable functional unit, which is used to execute the small computational intensive patterns or custom instructions across multiple application domains. To achieve this, we conduct analysis for sources of the speedup gained from using custom instructions. The specialized functional unit (SFU) is then instantiated regarding the design analysis results. We thoroughly evaluate the proposed SFU by integrating it into the processor pipeline. A vast range of applications are tested for its flexibility and applicability. By using the SFU as the primary processing element, we then take a step forward to build a novel specialized CGRA architecture (S-CGRA). By sharing the proposed S-CGRA among multiple cores, we come up with a novel customizable MPSoC architecture.

As our target coprocessor is a specialized CGRA, we first revisit the compilation problem for CGRA in Chapter 5. We demonstrate the CGRA mapping problem is a restricted graph minor problem. Together with the proof of NP-completeness, we also propose a practical tree-based search algorithm, which is adapted from graph theory domain and could produce near-optimal solutions. We further consider the compilation technique for the proposed S-CGRA. To exploit the capability of mapping more than one operations to one SFU in the S-CGRA, we design a clustering algorithm as a pre-processing step integrated into the CGRA compilation framework.

Finally, Chapter 6 formalizes the MPSoC customization problem in the presence of a shared reconfigurable fabric. The complications of the the problem reside in reconfigurations, task scheduling, alternative customizations and re-

source sharing. While the problem itself is NP-complete, we present a dynamic programming solution. We now have a design automation toolchain for dynamic customizable MPSoC, which consists of the concrete underlying architectural specifications, the compilation supports and the design space exploration techniques. In the end, we use the design automation toolchain to demonstrate the efficiency of our proposed customizable MPSoC architecture through concrete case studies using MP3 and JPEG encoders.

Chapter 2

Literature Review

In this chapter, an overview is presented of the existing research in single-core customizations and MPSoC customizations. Processor customizations in single cores have been extensively studied for both architectures and compiler perspectives. On the other hand, MPSoC customizations inherit the challenges presented in single-core scenario, but focus more on the new open problems of how to schedule and cooperate for multi-threaded applications. We will first give an overview of advances in single-core customizations. Subsequently, we will discuss MPSoC customization challenges and the initial attempts to overcome the challenges in literature.

2.1 Processor Customization

Although the thesis aims at creating an MPSoC system through customization techniques and the focus of MPSoC customization is much more different from single-core customization, it is still essential for us to gain a deep understanding of how customization is done in single-cores. In fact, to create a full system and develop a fully automated design tool chain, we have to revisit the customization problems in single-cores. In the following subsections, we will cover both the fine-grained and coarse-grained processor customization techniques and highlight how our works in the corresponding research area serve as an integral part in the final MPSoC customization framework.

2.1.1 Fine-Grained Processor Customization

In fine-grained processor customization, the accelerators are tightly integrated into the processor pipeline as custom functional units.

Static Fine-Grained Processor Customization

In ASIPs, custom instructions are implemented in application specific integrated circuits (ASICs). Obviously, given a set of custom instructions, the best performance can be achieved by implementing in ASICs using off-the-shell synthesis tools. The efficiency of the ASIP designs, thus, relies on the custom instruction identification and selection algorithms. Efficient algorithms are proposed to accelerate custom instruction identifications and selections. Mircro-architectural constraints are first introduced in custom instruction identifications and selections in [10, 103], which uses a tree-based search algorithm. The algorithm is further improved by using ILP [8] or iteratively selecting maximal convex subgraphs [9]. [132] enumerates the multi-inputs and multi-outputs (MIMO) convex subgraphs by combining subgraphs that have single-output and multi-outputs (SIMO). The selection algorithm for MIMO custom instructions under the resource constraints is presented in [131]. Cross-basic block custom instruction identification is also solved in [133]. Tensilica [64] is a company that commercializes customizable processors which are customized statically during the design time.

Dynamic Fine-Grained Processor Customization

While ASIPs suffer from limited flexibilities, reconfigurable ASIPs can support dynamic reconfiguration but with a tradeoff from performance. For reconfigurable ASIPs, extensive research have been carried out for the efficient designs of the reconfigurable fabric. Several excellent survey papers [59, 118] provide an overview of the contributions of prior reconfigurable computing projects focusing on a single processor core with an attached reconfigurable fabric. Theses architectures include Chimara [95] using a reconfigurable functional unit, One-chip [20] with an integrated FPGA, and Stretch [53], which includes an instruction-set-extension fabric.

Custom Instructions

Custom instructions can naturally cover the maximal parallelism leading to speedup. A large body of research works [10, 8, 103, 131, 132, 133] conclude that increasing the number of inputs or outputs can give more speedup. On the other hand, some works identify the benefits from chaining consecutive operations within the latency of one processor cycle. Interlock collapsing ALU [124] is probably the very first work considering the chaining effects. In fact, it was proposed to parallelize the execution of up to two interlocked consecutive

instructions rather than directly chaining the operations. A follow up work in [129] can collapse multiple instructions with up to 10 inputs and multiple outputs dynamically using special functional units, which are based on FPGA-like elements requiring large number of control bits and longer latencies. In both approaches, chaining the operations is realized by exploring the parallelism of the executions. Dynamic Strands [111] and Static Strands [112] reveal the potential of collapsing sequential instructions using closed-loop ALUs where the output of an ALU is forwarded to its inputs using self-bypass lines. Dataflow mini-graphs, [17, 18] also identifies the mini sequences but executions are carried out in an ALU pipeline, where three ALUs are chained together. These works have more concrete architectural designs. However, the supported custom instructions such as consecutive operations, strands and mini graphs are just a subset patterns of the custom instructions. [10, 8, 103, 131, 132, 133] focus on identifying more general custom instructions under certain micro-architectural constraints.

Another distinguishing feature of different customization approaches is the identification phase of the custom instructions – whether they are identified statically by compilers or dynamically during the execution. One major drawback of dynamic approaches such as [129, 111], is to push a large amount of overheads to the execution engine, which can potentially drain out all the speedup brought by using custom instructions. A different architectural design [31, 29] can relieve the identification overhead during the execution, however, note that a dynamic approach can hardly outperform the static compilation techniques. Additional works have focused on dynamic reconfiguration of programmable accelerators, including: RISPP [13], which dynamically reconfigures selected columns of an FPGA that implement custom accelerator functions; Warp processor [89], which transparently converts software binaries to placed-and-routed FPGA bitstreams; and KAHRISMA [75], a hybrid fine-grained/coarse-grained accelerator.

Summary

In summary, by dynamically supporting custom instructions in an extensible processor, one can expect its performance to match the ASIP design. In Chapter 4, we will first fully explore the essences of the benefits brought by custom instructions through extensive experimental evaluations. A specialized reconfigurable functional unit is then proposed following a systematic design procedure. We discover that the performance of an extensible processor can potentially match up with those of optimized ASIPs. More importantly, the proposed specialized functional unit will further serve as a processing element design for the shared reconfigurable fabric in MPSoC context.

2.1.2 Coarse-Grained Processor Customization

One historic debate in processor customization involves the granularity of the accelerator: should it be fine-grained, similar to an FPGA [20, 61, 122, 129], should it be coarse-grained, i.e., an array of ALUs with a programmable interconnect [31, 29, 54], or should the granularity be even coarser at the level of expressions [7] such as Expression Grained Reconfigurable Array (EGRA) [15]. In fact, coarse-grained accelerators are more favored by current research due to the much smaller reconfiguration overheads, e.g., less configuration bits. While coarse-grained accelerators could be used to execute custom instructions, they are more promising to be used as a coprocessor to execute larger segments of code. For coarse-grained processor customization, our targeting coprocessor architecture is coarse-grained reconfigurable arrays (CGRAs). CGRAs have been proposed especially for accelerating loops in multimedia and digital signal processing (DSP) applications in embedded systems.

Architectures

Year	*Name of CGRA*	*Size (Row×Column)*	*Private register file*	*Topology*
1998	[115] MorphoSys	8×8 FUs	Loop-back connection (4 16-bit registers)	2D mesh-plus; row and column connections (inside clusters); neighbor connections (across clusters)
1999	[90] CHESS	16×32 4-bit simple FUs	None	2D mesh-switch box
2003	[93] ADRES	8×8 FUs	Loop-back connection	2D mesh-only and mesh-plus
2003	[84] DRAA	8×8 FUs	Loop-back connection	2D mesh-only
2009	[83] FloRA	8×8 FUs	Loop-back connection	2D mesh-plus
2010	[21] SmartCell	4×4 clusters (4 FUs per cluster)	Input register banks	2D mesh-only
2011	[94] DR-SPE	10×10 FUs	Loop-back connection (1 register)	2D mesh-switch box
2011	[54] DySE	8×8 FUs	Input register banks each with 1 data register and 1 status register	2D mesh-switch box
2011	[101] SYSCORE	8×4 or 8×8 FUs	2 registers with dedicated input ports	2D mesh-East and West only

Table 2.1: Different CGRA architectures

In terms of architecture, some features of different CGRAs are listed in Table 2.1, including their sizes, register files and topologies. An interesting feature to note is that even though CGRAs have the potential to be scaled, current designs are still limited to very small architectures, e.g. an 8×8 array. Besides, most of the architectures in Table 2.1 are arranged in a 2D mesh-like structure. In a 2D mesh-only architecture, one functional unit is only connected to its neighborhood functional units. In a 2D mesh-plus architecture, one functional unit can communicate directly to other functional units in the same row or column. By using switch box, diagonal communications are added into the 2D mesh-only topology. For private register file design, most architectures use it

for a loop-back connection, which connects the output of a functional unit to its input ports. This enables the functional unit to use the output data for its future execution.

Compilation of loop kernels

Mapping a compute-intensive loop kernel of an application to CGRAs using modulo scheduling was first discussed in [92]. In this simulated annealing based approach, the cost function is defined according to the number of over-occupied resources. The simulated annealing approach can have long convergence time, especially for large dataflow graphs. Routing through register files and register allocation problems are further explored in [35], which extends the work in [92]. Register allocation is achieved by constraining the register usage during the simulated annealing place and route process. The imposed constraint is adopted from meeting graph [41] for solving loop cyclic register allocation in VLIW processors. In post routing phase, the registers are allocated by finding a Hamilton circuit in the meeting graph, which is solved as a traveling salesman problem [35]. This technique is specially designed for CGRAs with rotating register files. [60] also follows the simulated annealing framework but aims at finding better cost functions for over-used resources. SPR [43] is a mature CGRA mapping tool that successfully combines the VLIW style scheduler and FPGA placement and routing algorithms for CGRA application mapping. It consists of three individual steps namely scheduling, placement, and routing. The placement step of SPR also uses the simulated annealing approach.

List scheduling has been adopted in [12], which analyzes priority assignment heuristics under different network traversal strategies and delay models. The heuristics utilize the interconnect information to ensure that data dependent operations can be mapped spatially close to each other. [99] also gives priorities for operations and resources to obtain a quality schedule. The priorities are assigned according to the importance of routing from producer nodes to consumer nodes. This idea is further exploited in edge-centric modulo scheduling (EMS) [100], where the primary objective is routing efficiency rather than operation assignments. The quality of a mapping using specific priorities highly depends on efficient heuristics for assigning these priority values to both operations and resources.

There are various approaches to CGRA mapping using techniques from graph theory domain. [30] integrates subgraph isomorphism algorithm to generate candidate mapping between a DFG and the resource graph of a coarse-grained accelerator. SPKM [130] adopts the split and push technique [39] for planar

graph drawing and focuses on spatial mappings for CGRAs. The mapping in SPKM starts from an initial drawing where all DFG nodes reside in the same group. One group represents a single functional unit. The group is then split into two and a set of nodes are pushed to the newly generated group. The split process continues till each group contains only one node, which represents a one-to-one mapping from DFG to the planar resource graph of CGRA.

A number of CGRA mapping approaches follow the subgraph homeomorphism formalizations including [119, 5, 19, 48, 49]. The mapping algorithm in [119] is adapted from MIRS [134], a modulo scheduler capable of instruction scheduling with register constraints. The adaptations for CGRA mapping include a cost function for routing and considerations for conditional branches. [5] partitions the DFG into substructures called HyperOps and these HyperOps are synthesized into hardware configurations. The synthesis is carried out through a homeomorphic transformation of the dependency graph of each HyperOp onto the resource graph. [19] also formalizes the CGRA mapping as a subgraph homeomorphism problem. However, they consider general application kernels rather than loops. Particle swarm optimization is adopted for solving CGRA mapping problem in [48, 49]. The calculation for fitness, which is used to move particles (DFG nodes) in particle swarm optimization, is specifically designed to optimize multiple objectives for routing.

EPIMap [56] formalizes the CGRA mapping problem as a graph epimorphism problem with the additional feature of re-computations. The core of this approach consists of a subgraph isomorphism solver, which finds the maximum common subgraph (MCS) [86] between the DFG and the resource graph of CGRA. The idea is to transform the DFG iteratively by inserting dummy routing nodes or replicated operation nodes so that the routing requirements can be satisfied through the subgraph isomorphism solver. EPIMap can generate better scheduling results compared to EMS with similar compilation time. Most graph approaches solve a subset of the epimorphism problem defined in EPIMap.

Summary

In summary, the observation of various architectures and compilation techniques in CGRA research domain motivates us to re-consider the real difficulties of the mapping problem. Despite its simplistic illustration such as mapping a DFG to the resource graph of the target CGRA, the crucial features such as resource graph generation and explicit routing handling do dramatically complicate the mapping problem. In Chapter 5, we will sketch out the CGRA mapping problem as a graph minor problem. We will show how the formalization, which provides

a much clearer view for compiler designers can improve the performance and compilation time. Integrating into the MPSoC customization framework, the proposed graph-minor mapping technique will be utilized to generate alternative custom extensions for each candidate loop kernels. The generated alternative custom extension would be used as input data for MPSoC design space exploration in Chapter 6.

2.2 MPSoC Customization

Similar to processor customization techniques, MPSoC customization techniques could be divided into fine-grained or coarse-grained depending on the granularity of the accelerating candidates. Processor customization could be done through the compilation stage by taking all the micro-architectural constraints into consideration. The customization for an MPSoC system, however, is different from processor customization techniques, and it could involve more than just compilation. Given an application with a set of tasks, each core could be customized by mapping a subset of tasks to it. The customization problem is further complicated when each task has a set of alternative custom extensions. The custom extensions could be generated by traditional fine-grained or coarse-grained processor customization techniques according to architectural specifications. Design space exploration is required to decide the mapping from the tasks to the cores and choose appropriate custom extensions regarding the constraints imposed by the underlying MPSoC architecture. Depending on whether the MPSoC architecture support reconfiguration or not, the customizations could be divided into static MPSoC customization and dynamic MPSoC customization.

2.2.1 Mapping Strategies

In MPSoC scheduling problem, an application is usually decomposed into several independent and/or interdependent sets of cooperating tasks and then mapped onto a set of available processors for parallel execution. These sets of tasks are usually represented by a directed acyclic graph (DAG). For task mapping, Benoit et al. [14] classify the policies to map tasks onto a fixed number of processing elements (PEs, which can be viewed as processors) into three categories: one-to-one mapping, where each task gets its own dedicated PE; an interval-based policy, where only tasks that are contiguous in the task graph can be mapped on a single PE; and a fully general policy without restrictions. For complexity analysis, [14] further analyzed the structure of the platform detailed in communications contention and processors structure. As shown in Table 2.2, different platform

structures and different mapping policies will affect the problem complexity. For fully heterogenous processors, however, the problem is NP-complete for all the three mapping policies. Note that customization is more than just fixed architectures. In fact, customization means the processor architecture can be changed adaptively according to the task mappings and the quality constraints. Due to the high complexity of the problem, several works propose algorithms that generate approximately optimal solutions: [117] proposes an iterative heuristic approach, [120] uses evolutionary algorithms, and [114] uses heuristic approach as well. On the other hand, [67] focuses on optimal solutions using integer linear programming (ILP) but with restrictions to one-to-one mappings and a fixed number of PEs.

	Fully Homogeneous	**Communication Homogeneous**	**Fully Heterogeneous**
One-to-One	polynomial	polynomial	NP-complete
Interval-based	polynomial	NP-complete	NP-complete
General	polynomial	NP-complete	NP-complete

Table 2.2: Complexity Analysis

Interval-based Mapping Policy

As finding the optimal solution using the interval-based mapping policy for heterogeneous processors is NP-complete. So a good place to start might be trying to restrict the problem to chain structured task graph. If we only consider the chain structured task graph, the problem could be modeled as a chain-on-chain problem (CCP) [102]. The CCP problem has been widely studied, and various efficient polynomial time algorithms have been proposed [58, 65, 96].

An iterative refinement heuristic is proposed in [117] for interval-based mapping policy, where no guarantee of the optimality or near-optimality is provided. In [69], a much more matured and well-structured heuristic is proposed using a three phases framework including coarsening, partitioning and unpacking. By coarsening, a relatively smaller graph is generated and K-L heuristic [70] is used to decide the partition points. Then, in each step of unpacking, K-L refinement is performed to ensure that the final outcome solution will be close to the optimal solution. This coarsening and unpacking framework is not constrained in any particular heuristics used in any of the three phases. Any reasonable kinds of heuristics could be used during these phases.

Clustering for general mapping policy

General mapping policy corresponds to the clustering problem for scheduling DAGs onto multiprocessors, which has been extensively studied over the last two decades. An early paper for comparing different clustering heuristics is [47]. In the general mapping policy, a cluster could contain a set of tasks, which will execute on the same processor. The cluster has to be created meeting the intrinsic constraints, e.g. convexity constraint and capacity constraint. The clustering algorithm could aims at achieving different optimization objectives, such as communication cost [68, 104], number of processors [47], and etc. A widely accepted approach is to adapt genetic algorithm [38, 125, 79, 63] into multiprocessor scheduling context, due to the general applicability of genetic algorithm.

2.2.2 Static MPSoC customization

Design automation tools have been provided in industry for single-core processor customizations. These tools includes Tensilica Xtensa [52] and CoWare [123] tool chain. It could be straightforward that one MPSoC could be easily designed by integrating several ASIPs generated by these tools in one chip. However, as mentioned, MPSoC customization is a much more complex problem. Complex interdependencies arise while exploring the design space by simultaneously sweeping axes like task scheduling, custom extension selections and other constraints imposed by architectural components such as processing elements, memory hierarchies and chip interconnect fabrics. One of the very first papers discussing about all these design automation issues in configurable MPSoC could refer to [2].

With regards to architectural constraints, [6] proposes an integrated open framework for MPSoC design space exploration by combining the usage of LISATek processor design platform with MPARM system-level architecture, where MPARM provides extensive facilities for memory hierarchies and interconnects. In terms of MPSoC customization, a widely recognized work is [117], where they formalized the design space exploration problem of static customizations for application-specific custom MPSoCs. It is demonstrated that the design steps of custom extension selections and task scheduling are highly interdependent, and performing these design steps independently can significantly downgrade the quality of the resulting architecture. Their methodology pre-synthesizes the area cost and execution time for custom extensions and make assumption that these values won't be changed when combining the custom extensions with base processors.

However, as mentioned, this problem is too complex so that only heuristics are proposed without any solid guarantees provided. A widely used technique for the optimization in MPSoC design space exploration is by formalizing the problem using Integer Linear Programming (ILP) [33]. The static MPSoC customization problem is formalized in [113] as a Mixed ILP (MILP) problem, which has an intractable running time when the number of processors scales. Similar approaches includes [37], which partitions the task graph onto a set of available processors; [85], which provides an automation flow from custom instruction identification to synthesis using ILP; and [76] that considers hardware/software partitioning for pipelined tasks. For pipelined multimedia streaming applications, [67] gives an optimal solution based on ILP formulas with a case study using JPEG application. Another recent work [16] focusing on multimedia streaming applications proposes a design space exploration technique based on dynamic programming and worst-case execution time (WCET) analysis for instruction set customizable MPSoC.

Summary

In summary, the design space exploration problem for static MPSoC customization is a very complex problem. ILP is widely used to ensure the optimal solutions, while it has an intractable running time when the size of the problem scales. In chapter 3, we will expose the design issues faced in static MPSoC customizations and give an optimal solution for pipelined streaming applications using a hierarchical dynamic programming algorithm.

2.2.3 Dynamic MPSoC customization

Dynamic MPSoC customization can be realized by including reconfigurable fabrics, which are used to accommodate custom extensions. To support reconfigurability, it would be much more promising by using a large reconfigurable fabric shared among multiple processors in the system. However, there is a dearth of prior work in addressing how reconfigurable fabric can best benefit future MPSoCs. Many research efforts [22, 34, 45] have investigated the high level integration of a reconfigurable fabric on-chip. PRISC [106], Proteus [34], Stretch [53], Chimaera [95], and DPGA [36] tightly integrate the fabric with the processor as a specialized execution unit. The fabric predominates in DISC [127] and NAPA [109] with the processor serving largely to feed the reconfigurable hardware. Garp [20, 62] and PipeRench [51] fall in between. All of these, however, only investigate the integration with a single core, although Garcia and Compton [45] state that their technique could be extended to a multi-core system.

While multiple extensible cores sharing reconfigurable fabric is a relatively unexplored research direction, there are few representative works. ReMAP (Reconfigurable Multicore Acceleration and Parallelization) [126] is such a reconfigurable architecture for accelerating and parallelizing applications within a heterogeneous chip multiprocessor. In ReMAP, clusters of cores share a common reconfigurable fabric adaptable for individual thread computation or fine-grained communication with integrated computation. It pairs a specialized programmable logic (SPL) fabric with multiple cores of a chip multiprocessor (CMP). Cores are partitioned into different clusters, and the cores in the same cluster can temporally share the fabric in a round-robin fashion. The SPL controller can also spatially partition the fabric as needed to reduce inter-thread contention. Moreover, ReMAP also facilitates fine-grained communication among threads sharing the fabric, creating new opportunities for parallelization that are too costly using conventional software-based methods.

Shared reconfigurable coprocessor has also been proposed in [46] to improve the overall system throughput of multiple processes concurrently executing on a multi-core system. The focus of their work is on time-multiplexed sharing of the same physical kernel by multiple processes while maintaining process isolation. Sharing loosely coupled reconfigurable fabric is also addressed in [126]. Finally, [28] investigates the synergy between multi-core processors and rISE—an architecture where reconfigurable device is used to implement the custom instructions. However, they use dedicated reconfigurable logic per core.

Summary

In summary, dynamic MPSoC customization problem is quite challenging in diverse aspects due to the intrinsic problem complexity. Dynamic MPSoC customization inherits all the complexities from static MPSoC customization, while the new feature of reconfigurability and resource sharing further complicate this challenging problem. Meanwhile, dynamic MPSoC customization is a very new research topic and only few prior works have investigate this problem. In Chapter 6, we will formalize the dynamic MPSoC customization problem with the presence of a shared reconfigurable fabric. An optimal solution will then be proposed together with an efficient heuristic.

Chapter 3

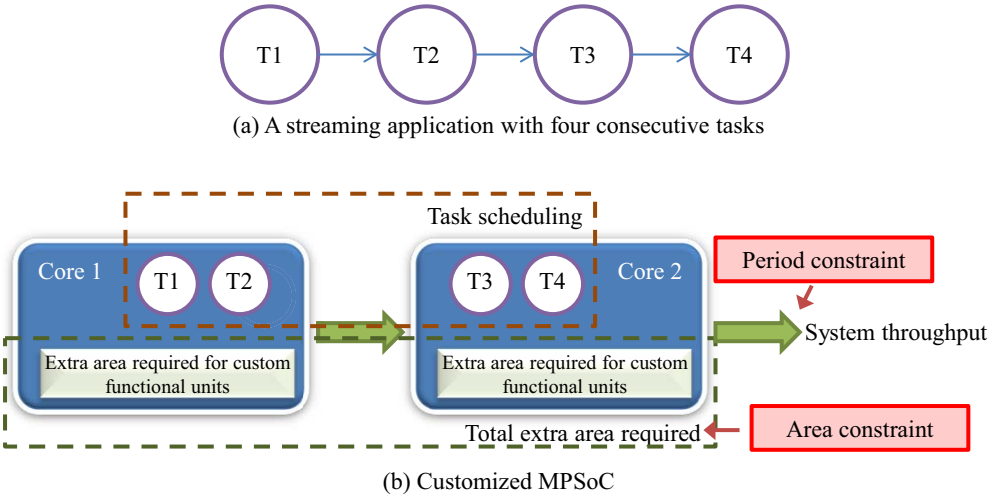
Design Space Exploration for Static Customizable MPSoCs

Observing the inevitable transition to multi-core era, automation tools are urgently required for MPSoC designs. On the other hand, single-core customization techniques have been extensively studied through the last decade. At first glance, it would be straightforward to create a heterogeneous MPSoC systems by directly using the inherited single-core customization techniques. However, a deep investigation shows that MPSoC customization is a much more complex problem. It not only includes all the challenges to deal with single-cores, but also many new aspects due to the requirements of efficient task scheduling, QoS constraints and resource sharing. In this chapter, we will take a first step to reveal the important factors to be considered in static MPSoC customization.

3.1 Overview

A heterogeneous MPSoC may consist of a number of extensible processor cores, where each core has been customized according to the application requirements. As all the processing elements (PEs) share the same base instruction set architecture (ISA) and a common core, application development on such platforms is relatively straightforward. MPSoC platforms consisting of extensible processor cores are an excellent match for streaming applications [67, 114]. These applications can be partitioned into multiple compute-intensive kernels or tasks and represented in the form of an acyclic task graph.

Our initial goal is to synthesize an optimal customized MPSoC platform for a given streaming application. This customization problem involves task scheduling and customization strategy selections. For example, in Figure 3.1(a), we



(a) A streaming application with four consecutive tasks

(b) Customized MPSoC

Figure 3.1: An example for MPSoC customization

have a streaming application with four consecutive tasks. Our MPSoC platform has two extensible processors. One possible task scheduling is shown in Figure 3.1(b), where the first two tasks are mapped to core 1 and the rest are mapped to core 2. Each core is then customized specifically according to the tasks mapped to it. The customization is done using custom functional units. The objective of the customization is to minimize the total extra area required for implementing custom functional units while minimizing the pipeline period (or equivalently maximizing the throughput). Mapping the tasks to PEs and the customization of each PE can dramatically influence the area and period of the entire system. Therefore, the design space exploration algorithm has to deal with task mapping and customization alternative selections under the area budget and period constraints to tune the processors in a synergistic manner.

In this chapter, we will show an efficient hierarchical algorithm that separates task mapping and custom instruction sets selections, and returns optimal solutions. Rather than focusing on fixed architectures with a given number of PEs [120, 117], or performing an one-to-one mapping of tasks to PEs [67, 114], we consider different number of PEs and interval-based mapping policy. Most importantly, rather than using a heuristic, we design a pseudo-polynomial time algorithm that returns the optimal solution in a fraction of the time required by an exhaustive approach.

3.2 Problem Definition

The input to our framework is a linear task graph modeling the application. Let $\langle T_1, T_2, \ldots, T_N \rangle$ be the N tasks in a linear task graph representing a streaming application. There are dependencies between consecutive tasks in this linear chain. Task T_{i+1} can start execution only after task T_i has completed execution for $1 \leq i < N$. Note that our framework is not limited to applications that can be modeled as linear task graphs. An application that is modeled with a general task graph can be easily transformed into a linear chain while respecting all the dependencies in the original task graph. The maximum tolerable period *period* (or minimum throughput) requirement of the application is also provided as an input.

We assume that each task in the task graph can be accelerated with the help of custom instructions. There are multiple implementations or versions of each task corresponding to different choices of custom instructions. We call each such implementation a custom instruction set or CIS, which consists of a set of custom instructions. Each CIS is associated with an area requirement and an execution time. The area requirement captures the additional area required to implement the specific functional units for the custom instructions. Increasing the area available allows more flexibility for the implementation and thereby reduces the execution time. Let $\{C_{i,0}, C_{i,1}, \ldots, C_{i,m_i}\}$ denote the different custom instruction sets corresponding to task T_i where $m_i + 1$ is the number of CISs for T_i. Let us also assume that $a_{i,j}$ is the additional area required and $t_{i,j}$ is the execution time for the CIS $C_{i,j}$. Moreover, we assume that $C_{i,0}$ is the software implementation version with $a_{i,0} = 0$ and $t_{i,0}$ is the software execution time. We order the rest of the CISs according to their area requirement. That is, $a_{i,0} < a_{i,1} < \cdots < a_{i,m_i}$ and as we only consider Pareto-optimal CISs, $t_{i,0} > t_{i,1} > \cdots > t_{i,m_i}$.

The application is mapped onto an underlying architecture consisting of a linear chain of P processing elements $(PE_1, \ldots, PE_P)$ where $P \leq N$. The PEs form the different pipeline stages of the application. We impose the constraint that only a consecutive sequence of tasks from the linear task graph can be mapped to a PE. This is known as interval-based mapping. In other words, the linear task graph is divided into P partitions $(S_1, \ldots, S_P)$ where each partition is a consecutive sequence of tasks in the task graph and partition S_i maps to PE_i for $1 \leq i \leq P$. The pipeline stage with the maximum execution time determines the period and the throughput.

We start with homogeneous multi-core architecture, that is, the base instruction-set architecture of all the P processing elements are identical. The base area of each PE is $areaPE$. However, each PE can be customized by adding CISs

according to the tasks mapped to it. So the final solution is a heterogenous multiprocessor system-on-chip (MPSoC) customized and optimized for the target application. The goal of our optimization strategy is to minimize the total area requirement of the MPSoC solution while satisfying the period or throughput constraint of the application. Both the base area of the PEs as well as the selected CIS versions of the tasks determine the area requirement of an MPSoC solution. In other words, our design space exploration need to explore (a) the number of PEs P, (b) the partitioning of the task graph into P partitions, and (c) the CIS choice for each of the N tasks.

So our problem definition can be formally stated as follows: Given a linear task graph consisting of N tasks with multiple CIS versions for each task and period constraint *period*, find the number of PEs P, the CIS version for each of the N tasks, and P partitions of the linear task graph so that the maximum execution time of each PE is less than *period* and the total area (the base area for P PEs and the additional area for all the selected CIS versions) for the MPSoC solution is minimized.

3.3 Exhaustive Design Space Exploration

We first start with a simple algorithm that exhaustively enumerates the entire design space. This helps us to visualize the complex tradeoff between area and performance. We will follow it up with more efficient approaches that can identify the resource-optimal solution under period constraint.

The exhaustive algorithm recursively enumerates all possible choices for each task. It processes the tasks in their linear order starting with task T_1. For task T_i, we enumerate all possible choices for CIS. For each such choice of CIS, we consider two alternative mapping choices for T_i. The first choice is to map T_i to the current PE. The other alternative is to map T_i to a new PE, in which case we add the base area of a PE $areaPE$ to our cumulative area variable $area$. At each point, we keep track of the period of the application, that is, the processing element with the maximum execution time. Once we have reached the last task, we simply plot the area requirement and the period of the solution.

Note that it is trivial to modify Algorithm 1 to compute the area-optimal solution under the a particular *period* constraint. In this case, we have to make sure that the execution time of any PE is always under the *period* constraint. If the constraint is violated at some point, we can simply prune away the rest of the recursions for that partial solution. We also need to keep track of the global optimal solution obtained so far. Once we have reached the last task, we check

Algorithm 1: Exhaustive Algorithm

```
1  P = 1;
2  area = areaPE;
3  time = 0;
4  period = 0;
5  Traverse(1,P,area,time, period);
6  procedure Traverse(i, P, area, time, period)
7      for j = 1 to m_i do
8          /* map task T_i to old PE */
9          tempArea = area + a_{i,j};
10         tempTime = time + t_{i,j};
11         if tempTime > period then
12             tempPeriod = tempTime;
13         if i < N then
14             Traverse(i + 1, P, tempArea, tempTime, tempPeriod);
15         else
16             plot {tempPeriod, tempArea};
17         /* map task T_i to new PE */
18         if i ≠ 1 then
19             tempArea = area + a_{i,j} + areaPE;
20             tempTime = t_{i,j};
21             if tempTime > period then
22                 tempPeriod = tempTime;
23             if i < N then
24                 Traverse(i + 1, P + 1, tempArea, tempTime,
                   tempPeriod);
25             else
26                 plot {tempPeriod, tempArea};
```

if the area requirement of the solution is better than the optimal solution and update the optimal area accordingly.

The complexity of the exhaustive design space algorithm is $O(m^N \times 2^{N-1})$ where m is the average number of CIS versions per task.

3.4 Integer Linear Programming (ILP) Formulation

We now present an Integer Linear Programming (ILP) formulation of the problem so that we can obtain an optimal solution with the help of an off-the-shelf ILP solver. However, as we will observe in the experimental evaluation section, ILP formulation does not scale well with the number of tasks N. So we will present an alternative scalable approach next.

Let $x_{i,j}$ be a binary variable that denotes whether CIS version $C_{i,j}$ is selected for task T_i.

$$x_{i,j} = \begin{cases} 1, \text{ if } C_{i,j} \text{ is selected} \\ 0, \text{ otherwise} \end{cases}$$

For each task T_i, only one CIS version can be selected.

$$\sum_{j=0}^{m_i} x_{i,j} = 1$$

Let $y_{i,k}$ be a binary variable that denotes whether task T_i is mapped to PE_k.

$$y_{i,k} = \begin{cases} 1, \text{ if } T_i \text{ is mapped to } PE_k \\ 0, \text{ otherwise} \end{cases}$$

Each task is mapped to exactly one PE.

$$\sum_{k=1}^{N} y_{i,k} = 1$$

In the summation term we have implicitly defined the number of processing elements to be N. This is necessary to keep the formulation linear. The solution may contain processing elements which have no tasks mapped to them and have to be eliminated. The number of valid processing elements P can be defined as

$$\sum_{i=1}^{N} y_{i,k} - U \times z_k \leq 0; \; \sum_{i=1}^{N} y_{i,k} + 1 - z_k > 0$$

$$P = \sum_{k=1}^{N} z_k$$

where U is a large constant greater than N. z_k is a binary variable which is equal to 1 if there is any task mapped to PE_k and 0 otherwise.

There is one important constraint that is imposed by interval-based mapping approach adopted in our framework. Two consecutive tasks T_i and T_{i+1} should either be mapped to the same PE or mapped to two adjacent PEs. In other words, if task T_i is mapped to PE_k, then task T_{i+1} can only be mapped to either PE_k or PE_{k+1}.

$$\sum_{k=1}^{N} k \cdot y_{i+1,k} \geq \sum_{k=1}^{N} k \cdot y_{i,k}$$

$$\sum_{k=1}^{N} k \cdot y_{i+1,k} \leq 1 + \sum_{k=1}^{N} k \cdot y_{i,k}$$

The period constraint can be imposed as follows.

$$\sum_{i=1}^{N} \sum_{j=0}^{m_i} t_{i,j} \cdot x_{i,j} \cdot y_{i,k} \leq period$$

This is a non-linear constraint. To linearize this constraint, we define a new binary variable $v_{i,j,k}$ where

$$v_{i,j,k} = 1 \Leftrightarrow (x_{i,j} = 1) \;\; AND \;\; (y_{i,k} = 1)$$

This condition can be expressed in linear form as follows.

$$v_{i,j,k} \leq x_{i,j}; \quad v_{i,j,k} \leq y_{i,k}; \quad v_{i,j,k} \geq x_{i,j} + y_{i,k} - 1$$

Now the period constraint can be re-written as

$$\sum_{i=1}^{N} \sum_{j=0}^{m_i} t_{i,j} \cdot v_{i,j,k} \leq period$$

Our objective function is to minimize the total area required

$$\text{Total area} = \sum_{i=1}^{N} \sum_{j=0}^{m_i} a_{i,j} \cdot x_{i,j} + P \cdot \text{area}PE$$

The most area-efficient solution can be obtained by minimizing the objective function under the constraints.

3.5 Dynamic Programming Algorithm

We now proceed to present a dynamic-programming based efficient algorithm that can compute, in pseudo-polynomial time, the area-optimal solution under a period constraint. The algorithm proceeds in two stages. In the first stage, we compute the minimal area required to map a subsequence of tasks on a PE such that the period constraint is not violated. In the second stage, we choose the best partitioning of the tasks.

3.5.1 Customization

The goal of this stage is to compute the area-optimal solution for a sequence of tasks mapping to a single PE under the period constraint. In other words, the total execution time of the tasks should be less than *period* while the area requirement of their selected CIS versions should be minimal.

Algorithm 2: Compute $area_{s,e}$ for all s, e

```
1  for s ← 0 to N do
2      for e ← s + 1 to N do
3          found = FALSE;
4          for A ← 0 to AREA do
5              for j ← 0 to m_e do
6                  if (a_{e,j} ≤ A) then
7                      time_{s,e}(A) = min(time_{s,e}(A), time_{s,e-1}(A − a_{e,j}) + t_{e,j})
8                  end
9              end
10             if (time_{s,e}(A) ≤ period AND !found) then
11                 area_{s,e} = A;
12                 found = TRUE;
13             end
14         end
15     end
16 end
```

Algorithm 2 computes the area-optimal solution for each possible subsequence $T_{s+1}, \ldots, T_e$ mapped to a PE under the *period* constraint. The execution time of the subsequence mapped to a PE can be defined as

$$time_{s,e} = \sum_{i=s+1}^{e} \sum_{j=0}^{m_i} t_{i,j} \cdot x_{i,j}$$

Note that according to our definition, $time_{s,e}$ corresponds to the execution time of the task subsequence $[T_{s+1}, T_{s+2}, \ldots, T_e]$. We assume that $time_{s,s} = 0$, which means that there is no task mapped to the PE. Similarly, we have $area_{s,s} = 0$. We can compute the minimum value of $time_{s,e}$ for all possible values of s, e under different area constraints through dynamic programming. The recursive equation is given as

$$time_{s,e}(A) = \min_{\substack{j=0,\ldots,m_e \\ a_{e,j} \leq A}} (time_{s,e-1}(A - a_{e,j}) + t_{e,j})$$

Basically, the dynamic programming algorithm works as follows. When we

are computing $time_{s,e}(A)$, we go through all the CIS versions of task T_e. For each CIS version $C_{e,j}$ that requires an area not more than A, we pre-allocate the required area and put the rest of the tasks T_{s+1} to T_{e-1} in the remaining area $A - a_{e,j}$. The execution time for this allocation is computed as $t_{e,j} + time_{s,e-1}(A - a_{e,j})$. We then choose the CIS version of task T_e with minimal resulting execution time value and record it as $time_{s,e}(A)$.

We now know how to compute the minimal execution time for the task sequence $T_{s+1} \dots T_e$ under various area constraints. For each task sequence, the algorithm increases the area budget at every iteration, and the execution time decreases correspondingly. Hence, the area budget of the very first iteration where the execution time falls below the period constraint defines the minimal area. The constant $AREA$ is set at a large value such that all the tasks can select their best possible CIS version. The complexity of the algorithm is $O(N^2 \times AREA \times m)$, where m is the average number of CIS versions per task.

We do not take into account the communication cost between the PEs. However, it is fairly straightforward to include communication cost into our framework. We simply need to add area and performance overhead of communication while computing $area_{s,e}$ in Algorithm 2.

3.5.2 Partitioning

Algorithm 3: Compute $Area_N|P$

```
1 for e ← 1 to N do
2 |   Area_e|1 = area_{0,e};
3 end
4 for p ← 2 to N do
5 |   for e ← 1 to N do
6 |   |   Area_e|p = min_{k=1,...,e} (Area_k|(p − 1) + area_{k,e} + areaPE)
7 |   end
8 end
```

Now we focus on partitioning the tasks. We define $Area_N|P$ as the minimal area required to execute tasks $T_1, \dots, T_N$ on P processing elements such that the period constraint is not violated. Again we employ dynamic programming algorithm to compute this value. Clearly, $\min_{p=1,\dots,N} Area_N|p$ denotes the minimal area required to execute the entire task sequence $T_1, \dots, T_N$ on at most N processing elements.

Algorithm 3 returns the values of $Area_N|P$. The algorithm iterates over the number of processing elements p. Given a fixed number of processing elements p,

we iterate over the number of tasks e. Note that $Area_e|p$ computes the minimal area required to execute tasks $T_1, \ldots, T_e$ on p PEs such that the period constraint is not violated. We need to create p partitions such that each partition will be mapped to one PE. The recursive equation is defined as

$$Area_e|p = \min_{k=1,\ldots,e}(Area_k|(p-1) + area_{k,e}) + areaPE$$

When there is only one PE, all tasks are simply mapped to it, which is the initialization statement for $Area_e|1$. The basic idea of the recursive step is to check all possible partition points for the last PE. A partition point k partitions the task chain into two parts: task subsequence $[T_1, \ldots, T_k]$ and task subsequence $[T_{k+1}, \ldots, T_e]$. The second task subsequence $[T_{k+1}, \ldots, T_e]$ is mapped to the last PE and the first task subsequence is mapped to $p-1$ processing elements. In that case, the minimal area requirement for the last PE will be $area_{k,e} + areaPE$ where $area_{k,e}$ is the area corresponding to CIS versions computed using Algorithm 2. As we are computing our solutions iteratively, we have already computed $Area_k|(p-1)$ which corresponds to the minimal area solution for the first task sequence on $p-1$ PEs. The summation of the two returns the minimal area with last partitioning point at T_k. Among all the partitioning points ($k = 1, \ldots, e$), we select the one with the minimal area requirement.

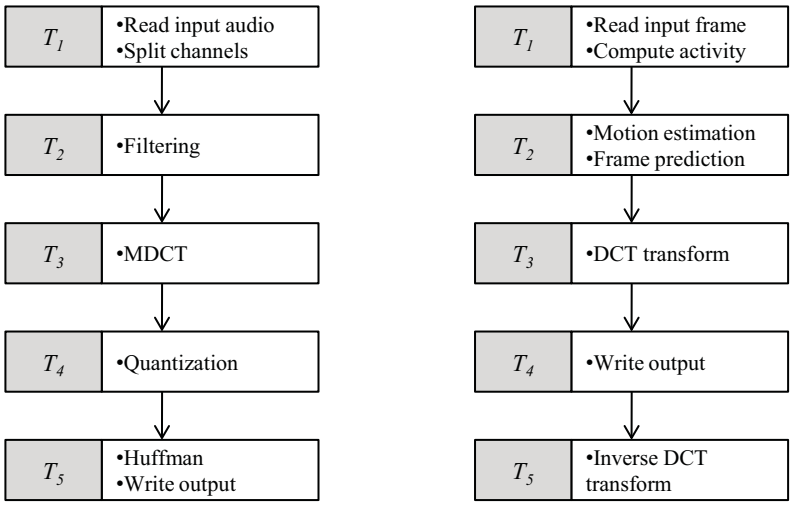
Notice that when $k = e$, the second task subsequence will be empty and $area_{e,e} = 0$. This will essentially create additional idle PE in the end, which will increase the area by $areaPE$ without any performance benefit. Hence this solution will be eliminated. Similarly, if $e < p$, that is, the number of tasks is less than the number of PEs, we will also get some idle PEs. These idle PEs will add to area without contributing to performance. Again these partial solutions with idle PEs will not be part of the optimal solution.

The complexity of Algorithm 3 is $O(N^3)$.

3.6 Experiment Evaluation

For the experiment evaluation, we use two popular streaming applications, an MP3 encoder and an MPEG-2 encoder. As shown in Figure 3.2, each application consists of a number of tasks, which are the compute-intensive kernels.

The base processing elements used in our experiments are the extensible Tensilica Xtensa LX2 processor cores that can be configured for applications-specific instruction set extension. Together with a hardware multiplier, 32KB of data caches, and 4KB of instruction cache, each Xtensa LX2 processor requires about 231K gates, and can run at 326MHz using 0.13μm LV manufacturing



a) Tasks of MP3 b) Tasks of MPEG

Figure 3.2: Task graphs of MP3 encoder and MPEG-2 encoder.

process.

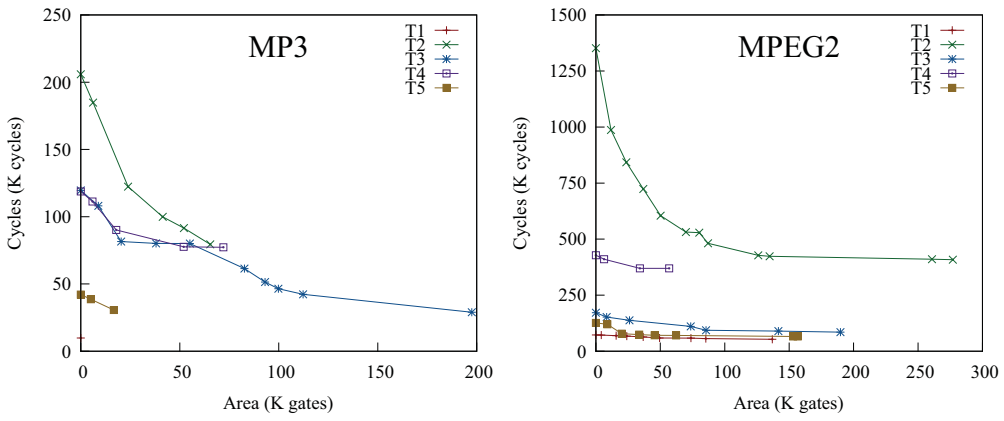


Figure 3.3: Custom instruction sets for the tasks in MP3 and MPEG-2

For each task, we use XPRES compiler provided by Tensilica to generate a number of different configurations with varying trade-offs between area and performance. The CIS versions for each task are shown in Figure 3.3. The X-axis represents the area (in gates) and the Y-axis represents the execution time of the task. Some of the CIS versions require almost the same area as the base PE.

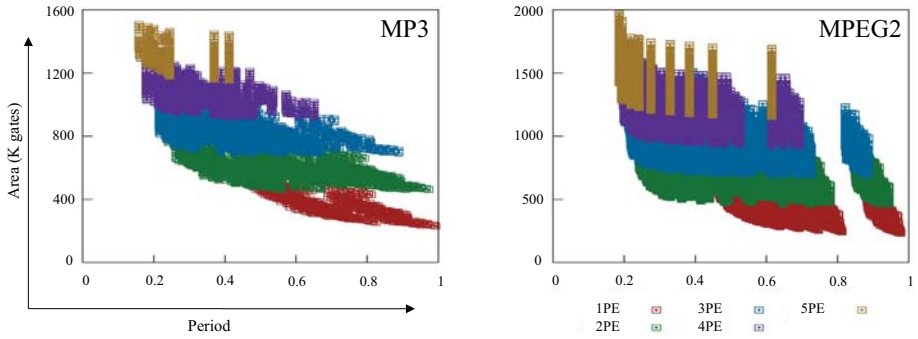


Figure 3.4: Design space for MP3 encoder and MPEG-2 encoder

We first plot the result of the exhaustive design space exploration shown in Figure 3.4. There are 14,400 points in the MP3 encoder design space and 387,072 points in the MPEG-2 encoder design space. The X-axis represents the period normalized with respect to the completely software based implementation on a single PE. The Y-axis represents the total area required by the MPSoC solution. Each color corresponds to the number of PEs in the solution. As can be seen from the figure, the design space is quite complex. It is possible to meet the same period constraint either with a small number of PEs each customized heavily or with a larger number of PEs devoid of customization.

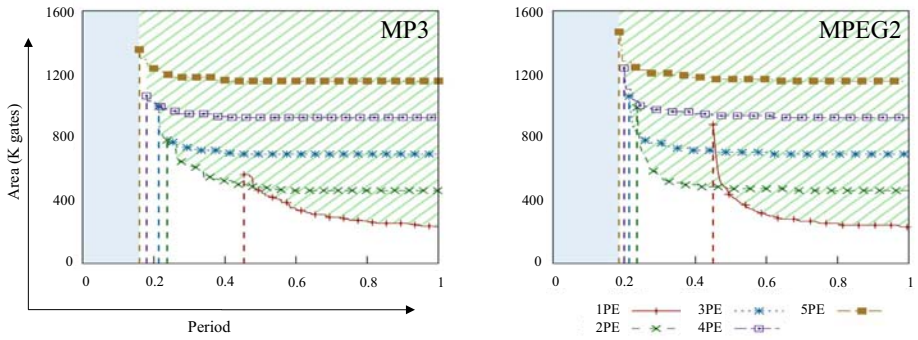


Figure 3.5: Minimal area cost versus period constraint for MP3 and MPEG-2 for different numbers of PEs

Now we focus on generating the area-optimal solution under a given period constraint. For each application, we vary the period constraints from 0 to 1.0 (in steps of 0.01) of the period with pure software implementation on a single PE without any customization. The software execution on a single PE without customization is the solution with minimum area. For clarity, we plot the results

Number of tasks	EA	ILP	DP
5	0.01 sec	1 sec	0.01 sec
7	1.18 sec	5 min	0.01 sec
10	12 min	9 min	0.03 sec
12	18 hour	2 hour	0.05 sec
15	-	-	0.09 sec
20	-	-	0.20 sec

Table 3.1: Analysis time for exhaustive (EA), ILP, and the dynamic programming (DP) approach

for different number of PEs though our algorithm can easily identify the optimal number of PEs.

Figure 3.5 plots the results for the two applications. The light blue region in the left of each graph corresponds to the infeasible region where the period constraint is too small. The white region under the curves corresponds to the infeasible design space due to tight area budget. The third region, in light green, is the feasible design space. The Pareto-optimal solutions in this feasible design space are highlighted in the figure. Given a period constraint, the corresponding optimal point tells us how many PEs should be used and the minimal area cost. The vertical dashed lines indicate the maximum accelerations that can be gained for different numbers of PEs.

Finally, we compare the analysis time for exhaustive algorithm (EA), ILP solver and our proposed dynamic programming algorithm (DP) on Intel Xeon 2.53GHz processor with 16GB memory. We used LINGO, a commercial ILP solver [87] for our experimental evaluation. For this set of experiments, we generate synthetic task graphs with number of tasks varying from 5 to 20. The average number of CIS version per task is set at 5. The performance gain of each CIS version ranges between 1,000 to 10,000 time units. The hardware area is between 1 to 100 units. The performance gain increases with hardware area.

Table 3.1 shows the analysis time for the three methods. The analysis time corresponds to finding the area-optimal solutions given a fixed period constraint. Given an application and a fixed period constraint, the analysis time remains unchanged for different runs of exhaustive algorithm and dynamic programming approach. However, for the ILP solver, analysis time can vary; so we report the average analysis time.

As shown in the table, dynamic programming approach improves the analysis time dramatically and still produces the optimal solution. With 15 tasks and more, exhaustive algorithm and ILP solver fail to return optimal solutions within a reasonable time. However, dynamic programming approach still manages to

identify the optimal solution within short time.

The exhaustive algorithm is more powerful than ILP solver if the designer is interested in all the Pareto-optimal solutions, that is, the tradeoff between area and period. The exhaustive algorithm can explore the entire design space in one go. The ILP solver, on the other hand, needs to be invoked with different period constraints. Even the dynamic programming approach needs to be invoked with different period constraints. However, our experiments show that dynamic programming approach is way faster than exhaustive algorithm for a task graph with 12 tasks and 100 different period constraints.

3.7 Chapter Summary

In this chapter, we expose the challenges of customizing the MPSoC system statically using hardwired circuits. This first step towards the MPSoC customization problem highlights complexity of the design space exploration technique, which should consider multiple design factors such as task mapping, selections of alternative custom instruction sets, resource competition among the cores and others. An efficient hierarchical algorithm is then proposed to design the most resource-efficiently customized MPSoC platform for mapping linear task graphs of streaming applications under all the constraints. The proposed dynamic programming algorithm achieves optimal solutions while decoupling the task mapping and the customizations. Using two popular streaming applications (MP3 encoder and MPEG-2 encoder) with Tensilica extensible processors, the experimental validation confirms the efficiency of our approach.

Chapter 4

S-CGRA: Customizable MPSoC design

In the previous chapter, we have highlighted the challenges in static MPSoC customization. When reconfigurability is taken into consideration, the MPSoC customization becomes a much more complicated problem. Our main goal in this dissertation is to propose a full design automation tool chain for dynamic MPSoC customization, covering the three major topics including architectural designs, compilation supports and design space exploration. In this chapter, we will focus on the first topic, architectural design. In order to provide reconfigurability, the MPSoC system should include a shared reconfigurable fabric. The fabric could either be tightly coupled within multiple processor pipeline or used as a coprocessor. The tightly coupling approach will essentially create a conjoined-core chip [77], which is not scalable with number of cores. Thus, in our design, we will mainly focus on coprocessor design, which could be shared among multiple cores without introducing large overheads.

4.1 Overview

A widely adopted reconfigurable coprocessor is the coarse-grained reconfigurable array (CGRA), which is used to accelerate computational intensive loop kernels. A CGRA is basically formed by arranging a set of functional units in a two-dimensional topology. In order to design the reconfigurable coprocessor, we have to first focus on the design of the primary processing element in the CGRA, the functional unit. Normally, one functional unit is designed to execute one operation each cycle. However, the cycle time of the functional unit is long enough to accommodate multiple operations within one cycle. In this context,

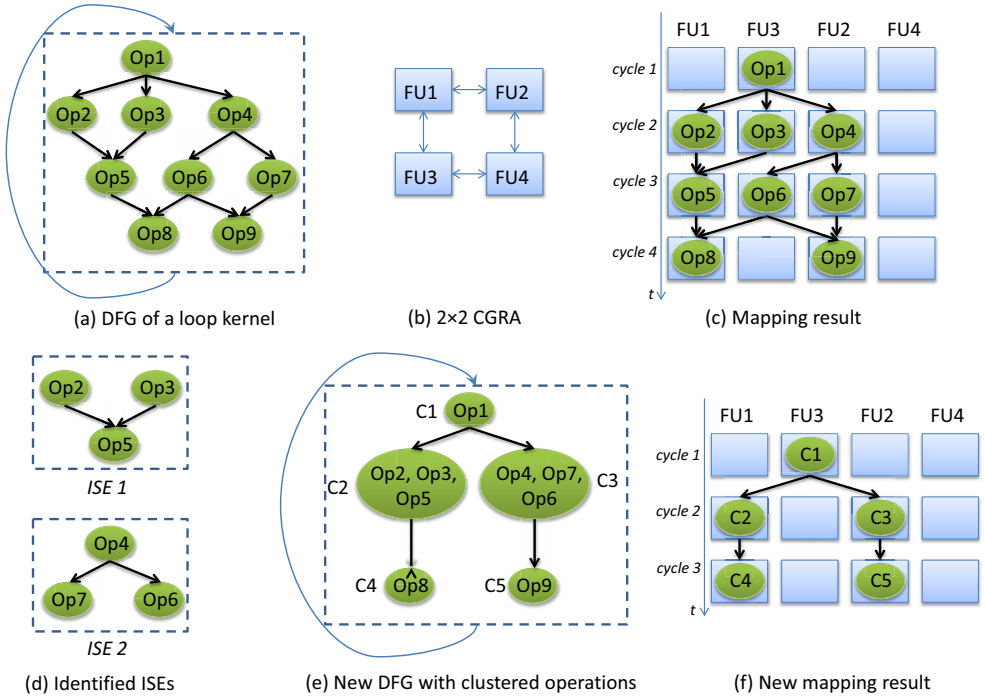


Figure 4.1: A motivating example

each functional unit can be specialized to execute one pattern of the computational intensive loop kernels every cycle. Recall that an ISE is generated by encapsulating a computational intensive pattern, which share the similarity of the function defined here for functional unit in CGRA. Thus, we can potentially use ISEs from the context of fine-grained customization in the functional unit design for our CGRA coprocessor. This also enables us to view the entire loop kernel as a DFG consisting of multiple clusters, where each cluster could be either a operation or an ISE containing a group of operations. Figure 4.1 shows a motivating example. Figure 4.1(a) shows the DFG of a loop kernel. With a one to one mapping strategy, Figure 4.1(c) shows the mapping result by mapping the DFG to a 2×2 CGRA, which is shown in Figure 4.1(b). Figure 4.1(d) gives the ISEs identified from the given loop kernel. These ISEs are represented as clustered nodes in the clustered graph shown in Figure 4.1(e). The final mapping result is presented in Figure 4.1(f). In this example, enabling multi-operation execution saves one cycle execution time for one loop iteration, which highlights the potential benefits of using a complex functional unit design.

Thus, a promising approach for the functional unit design is to support ISEs across multiple application domains. In the following, we first perform an empirical ISE analysis of a set of representative embedded applications. The application analysis classifies commonly occurring sequences of arithmetic and logical oper-

ations, which are essentially candidates for selection as ISEs. To support these sequences, we identify several smaller functional units that can be combined to form a reconfigurable multi-stage ALU (arithmetic logic unit). This accelerator, which we call a *Specialized Functional Unit (SFU)*, can execute ISEs from a variety of applications. The efficiency of the proposed SFU is evaluated by integrating into the processor pipeline in parallel with the ALU. The SFU is accessed using a non-traditional instruction fetch and decode mechanism, which we call a *Just-in-Time Customizable (JITC)* core. When an opcode matching an accelerator function is read from the instruction cache, the traditional fetch-and-decode mechanism is suppressed, enabling execution of an ISE on the SFU. After the evaluation of the SFU, we will propose a novel specialized CGRA (S-CGRA) design using the SFU as its primary processing element. The proposed S-CGRA is further coupled with multiple cores to complete our final customizable MPSoC system design.

4.2 SFU as the Primary Processing Element

As mentioned, our first step is to propose an efficient design for the primary processing element in the reconfigurable coprocessor. The processing element is used to accelerate commonly occurring expressions encapsulated as ISEs.

4.2.1 Analysis of ISEs

We first analyze the ISEs found across a range of applications to identify the characteristics that lead to performance acceleration. With the analysis results Section 4.2.5 describes the experimental setup used for this analysis.

Figure 4.2 shows the dataflow graph (DFG) representing an ISE with four inputs, two outputs, and six arithmetic and logical operations. The ISE obtains speedup by either exploiting *instruction- level parallelism (ILP)*, or chaining consecutive operations in a single-cycle. The out-of-order processors with dynamic instruction scheduling can extract ILP automatically, but with high area overhead and energy consumption; alternatively, a compiler can extract ILP and schedule operations statically as in a VLIW architecture. Operation chaining, in contrast, depends on the frequency of the processor; for the DFG in Figure 4.2, a multiply-accumulator could execute the multiply-add portion of the ISE in one cycle, while a chain of arithmetic and logic operators could execute the shift and logical-AND operations in a second cycle. The fundamental question that we answer in this section is whether parallelism or operation chaining has a stronger correlation with the speedup obtained by an ISE.

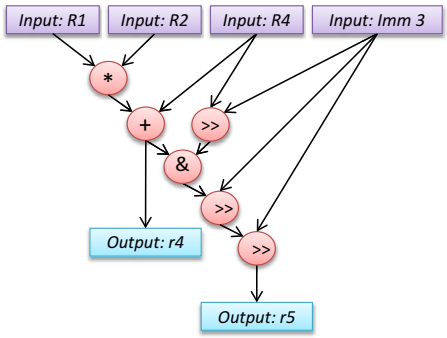


Figure 4.2: Dataflow Graph (DFG) of an ISE

Exploring Inter-Operation Parallelism

ILP extraction and exploitation is fundamental to computer architecture research. In terms of ISE identification, increasing the I/O bandwidth to/from an ISE leads to wider dataflow graphs with higher ILP [10, 8, 9, 131]; however, *prior work [31, 29, 131] has reported that up to four inputs and two outputs are sufficient to achieve near-optimal speedup for ISEs in most cases.* Therefore, we study ISEs with at most four input and two output operands in this work. The average parallelism of an ISE is defined as

$$avarage\ parallelism = \frac{\#\ of\ total\ operations}{critical\ path\ length}$$

where the critical path length is the number of operations along the longest path in the DFG. For example, in Figure 4.2, the critical path length is 5 and the average parallelism is $6/5 = 1.2$. The average parallelism captures the ILP available within ISEs, i.e., the average number of operations that can execute in parallel per cycle. The maximal parallelism is the maximum number of DFG operations executing concurrently using the *As Late As Possible (ALAP)* scheduling policy. For example, in Figure 4.2, with an ALAP scheduling, the ADD operation executes in parallel with the first shift, so the maximal parallelism for the ISE is 2.

We analyzed the average and maximal parallelism of ISEs found in 21 Mibench and Mediabench applications. The average parallelism is close to 1, and the maximum parallelism never exceeds 2, across all of the applications as shown in Figure 4.3. This confirms that ISEs with up to 4 inputs and 2 outputs have limited ILP, and at most two parallel functional units should suffice to exploit this limited parallelism.

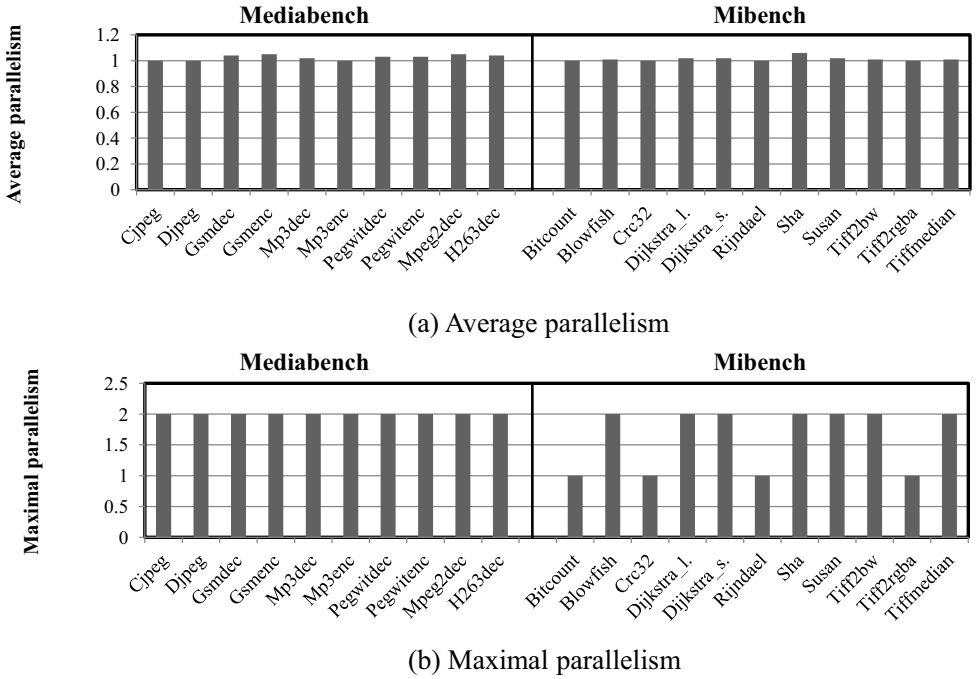


Figure 4.3: Parallelism explorations for Mediabench and Mibench benchmark suits

Exploring Critical Path Length

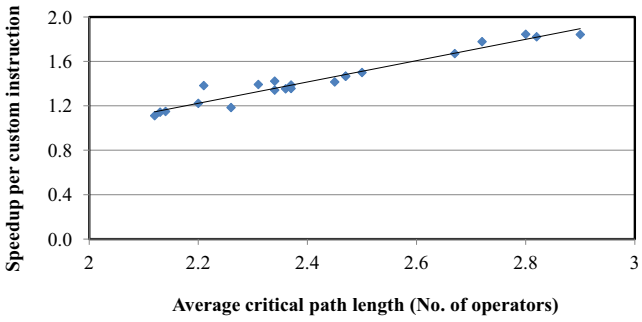


Figure 4.4: Correlation between critical path length & speedup

Impact of operation chaining on speedup To investigate the impact of operation chaining on the speedup of ISE, we measure and report the ISE's average critical path length. This metric is closely related to the number of dependent operators that could be chained and executed in one cycle. Figure 4.4 shows the correlation between the average critical path length and the average speedup per ISE. Each point in the graph corresponds to one particular application from the set of 21 Mibench and Mediabench applications. The linear trend line establishes

a linear correlation between the two variables. Thus, *ISEs with a longer critical path tend to achieve the highest speedups.*

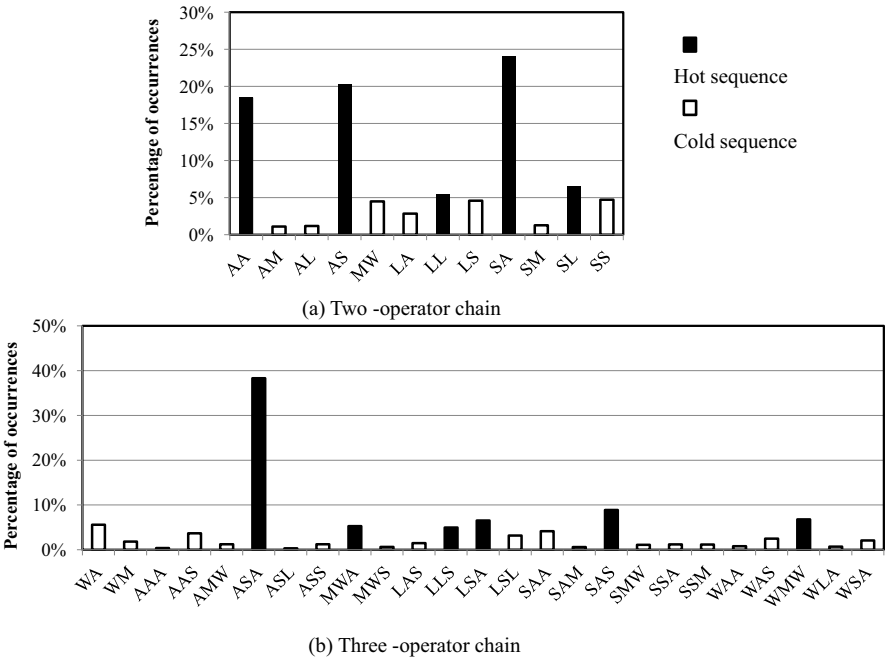


Figure 4.5: Hot sequences in operation chaining

Hot sequences in operation chaining Our objective is to design an SFU that exploits operation chaining; however, the SFU cannot possibly support every possible operation sequence. Thus, the first step is to identify "hot" sequences, i.e., those that appear frequently in ISEs across a wide variety of applications. The primary objective of the SFU is to execute the hot sequences efficiently.

First, we classify the operations into five groups: arithmetic operations (A type), logical operations (L type), shift (S type), wire (W type), and multiplication operations (M type). W type operations are essentially move instructions that are converted to wiring when synthesized as part of an ISE. All operations belonging to each class have approximately equal latencies, and can be implemented using a single physical execution block. A sequence is hot if it occurs above a certain threshold; we use a threshold value of 5%, which captures the most frequent sequences. We restrict the number of operators per sequence to 3, as the average critical path length does not exceed 3 (see Figure 4.4).

Figure 4.5 shows the hot sequences from an analysis of the 21 Mibench and Mediabench applications. Only a handful of operator chains (bars filled in black) appear frequently: there are five hot two-operator sequences and six hot three-

operator sequences. The frequency of occurrence of each sequence is averaged across the benchmarks; certain sequences may occur frequently in some, but not all, of the applications. For example, the sequence *MWA* has a frequency of 66% in the `Tiff2bw` application; however it only has 5% frequency averaged across all benchmarks.

4.2.2 SFU Design

The SFU is designed to support the 11 hot sequences that appear frequently in ISEs across the 21 MiBench and MediaBench applications that we analyzed in the preceding section. The hot sequences are: *AA, AS, LL, SA, SL, ASA, LLS, LSA, SAS, MWA, WMW*. The SFU is designed to execute each of these hot sequences in one clock cycle. We build up the SFU incrementally, starting from a basic functional unit, described next.

Basic functional unit (BFU) Figure 4.6(a) illustrates a basic functional unit, which is an ALU followed by a shifter. The BFU supports five operation sequences: *A*, *L*, *S*, *AS*, and *LS*. To support sequences *A* and *L*, the shift length is set to zero; to support sequence *S*, the ALU performs an identity operation (e.g., OR identical inputs) on the input operand. A regular expression that enumerates all possible sequences supported by the basic functional unit is $(A \mid L \mid \varepsilon)(S \mid \varepsilon)$ where ε is the empty string.

Fused basic functional units The basic functional unit can only support one (*AS*) out of eleven hot sequences. The fused basic functional unit chains two basic functional units sequentially along with a bypass line, as shown in Figure 4.6(b). Using regular expression notation, the fused basic functional unit supports sequences $((A \mid L \mid \varepsilon)(S \mid \varepsilon))^2$, which encompasses all hot sequences that do not include a multiplier.

Complex functional unit The two remaining sequences are *MWA*, *WMW*, where *W* selects the upper- or lower 32-bit portions of the 64-bit output of a 32×32-bit multiplier; this is done using multiplexers and control signals. The remaining subsequence, $M(A \mid \varepsilon)$, is a fused multiply-addition operation that can be implemented using an MAC unit. We refer to this operator as a complex functional unit, as shown in Figure 4.6(c). The rationale for including the ALU and the shifter in the complex functional unit is to provide additional parallelism in the SFU as explained next.

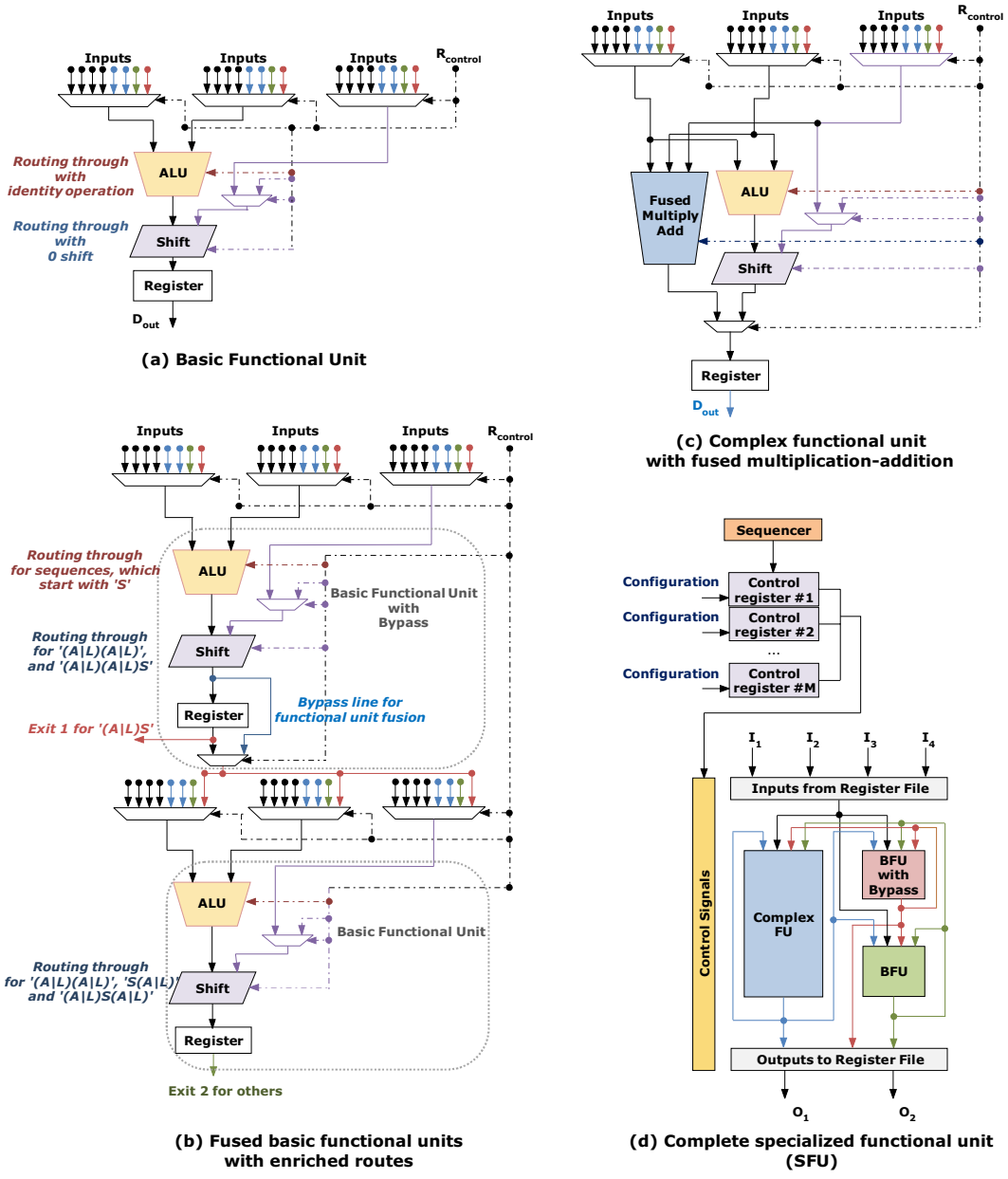


Figure 4.6: Design of the Specialized Functional Unit (SFU)

Component	Area(μm^2)	delay(ns)
Basic Functional Unit	9856.7078	0.7231
Fused Basic Functional Unit	27913.4943	1.5424
Complex Functional Unit	49780.5275	1.6379
SFU	80502.7823	1.6499

Table 4.1: Area and delay for the SFU components

Specialized functional unit (SFU) We synthesized the different components of the SFU in Synopsys Design Compiler with PDK 45nm standard cell library. Table 4.1 provides the area and delay values for the components. We observe that the fused basic functional unit in Figure 4.6(b) has a latency of 0.7231ns while the complex functional unit supporting multiplication in Figure 4.6(c) has a latency of 1.6379ns. Therefore, we form the SFU by placing a complex functional unit in parallel with a fused basic functional unit, without extending the critical path. This dual-path architecture supports the maximum parallelism of 2 that is present in the ISEs. Functional units within the SFU are fully connected, and each internal functional unit has access to four input and two output registers. Fig. 4.6(d) shows the SFU architecture, which supports single-cycle execution of all eleven hot sequences at 606MHz clock frequency and $0.08mm^2$ area.

The SFU requires 62 bits for control signals. The fused basic functional unit has six 8-input multiplexers, and the complex functional unit has three 8-input multiplexers; as each 8-input multiplexer requires 3 control bits, the SFU requires 27 bits to select the inputs to the various internal functional units. With two 4-input multiplexers at the output, 4 additional bits are required for output register selection. The SFU has three shifters, each of which has a 2-input multiplexer driving one of its inputs, so 3 additional bits are required to control these multiplexers. 15 bits are required for register storage (5 bits per functional unit), 12 bits for operation control (each ALU and MAC supports up to 16 operations, so 4 control bits for each are required), and 1 bit is required for bypass selection within the fused basic functional unit. This adds up to 62 control bits in total.

Multi-cycle execution of ISE on SFU The SFU supports single-cycle execution of most ISEs whose critical paths consist of up to 3 operators. Indeed, *almost 90% of the ISEs can be executed in one cycle on the SFU.* But the SFU can also support multi-cycle execution of ISEs with longer critical paths through reconfigurations. Reconfiguring the SFU involves changing the values of the 62 control bits each cycle. The compiler has to exploit the reconfiguration ability of SFU, which we will detail in Section 4.2.4. We observe that almost 99.77% of the ISEs can be executed within 4 cycles.

4.2.3 JITC Architecure

This section describes how to integrate one or more SFUs into a processor pipeline to achieve Just-in-Time Customization. To simplify discussion, we as-

sume a simple 5-stage, single-issue in-order RISC pipeline. Figure 4.7 shows the modified pipeline structure.

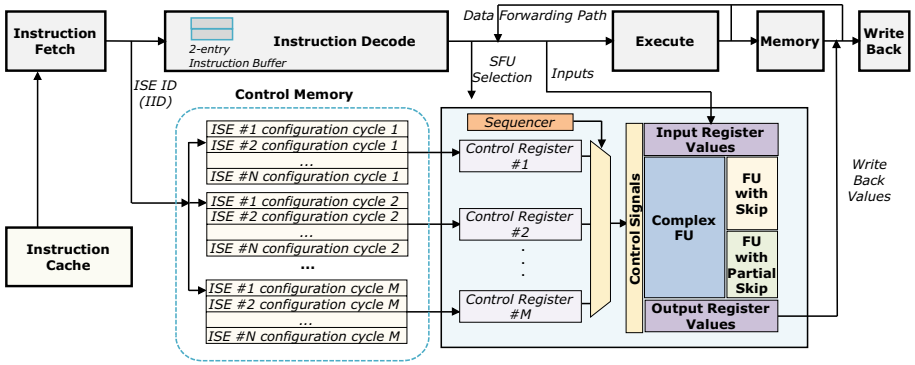


Figure 4.7: Just-in-Time Customizable (JITC) processor architecture: Integration of SFUs in the pipeline datapath

ISE Encoding and Decoding We assume 32-bit ISA with 4-bits per register encoding corresponding to a 16-entry register file. As shown in Figure 4.8(a), 12 bits are required to encode two source and one destination registers and 8-bit opcode supports 256 instructions.

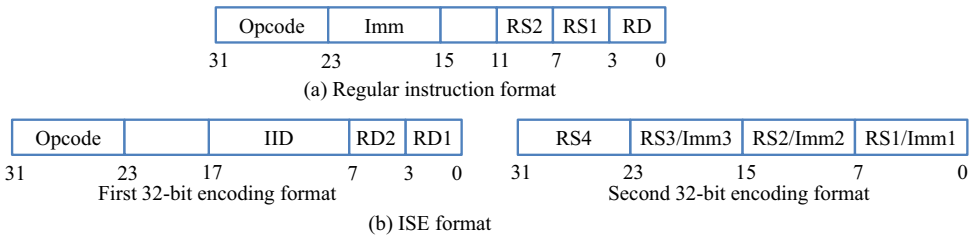


Figure 4.8: ISE encoding format

With a 32-bit instruction format, it is necessary to encode ISEs using two consecutive 32-bit words, as shown in Figure 4.8(b). The first 32-bit is used to encode 8-bit Opcode and two 4-bit destination registers. We can support at most one ISE if one of the opcodes (out of 256) is reserved for custom instructions. To extend the number of ISEs, we use 10 unused bits to encode the *IID* or ISE ID. When the opcode signifies a ISE, the *IID* field indicates which ISE it is. This allows us to encode at most 1024 ISEs. The second 32-bit encodes the source registers and/or immediate values. Note that we support ISEs with different addressing modes. For example, an input operand can be a register

index or an immediate value. As at least one input operand should be a register, we assume the first input operand is always a register index; the other three are either registers or immediate. So we have four addressing modes: RRRR (all registers), RRRI (one immediate), RRII (two immediates), and RIII (three immediates), which can be supported by reserving four opcodes in the ISA.

To support ISE decoding, we require a 2-entry instruction buffer between the fetch and the decode stage so that the decoder has access to the entire 64-bit custom instruction. When the decoder detects a ISE opcode, it decodes the second half of the ISE in the buffer for the source operands.

Multi-banked control memory ISEs that execute on the SFU require more control signals than regular instructions that access the ALU and/or memory. We store the control signals for each ISE in an on-chip control memory that is accessed in parallel with the instruction decode phase of the pipeline when an ISE is decoded. The IID field is used as an index into the control memory. The control memory consists of M banks. where the i^{th} bank stores the control signals required for the execution of an ISE on the SFU in the i^{th} cycle ($M = 4$ in our design). The banks are accessed in parallel to retrieve all the control signals of an ISE. With 10-bit IID field, storage space for 1024 entries is required, where each entry holds 62 bits; the approximate size of the control memory is 32KB. Additionally, the control memory needs to store the number of cycles required to execute each ISE. The number of cycles and the control bits read from the control memory are written into the SFU's sequence and control registers, respectively.

Execution of ISEs on SFU Only one SFU needs to be integrated into a single-issue in-order pipeline; however, for multi-issue out-of-order execution, our experiments confirm that four SFUs achieve near-optimal acceleration. When all of the input operands to an ISE are ready, the ISE can start execution on the SFU. When the ISE execution inside the SFU completes, the output operands are written to the register file and the SFU becomes free to execute another ISE.

4.2.4 Compiler Support

ISE identification and selection Our JITC architecture needs compiler support to automate the process of identifying the ISEs and mapping them onto the SFU.

Given an application, we first detect the "hot" basic blocks through profiling. The DFGs of these hot basic blocks are then analyzed to identify all the ISE candidate patterns [132]. We impose the restriction of at most 4 input operands

and 2 output operands per candidate pattern as noted earlier [31, 29, 131]. We also do not allow memory accesses and control flow operations within an ISE. Once all the candidate patterns have been identified, we select a subset of these patterns such that (a) each node in the DFG of a basic block is covered by at most one candidate pattern, and (b) the cumulative speedup of the selected patterns is maximized. The speedup of a pattern is defined as $\frac{t_{sw}}{t_{custom}}$, where t_{sw} is execution cycles on the base processor core and t_{custom} is the execution cycles when the pattern is implemented in ASIC.

Mapping algorithm We employ a greedy heuristic for mapping of an ISE on the SFU. Our objective is to minimize the number of cycles required to execute the ISE on the SFU. We borrow the notion of Routing Resource Graph (RRG) [91] from the FPGA domain to represent the resources of SFU in different cycles and the connections among them. The connections are generated such that the components of the SFU in one cycle are connected to the components of the SFU in the next cycle. For example, the RRG in Figure 4.10(a) shows how the components of the SFU in cycle 1 are connected to the SFU in cycle 2. Basically, each of the three functional units in cycle 1 is connected to any one of the three functional units and the output registers in cycle 2, which means that the value generated by one functional unit could be read by any functional unit or stored in the output registers in the next cycle. Note that the input registers are connected to the functional units in the same cycle as their values could be read within the period of one cycle. Similarly, the connection between two fused functional units also appears in the same cycle.

Algorithm 4 presents the pseudo-code of our mapping algorithm. We first assign level values to each of the nodes in the DFG according to an *As Late As Possible (ALAP)* scheduling policy. Note that any advanced scheduling policy that helps to align the predecessors close to their successors can be adopted here. However, choosing the best policy is not the main focus here. The nodes (functional units) in the RRG are also ordered according to their time cycle. We also ensure that the two basic functional units (BFU) have higher priorities compared to the complex functional unit within a cycle.

The greedy heuristic maps the nodes of the DFG to the RRG in the level order. Consider a node u in the DFG. Suppose we are mapping a node u in the DFG that has predecessors $v_1, v_2, \ldots v_x$. We identify the closest common free successor functional unit of $Map(v_1), Map(v_2), \ldots, Map(v_x)$. $Map(v)$ stands for the functional unit to which operator v has been mapped to. We simply map u to this free functional unit. If u has no predecessors, then the chosen free

Algorithm 4: Mapping_algorithm

```
Input: The data flow graph (DFG) of the ISE and the routing resource graph (RRG) of the
       SFU.
Output: The generated configuration if mapping is successful.
Begin
    max_level = Assign_level_ALAP(DFG);
    Initialize(RRG);
    For i ← 1 to max_level do
        For All each operator u in DFG do
            If u→level == i then
                successful = 0;
                If u has only one immediate predecessor v then
                    If u is v's only immediate successor then
                        If Map(v)→Res(u) == Available and Map(v)→Res(v) is
                        connected to Map(v)→Res(u) then
                            Map(v)→component(Res(v)) = Occupied;
                            successful = 1;
                        Endif
                    Endif
                Endif
                If successful == 0 then
                    For cycle = 1 to 4 do
                        If successful == 1 then
                            break;
                        Endif
                        For all functional unit n in RRG(cycle) do
                            If n→status == Free and n→component(Res(v)) ==
                            Available then
                                Feasible = 1;
                                For each immediate predecessor v of u do
                                    If Map(v) is not connected to n in RRG then
                                        Feasible = 0;
                                    Endif
                                Endfor
                                If Feasible == 1 then
                                    n→status = Mapped; n→component(Res(v)) =
                                    Occupied;
                                    successful = 1;
                                    break;
                                Endif
                            Endif
                        Endfor
                    Endfor
                Endif
                If successful == 0 then
                    Return FAIL;
                Endif
            Endif
        Endfor
    Endfor
    Return Gen_conf(DFG, RRG);
End
```

functional unit would be the one with minimal cycle time stamp.

The only special case we need to take care of is when u has only one immediate predecessor v and v has u as its only immediate successor. In this case, we have to explore the components within the functional unit $Map(v)$. Suppose $Res(u)$ stands for the component resource u requires. If in functional unit $Map(v)$,

there is an available component $Res(u)$ and its component $Res(v)$ is connected to component $Res(u)$, then we can directly map u to functional unit $Map(v)$. This takes care of operator chaining.

The process ends once all the nodes of the DFG have been mapped to the RRG. In the rare event that the mapping fails because a pattern requires more than 4 cycles, the pattern cannot be accelerated using our SFU and is eliminated from further consideration. Once the mapping has been finalized, we generate control signals corresponding to each cycle of execution of the DFG on the SFU.

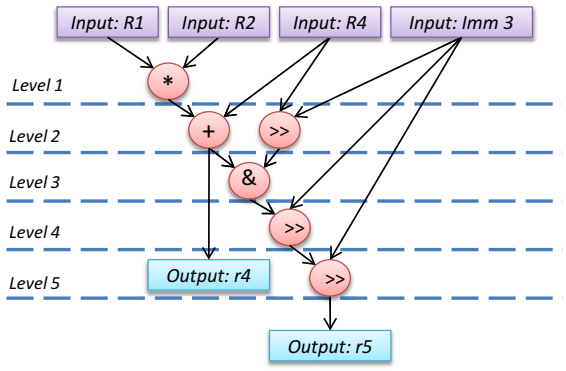
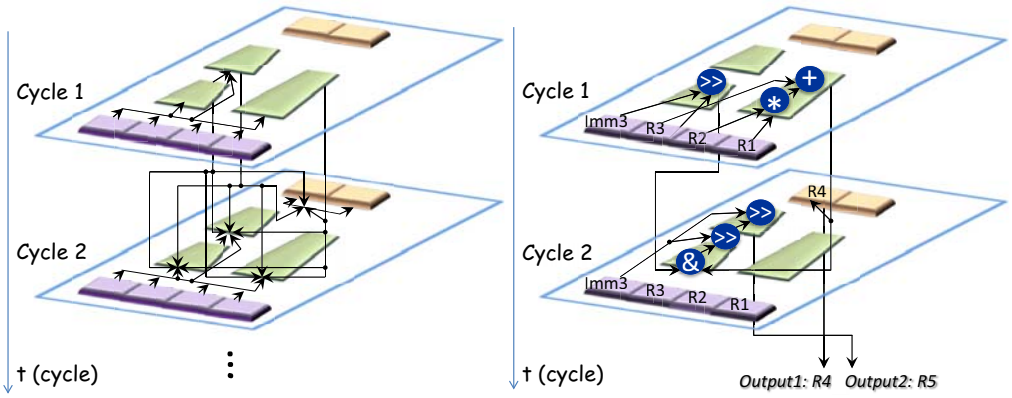


Figure 4.9: *L*evel order assignment for DFG nodes with ALAP scheduling



(a) Routing Resource Graph (RRG) of SFU

(b) Final mapping

Figure 4.10: Routing Resource Graph (RRG) of the SFU and the final mapping of the DFG to the RRG

Mapping Example We now show an example of how the DFG in Figure 4.2 is mapped to the RRG of SFU. First, we assign level order to the DFG nodes using *ALAP* scheduling policy; the results are shown in Figure 4.9. Then, we try to map all the operators in DFG level by level from level 1 to level 5. The

first operator we encountered is a *multiplication*; so we find the first free complex functional unit with the lowest cycle time, which is the complex functional unit in cycle 1 and map this *multiplication* to the multiplier component of it.

We continue the mapping in level 2 and find an *addition* operator to be mapped. The *addition* has only one immediate predecessor, which is the *multiplication* we just mapped and the *multiplication* has this *addition* as its only immediate successor. So we can try to map this *addition* to the same complex functional unit. Fortunately, we find the MAC inside the complex functional unit can support this mapping with the connection requirement satisfied.

The next operator in level 2 is a *shift*. We simply find the first free functional unit in cycle 1 and map it there. Now we continue to map the *and* operator in level 3. The *and* operator has two predecessors; so the earliest common successor should be a functional unit in cycle 2. So we pick the first basic functional unit in cycle 2 as it has higher priority. In the next level, the operator *shift* can be mapped to the same functional unit as the *and* operator. Finally, another *shift* operator in level 5 is mapped to the second basic functional unit in cycle 2 as it is the closest one. The final mapping is shown in Figure 4.10(b).

Binary executable and configuration generation Once the compiler decides on the mapping of an ISE to the SFU, it generates the corresponding control signals for the ISE. The compiler then generates the binary executable that replaces, for each occurrence of a candidate pattern, a sequence of instructions from the base ISA with the corresponding custom instruction. Finally, the control signals are loaded into the control memory before the application initiates execution. Note that as the subset of ISEs selected is different for different applications, the content of the control memory is different for each application. In other words, the JITC architecture achieves flexibility by changing the content of the control memory and thereby instantiating different custom instructions per application.

4.2.5 Experimental Evaluation for SFU Design

Experimental Setup We evaluate the performance of JITC core compared to ASIPs [10, 8, 64, 103, 131, 132]. As mentioned earlier, we selected 21 benchmark applications from MiBench [55] and MediaBench [82] to derive the design of the SFU. Here *we use 14 additional applications from SPECInt, HPEC, Olden, and Encrypt benchmark suites to perform cross validation of the SFU design.*

For a fair comparison, we design both the ASIP and the JITC core by augmenting a RISC-like baseline core [11] with no accelerator. For each of the 35

applications, we custom design an ASIP following the standard ISEs identification and selection methodology [10, 8, 103, 131, 132]. That is, we design a total of 35 individual ASIPs, where each ASIP is capable of accelerating the specific application it is design for.

We assume that the clock period of the baseline core is determined by the latency of the MAC unit [10, 131], which also has roughly the same latency as a multiplier [116]. All the designs are synthesized using Synopsys Design Compiler version E-2010.12-SP4 with Free PDK 45nm standard cell library. The MAC unit has a latency 1.58ns; thus the frequency of the baseline core and all the ASIPs are set at 633MHz. JITC core, however, has a frequency of 606MHz constrained by the SFU latency as shown in Table 4.1. Further optimizations could lead to higher frequency of JITC core.

Following prior works [31, 29, 131], we assume that each ASIP can support ISEs with at most 4 input and 2 output operands, and cannot include any memory or control operations. The latency of an ISE in ASIP is obtained by dividing the latency along the critical path by the clock period of the baseline core. The area of each individual ASIP is restricted to the area of the JITC core. This area restriction leads to only 1.5% average performance degradation compared to the theoretical speedup of an ASIP with unlimited area.

We modified the SimpleScalar simulator [11] to integrate the SFUs and corresponding control memory in the pipelined datapath. We modeled both in-order and out-of-order pipelines for JITC core and the ASIPs. For the ASIPs, we assume that all ISEs are implemented as dedicated functional units in the pipeline. We extended the instruction set to support the ISE formats, and modified the gcc cross-compiler for SimpleScalar to identify the ISEs for each application and to generate binary executables that include calls to the ISEs. Table 4.2 shows the configurations for both in-order and out-of-order micro-architecture in SimpleScalar simulator setup. The configuration parameters are chosen to closely match realistic in-order (ARM Cortex-A7) and out-of-order (ARM Cortex-A15) embedded processors.

Results for profiling benchmarks Let us fist focus on performance comparison with profiling benchmarks (MiBench, MediaBench) used to derive the design of the JITC core (left of Figure 4.11). For in-order pipeline, Figure 4.11(a) shows the performance of JITC core and the ASIPs compared to the baseline core with no accelerator. The speedup is defined as $\frac{t_{sw}}{t_{custom}}$ where t_{sw} is execution cycles on the baseline core and t_{custom} is the execution cycles on ASIP or JITC core. We also plot the theoretical speedup for ASIPs — the speedup achievable

	In-order architecture	Out-of-order architecture
Pipeline	1 way	4 ways
RUU size	2 entries	128 entries
IFQ size	4 entries	16 entries
LSQ size	2 entries	16 entries
L1 I-Cache	32KB, 2-way, 1 cycle hit	
L1 D-Cache	32KB, 2-way, 1 cycle hit	
Unified L2	512KB, 4-way, 10 cycle hit	
Control memory	32KB	

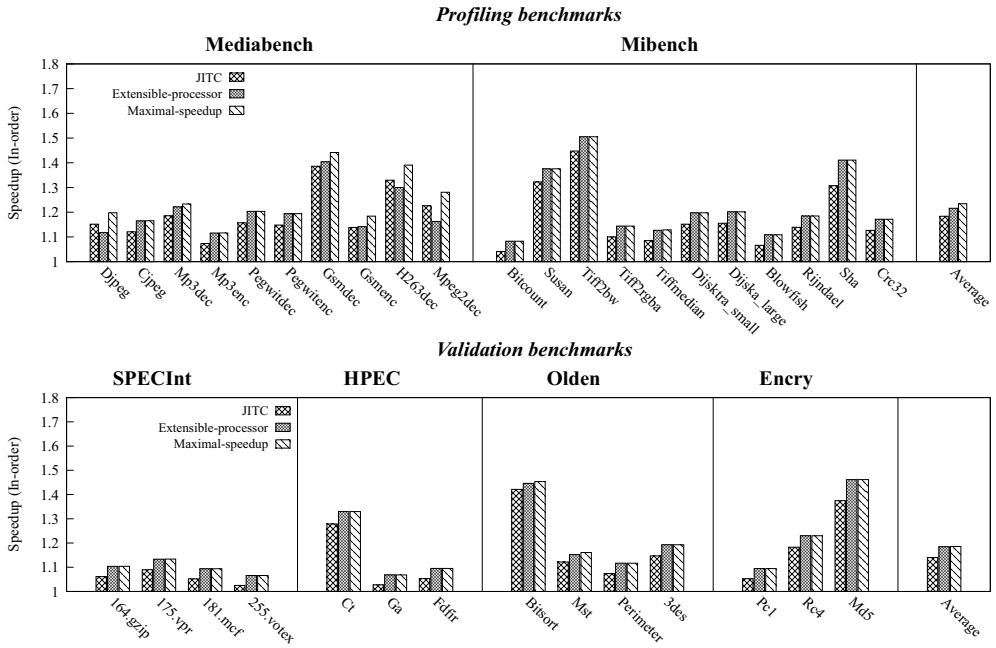
Table 4.2: Simulated processor configurations

for an ASIP without any area constraint. JITC architecture achieves an average speedup of 1.184X, which is 97.40% of the speedup achieved by ASIPs (1.216X) and 94.93% of the theoretical speedup (1.234X). The slight loss in performance of the JITC core comes from two sources: the reduced clock frequency and multi-cycle execution of 10% ISEs on the SFU. The remaining 90% ISEs can execute in single-cycle on the SFU. More importantly, *JITC has huge advantage in terms of flexibility: we need a different ASIP to accelerate each application, while a single JITC core can accelerate all the different applications with minimal performance loss.*

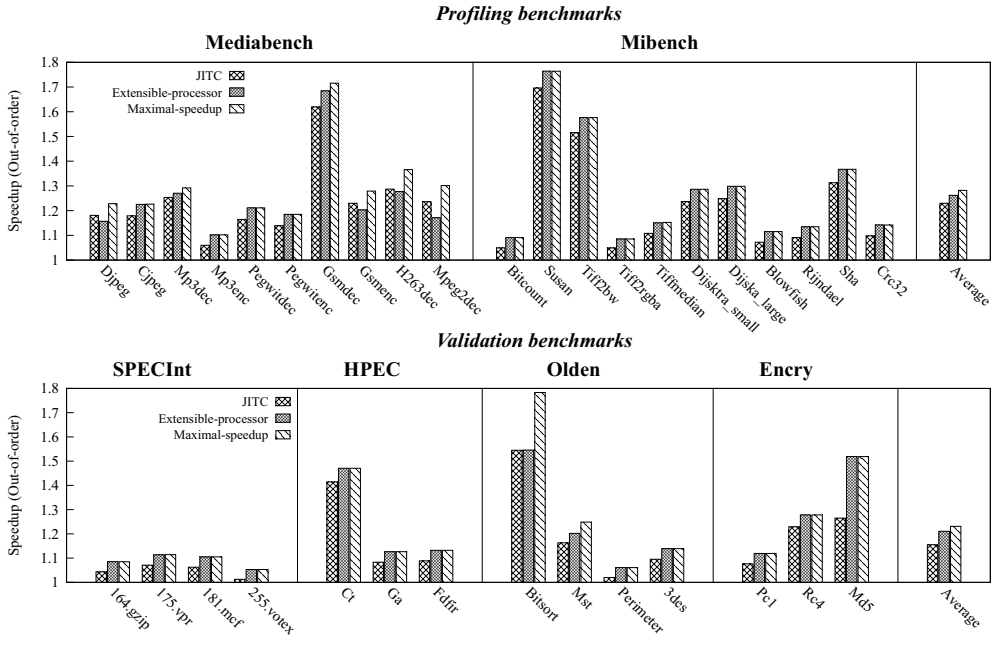
For out-of-order pipeline, we use 4-way decode, issue, execute, and commit. As expected, we need at most 4 SFUs in this case to achieve maximal speedup. For out-of-order pipeline, Figure 4.11(a) shows the performance of JITC core and ASIP compared to the baseline processor with no accelerator. Here JITC achieves an average speedup of approximately 1.230X across all benchmarks in Mediabench and Mibench, which is 97.54% of the speedup achieved by the ASIP (1.262X) and 95.98% of the theoretical speedup (1.282X).

Results for validation benchmarks The benchmarks from MiBench and MediaBench were used to derive the design of JITC core. Our objective, however, is to design a flexible architecture that can support any contemporary or emerging application domains. In order to stress test the design of our architecture, we attempt to accelerate applications from benchmark suites with completely different characteristics compared to the embedded space. We chose SPECInt [98], Encryption, Olden, and HPEC [110] benchmark suits for this evaluation. HPEC is derived from HPCC [88, 40] and PCA [81] both targeting general-purpose high-performance computing.

These validation results are shown in the right of Figure 4.11. JITC still achieves similar speedup to ASIPs, around 96% on an average for both in-order



(a) In-order architecture



(b) Out-of-order architecture

Figure 4.11: Speedup of JITC and ASIP over the baseline processor and the theoretical speedup for ASIP with unlimited area

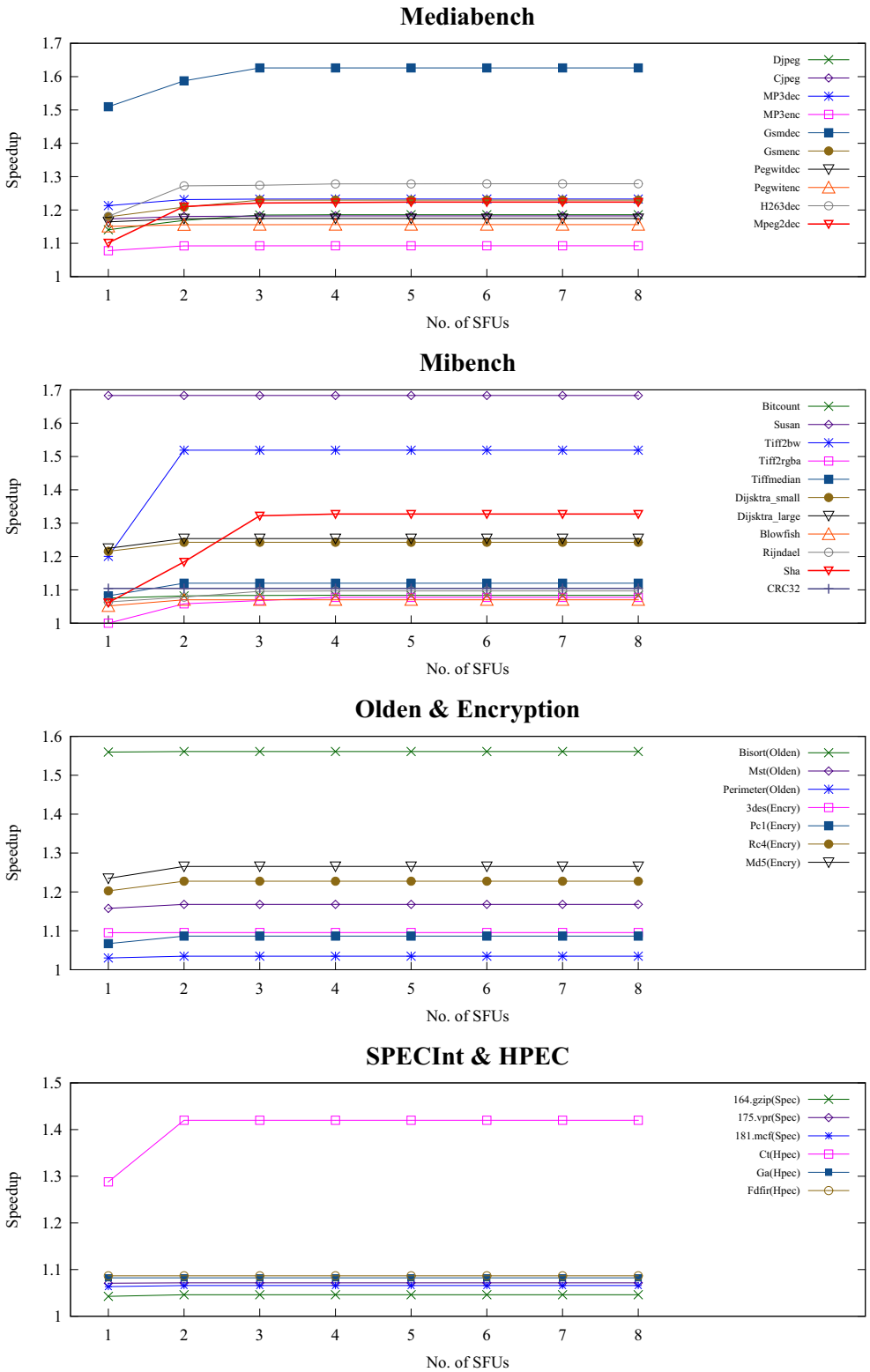


Figure 4.12: Experimental evaluation for the optimal number of SFUs in out-of-order execution

and out-of-order executions. This confirms that *even though the JITC core was designed to accelerate embedded applications, the design is flexible enough to support a completely different application domain*, e.g., SPEC and HPEC. However, for these benchmarks, the speedup achieved using customization (ASIP or JITC) is limited to around 1.10X. This is because these benchmarks have lower ratio of ALU operations and smaller basic blocks [17], characteristics that are not ideal for customization.

Optimal number of SFUs in out-of-order execution As mentioned, in our experimental evaluations, four SFUs are used in the out-of-order execution. We elaborate the setup by varying the number of SFUs to be integrated into the out-of-order processor pipeline. Figure 4.12 shows the experimental results for the performances when different number of SFUs are used. It is shown that performance will increase when number of SFUs is increased. Peak performance will be achieved when four SFUs are used. In fact, for most of the applications we evaluated, 2 to 3 SFUs are sufficient to achieve 99% of the maximal speedup. This is because there is only limited amount of parallelism to exploit across multiple ISEs.

4.3 S-CGRA Design using SFU

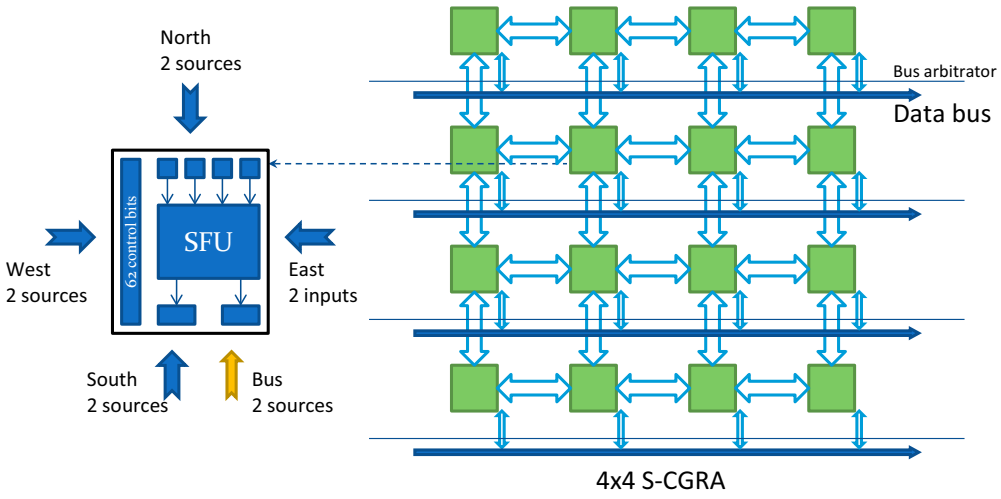


Figure 4.13: A 4×4 S-CGRA design

We have evaluated the efficiency of the SFU design and now we use the proposed design to build the reconfigurable coprocessor. We propose our novel specialized CGRA or S-CGRA by arranging a set of SFU in a two-dimensional

topology. A 4×4 S-CGRA is shown in Figure 4.13. As mentioned in Section 4.2.2, each SFU has four inputs and two outputs. The four inputs are selected from ten sources produced by its four neighbors and the data bus. The data bus in each row is shared among all the SFUs in that row through an arbitration policy. The arbitrator requires two bits for four SFUs in each row. The data bus is designed to be capable of carrying both the two outputs from a single SFU. Assuming each output is 32-bit, then the width of the data bus is 64-bit. To select ten sources as four inputs, a 16×4 multiplexer with 20 configuration bits is required. So, we have

$$\#Network\_conf\_bits = 20 * \#SFUs + 2 * \#Rows$$

where $\#Netowrk\_conf\_bits$ is the number of bits required for network configuration, $\#SFUs$ is the total number of SFUs, and $\#Rows$ is the total number of rows in the S-CGRA. For a 4×4 S-CGRA, the number of extra bits is 328 bits.

The total number of configuration bits would be the extra bits plus the number of bits required for functional configurations.

4.4 Customizable MPSoC Architecture with Shared S-CGRA

By sharing the proposed S-CGRA with multiple cores, we come up with our novel customizable MPSoC architecture.

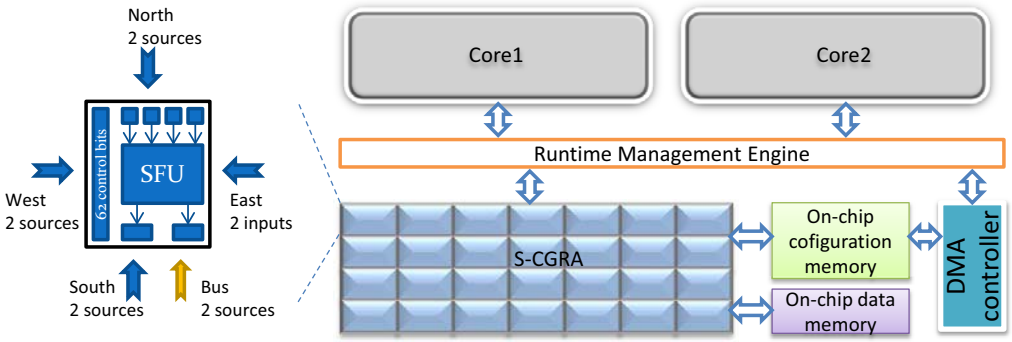


Figure 4.14: Proposed multi-core architecture with shared CGRA

The full system overview is shown in Figure 4.14. A runtime management engine is designed to synchronize the communications between the cores and the CGRA coprocessor. At runtime, different computationally intensive loop kernels could be executed by offloading to the CGRA. In sequential execution mode, the core will be suspended and listening on the acknowledgement sockets.

An on-chip configuration memory is used to store all the configurations required for the loop kernels. Any local data to be processed by the loop kernels would be stored in an on-chip data memory if necessary. Once a core offloads loop to CGRA by sending an execution request to the runtime management engine, the engine will trigger the DMA controller to load the corresponding configurations from the on-chip configuration memory and start the execution in CGRA. The configuration loading is done by writing each context register of the functional unit in the CGRA. When the execution in CGRA is finished, the management engine will acknowledge the completion to the requesting core, which will be resumed afterwards.

4.5 Chapter summary

In this chapter, we first propose a specialized functional unit design by revisiting the processor customization problem in the presence of reconfigurability. We conduct extensive experiments to investigate the intrinsic properties of the ISEs. It is revealed that the ISEs exploit limited parallelism, and there exist many common hot sequences of operations within custom instructions from multiple application domains. The specialized functional unit is then designed to fully explore the speedup benefits by supporting the required parallelism and the hot sequences. We confirm the efficiency of the proposed specialized functional unit by integrating it into the processor pipeline to create a just-in-time configurable processor. The instrumented processor is then proved to be able to provide ASIP-like performance. By using the proposed SFU as the primary processing elements, we create a specialized CGRA, namely S-CGRA, to support the efficient executions of the computational intensive loop kernels. Finally, the S-CGGA is further shared among multiple cores to create a customizable MPSoC system.

Chapter 5

Compilation of Computational Kernels on S-CGRA

In the previous chapter, we have designed a dynamic customizable MPSoC architecture called S-CGRA. In this chapter, we will detail the compilation supports in the MPSoC design automation tool. Thus, in this chapter, we will focus on the compilation technique for the S-CGRA. However, as S-CGRA is a specialized version of CGRA, we first revisit the application mapping problem on CGRAs. Then we consider the specialized functional units (SFUs) of S-CGRA by developing a pre-processing step, which helps to cluster operations that could be executed in the SFUs.

5.1 Overview

CGRAs are promising alternatives between ASICs and FPGAs. Traditionally in embedded systems, compute intensive kernels of an application are implemented as ASICs, which have high efficiency but limited flexibility. Current generation embedded systems demand flexibility to support diverse applications. FPGAs provide high flexibility, but may suffer from low efficiency [78]. To bridge this gap, CGRA architectures, such as CHESS [90], MorphoSys [115], ADRES [93], DRAA [84], FloRA [83] etc., have been proposed. Typically these architectures arrange coarse-grained functional units (FUs) in a mesh structure. The FUs can be reconfigured by writing to a control (context) register on per cycle basis. Figure 5.1 shows a 4×4 CGRA with FUs connected in a mesh topology; each FU has a local register file and a configuration cache.

The compute-intensive loop kernels are perfect candidates to be mapped to CGRAs containing multiple FUs targeting high instruction-level parallelism.

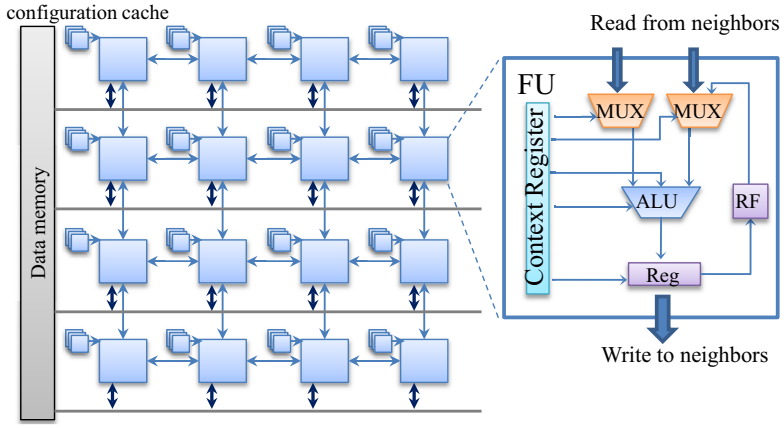


Figure 5.1: A 4×4 CGRA

Some CGRA mapping algorithms [92, 12, 60, 43, 35, 48] are inspired by compilation techniques for VLIW architectures as well as FPGA synthesis. For example, CGRA mapping algorithms adopt placement and routing techniques from FPGA synthesis domain and software pipelining based techniques such as modulo scheduling from VLIW compilation process. Note that the inherent structure of the CGRAs is very different from both FPGAs and VLIW architectures. More concretely, the connectivity among the functional units in CGRAs is usually fixed unlike FPGAs where the interconnections can be reconfigured. Thus, the mapping algorithms based on FPGA place and route techniques may find it challenging to identify feasible routing paths in fixed interconnect structure of CGRAs. Similarly, unlike VLIW architectures where all the FUs typically share a common register file, the FUs in most CGRAs have limited and explicit connections to the register files. Thus, it is not prudent to perform register allocation as a post-processing step as is commonly done in VLIW scheduling. Instead, register allocation should be integrated in the early stage with scheduling (place and route) to achieve quality mapping.

In this work, we focus on developing an efficient CGRA mapping algorithm that generates high quality solution with fast compilation time. To first formalize the CGRA mapping problem, we notice that many recent works [119, 5, 19, 48, 49] follow subgraph homeomorphism [42] formalization. The idea is to test if the data flow graph (DFG) representing the loop kernel is subgraph homeomorphic to the modulo routing resource graph (MRRG) representing the CGRA resources and their interconnects. Homeomorphism formulation allows subdivision of the DFG edges when being mapped onto the MRRG, i.e., a DFG edge can be mapped as a chain of edges (path) on the MRRG. Alternatively, additional vertices on a path consisting of a chain of edges on the MRRG can

be smoothed out to create a single DFG edge. The additional nodes by subdivisions model the routing of data from the source to the target FUs if they are not connected directly. However, subgraph homeomorphism requires the edge mappings to be node-disjoint (except at end points) or edge-disjoint [42]. While subgraph homeomorphism provides an elegant formulation of the CGRA mapping problem, it excludes the possibility of sharing the routing nodes among single source multiple target edges [100] (also called multi-net [35]) leading to possible wastage of precious routing resources.

Figure 5.2 illustrates the subgraph homeomorphism formulation. Figure 5.2(a) shows a simple DFG (for simplicity we have removed the loop back edge) being mapped onto a 2x2 homogeneous mesh CGRA shown in Figure 5.2(b). The DFG is homeomorphic to the subgraph of the MRRG shown in Figure 5.2(c) and thus the subgraph represents a valid mapping (again for simplicity we have removed additional nodes of the MRRG). In this homeomorphic mapping, edges (1,3) and (1,4) have been routed through three additional routing nodes marked by R. Notice that each routing node has degree 2 and has been added through edge subdivision (marked by dashed edges). Alternatively, the routing nodes in the MRRG subgraph can be smoothed out to obtain the original DFG. As mentioned earlier, by definition, edge subdivision cannot support route sharing.

In contrast, we model the CGRA mapping problem as graph minor containment problem, which can explicitly model route sharing. A graph H is a minor of graph G if H can be obtained from a subgraph of G by a (possibly empty) sequence of edge contractions [107]. In graph theory, an edge contraction removes an edge from a graph while simultaneously merging the two vertices it previously connected. In our context, we need to test if the DFG is a minor of the MRRG, where the edges to be contracted represent the routing paths in the MRRG. Unlike edge subdivision (or its reverse operation smoothing), edge contractions are not restricted to simple paths. Thus, graph minor formalism naturally allows for route sharing. Figure 5.2(d) shows a mapping under graph minor approach. It is a subgraph of the MRRG, from which the DFG can be derived through two edge contractions as shown in Figure 5.2(e)-(f). In this example, we reduce the number of routing nodes from 3 (in subgraph homeomorphism mapping) to 2 (in graph minor mapping). While it is possible to support route sharing in [100, 35], we provide a formalization of the CGRA mapping problem under route sharing. This formalization enables us to design a customized exact graph minor testing approach that fully exploits the structure of the DFG and the CGRA interconnects to effectively navigate and prune the mapping alternatives.

In parallel to our graph minor formalization [24] for CGRA mapping problem,

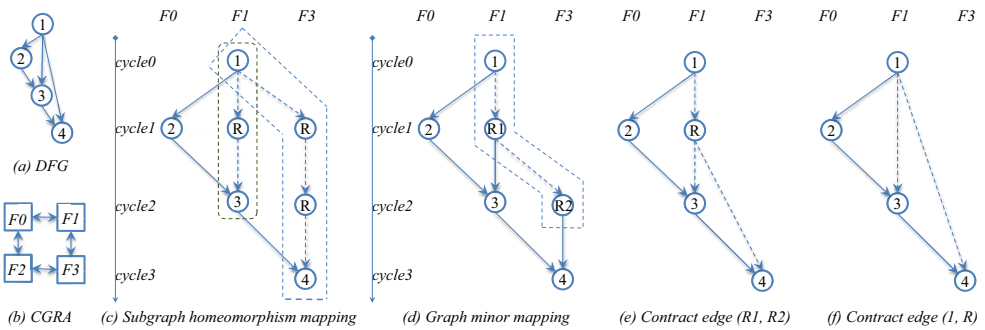


Figure 5.2: Subgraph Homeomorphism versus Graph Minor formulation of CGRA mapping problem

[56] proposed graph epimorphism formalization for the same problem. Their approach, called EPIMap, is quite elegant and models the novel concept of re-computation in addition to route sharing. Re-computation allows for the same operation to be performed on multiple FUs if it leads to better routing. In EPIMap approach, the DFG H is morphed into another graph H' (through introduction of routing/re-computation nodes and other transformations) such that there exists subgraph epimorphism from H' to H (many to one mapping of vertices from H' to H and adjacent vertices in H' map to adjacent vertices in H). Then, EPImap attempts to find the maximal common subgraph (MCS) between H' and the MRRG graph G using standard MCS identification procedure. If the resulting MCS is isomorphic to H', then a valid mapping has been obtained; otherwise H is morphed differently in the next iteration and the process repeats.

The key difference with our approach is that while we develop a customized graph minor testing procedure that exploits structural properties of our graphs, EPIMap relies on off-the-shelf MCS identification algorithm. This can potentially lead to faster compilation time for graph minor approach. Both approaches introduce heuristics to manage the computational complexity; the transformation of the DFG as well as MCS identification require heuristics in EPIMap, while graph minor approach restricts the subgraph mapping choices. Thus, the quality of the solutions in both approaches depend on the loop kernel and the underlying CGRA architecture. On the other hand, the re-computation concept in EPIMap enables additional scheduling and routing options that can potentially generate better quality solutions for certain kernels. Finally, graph epimorphism and graph minor are quite unrelated concepts even though a detailed discussion on this topic is out of scope here. Instead, we provide quantitative comparison of the two approaches in Section 5.6.

The concrete contributions of this work are as follows. We observe that the

CGRA mapping problem in the presence of route sharing can be formulated as a graph minor containment problem. This allows us to develop a systematic and customized mapping algorithm that directly works on the input DFG and MRRG to explore the inherent structural properties of the two graphs during the mapping process. Experimental results confirm that our graph minor approach can achieve high quality schedules with minimal compilation time.

In this chapter, we will first provide backgrounds on modulo scheduling in CGRA mapping problem in Section 5.2. We then formalize the CGRA mapping problem as a graph minor containment problem in Section 5.3. The proposed graph minor testing algorithm will be detailed in Section 5.4. Experimental evaluations comparing graph minor approach with different techniques are presented in Section 5.6.

5.2 Modulo Scheduling for CGRA

Given a loop from an application and a CGRA architecture, the goal of mapping is to generate a schedule such that the application throughput is maximized. The loop is represented as a data flow graph (DFG) where the nodes represent the operations and the edges represent the dependency among the operations. Figure 5.4(a) shows the DFG of a simple loop. Figure 5.4(b) shows a 2x2 CGRA consisting of four functional units (FUs) where the loop should be mapped to. The mapping problem consists of (a) scheduling the operations in space and time so as to satisfy the dependency constraints, and (b) explicit routing of the operands from the producers to the consumers.

5.2.1 CGRA Architecture

For simplicity of exposition, in the algorithm description we assume a homogeneous CGRA architecture with comprehensive FUs that can support all possible operations. However, our mapping approach can support diverse CGRA architectures through parameterizations. Our register file modeling approach can also support many different register file configurations such as NORF (architecture with no RF shown in Figure 5.3(a)), LRF (architecture with local shared RF shown in Figure 5.3(b)) and CRF (the architecture with central shared RF shown in Figure 5.3(c)). Heterogeneities for functional units are also supported in our framework. Experimental evaluations for different CGRA architectures will be presented in Section 5.6.

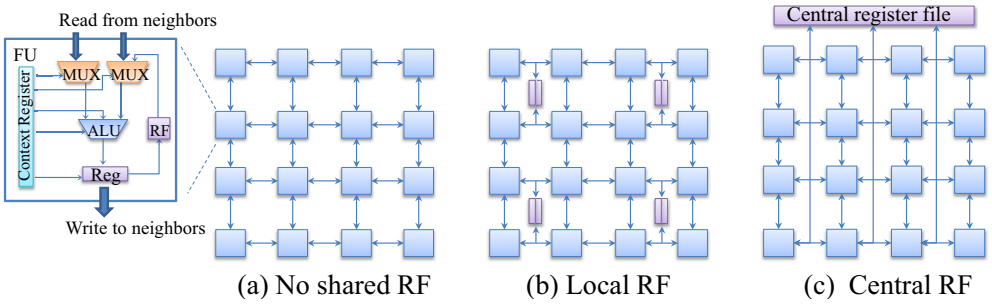


Figure 5.3: 4×4 CGRAs with different register file configurations

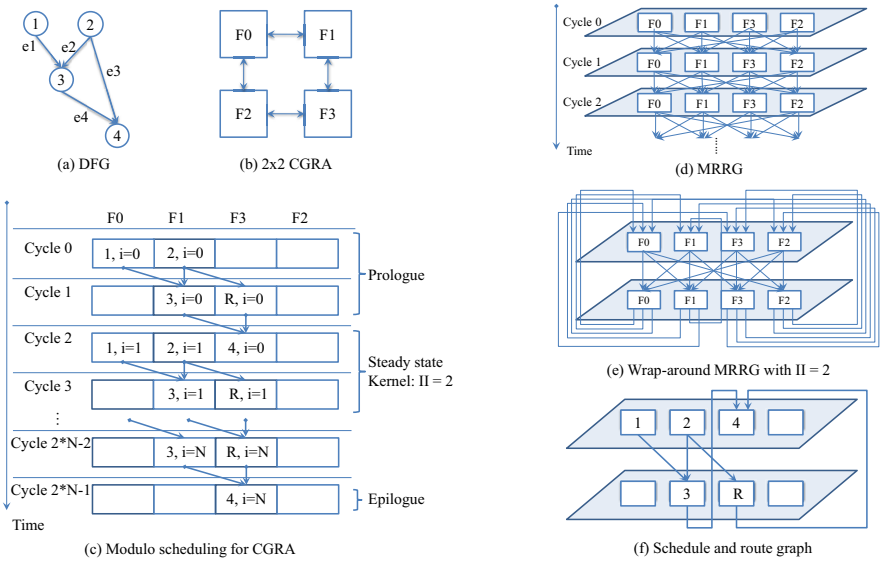


Figure 5.4: Modeling of loop kernel mapping on CGRAs: An illustrative example

5.2.2 Modulo Scheduling

Modulo scheduling is a software pipelining technique used to exploit instruction-level-parallelism in the loops by overlapping consecutive iterations [105]. The schedule produced includes three phases: the prologue, the kernel, and the epilogue. The kernel corresponds to the steady state execution of the loop and comprises of operations from consecutive iterations. The schedule length of the kernel, which is also the interval between successive iterations, is called the initiation interval (II). If the number of loop iterations is high, then the execution time in the kernel is dominant compared to the prologue and the epilogue. Thus, the goal for modulo scheduling is to minimize the II value. Initially, the scheduler selects the minimal II (MII) value between resource-minimal II and recurrence-minimal II, and attempts to find a feasible schedule with that II value. If the scheduling fails, then the process is repeated with an increased II value.

Figure 5.4(c) shows the modulo-scheduled version of the loop in Figure 5.4(a)

to the CGRA architecture in Figure 5.4(b) with prologue, kernel, and epilogue where II=2. Notice that *operation 4* from the i^{th} iteration is executing in the same cycle with *operation 1* and *operation 2* from the $(i+1)^{th}$ iteration in the steady state. Also, we need to hold the output of *operation 2* in a routing node (R) till it gets consumed by *operation 4*. This explicit routing between FUs is what sets apart modulo scheduling in CGRAs from conventional modulo scheduling, where FUs are fully connected through the central register file (RF) and routing is guaranteed. In CGRAs, the modulo scheduler has to be aware of the underlying interconnect among the FUs and the RFs to route data.

5.2.3 Modulo Routing Resource Graph (MRRG)

Mei et al. [92] defined a resource management graph for CGRA mapping, called Modulo Routing resource graph (MRRG), which has been used extensively in subsequent studies. In MRRG, the resources are presented in a time-space view. The nodes represent the ports of the FUs and the RFs, and the edges represent the connectivity among the ports. We adopt a simplified form of MRRG proposed in [99] where a node corresponds to FU or RF rather than the ports. Our mapping technique integrates register allocation with scheduling. We model each RF as one node per cycle in the MRRG. The individual registers within RF are treated as identical elements and represented by the capacity of the RF as in *compact register file model* [35]. The usage of registers is tracked and constrained during the mapping procedure. The number of read and write ports per RF is also included as a constraint.

The MRRG is a directed graph G_{II} where II corresponds to the initiation interval. Given a graph G, we denote the vertex set and the edge set of G by $V(G)$ and $E(G)$, respectively. Each node $v \in V(G_{II})$ is a tuple *(n, t)*, where n refers to the resource (FU or RF) and t is the cycle. Let $e = (u, v) \in E(G_{II})$ be an edge where u = *(m, t)* and v = *(n, t+1)*. Then the edge e represents a connection from resource m in cycle t to resource n in cycle *t+1*. Generally, if resource m is connected to resource n in the CGRA, then node u = *(m, t)* is connected to node v = *(n, t+1)*, $t \geq 0$.

For example, Figure 5.4(d) shows the MRRG corresponding to the CGRA shown in Figure 5.4(b). The resources of the CGRA are replicated every cycle along the time axis, and the edges point forward in time. During modulo scheduling, when a node *v=(n, t)* in the MRRG becomes occupied, then all the nodes *v'=(n, t+k×II)* (where $k > 0$) are also marked occupied. For example, in the modulo schedule with II=2 shown in Figure 5.4(c), as F1 is occupied by *operation 2* in cycle 0, it is also occupied by *operation 2* every $2 \times k$ cycle. In

most CGRA mapping techniques, this modulo reservation for occupied resources is done through a modulo reservation table [92].

5.2.4 MRRG with Wrap-Around Edges

The goal of CGRA modulo scheduler is to generate II different configurations for the CGRA where each configuration corresponds to a particular cycle in the kernel. These configurations are stored in the configuration caches and provide configuration contexts to FUs every cycle. As these configurations are repeated every II cycles, the output from the resources involved in the last configuration cycle are consumed by the resources involved in the first configuration cycle. Thus, instead of using MRRG where the time axis grows indefinitely till the steady state is achieved, we could restrict the time axis to the target II. We then need to add wrap around edges from the last cycle to the first cycle as shown in Figure 5.4(e) (similar graph is used in [43]). The modulo scheduled kernel in Figure 5.4(c) can now be simplified to the graph in Figure 5.4(f). We refer to this simplified graph as *schedule and route graph (SRG)*, which captures the scheduling plus routing information and is a subgraph of the MRRG. So instead of using a modulo reservation table, we can directly use MRRG with wrap around edges, which provides us an integrated view during mapping. *In the following sections, the term MRRG will be used to refer to MRRG with wrap around edges.*

5.3 CGRA Mapping Problem Formalization

We first present the formalization of the CGRA mapping problem in the form of subgraph isomorphism when no data routing is required and subgraph homeomorphism when routes are not shared. We then model the CGRA mapping as a graph minor problem [107] between the DFG and the MRRG in the presence of route sharing. Meanwhile, we point out the necessary restrictions imposed in the formalization. We also provide the NP-completeness proof for the CGRA mapping problem under our graph minor formalization.

5.3.1 Subgraph Isomorphism and Homeomorphism Mapping

Let H be a directed graph representing the DFG and G_{II} be a directed graph representing the MRRG with initiation interval II. We are looking for a mapping from the input graph H to the target graph G. In the ideal scenario of full connectivity among the FUs, all the data dependencies in the DFG can be mapped to direct edges in the MRRG. That is, for any edge $e = (u, v) \in E(H)$,

there is an edge $e = (f(u), f(v)) \in E(G)$ where f represents the vertex mapping function from the DFG to the MRRG. This matches the definition of subgraph isomorphism in graph theory. Thus, the CGRA application mapping problem can be solved using techniques for subgraph isomorphism from the graph theory domain [121, 32].

In reality, data may need to be routed through a series of nodes rather than direct links. For example, the edges $(1, 3)$ and $(1, 4)$ in Figure 5.2(a) are routed through additional nodes. If an edge $e = (u, v) \in E(H)$ in the DFG can be mapped to a path from *f(u)* to *f(v)* in the MRRG G, it matches the subgraph homeomorphism definition [42]. The subgraph homeomorphism techniques for CGRA mapping problem has been adopted in [119, 5, 19, 48, 49]. Subgraph homeomorphism, however, requires the edge mappings to be node-disjoint (or edge-disjoint), which means the nodes (or the edges) in the mapping paths for the edges carrying the same data cannot be shared.

5.3.2 Graph Minor

We now present graph minor [107] based formulation of the application mapping problem on CGRAs with route sharing. In graph theory, an undirected graph H is called a minor of the graph G if H is isomorphic to a graph that can be obtained by zero or more edge contractions on a subgraph of G. An edge contraction is an operation that removes an edge from a graph while simultaneously merging together the two vertices it used to connect. More formally, a graph H is a minor of another graph G if a graph isomorphic to H can be obtained from G by contracting some edges, deleting some edges, and deleting some isolated vertices. The order in which a sequence such operations are performed on G does not affect the resulting graph H.

A *model* of H in G is a mapping ϕ that assigns to every edge $e \in E(H)$ an edge $\phi(e) \in E(G)$, and to every vertex $v \in V(H)$ a non-empty connected tree subgraph $\phi(v) \subseteq G$ such that

1. the graphs $\{\phi(v)|v \in V(H)\}$ are mutually vertex-disjoint and the edges $\{\phi(e)|e \in E(H)\}$ are pairwise distinct; and

2. for $e = \{u, v\} \in E(H)$, the edge $\phi(e)$ connects subgraph $\phi(u)$ with subgraph $\phi(v)$.

H is isomorphic to a minor of G if and only if there exists a model of H in G [4].

5.3.3 Adaptation of Graph Minor for CGRA Mapping

We need to adapt and restrict the definition of graph minor. Graph minor is usually defined for undirected graphs. For directed graphs, the definition of edge contraction is similar to the undirected case [108]. Figure 5.2(e)-(f) show examples of directed edge contractions.

We call the subgraph $M \subseteq G$ defined by the union of $\{\phi(v)|v \in V(H)\}$ and $\{\phi(e)|e \in E(H)\}$ as the schedule and route graph (SRG) of H in G. The SRG M is essentially the model of H in G. The edge set of M is partitioned into the *contraction edges* (the edges in $\{\phi(v)|v \in V(H)\}$) and the *minor edges* (the edges in $\{\phi(e)|e \in E(H)\}$). The minor edges support the data dependencies in the dataflow graph, while the contraction edges represent data routing through additional nodes. For example, in Figure 5.2(d), $\phi(1)$ is the subgraph inside the dashed region rooted at node 1. The dashed edges are the contraction edges, while the solid edges are the minor edges.

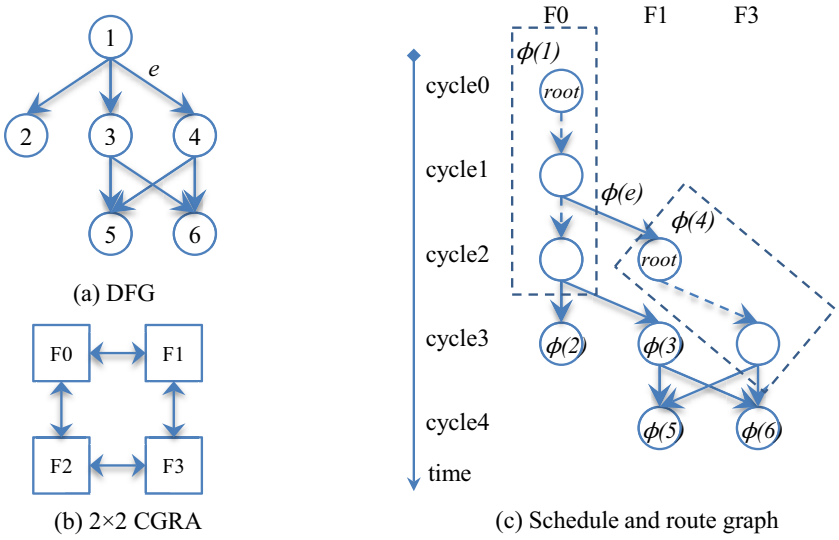


Figure 5.5: Minor relationship between DFG and MRRG

Minor edge constraint In graph minor definition, for $e = (u, v) \in E(H)$, the minor edge $\phi(e)$ connects $\phi(u)$ with $\phi(v)$. In other words, it is sufficient for $\phi(e)$ to connect any node in the subgraph $\phi(u)$ with any node in the subgraph $\phi(v)$. However, for our problem, we need to define one particular node in the subgraph $\phi(v)$ where the actual operation $\phi(v)$ takes place and it has to receive all the required inputs. The remaining nodes in $\phi(v)$ are used to route the result of the operation. More concretely, for our mapping, each subgraph $\phi(v) \subseteq G$ is a tree rooted at the node where the computation takes place. Let $root(\phi(v))$ be the root

of the tree $\phi(v)$. Then, we introduce the restriction that for $e = (u, v) \in E(H)$, the minor edge $\phi(e)$ connects $\phi(u)$ with $root(\phi(v))$. For example, the DFG in Figure 5.5(a) has an edge e that connects the DFG nodes 1 and 4, and it is mapped to a 2×2 CGRA shown in Figure 5.5(b). Then, in the SRG, $\phi(1)$ has to connect to the root of $\phi(4)$ through a direct link $\phi(e)$ as shown in Figure 5.5(c).

Timing constraint The wrap-around nature of the MRRG introduces another restriction. For SGR M to be a valid mapping, it has to satisfy the timing constraints as follows. For simplicity, let us first ignore the recurrence edges in the DFG. Then the DFG H is a directed acyclic graph. Let $u \in V(H)$ be a node in the DFG without any predecessor and $root(\phi(u)) = (m, t) \in M$ where $0 \leq t < II$ and M is the SRG, a subgraph of the MRRG. That is, u has been mapped to FU m in configuration t in the MRRG. We define the timestamp of u as $cycle(u) = t$, assuming u is executed in cycle t. Let $v \in V(H)$ be a DFG node with u as its predecessor node and $route(u, v)$ be the number of nodes (possibly zero) in the connecting path between $root(\phi(u))$ and $root(\phi(v))$ in the SRG M. For a mapping M to be valid, the following timing constraint, which ensures identical cycle along all input edges of v, must be satisfied for each internal DFG node v.

$$\forall u, u' \in pred(v): \quad cycle(u) + route(u, v) = cycle(u') + route(u', v)$$

We also define

$$\forall u \in pred(v): \quad cycle(v) = cycle(u) + route(u, v) + 1$$

where $pred(v)$ is the set of all predecessors of v in the DFG. Note that we are not doing modulo operation (w.r.t. II) while computing the cycle values. Figure 5.6 shows this timing computation. In the SRG, $root(\phi(2))$ is in cycle 0 and $root(\phi(3))$ is in cycle 1. However, $root(\phi(2))$ has to go through three routing nodes to reach $root(\phi(4))$; and $root(\phi(3))$ can directly pass the data to $root(\phi(4))$ in the next cycle. The timing constraint is then violated, leading to an invalid mapping.

For a recurrence edge $e = (u, v) \in V(H)$ in the DFG, we introduce additional timing constraint

$$route(u, v) = II \times d + cycle(v) - cycle(u) - 1$$

where d is the recurrence distance of e. Figure 5.7 shows how this timing constraint is used. Suppose $root(\phi(1))$ is executed in cycle 0; then it will receive the

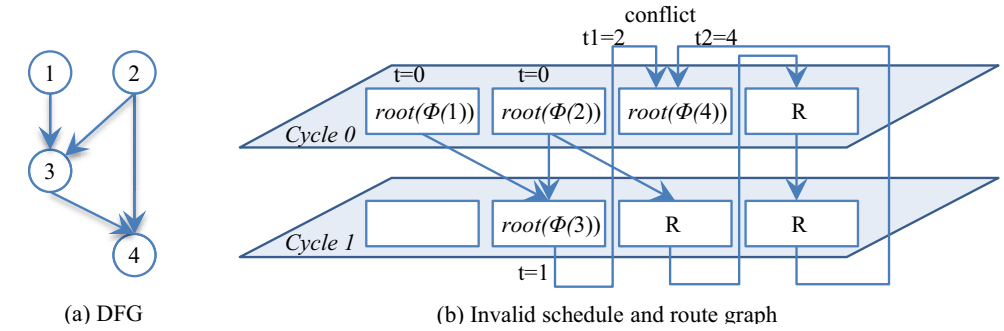


Figure 5.6: Invalid mapping under timing constraint

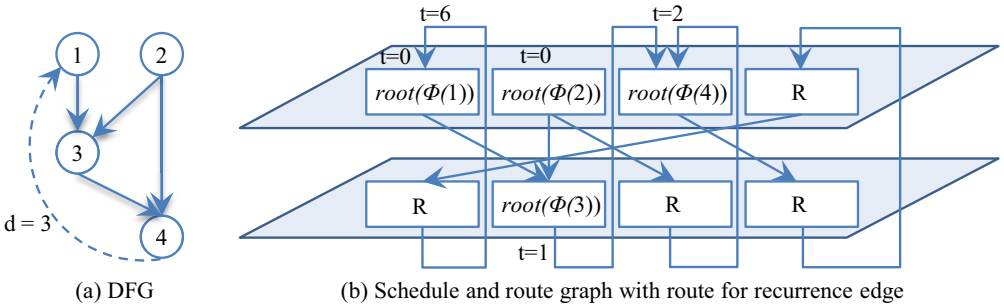


Figure 5.7: Mapping with recurrence edge under timing constraint

output of $root(\phi(4))$ 6 cycles (3 iterations) later. As $root(\phi(4))$ is executed in cycle 2 ($cycle(4) = 2$), the length of the route from $root(\phi(1))$ to $root(\phi(4))$ should be 2*3+0-2-1 = 3, which means the route contains three routing nodes. In fact, the timing constraint of normal edges are just special cases where distance d is equal to 0.

Attribute constraint Each node in the DFG and the MRRG has an attribute that specifies the functionality of the node. For example, a node in the DFG can have memory operation as its attribute, while a node in the MRRG can have an attribute that signifies that it can support memory operations. Attribute constraint ensures that a DFG node is mapped to an MRRG tree subgraph whose root has a matching attribute. For example, the root of the tree subgraph for mapping a memory operation can only be a functional unit supporting memory accesses. However, other nodes in the tree subgraph can be any type of functional unit or register file.

Register file constraint The mapping must ensure availability of register file read/write ports and capacity in the corresponding cycle if a link from/to the register file is used.

Restricted Graph Minor We can now define application mapping on CGRAs as finding a valid subgraph (schedule and route graph) M of the MRRG such that the DFG can be obtained through repeated edge contractions of M. We call the DFG a restricted minor of the MRRG and the subgraph M represents the mapping. Alternatively, the DFG H is a minor of G if and only if there exists a model of H, represented by the schedule and route graph M, in G.

lemma 1. *The restricted graph minor problem for directed graphs is NP-complete.*

Proof. We first show that the restricted graph minor problem for directed graphs is in the set of NP. Given a mapping in the form of SRG $M \subseteq G$, we can check in polynomial time (a) the graphs $\{\phi(v)|v \in V(H)\}$ are mutually vertex-disjoint and the edges $\{\phi(e)|e \in E(H)\}$ are pairwise distinct, (b) for $e = (u, v) \in E(H)$, the edge $\phi(e)$ connects subgraph $\phi(u)$ with $root(\phi(v))$, and (c) the timing constraints as defined earlier are satisfied. That is DFG H is a minor of the G.

We now show that for general directed graphs, the restricted graph minor problem can be reduced to the Hamiltonian cycle problem, which is an NP-complete problem. The Hamiltonian cycle problem is to find a cycle in a directed graph G visiting each node exactly once. We can construct a graph H, which is a directed cycle with $|V(G)|$ nodes. Finding the Hamiltonian cycle in G can now be reduced to finding a restricted graph minor between H and G. As $|V(G)| = |V(H)|$, each subgraph $\phi(v)$ can only consist of a single vertex and each edge mapping $\phi(e)$ where $e = (u, v) \in E(H)$ directly connects vertex $\phi(u)$ to vertex $\phi(v)$. This matches the exact definition of Hamiltonian cycle. Thus, the restricted graph minor problem for directed graphs is NP-complete. □

5.4 Graph Minor Mapping Algorithm

Our solution for restricted graph minor containment problem is inspired by the tree search method (also called state space search) widely used to solve a variety of graph matching problems [97]. The contribution of our solution is the introduction of customized and effective pruning constraints in the search method that exploit the inherent properties of the data flow graph and the CGRA architecture. We first present the exact restricted graph minor containment algorithm followed by description of additional strategies to accelerate the search process.

5.4.1 Algorithmic Framework

Our goal is to map a DFG H to the CGRA architecture. Similar to the traditional modulo scheduling, we start with the minimum possible II, which is the

maximum of the resource constrained II and the recurrence constrained II, that is, $II = \max(ResMII, recMII)$. Given this II value, we create the MRRG G_{II} corresponding to the CGRA architecture. If H is a minor of G_{II}, then the DFG can be mapped with initiation interval II. To check graph minor containment, we check if there exists a model or mapping of H in the form of a valid SRG $M \subseteq G_{II}$. If such SRG M does not exist, we increment the II value by one, create the MRRG corresponding to this new II value, and perform graph minor testing for this new MRRG. This process is repeated till we have generated an MRRG with sufficiently large value of II so that the DFG can satisfy the graph minor test. Algorithm 5 provides a high-level view of our mapping framework.

The core routine of the mapping algorithm *Minor()* performs graph minor testing. We consider all possible mapping between the DFG and the MRRG; thus our algorithm is guaranteed to generate a valid mapping if it exists. Clearly, the number of possible mappings between the DFG and the MRRG is exponential in the number of nodes of the DFG. That is, our search space is large. Our goal is to either (a) quickly identify a mapping such that the DFG passes the restricted minor test, or (b) establish that no such mapping exists. As mentioned earlier, we employ powerful pruning strategies to efficiently navigate this search space. We also carefully choose the order in which we attempt to map the nodes and the edges so as to achieve quick success in finding a valid mapping or substantial pruning that helps establish the absence of any valid mapping.

The procedure *Minor()* starts with an empty mapping. As mentioned earlier, restricted graph minor mapping for our problem requires mapping each vertex $v \in V(H)$ in the DFG to a tree $\phi(v) \subseteq G$ in the MRRG. Each edge $e = (u, v) \in E(H)$ is simply mapped to an edge $\phi(e) \in E(G)$ that connects some node in $\phi(u)$ to $root(\phi(v))$. Following this definition, we attempt to map the nodes one at a time in some pre-defined priority order, which will be detailed in Section 5.4.2.

There exist many possibilities to map a node $v \in H$ to a tree subgraph $\phi(v) \subseteq G$. However, the *min_map()* function in Algorithm 5 returns a set Γ of minimal valid mappings $\phi(v)$. Each minimal valid mapping contains minimal number of nodes and satisfies various constraints, including minor edge, timing, attribute and pruning constraints. The minor edge constraint ensures that all the edges connecting the mapped direct predecessors and successors of v can be mapped. More specifically, while mapping node v, we identify all its mapped direct predecessors P and successors S. We ensure that minor edge constraint can be satisfied between each node $p \in P$ and v as well as between v and each node $s \in S$. In other words, if node v has mapped direct successors, then

Algorithm 5: Graph Minor Mapping Algorithm

```
1  begin
3      order_list := DFG_node_ordering(H);
5      II := max(resMII, recMII);
7      while do
8          /*Create MRRG with II*/;
10         G_II := Create_MRRG(G, II); M := ⊥;
12         for all v ∈ V(H) and e ∈ E(H) do
14             φ(v) := ⊥;  φ(e) := ⊥;
15         end
17         add all φ(v), φ(e) to M;  /* empty mapping */
19         if Minor(H, G_II, M) then
21             return(M);
22         end
24         II++;
25     end
26 end
27 Function Minor(H,G,M)
1  begin
3      if no unmapped node in H then
5          return(success);
6      end
8      v := next unmapped node in H according to order_list;
10     P := {p |p ∈ pred(v) ∧ φ(p) ≠ ⊥};  /*mapped predecessors of v */
12     S := {s |s ∈ succ(v) ∧ φ(s) ≠ ⊥};  /*mapped successors of v */
13     /*All candidate mappings are generated satisfying minor edge, timing,
       attribute, pruning constraints */
15     Γ := min_map(v, P, S);
17     for each φ(v) ∈ Γ do
19         update M with φ(v);
21         if Minor(H, G, M) then
23             return(success);   /* mapping completed */
24         end
25     end
27     if Γ = ⊥ then
28         /* No feasible node mapping; expand predecessors */
30         for each possible expansion do
32             expand_map(v, P, M);
33             /* attempt mapping v again */
35             if Minor(H, G, M) then
37                 return(success);
38             end
39         end
40     end
41     /* No node mapping; backtrack to the predecessor */
42     return(failure);
43 end
```

we attempt to generate $\phi(v)$ containing additional routing nodes to ensure that $root(\phi(s))$, $s \in S$, can be reached from some node in $\phi(v)$. Meanwhile, $root(\phi(v))$ should be linked from every $\phi(p)$, $p \in P$. If node v does not have any mapped direct successor, $\phi(v)$ is generated containing a single node. Through *min_map()* function, edge mapping is automatically performed under minor edge constraint checking and we do not need to explicitly map the edges.

In addition, we check for timing constraint between v and its predecessors/successors to ensure that the data is routed correctly. Attribute compatibilities are checked between the DFG node v and the root of the candidate tree subgraph $root(\phi(v))$. If the target CGRA contains register files, the register constraint is used to check for available ports and capacity. Finally, we also apply aggressive pruning constraints to eliminate mappings that are guaranteed to fail in the future.

If we get non-empty Γ for each node v, then we will eventually obtain a complete feasible solution. However, Γ could be empty if there is no minimal valid mappings. In this case, we have to explore more elaborate tree subgraph mappings for the candidate node v. This is done through *expand_map()* function. In *expand_map()* function, we add one extra node in $\phi(p)$ for each $p \in P$, which helps to enhance the routing path from $\phi(p)$ to $\phi(v)$. If we cannot map v even after all the possible expansions, then we backtrack and attempt a different mapping.

The mapping process continues till we have either mapped all the DFG nodes (i.e., the DFG is a restricted minor of the MRRG) or we have discovered that no such mapping is possible (i.e., the DFG is not a restricted minor of the MRRG) and we have to increment the II value.

5.4.2 DFG Node Ordering

An appropriate ordering of the DFG nodes during mapping is crucial to quickly find a feasible solution. We impose the constraint that the nodes along the critical path have higher priority, i.e., they appear earlier. This is because if the critical path cannot be mapped with the current II value, then we can terminate the search process and move on to the next II value.

In addition, we employ an ordering that helps us validate the timing constraints as discussed in Section 5.3.3. A node v is mapped only when at least one of its direct predecessor or successor has been mapped. That is v should appear in the ordering after at least one of its direct predecessor or successor nodes. The only exception is the first node in the ordering. The advantage of this ordering is that the timestamps $cycle(v)$ are generated appropriately for the

nodes so that timing conflicts can be avoided early. When the DFG contains disjoint parts, a new timestamp is regenerated and propagated for every disjoint component during the mapping process.

Figure 5.8(b) shows a DFG and the ordering of the nodes through the arrow signs. We start with the input node 1 on the critical path. We proceed along the critical path to node 3 and node 4. Notice that we could not include node 2 after node 1 because none of its direct predecessors or successors would have appeared in the ordering by then. After node 4, we include node 2 in the ordering.

5.4.3 Mapping Example

Suppose we have a DFG as shown in Figure 5.8(b) and we are attempting to map it to a 2×2 CGRA array. Let us assume that we are currently considering II=2. For simplicity of exposition, we only draw the occupied edges in the MRRG. The entire mapping process is illustrated in Figures 5.8(c-g).

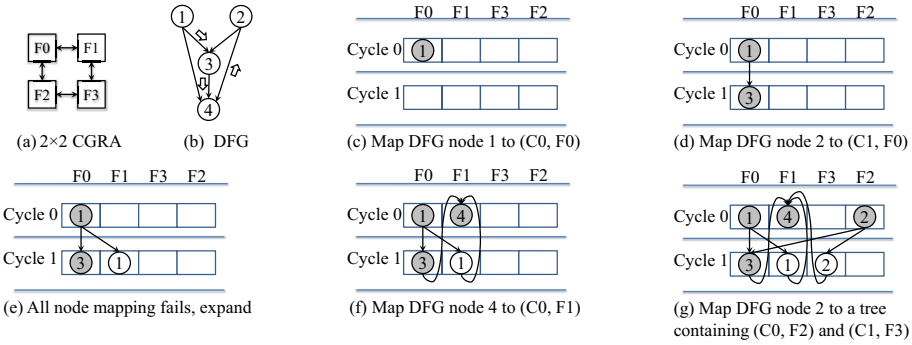


Figure 5.8: An example of mapping process during the restricted graph minor test

The process starts with mapping node 1. Node 1 is the initial node and it has no mapped direct successor. So the first tree subgraph generated by *min_map()* function contains only one node as shown in Fig 5.8(c): F1 in cycle 0 denoted as (C0, F0). Then, we pick the next node in the priority list, which is node 3. Again, this node has no mapped direct successors; so its tree mapping also contains only one node. However, we need to make sure that $\phi(1)$ is directly connected with $root(\phi(3))$ according to the edge constraint imposed by the edge e = (1, 3) in DFG. Mapping node 3 to (C1, F0), as shown in Figure 5.8(d), can satisfy the constraint.

The next node in the priority list to be mapped is node 4. However, this time we fail to find any feasible node directly connected to the mapped direct predecessors $\phi(1)$ and $\phi(3)$. As mapping for node 4 fails, we expand its predecessor's mapping. An extra node (C1, F1) is added to $\phi(1)$ in Figure 5.8(e). Notice that

to distinguish between root nodes and other nodes, the root nodes have been shadowed. Now node 4 can be mapped to (C0, F1) in Figure 5.8(f).

The final node in the list is node 2. This time, node 2 has two mapped successors, node 3 and node 4. Thus, we find a tree subgraph $\phi(2)$ containing (C0, F2) and (C1, F3) (see Figure 5.8(g)) that satisfy both the minor edge constraint (direct links to root nodes of $\phi(3)$ and $\phi(4)$) and the timing constraints at node 3 and 4. As all the nodes and the minor edges have been mapped successfully, DFG is a minor of $MRRG$ with $II = 2$.

5.4.4 Pruning Constraints

Pruning constraints are important to reduce the compilation time. Pruning constraints look ahead and quickly identify if the current mapping can be extended to a successful final mapping. This lookahead helps to eliminate mappings that are guaranteed to fail in the future. Note that the pruning constraints do not affect the optimality of the solution.

Available resource constraint This constraint simply checks that the number of available FUs of each type in the MRRG is larger than or equal to the number of unmapped DFG nodes of the same type. For example, the number of remaining available memory FUs must be larger or at least equal to the number of unmapped memory operations in the DFG. Global variables are used to record information about the available FUs and the unmapped DFG nodes and are updated every time the partial mapping changes. Thus, both time and space complexity of this constraint are *O(1)*.

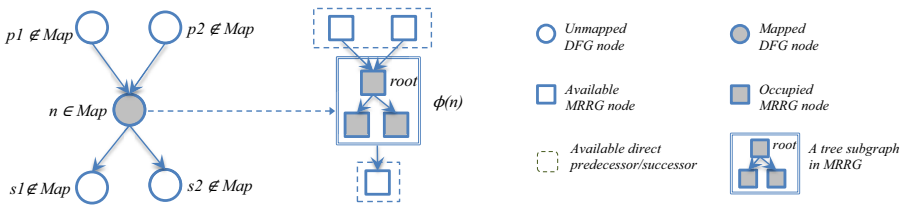


Figure 5.9: Illustrations of degree pruning constraint

Degree constraint This constraint considers the local structures between the DFG H and the MRRG G. Let $\phi(n) \subseteq G$ be the tree subgraph representing the mapping of node $n \in V(H)$. The number of unmapped direct predecessors of n in the DFG must be smaller than or equal to the number of available direct predecessors of $root(\phi(n))$ in the MRRG.

On the other hand, if n has any unmapped direct successors, then the number of available direct successors of $\phi(n)$ must be at least one. This is because the data from $\phi(n)$ can be routed through any available outgoing node. For example, in Figure 5.9, DFG node n is mapped to $\phi(n)$ in the MRRG. It has two unmapped direct predecessors and two unmapped direct successors. So $root(\phi(n))$ must have at least two available direct predecessors and there must be at least one available direct successor of $\phi(n)$ in the MRRG. Notice that the available direct successors of $\phi(n)$ are those available MRRG nodes directly connected from any node in $\phi(n)$.

The degree pruning constraint checks for all the DFG nodes in the current mapping. The time complexity for this pruning constraint is $O(cN)$, where N is the number of DFG nodes and c is the average number of producer nodes in $\phi(n)$ across all mapped DFG nodes n.

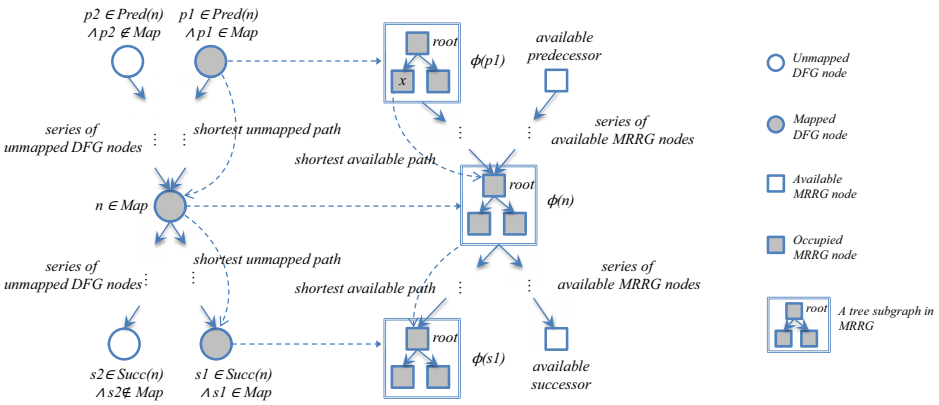


Figure 5.10: Illustration of predecessor and successor constraints

Predecessor and successor constraint We further exploit structural patterns formed by each mapped DFG node n and its predecessors/successors as shown in Figure 5.10. We check the timing constraint inherently imposed by these patterns. We first calculate the shortest path lengths in both DFG and MRRG. The shortest paths defined here only consists of unmapped DFG nodes or available MRRG nodes except the two end nodes. For any mapped predecessor p of n, if p and n are connected through the shortest unmapped path $r = (p \rightsquigarrow n)$, then $\phi(p)$ and $\phi(n)$ should also be connected by a shortest available path $R = (x \rightsquigarrow root(\phi(n)))$, $x \in \phi(p)$, in MRRG. Thus, we have $cycle(root(\phi(n)) - cycle(x) \geq max(length(R), length(r))$, which uses the fact that the timestamp differences must be at least equal to the length of the shortest path connecting the corresponding nodes either in the MRRG or in the DFG. Similar constraints are applied to the patterns formed by n and its successors.

We also consider the relationships between a mapped DFG node n and its unmapped predecessors/successors. However, as these predecessors/successors have not been mapped yet, there is no explicit structural information to be used for pruning purpose. Instead, we calculate the number of available MRRG nodes those could be connected to $root(\phi(n))$ (or reached from $\phi(n)$) through available MRRG paths. The number must be at least equal to the number of unmapped predecessors (or successors) of n, which can be connected to (or from) n through unmapped DFG paths.

To obtain the reachability information in both the DFG and the MRRG during the mapping, two reachability matrices are built using an efficient algorithm by Italiano et al. [66]. The algorithm has a time complexity $O(K)$ with $O(K^2)$ space overhead, where K is the number of nodes in the input graph. Each element (u, v) in the matrix represents the shortest path length between the node u and node v. To build the reachability matrix for M MRRG nodes, the time complexity is $O(M^2)$. As the computation for reachability matrices is the most time consuming step, the overall time complexity for the pruning constraint is $O(M^2)$.

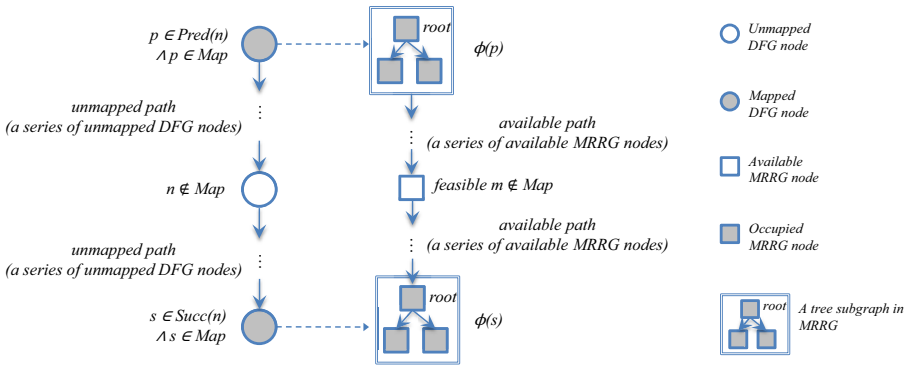


Figure 5.11: Illustration of feasibility constraint

Feasibility constraint In the final pruning constraint, we exploit the structural patterns of the unmapped DFG nodes. As shown in Figure 5.11, for each unmapped DFG node, we find all its mapped predecessors and successors reachable through unmapped paths. There must be at least one MRRG node that has the same connectivity to all the subgraphs the corresponding predecessors and successors have been mapped to. More specifically, let n is such an unmapped DFG node, p is a mapped predecessor of n and p is connected to n through an unmapped path. Then, in the MRRG, there must be at least one available node m such that m could be connected from $\phi(p)$ through an available path. As this

pruning constraint also depends on the reachability matrices, the complexity is $O(M^2)$.

5.4.5 Acceleration Strategies

We now introduce additional strategies to further accelerate the compilation time. These strategies are integrated in the preprocessing step and the constraints in the algorithm infrastructure. All the strategies are designed in such a way that they do not impact the optimality of the mapping.

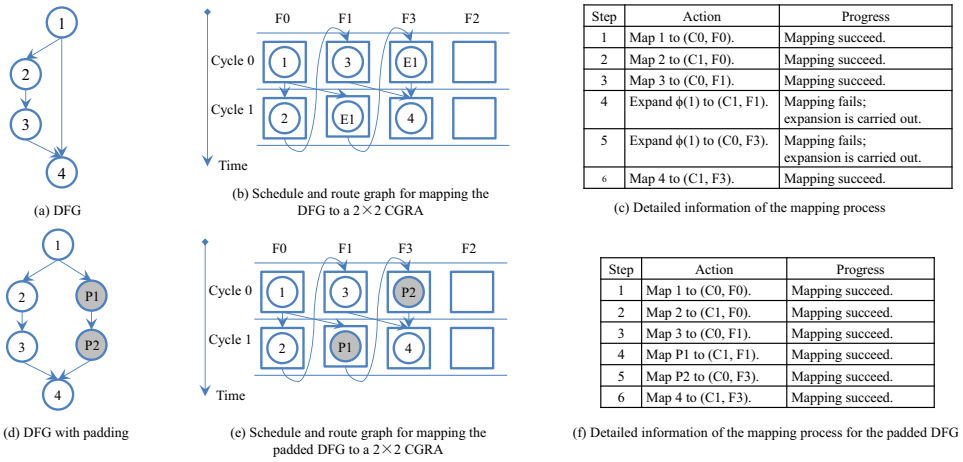


Figure 5.12: A motivating example for dummy node insertion

Dummy nodes in the DFG

We introduce dummy nodes in the DFG during the preprocessing step. These dummy nodes are *only* used for routing, which means they can be mapped to non-computation nodes in the MRRG, e.g., register file nodes. Basically, the idea is based on the observation that expanding the tree mapping $\phi(v)$ for any node v is quite expensive. This is because $\phi(v)$ is expanded only after all attempts to map subsequent nodes have failed. Also the expansion is carried out incrementally, i.e., $\phi(v)$ is expanded one node at a time. The goal of introducing dummy nodes is to avoid the expansions as much as possible without affecting the quality of the solution.

Figure 5.12 shows an example of how dummy nodes can avoid expansion of node mapping. We want to map the DFG in Figure 5.12(a) to 2×2 CGRA. The mapping order is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. The first three nodes 1, 2, and 3 can be mapped successfully. However, when we try to map node 4, the mapping attempt fails (Γ is empty) and we have to expand $\phi(1)$ twice in order to find the

final feasible mapping for node 4. The final schedule and route graph is shown in Figure 5.12(b) with the expansion nodes for $\phi(1)$ denoted as $E1$. The detailed search process is also listed in Figure 5.12(c).

To avoid the mapping failures and expansions, we can add two dummy nodes $P1$ and $P2$, as shown in Figure 5.12(d). Suppose the mapping order for the new DFG is $1 \rightarrow 2 \rightarrow 3 \rightarrow P1 \rightarrow P2 \rightarrow 4$. After mapping the three nodes 1, 2 and 3, we will continue to map $P1$ and $P2$ without any failure. Finally, node 4 will be mapped successfully at the first attempt. The final schedule and route graph is shown in Figure 5.12(e) and the detailed mapping process is listed in Figure 5.12(f).

Clearly, dummy node insertion is useful in guiding the mapping process. So we add dummy nodes as part of DFG pre-processing step. We first assign scheduling levels to each DFG node using as soon as possible ($ASAP$) scheduling policy and as late as possible ($ALAP$) scheduling policy. The number of dummy nodes inserted to a DFG edge $e = (u, v) \in E(H)$ is equal to the difference between the $ASAP$ level of v and the $ALAP$ level of u. This is somewhat similar in concept to node balancing in [56]. However, the difference is that we insert dummy nodes to accelerate the search process to obtain a feasible schedule. In the previous approach [56], adding more balancing nodes is a requirement to obtain a valid schedule.

Fast implementation of pruning constraints

For large DFGs, the pruning constraints can increase the compilation time. The most expensive part is the reachability matrices computation. To reduce this overhead, we bypass updating the reachability matrix of the MRRG at each step. We do, however, generate the reachability information for the DFG statically in the beginning and for the MRRG at its generation step for each II value. We believe that the two static matrices provide limited but enough information for the pruning purposes. The static reachability matrices now record the reachability information between any two arbitrary nodes in the absence of any mapping, e.g., the element (x, y) in the MRRG matrix records the static shortest path length between nodes x and y. With only static reachability matrix, the pruning constraints have to be redesigned as follows.

Fast implementation of predecessor and successor constraints Unlike the original constraint, the fast implementation only focuses on the structural patterns related to current mapping. Suppose the candidate DFG node n is mapped to $\phi(n)$ in the MRRG. For every mapped predecessor p of n, we can

have the length value for the static shortest path $r_s = (p \rightsquigarrow n)$, from static DFG matrix. Let $R_S = (x \rightsquigarrow \phi(n))$, $x \in \phi(p)$, be the static shortest path between $\phi(p)$ and $\phi(n)$ in MRRG. x can be identified by checking the static MRRG matrix for all the nodes in $\phi(p)$. Utilizing the same fact used in the original constraint, we have $cycle(root(\phi(n))) - cycle(x) \geq max(length(R_S), length(r_s))$

Similarly, constraints are also imposed for the structural patterns formed by the candidate node and its mapped successors. The fast implementation reduces the runtime complexity from $O(M^2)$ to $O(cN)$ where c is the average number of nodes in $\phi(n)$ for each DFG node n.

Fast implementation of feasibility constraint The basic idea for designing fast implementation of feasibility constraint is to consider the local effects of consuming one MRRG node for the remaining unmapped DFG nodes. Suppose the candidate MRRG node to be used for mapping is m, then the consumption will affect the potential mappings of those who also require m. If m is directly linked from any node in $\phi(p)$, p is a mapped DFG node, then the consumption of m can affect the mapping for the unmapped child $child\_p$ of p. In other words, we need to ensure that apart from m, there is another available MRRG node m' that can be used to map $child\_p$ satisfying certain timing constraints. For every mapped successor s of $child\_p$, we can have the static shortest path $r_s = (child\_p \rightsquigarrow s)$. Let R_S be the static shortest path connecting m' and $root(\phi(s))$, $R_S = (m' \rightsquigarrow root(\phi(s)))$. Following the same reasoning used before, we have

$$cycle(root(\phi(s))) - cycle(m') \geq max(length(R_S), length(r_s))$$

If m is a direct predecessor of the root node of $\phi(s')$, where s' is a mapped DFG node, similar constraints are used for the unmapped parent node of s'. The time complexity is also $O(cN)$.

5.4.6 Integration of Heuristics

Our modulo scheduling algorithm (Algorithm 5) can achieve the optimal II by definition. This is because it checks if the DFG is a minor of the MRRG for each value of II, starting with the minimum possible value. However, even with the pruning and acceleration strategies, the runtime of the optimal algorithm can be prohibitive when both the number of DFG nodes and the number of CGRA functional units are quite large. Therefore, we integrate some heuristics in the algorithm to speed up the search process. This may introduce sub-optimality,

i.e., the search process may miss a valid mapping at lower II value even though it exists. But the compilation time improves significantly.

The first heuristic avoids backtracking between two unrelated nodes. In the optimal search process, if a node m cannot be mapped, then we backtrack to the node n, which appears just before m in the DFG node ordering. However, node n may not be a predecessor or successor of node m in the DFG and hence may not be able to steer the search towards a successful mapping to m. Instead, we directly backtrack to the last predecessor or successor of node m in the ordering.

The second heuristic is motivated by the edge-centric mapping [100]. In graph minor testing, instead of enumerating all possible tree subgraphs for node n, the procedure aims to find limited number of feasible subgraphs. The feasible subgraphs are chosen to be those with minimal number of nodes. After all the specified subgraphs have been explored, the node mapping fails.

The final heuristic makes it possible to escape from extensive subgraph expansions. We put a counter for each node mapping. The counter is increased every time an expansion is carried out. Once the counter reaches a pre-defined threshold value, we eliminate current mapping and backtrack to previous mappings. Our experimental evaluation reveals that this is the only heuristic that sometimes prevent us from reaching a feasible solution even if one exists.

5.5 Clustering preprocessing for S-CGRA

Till here, we have fully presented our G-Minor CGRA mapping algorithm. We now consider the compilation support for the S-CGRA architecture presented in Chapter 4, which contains SFU as its fundamental element.

5.5.1 Hierarchical scheduling technique

The compilation technique for CGRA mapping proposed so far requires to create a one-to-one mapping from the nodes in the DFG to the resource nodes in the MRRG. For S-CGRA, however, each SFU in it could perform multiple operations each cycle as depicted in Chapter 4. To fully exploit the capability of multi-operation execution, we will present a clustering algorithm, which is to be adopted as a pre-processing step in our G-Minor framework.

In literature, a clustering preprocessing step followed by task scheduling is called a hierarchical scheduling technique [71], which is well accepted in MPSoC mapping and scheduling context. Our problem presented here, while sharing many similarities of MPSoC mapping and scheduling problems, does have its own unique properties. Although it is not entirely fit for our context, we

do acknowledge that the widely adopted GA clustering algorithm for solving the MPSoC mapping and scheduling problem could be adapted due to its general applicability. In the following subsections, we will present the adapted GA heuristic and then propose our greedy heuristic. A simplistic greedy heuristic has been presented in Chapter 4 for mapping custom instruction to one SFU. Here, the greedy heuristic will be further extended for the clustering purpose.

5.5.2 Genetic Algorithm for Clustering

Genetic algorithm (GA) [50], well known for its robustness, is a technique that starts from an initial population of randomly generated potential solutions to a problem, and gradually evolves towards better solutions through a repetitive application of genetic operations such as selection, crossover and mutation. The evolution process proceeds through generations. Each next generation is created by producing offsprings from the current population through a crossover operator. Evolution is ensured by selecting appropriate offsprings according to preset fitness functions. The evolution process is repeated until certain criteria are met. GA has been successfully deployed for task matching/scheduling problem in multiprocessor computing environments [38, 125, 79, 63].

Algorithm 6: Genetic_algorithm

```
1  Begin
2      population = Initialize_population(N);
3      Evaluation(population);
4      while Stop criteria not met do
5          new_population = empty;
6          /*Generate new generation.*/
7          For i = 1 to N do
8              parents = Selection(population);
9              crossover_offsprings = Crossover(parents, cross_rate);
10             mutation_offsrpings = Mutation(crossover_offsprings);
11             Evaluation(mutation_offsrpings);
12             Insert(new_population, mutation_offsprings);
13         Endfor
14     end
15     Return Best_solution_found;
16 End
```

To adapt the genetic algorithm in our S-CGRA context, the very first step is to define a proper chromosomal representation. Two functions, crossover and mutation, have to be modeled to generate the new population. Moreover, the fitness criteria has to be formulated for the selection in each evolution step.

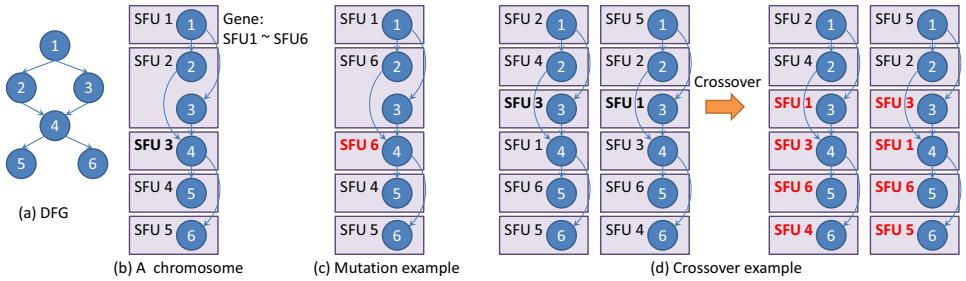


Figure 5.13: Examples for chromosomal representation, mutation and crossover

Chromosomal representation For the chromosomal representation , we specify the gene pool as the available SFUs. The DFG is linearized to constitute the main structure of the chromosome, and in each of the position, one gene is combined to each DFG node. The combination in each position stands for the mapping relationship from the DFG node to the SFU. For example, for the DFG shown in Figure 5.13(a), one of its chromosome is shown in Figure 5.13(b) with each gene representing one particular SFU. One DFG node could be mapped to one SFU, e.g. DFG node 1 is mapped to SFU1; more than one DFG nodes could be mapped to the same SFU, e.g. DFG node 2 and 3 are mapped to SFU2; and there could be SFU with no nodes mapping to it, e.g. SFU6 is not mapped by any DFG nodes.

Initial population The initial population is generated randomly. The population generator assigns a random SFU to each DFG node. The number of the population N is set as 1000 in our experimental evaluation.

Selection The selection is to fetch two parent chromosomes from the population according to their fitness. The fitness metric is defined as the following.

$$fitness = \sum_{g}^{M} (Convex_g * Non\text{-}loop_g * Feasible_g) \tag{5.1}$$

$$Convex_g = \left\{ \begin{array}{l} 1, \ \ if\ mapping\ of\ gene\ g\ satisfies\ DFG\ convexity\ constraint \\ 0, \ \ else \end{array} \right\} \tag{5.2}$$

$$Non\text{-}loop_g = \left\{ \begin{array}{l} 1, \ \ if\ mapping\ of\ gene\ g\ satisfies\ DFG\ non\text{-}loop\ constraint \\ 0, \ \ else \end{array} \right\} \tag{5.3}$$

$$Feasible_g = \left\{ \begin{array}{l} 1, \ \ if\ mapping\ of\ gene\ g\ satisfies\ SFU\ architectural\ constraint \\ 0, \ \ else \end{array} \right\} \tag{5.4}$$

So the fitness metric is defined according to the convexity, non-loop and feasibility constraints for each of the M SFUs, M is the total number of SFUs.

The convexity and non-loop constraints are imposed by the structure of the DFG graph, while the feasibility constraint is imposed by the architecture of the SFU. Notice that the each gene g stands for one SFU. The set of DFG nodes mapped to an SFU g satisfies **convexity constraint** if the immediate nodes along all of the paths between any two nodes in the set are also contained inside the set. For the **feasibility constraint**, if the set of DFG nodes is mapped to an SFU g in GA, then there should be a feasible mapping from this set of DFG nodes to the underlying architecture of SFU g. The **non-loop constraint** is imposed for the DFG nodes mapped to two genes. For any two paths p connecting the node u to the node v in the DFG, $p = <u \rightsquigarrow v>$, and $p' = <u' \rightsquigarrow v'>$ connecting u' and v', if u and v' is mapped to gene $g1$, then u' and v should not mapped to the same gene $g2$ other than $g1$. The illustrated example is given in Figure 5.14. Assuming that u' is mapped to gene $g2$, we can see that mapping u and v to the same gene $g1$ will generate a data dependency from gene $g2$ to gene $g1$ shown in Figure 5.14(a). If we continue to map v' to gene g2, which u' has been mapped to, another data dependency from gene $g1$ to gene $g2$ is created. This will generate a data dependency loop leading to a dead lock as shown in Figure 5.14(b). Notice that convexity constraint is to avoid dead lock within one gene, while non-loop constraint is to avoid dead lock across genes.

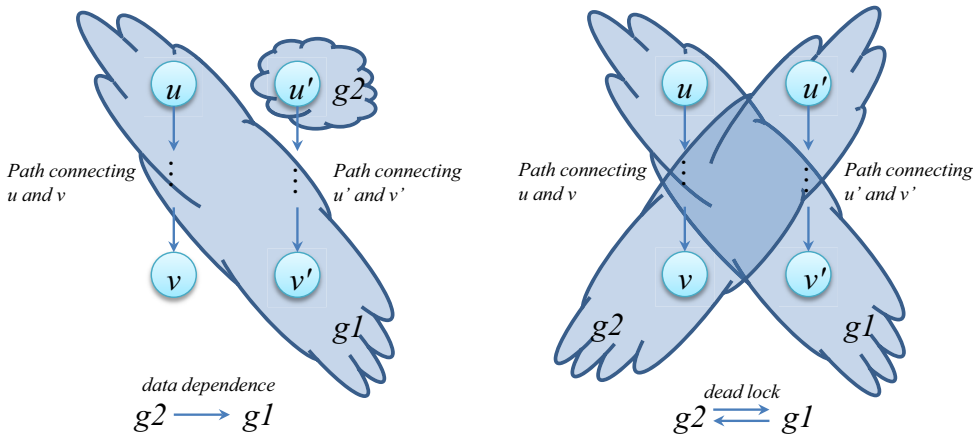


Figure 5.14: An illustrative example for non-loop constraint

Each chromosome is associated with a fitness value after it is generated. In the selection phase, the higher the fitness score of one chromosome is, the higher chance the chromosome will be selected.

Crossover In crossover, two chromosomes exchange their genes from a certain position. For the mapping, this stands for exchanging the mapping information from that position. For example, in Figure 5.13(d), the two chromosomes in the left exchange genes from the third position. Consequently, all the mapping information from DFG node 3 are exchanged between the two chromosomes. For crossover rate, we practically choose this value as 0.7.

Mutation In mutation, one gene in a certain position in the chromosome could be changed to another gene. In the mapping context, this means that the DFG node in that position, originally mapped to a particular SFU, now is mapped to a different SFU. Figure 5.13(c) shows an example where the fourth position of the chromosome in Figure 5.13(b) is mutated from SFU3 to SFU4. The mutation rate is chosen as 0.01 in our experimental evaluation.

Termination criteria When GA finds a feasible mapping, it will terminate the evolving procedure. The feasible mapping is defined as satisfying both convexity and feasibility constraints. So every time after one new population is generated, the chromosome with the highest fitness value is checked. If the value is equal to the total number of SFUs, then GA will report it as the final feasible solution. On the other hand, if GA could not find a feasible solution when it comes to a sufficient large number of generations, it will also terminate the procedure. The stopping bound is set as 1000 generations.

Optimal number of SFUs To find the optimal number of SFUs, the generic algorithm could be iterated n times by setting the number of SFUs from 1 to n, where n is the total number of DFG nodes serving as an upper bound. To restrict the number of iterations, the lower bound could be identified by examine the underlying architecture of SFU. As there are only 6 components inside one SFU, so the lower bound could be set as $n/6$. This could be further refined specifically according to the numbers of each type of operations and components. The searching for optimal number of SFUs start from the lower bound of available SFUs and increase the value in each step until it finds the GA returns a feasible chromosome.

5.5.3 A Derived Greedy Heuristic

Our greedy heuristic is derived from the mapping heuristic proposed in Chapter 4. In Chapter 4, the heuristic is designed to synthesize the ISEs onto one SFU in multiple cycles. Here, however, the algorithm is designed to cluster the

Algorithm 7: Clustering_heuristic

Input: The data flow graph (DFG) of the nested loop and the resource routing graph (RRG) of the SFU.
Output: The generated configuration if mapping is successful.

```
1  Begin
2    max_level = Assign_level_ALAP(DFG);
3    For i ← 1 to max_level do
4      For All each operator u in DFG do
5        If u→level == i then
6          successful = 0;
7          For Each of u's predecessor v do
8            If u has only one immediate predecessor v and u is v's
             only immediate successor then
9              If SFU(v)→FU(v)→component(Res(u)) ==
               Available and SFU(v)→FU(v)→Res(v) is connected
               to SFU(v)→FU(v)→Res(u) then
10               SFU(v)→FU(v)→component(Res(v)) =
                 Occupied;
11               successful = 1;
12             Endif
13           Endif
14           Else
15             If ∃ FU n ∈ SFU(v), n→status = Free and
               n→component(Res(u)) == Available then
16               n→status = Mapped;
17               n→component(Res(u)) = Occupied;
18               successful = 1;
19             Endif
20           Endif
21         Endfor
22         If successful == 0 then
23           Assign a new SFU s;
24           Get the first available FU n in SFU s;
25           n→status = Mapped;
26           n→component(Res(u)) = Occupied;
27         Endif
28       Endif
29     Endfor
30   Endfor
31   Return Build_cluster_graph();
32 End
```

DFG nodes under the architectural constraints. The architectural constraints is imposed by using the resource routing graph (RRG) of the SFU. The basic idea

of the greedy heuristic is to place each operation within the same cluster that one of its previous predecessors has been mapped to. Intuitively, this would lead to less number of clusters or SFUs as the mapping become more compact. The detailed algorithm is shown in Algorithm 7. We assign operations according to their as late as possible (ALAP) order. For each operation u, we find the cluster that one of its predecessors v has been mapped to. If v is *u's* only immediate predecessor and v has u as its only immediate successor, then we map u to the available component *component(Res(u))* corresponding to the required resource *Res(u)* in the SFU *SFU(v)*. Otherwise, we map u to one free functional unit inside *SFU(v)*, where v is one of its predecessor. If we cannot cluster u with any of its predecessors, we allocate a new cluster/SFU and map it there. The output of the algorithm will give the clustering graph with data dependencies among the clusters generated according to the data dependencies in the input DFG. This is done by calling the function *Build_cluster_graph()*.

The algorithm has a linear running time complexity *O(N)*, N is the total number of nodes in the input DFG. The deadlock presented in the non-loop constraint will not occur as the greedy heuristic simply assigns a new cluster/SFU to the unsuccessful clustering with candidate operation's predecessors. The final mapping solution will always be feasible as the architectural constraints are considered during the clustering process.

5.6 Experimental Evaluation for Mapping on CGRA

We now proceed to evaluate the quality and the efficiency of our mapping algorithm. We initially target a 4×4 CGRA with 2D mesh network architecture and no shared or central register file. The 4×4 array is the basic structure in many CGRA architectures and has been widely used to evaluate various mapping algorithms [99, 100, 80, 60, 72, 12]. For our initial experiments that compare against previous approaches, we assume each functional unit is comprehensive and is capable of handling any operation including memory operations. Later, we evaluate the versatility of graph minor mapping approach in supporting diverse CGRA architectures, such as heterogeneous functional units and various register file configurations. We also evaluate the scalability issue by mapping to 4×8, 8×8, 8×16 and 16×16 CGRAs.

We select loop kernels from MiBench benchmark suite [55], SPEC2006 benchmark suite, and the benchmarks used in the EPIMap approach [56]. Most of the benchmarks have an easily identifiable compute-intensive loop that performs the main functionality of the application and we select that loop for our experiments.

Benchmark	*#ops*	*#MEM ops*	*#edges*	*Benchmark*	*#ops*	*#MEM ops*	*#edges*
SOR	17	6	11	***osmesa***	16	9	17
swim_cal1	59	23	39	***texture***	29	7	31
swim_cal2	62	26	44	***quantize***	21	8	24
sobel	27	7	34	***rgb2ycc***	41	15	44
lowpass	23	9	19	***rijndael***	32	13	35
laplace	20	8	16	***fft***	40	20	42
wavelet	12	4	6	***tiff2bw***	42	20	50
sjeng	36	13	21	***fdctfst***	59	16	80
scissor	12	4	13	***idctflt***	87	25	114

Table 5.1: Benchmark characterisitics

For the few benchmarks with multiple loop kernels, we choose the representative one of them. *Rijndael* implements the AES standard where we choose the nested loop in its encryption subroutine. *Tiff2bw* converts a color TIFF image to greyscale image where we choose the nested loop in the first step that converts 16-bit color map to 8-bit. The benchmarks *Wavelet, Fdctfst, Idctfst* have multiple identical or similar loops and we choose one of them.

The DFGs for the loop kernels are generated from Trimaran [1] back-end using Elcor intermediate representation [3]. Benchmark characteristics are listed in Table 5.1 including the number of operations and the number of load/store operations. We assume that the memory operation includes both the address generation and the actual load/store operation.

Comparison with different techniques There exist a number of approaches to CGRA mapping in the literature. We compare our graph minor approach (abbreviated as G-Minor here) with two previous techniques: simulated annealing based approaches and EPIMap [56]. Simulated annealing (SA) based approaches [92] are widely considered to provide high-quality mapping solutions with (possibly) longer compilation time. EMS, the edge-centric mapping approach [100], provides significantly reduced compilation time with some degradation in the quality of the schedule compared to SA. As mentioned in Section 5.1, in parallel to G-Minor approach, [56] have proposed graph epimorphism based mapping approach EPIMap that produces better quality solutions than EMS with similar compilation time. We compare G-Minor with EPIMap as it represents state-of-the-art CGRA mapping approach. For the comparison, we have re-implemented the EPIMap approach [56] and the simulated annealing (SA) algorithm [92] for 4×4 mesh CGRA with comprehensive functional units and no shared/central register file similar to the setup in [56]. Our implementations of these two approaches allow route sharing. To demonstrate the benefits gained from using route sharing, we also create a subgraph homeomorphism mapping kernel. Moreover, we also integrate re-computation methodology introduced in EPIMap

as a DFG pre-processing step in our G-Minor framework.

Figure 5.15 compares the scheduling quality for 18 benchmarks. The Y-axis represents the achieved II value. The first bar represents the minimal II value achievable considering only recurrence minimal and resource minimal II for each kernel. The remaining bars from left to right represent the II achieved for G-Minor, EPIMap, simulated annealing (SA), subgraph homeomorphism, and G-Minor with re-computation pre-processing (Rec-G-Minor), respectively.

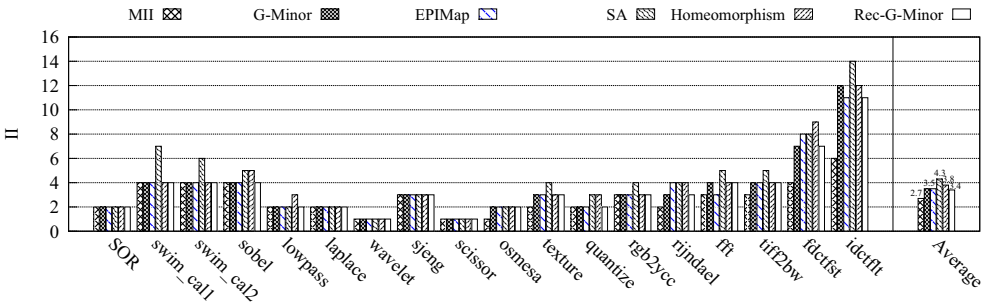


Figure 5.15: Scheduling quality for G-Minor, EPIMap, SA, subgraph homeomorphism and G-Minor with re-computation

We first observe that the scheduling quality generated by EPIMap and G-Minor are quite similar. The achieved II value is different between the two for only 4 out of 18 benchmarks. For example, G-Minor produces better scheduling results for *rijndael* and *fdctfst*, while EPIMap performs better for *fft* and *idctflt*. Even for these benchmarks, the difference is only one cycle. The two reasons for the competitive results between G-Minor and EPIMap are the following. G-Minor exhaustively searches for minor with all routing possibilities, while EPIMap restricts the number of routing nodes. On the other hand, EPIMap provides extra choices for mapping the DFGs such as replication (or re-computation) for high fan-out nodes. An interesting possible future research direction would be to combine the relative strengths of G-Minor and EPIMap. We conduct preliminary evaluation by integrating re-computation with our G-Minor framework. It is shown in Figure 5.15 that in most cases, Rec-G-Minor can generate better scheduling results than G-Minor and EPIMap.

We observe that for a large subset of benchmarks (11 out of 18), both G-Minor and EPIMap achieve Minimal II (MII). SA, on the other hand, achieves minimal II value for 6 benchmarks. In general, G-Minor and EPIMap provide better schedules compared to SA. A possible reason is that in SA, a random operation is picked, replaced and routed in each step. It is inefficient in considering the placement and routing impacts among operations. This inefficiency gets

worse when the routing resources are limited such as in a 4×4 mesh CGRA. G-Minor and EPIMap, on the other hand, directly explore the structural properties of the graphs and hence the relationships among operations.

We carry out additional experiments to demonstrate the benefits of route sharing. We disable route sharing in our G-Minor algorithm to create a subgraph homeomorphism kernel. As shown in Figure 5.15, subgraph homeomorphism generates far worse schedules compared to G-Minor.

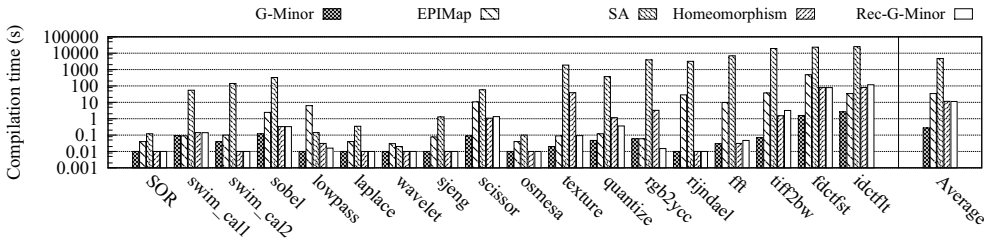
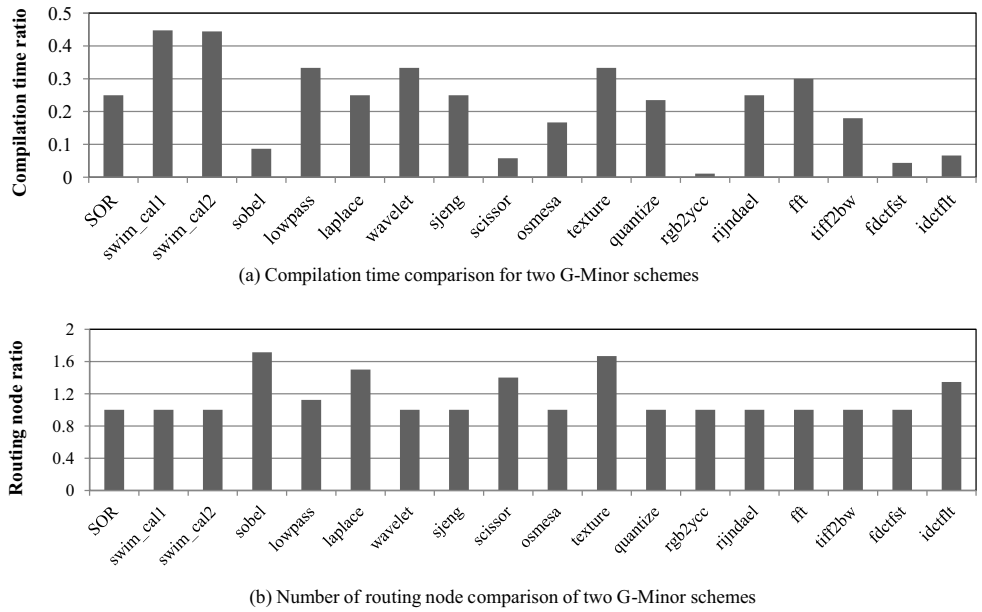


Figure 5.16: Compilation time for G-Minor, EPIMap, SA, subgraph homeomorphism and G-Minor with re-computation

The runtime of the different approaches for all the benchmarks are shown in Figure 5.16, which is reported based on an Intel Quad-Core running at 2.83GHz with 3GB memory. It is well known that SA approaches require longer compilation time [100] specially for large kernels. Similar compilation time has been reported in [57]. G-Minor and EPIMap reduce compilation time significantly using more guided approach to mapping. The average compilation time for EPIMap is 34.26 sec, which is consistent with the timing reported in [56]. G-Minor provides extremely fast compilation time of only 0.27 sec on an average. This is because the graph minor testing algorithm in G-Minor has been highly optimized using various pruning constraints and different acceleration strategies. EPIMap transforms the DFG and uses it as an input to an off-the-shelf maximal common subgraph (MCS) kernel [86]. Thus, the compilation time for EPIMap depends on the efficiency of the chosen MCS kernel. Besides, EPIMap might need to transform the DFG and repeat the MCS kernel computation multiple times when the mapping fails. This potentially leads to longer compilation time.

Impact of acceleration strategies and heuristics We evaluate reduction in compilation time using the acceleration strategies presented in Section 5.4.5. We compare compilation time for two different versions of G-Minor: the slow mode and the fast mode in Figure 5.17. The fast mode uses the acceleration strategies. Both modes achieve identical II for all the benchmarks because the acceleration strategies are designed such that they do not impact the quality



(a) Compilation time comparison for two G-Minor schemes

(b) Number of routing node comparison of two G-Minor schemes

Figure 5.17: Experimental results for fast G-Minor scheme (with acceleration strategies) compared to slow G-Minor scheme

of the solutions, but provide better guidance for the search process. In Figure 5.17(a), the compilation time of the fast mode is normalized w.r.t. the slow mode. The fast mode can effectively reduce the compilation time by more than 50%. The penalty for the fast mode is in the form of using more routing nodes. Figure 5.17(b) compares the number of routing nodes for the two schemes. The average ratio is around 1.15, which means there are 15% extra routing nodes used in fast mode because the fast pruning constraints using static shortest path connectivity information can lead to more node expansions. The heuristics play crucial roles in achieving reasonable compilation time. In our experiments, 9 out of the 17 benchmarks will fail to return a feasible solution within 10 hours without the heuristics. Meanwhile, the II values of the remaining benchmarks match the results generated with heuristics.

Different CGRA configurations As mentioned in Section 5.2.1, our approach can support different CGRA configurations. The experiment results for 4×4 CGRAs with different number of memory units and different register file configurations are shown in Figure 5.18. MxC denotes the availability of x columns of memory FUs in the array; and y is the number of registers in a register file. So an architectural configuration MxC-LRF-yR corresponds to an array with x columns of memory units and locally shared register files, each of

which contains y registers. Each register file is associated with two read ports and one write port. The results indicate that memory units are the most critical resources. Adding more memory units brings substantial benefit by reducing the achieved II. However, adding more registers may not necessarily improve II. This is because the intelligent exploration of the search space can find mappings within limited routing resources. Adding more routing resources such as increasing the size of local/global register files can reduce the mapping efforts but could also end up with resource wastage. We notice that starting from M2C-LRF-1R configuration, increasing the number of registers and providing more connectivity through registers for routing do not reduce the value of the achieved II.

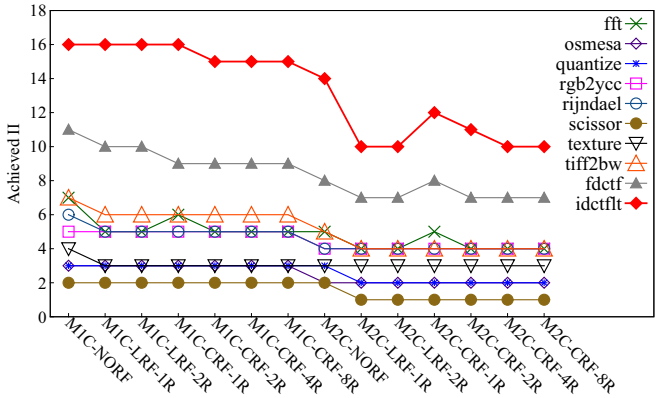


Figure 5.18: Achieved II for different CGRA configurations

Scalability Our G-Minor fast mode can dramatically accelerate the compilation time. We test the scalability by configuring the size of NORF CGRA to 4×8, 8×8, 8×16 and 16×16 2D-mesh. To further stress the scalability, we generate 100 random DFGs where number of nodes is uniformly distributed in the range (0, 100]. We present the average compilation time for G-Minor and EPIMap with different CGRA sizes in Table 5.2. The results confirm that G-Minor provides better scalability to map kernels on large CGRAs. We do not report compilation time for SA approaches as it takes too long to generate solutions for large CGRAs.

	4×4 CGRA	4×8 CGRA	8×8 CGRA	8×16 CGRA	16×16 CGRA
Avg. compilation time (s) of G-Minor	0.23	0.61	1.51	3.12	7.08
Avg. compilation time (s) of EPIMap	54.78	570.72	837.92	1235.18	1385.27

Table 5.2: Compilation time for CGRAs with different sizes

5.7 Experimental Evaluation for Mapping on S-CGRA

We now conduct experimental evaluation for the mapping on S-CGRA. We demonstrate the efficiency of our proposed heuristic by comparing to the adapted GA algorithm. Besides, to show the efficiency of the S-CGRA architecture, we also compare the schedules generated by the S-CGRA and the normal CGRA.

We choose a set of loop kernels from DSP applications, Mediabench [82] and Mibench [55]. The DFGs of the loop kernels are generated at the back-end of the gcc cross compiler for simplescalar [11]. These loop kernels are then mapped to both S-CGRA and the normal CGRA through our graph minor framework with/without clustering pre-processing step.

As mentioned, the adapted genetic algorithm and the proposed heuristic both aim at optimizing the computation resource usages by minimizing the number of clusters. Interestingly, although genetic algorithm is well-known to converge to near-optimal solutions, when adapting to our clustering context with the considerations of underlying architectural constraints and DFG constraints, the genetic algorithm in fact performs worse than our greedy heuristic. From the experimental results shown in Figure 5.19(a), the greedy heuristic consistently outperforms genetic algorithm (GA) in terms of number of clustering nodes generated across all the benchmarks. The smaller the number is, the better clustering effect would be. This number is reported as the node ratio to the number of nodes in the original DFG in Figure 5.19(a). In average, by clustering, the greedy heuristic can reduce the DFG size to 65.7% of its original, while GA can only achieve reduction to 76%. A possible reason for why our heuristic outperforms GA is that the evolution process in GA can randomly map unrelated DFG nodes to one gene, which still satisfy all the constraints used to calculate the fitness value. However, grouping unrelated DFG nodes together would lead to inefficient resource usage as the resources in the SFU are occupied and could not be used for mapping the direct successors. Our heuristic exactly solves this problem and its efficiency is confirmed.

Then we compare the the schedules generated by using the S-CGRA and the normal CGRA. The schedules for the S-CGRA are generated by passing the clustered DFGs to our G-Minor mapper proposed in Chapter 5. And the schedules for the normal CGRA are generated by mapping the original DFGs to the normal CGRA using the G-Minor mapper. The schedules generated for the normal CGRA also represent the schedules generated for the S-CGRA without a pre-processing step. Thus, the comparison results also demonstrate the importance of using a pre-processing for the S-CGRA. With the clustering pre-processing step, the size of one DFG is reduced through clustering. The number

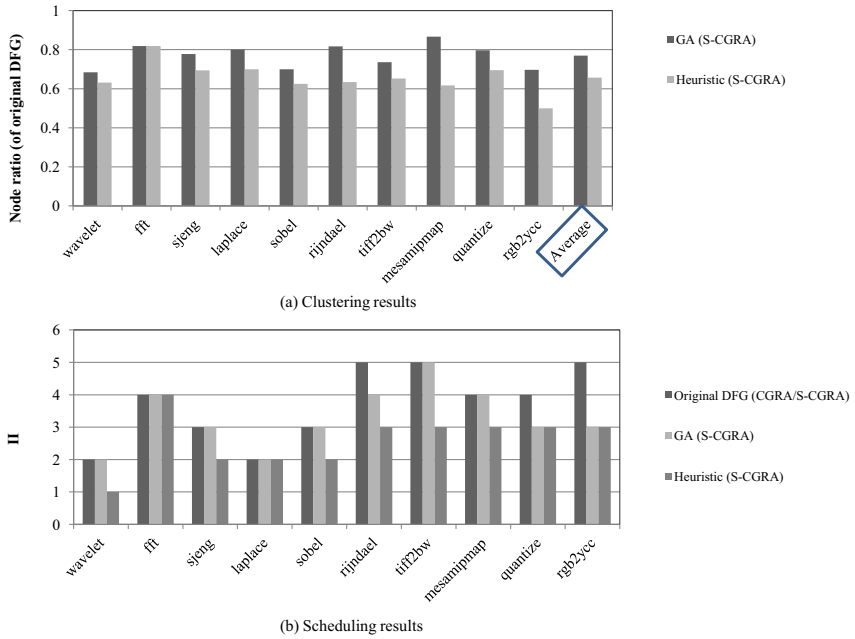


Figure 5.19: Experimental results for genetic algorithm and proposed heuristic

of nodes in the new DFG is smaller than the original DFG. So it is expected that the schedules could be much more effective for mapping DFGs into the S-CGRA comparing to the schedules generated for the normal CGRA. The experimental results for the comparisons are shown in Figure 5.19(b). As expected, the schedules generated for the S-CGRA with a clustering pre-processing step using either the GA or the heuristic are better than the schedules generated for the normal CGRA or the S-CGRA without a pre-processing step. Also notice that, less computational nodes does not mean an absolute reduction in II. It is possible that even though a clustering pre-processing step using GA or greedy heuristic reduces the number of required computational nodes, the II values would still not improved, e.g. FFT benchmark shows three identical IIs. The quality of the schedules are highly dependent on how many available resources could be utilized and the efficiency of the mapper.

5.8 Chapter Summary

In this chapter, we establish the compilation techniques to be used in the MPSoC customizations for architectures with shared CGRA and S-CGRA. We formalize the CGRA mapping problem as a restricted graph minor containment with the data flow graph representing the computation kernel and the modulo routing resource graph representing the CGRA architecture. We design a customized and

efficient graph minor search algorithm for our problem that employs aggressive pruning and acceleration strategies. We conduct extensive experimental evaluations of our approach and show that it achieves quality schedule with minimal compilation time. The graph minor compilation framework is then integrated with a pre-processing step to support the compilation for the S-CGRA. The proposed CGRA and S-CGRA compilation techniques are essential parts of the whole MPSoC design automation tool chain. They will serve to provide alternative custom extensions, which would be used in the design space exploration step in the next chapter.

Chapter 6

Mapping Multi-threaded Applications on S-CGRA

In Chapter 4, we have proposed our novel customizable MPSoC architecture. The compiler support is then designed for mapping loop kernels into S-CGRA in Chapter 5. This chapter covers designs space exploration for mapping multi-threaded applications on the dynamic customizable MPSoC. The design space exploration algorithm has to be aware of the architectural specifications and takes in the deign alternatives generated from the compiler. Recall that we have already discussed design space exploration in static MPSoC customization in Chapter 3. The high complexity resides in the interdependent task mappings, resource sharing and individual customizations. By further involving the reconfigurability, the design space could be drastically increased. We do, however, set our objective as finding out the optimal or near optimal customization solutions with the considerations of all the design factors.

6.1 Overview

In the MPSoC system that could be dynamically customized using a shared reconfigurable fabric, other than all the design challenges presented in static MPSoC customizations depicted in Chapter 3, one needs to consider reconfigurations by selecting the appropriate set of custom extensions and partitioning them into different configurations to maximize the performance of a multi-threaded application. In this chapter, we provide an optimal solution for temporal and spatial partitioning of the custom extensions. The optimal solution has limited scalability due to its high computational complexity. Hence we propose an iterative refinement algorithm that quickly attains good quality solution. We

evaluate our technique with real embedded applications for the customizable MPSoC architecture presented in Chapter 4. The evaluations confirm that sharing reconfigurable fabric among the cores leads to better solutions compared to per-core dedicated fabric. Moreover, a recommendation is given for the architecture preference between MPSoC architectures with CGRA and S-CGRA.

6.2 Problem Definition

Our architecture is a multi-core system with N cores where all the cores share a reconfigurable fabric (RF). Let $AREA$ be the area of the shared RF and ρ be the reconfiguration latency.

We assume a multi-threaded application with at most N threads running on this multi-core system, i.e., at most one thread is mapped to each core. A thread T_i is modeled as a sequence of n_i tasks $T_{i,1} \ldots T_{i,j} \ldots T_{i,n_i}$. Note that our technique is *not* restricted to linear chain of tasks per thread. If a thread is modeled as a task graph, the tasks can be scheduled through a topological sort of the task graph that respects the dependencies among the tasks. The resulting linear schedule is used as input to our technique. Moreover, it is easy to model applications with pipelined parallelism (e.g., streaming application). Each pipeline stage of the application can be modeled as a thread that maps to a core. All the tasks corresponding to a pipeline stage can be scheduled to create a sequence of tasks for that thread.

Each task is associated with multiple custom extensions (CEs). A CE could consist of a set of custom instruction or loop kernel configurations depending on the architectural specifications. The CEs are generated according to the tradeoff between area and execution time. Let $\{c^0_{i,j}, \ldots, c^{m_{i,j}}_{i,j}\}$ denote the set of possible CEs for task $T_{i,j}$. In addition, let $t^k_{i,j}$ and $a^k_{i,j}$ denote the execution time and area requirement of the CE $c^k_{i,j}$. We assume $c^0_{i,j}$ corresponds to the completely software implementation of the task, i.e., $a^0_{i,j} = 0$. That is, for each task $T_{i,j}$, we have a choice of one software implementation and $m_{i,j}$ implementations accelerated with custom extensions. In addition, $a^0_{i,j} < \ldots a^k_{i,j} < \ldots < a^{m_{i,j}}_{i,j}$ and $t^0_{i,j} > \ldots t^k_{i,j} > \ldots > t^{m_{i,j}}_{i,j}$. The CE of a task must fit into the available area, i.e., $a^k_{i,j} \leq AREA$.

Example Figure 6.1(a) shows an example of two threads with CEs. We assume $AREA = 10$. The first thread has 4 tasks while the second thread has 5 tasks. Each task has multiple CEs. For example, task $T_{1,1}$ has 3 CEs with 0 (software), 1 and 4 area units. The corresponding execution times are 300, 50, and 30 time units.

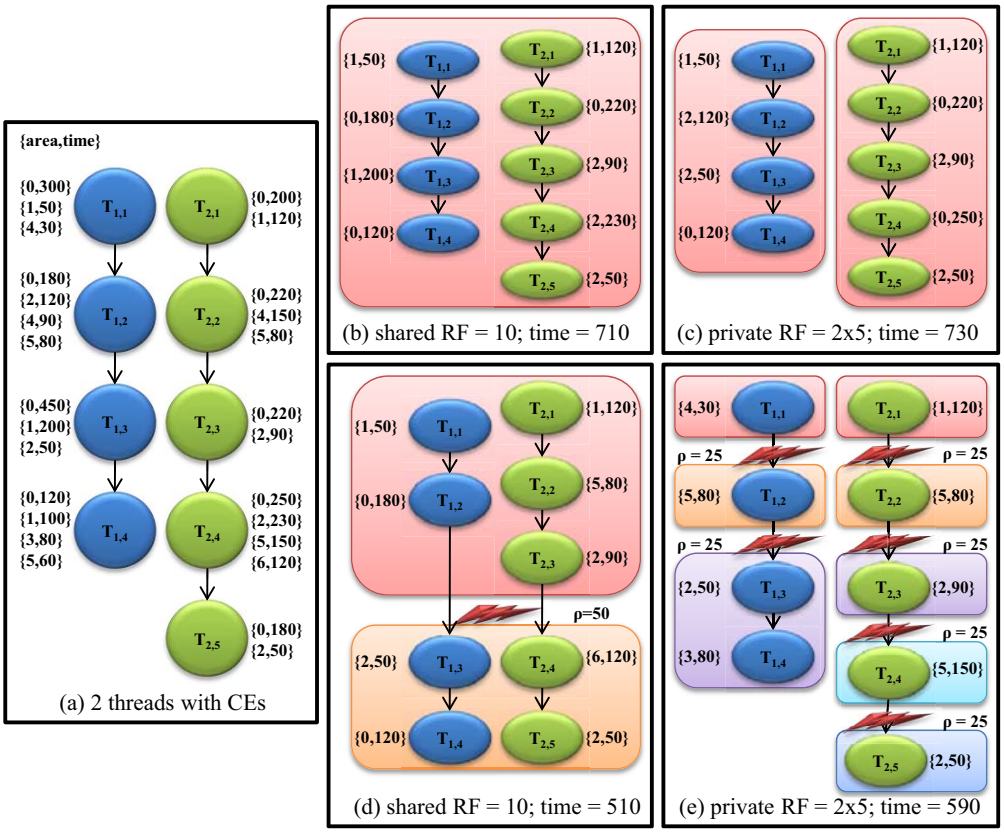


Figure 6.1: Motivating Example

Our objective is to select a CE for each task and appropriate reconfiguration points so as to minimize the execution time of the multi-threaded application, i.e., minimize the execution time of the critical thread.

Static configuration Let us first concentrate on a restricted version of the problem where we do not allow any dynamic reconfiguration of the fabric. Let $x_{i,j}^k$ be a binary variable that is set to 1 if the CE $c_{i,j}^k$ is chosen corresponding to task $T_{i,j}$ and 0 otherwise. Then, our goal is to **minimize** the following objective function:

$$\max_{i=1,\ldots,N} \sum_{j=1}^{n_i} \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times t_{i,j}^k$$

subject to the following constraints

$$\sum_{i=1}^{N} \sum_{j=1}^{n_i} \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times a_{i,j}^k \leq AREA; \quad \forall i,j \quad \sum_{k=0}^{m_{i,j}} x_{i,j}^k = 1$$

This is 0-1 Integer Linear Programming (ILP) problem.

Example Figure 6.1(b) and 6.1(c) show the optimal solutions with shared and private RFs, respectively. In case of private RF per core, we assume each core has access to $10/2 = 5$ units of RF. It is easy to prove that shared RF will always lead to better execution time than private RFs. In our example, we get 710 units of execution time with shared RF compared to 730 units of execution time with private RFs.

Dynamic reconfiguration Allowing dynamic reconfiguration of the fabric adds significant complexity to the problem. Let P be the total number of configurations. In the worst case, each task can have its exclusive configuration, i.e., $P \leq \sum_{i=1}^{N} n_i$. Let $p(T_{i,j})$ be the configuration that $T_{i,j}$ belongs to. Then, we have the constraint $p(T_{i,j}) \leq p(T_{i,j+1})$ as partitions contain consecutive tasks. Clearly, each configuration must satisfy area constraint. Therefore

$$\sum_{\forall i,j \;\; s.t. \;\; p(T_{i,j})=q} \sum_{k=0}^{m_{i,j}} x_{i,j}^{k} \times a_{i,j}^{k} \leq AREA \quad q \in \{1 \dots P\} \tag{6.1}$$

The execution time of the application is the summation of the execution time of each configuration plus the reconfiguration latency. The execution time in each configuration corresponds to the critical thread in that configuration. So our goal is to **minimize** the following objective function:

$$\rho \times (P-1) + \sum_{q=1}^{P} \max_{i=1,\dots,N} \left(\sum_{\forall j \;\; p(T_{i,j})=q} \sum_{k=0}^{m_{i,j}} x_{i,j}^{k} \times t_{i,j}^{k} \right) \tag{6.2}$$

Concretely, our goal is to select the CE of each task (i.e., assign the $x_{i,j}^{k}$ binary variables) and assign the configuration for each task $p(T_{i,j})$ such that the total execution time specified by Equation 6.2 is minimized.

Example Figure 6.1(d) and 6.1(e) show the optimal solutions for shared and private RFs with dynamic reconfiguration. Reconfiguration latency (ρ) is 50 and 25 corresponding to shared and private RFs, respectively. For shared RF, the application has been partitioned into 2 configurations with execution times 290 unit and 170 unit, respectively. Hence the total execution time is (290+170+50 = 510) unit. Shared RF allows flexibility in terms of allocating area to each thread. However, the reconfiguration for all the threads have to be synchronized assuming no partial reconfiguration is supported for the reconfigurable fabric. Thus, load-imbalance among the threads can have a negative impact. If the

threads have private RFs as in 6.1(e), each thread can reconfigure its own fabric independently and asynchronously. In our example, T_1 reconfigures 2 times while T_2 reconfigures 4 times. Still the optimal solution with private RFs requires 590 time units compared to 510 unit for shared RF. This is because T_2 has inherently more requirement of CEs that can be satisfied with shared RF. Therefore, the design space exploration algorithm needs to carefully take into account the tradeoff between imbalance in load and area requirement among the threads.

The presence of both the partition variables and the CE selection variables in the objective function introduces non-linearity making ILP solution infeasible. A much simpler version of the partitioning problem where all the threads have identical number of tasks and the same partitioning is applied to all the threads (there is no reconfiguration delay and CEs) is known as the multistage linear array assignment problem (MLAA) [74]. The MLAA problem has been shown to be NP-complete. We now present an optimal solution to our problem followed by an efficient iterative refinement algorithm that achieves close to the optimal solution.

6.3 Optimal Solution

The optimal solution is constructed in a bottom-up fashion by first computing the solutions per thread, then combining them for multi-threading without reconfiguration before finally proceeding to incorporate multiple configurations.

Algorithm 8: Compute $time_{i,s,e}(A)$ for all i, s, e, A

```
1  for i ← 1 to N do
2      for s ← 0 to n_i do
3          for e ← s + 1 to n_i do
4              for A ← 0 to AREA do
5                  for k ← 0 to m_{i,j} do
6                      if (a^k_{i,e} ≤ A) then
7                          time_{i,s,e}(A) = min(time_{i,s,e}(A), time_{i,s,e-1}(A − a^k_{i,e}) + t^k_{i,e})
8                      end
9                  end
10             end
11         end
12     end
13 end
```

Single thread The term $\sum_{\forall j\ p(T_{i,j})=q} \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times t_{i,j}^k$ in Equation 6.2 defines the execution time of thread T_i in configuration q. Only a consecutive subsequence of tasks from T_i can be mapped to a configuration. Let $T_{i,s+1}$ and $T_{i,e}$

($s \leq$e) be the start and end task of the subsequence of tasks from T_i mapped to a particular configuration. Then the execution time of the subsequence can be defined as

$$time_{i,s,e} = \sum_{j=s+1}^{e} \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times t_{i,j}^k$$

Note that according to our definition $time_{i,0,n_i}$ corresponds to the execution time of the entire thread from task $T_{i,1}$ to task T_{i,n_i}. *Moreover, we assume that* $time_{i,s,s} = 0$ *corresponds to the execution time of an empty sequence of tasks.*

We first pre-compute the minimum value of $time_{i,s,e}$ for all possible values of i, s, e under different area constraints. We design a dynamic programming algorithm to compute these values. The recursive equation is

$$time_{i,s,j}(A) = \min_{\substack{k=0,\ldots,m_{i,j} \\ a_{i,j}^k \leq A}} (time_{i,s,j-1}(A - a_{i,j}^k) + t_{i,j}^k)$$

where $time_{i,s,j}(A)$ (with $s < j$) is the minimum execution time of the subsequence $T_{i,s+1} \ldots T_{i,j}$ under area constraint A. Basically, we start with the task $T_{i,s+1}$ and add one task at a time till we reach the task $T_{i,e}$. For the task $T_{i,j}$, we go through all its CEs that can fit in the area A. For each such CE $c_{i,j}^k$, we allocate its area $a_{i,j}^k$ and the remaining area $A - a_{i,j}^k$ is given to the tasks $T_{i,s+1} \ldots T_{i,j-1}$. The execution time under this allocation is the execution time of the task $T_{i,j}$ with CE $c_{i,j}^k$ and the minimum execution time of the previous tasks under the remaining area constraint $time_{i,s,j-1}(A - a_{i,j}^k)$. Then, we choose the CE that produces minimum execution time under this scenario. In other words, we set $x_{i,j}^k = 1$ for that CE and 0 for all the other CEs. Algorithm 8 illustrates this computation. The complexity of the algorithm is $O(N \times n^2 \times m \times AREA)$ where n is the average number of tasks per thread and m is the average number of CEs per task.

Algorithm 9: Compute $time_{\langle s1,e1\rangle \ldots \langle sN,eN\rangle}$

```
1  for i ← 1 to N do A_i = 0;
2  for A ← 0 to AREA do
3  |   critical = 0; maxTime = 0;
4  |   for i ← 1 to N do
5  |   |   if time_{i,si,ei}(A_i) >= maxTime then
6  |   |   |   maxTime = time_{i,si,ei}(A_i); critical = i;
7  |   |   end
8  |   end
9  |   A_critical = A_critical + 1;
10 end
11 return maxTime;
```

Multi-threading with Static Configuration Let us suppose subsequences $[T_{1,s1+1} \dots T_{1,e1}] \dots [T_{N,sN+1} \dots T_{N,eN}]$ have been mapped to a particular configuration. The execution time of this configuration will be determined by the subsequence with maximum execution time. We also need to satisfy the area constraint of the configuration (see Equation 6.1). We define $time_{\langle s_1,e_1\rangle \dots \langle sN,eN\rangle}$ as the execution time of the subsequences $[T_{1,s_1+1} \dots T_{1,e_1}] \dots [T_{N,sN+1} \dots T_{N,eN}]$ mapped to a configuration. We propose Algorithm 9 to efficiently compute the minimum value of $time_{\langle s1,e1\rangle \dots \langle sN,eN\rangle}$.

Our goal is to partition $AREA$ among all the threads to minimize the execution time of the critical thread. Initially, we set the area assigned to each thread (A_i) to 0. In each step, we allocate unit area to the critical thread to reduce its execution time. The correctness of the algorithm can be easily proved through induction on area A, as the execution time can be potentially decreased only by assigning the area increment to the critical thread.

Note that $time_{\langle 0,n_1\rangle \dots \langle 0,n_N\rangle}$ corresponds to the minimum execution time of the entire application with single configuration. That is, Algorithm 9 can generate the optimal solution for N threads with shared RF without reconfiguration. The complexity of this algorithm is $O(N \times AREA)$.

Algorithm 10: Optimal Algorithm

```
1  P = 1;
2  repeat
3  |  P = P + 1;
4  |  improve = false;
5  |  for all combinations of e_i (0 ≤ e_i ≤ n_i) do
6  |  |  min = time_⟨0,e1⟩...⟨0,eN⟩|(P − 1);
7  |  |  for all combinations of v_i (0 ≤ v_i ≤ e_i) do
8  |  |  |  temp = time_⟨0,v1⟩...⟨0,vN⟩|(P − 1) + ρ + time_⟨v1,e1⟩...⟨vN,eN⟩;
9  |  |  |  if temp < min then
10 |  |  |  |  min = temp;
11 |  |  |  |  improve = true;
12 |  |  |  end
13 |  |  end
14 |  |  time_⟨0,e1⟩...⟨0,eN⟩|P = min;
15 |  end
16 until !improve;
17 opt = time_⟨0,n1⟩...⟨0,nN⟩|P;
18 return opt;
```

Multi-threading with Dynamic Reconfiguration We now proceed to introduce reconfiguration. Let us define $time_{\langle s_1,e_1\rangle\ldots\langle s_N,e_N\rangle}|P$ as the minimum execution time of the task subsequences $[T_{1,s1+1}\ldots T_{1,e1}]\ldots[T_{N,sN+1}\ldots T_{N,eN}]$ with P configurations including reconfiguration overhead of $\rho\times(P-1)$. For one configuration, $time_{\langle s_1,e_1\rangle\ldots\langle s_N,e_N\rangle}|1 = time_{\langle s_1,e_1\rangle\ldots\langle s_N,e_N\rangle}$. We define a recursive equation to compute the execution time for P configurations given the execution times for $P-1$ configurations as follows.

$$time_{\langle s_1,e_1\rangle\ldots\langle s_N,e_N\rangle}|P = \min_{\forall i\ s_i\leq v_i\leq e_i}(time_{\langle v_1,e_1\rangle\ldots\langle v_N,e_N\rangle} + \rho + time_{\langle s_1,v_1\rangle\ldots\langle s_N,v_N\rangle}|(P-1))$$

The equation states that we need to explore all possible combination of starting points in each thread for the P^{th} configuration. This is achieved by setting v_i (starting points of P^{th} configuration) between s_i and e_i for each thread T_i. Then, $time_{\langle v_1,e_1\rangle\ldots\langle v_N,e_N\rangle}$ denotes the execution time of the P^{th} configuration. The remaining tasks are assigned to the $P-1$ configurations and $time_{\langle s_1,v_1\rangle\ldots\langle s_N,v_N\rangle}|(P-1)$ denotes the corresponding execution time. We add the reconfiguration overhead. The combination of starting points that provides the minimum execution time is the optimal solution.

Algorithm 10 describes the dynamic programming algorithm to find the optimal solution. We start with $P=1$ configuration and increment the number of configurations by one in each step. We compute the execution time for all possible partitions and then select the one with the minimum execution time. If the execution time improves with the additional configuration, we continue. Otherwise, the algorithm terminates. Clearly, the algorithm has exponential complexity of $O(n^N)$ where n is the number of tasks per thread. However, this algorithm produces the optimal solution and provides a solid reference point.

6.4 Iterative Refinement

Now we present an iterative refinement technique (see Algorithm 11) that avoids the exponential complexity of the optimal algorithm while achieving close to optimal solution. The basic idea is to start with the static configuration and partition one of the configurations in each step. Suppose we have P configurations (represented by $SetP$) after $P-1$ partitioning steps. Corresponding to each configuration, we maintain the start and end tasks of each thread ($Start$, End), the area required by each thread ($Area$), and the execution time ($Time$). We then choose the configuration p with the maximum execution time and attempt

Algorithm 11: Iterative Refinement (IR) Algorithm

```
1  add static configuration to SetP; min = time_⟨0,n1⟩,...⟨0,nN⟩;
2  while SetP ≠ φ do
3      choose config p from SetP with max execution time;
4      for i ← 1 to N do
5          s_i = Start[p][i]; e_i = End[p][i]; A = Area[p][i];
6          find v_i with min |time_{i,s_i,v_i}(A) − time_{i,v_i,e_i}(A)|;
7      end
8      temp = time_⟨s1,v1⟩...⟨sN,vN⟩ + time_⟨v1,e1⟩...⟨vN,eN⟩ + ρ;
9      if temp < Time[p] then
10         min = min − (Time[p] − temp);
11         replace p with partitions of p in SetP;
12         update Start, End, Area, Time;
13     end
14     else
15         remove p from SetP;
16     end
17 end
18 return min;
```

to partition it. The heuristic partitions each thread independently as follows. If in configuration p, thread T_i was allocated area $Area[p][i]$, then we allocate the same area to each of its partition. We then select the point v_i to maximize the balance between the two partitions of T_i. Once the partitioning points for all the threads have been selected, we compute the actual execution time per partition by invoking Algorithm 9 and add the reconfiguration overhead. If the execution time of p reduces with partitioning, then we add the new configurations to $SetP$. Otherwise, we remove p from further consideration. The algorithm terminates when we cannot optimize any configuration through partitioning. The complexity per iteration is $O(N \times n + N \times AREA)$. As the number of reconfigurations is typically quite small, the algorithm terminates quickly.

Example We illustrate the algorithm with the same example used in Figure 6.1. The threads and their CE information are shown in Figure 6.2(a). We start with the static configuration, i.e., the solution in Figure 6.2(b) with execution time 710. Here T_1 occupies 2 units of area, whereas T_2 occupies 7 units of area. We try to partition each thread independently as shown in Figure 6.2(c). Each partition of T_1 is assigned 2 units of area. With this constraint, the best partitioning point is after task $T_{1,2}$. As T_2 is the critical thread, 1 unit of unassigned area is added to its allocated 7 units of area. With area 8, the best

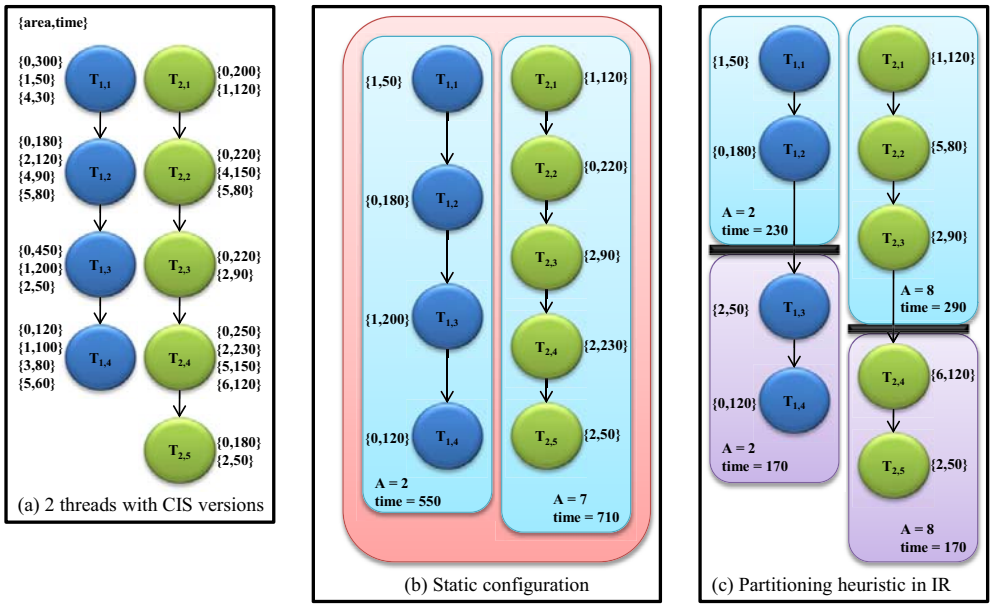


Figure 6.2: An illustrative example for iterative heuristic

partitioning point of T_2 is after task $T_{2,3}$. Now we determine the execution time of the configuration $\{\langle T_{1,1}T_{1,2}\rangle, \langle T_{2,1}T_{2,2}T_{2,3}\rangle\}$, which is 290. The execution time of the other configuration $\{\langle T_{1,3}T_{1,4}\rangle, \langle T_{2,4}T_{2,5}\rangle\}$ is 170. Hence the execution time of 2-configuration solution is (290+170+50=510), which is better than 1-configuration solution. Next we try to partition each of the configurations. But further partitioning does not improve execution time. So the algorithm returns the 2-configuration solution, which is also the optimal solution as shown in Figure 6.1(d).

6.5 Experimental Evaluation

6.5.1 Design Automation Tool Overview

Combining the architectural specifications depicted in Chapter 4, compilation supports detailed in Chapter 5 and the design space exploration techniques explained in this chapter, we have a full design automation tool chain to support dynamic MPSoC customization. The whole design automation flow is shown in Figure 6.3.

At the first step, the source code is fed into a profile tool to extract computationally intensive kernels. In our experimental evaluations, computationally intensive kernels are identified as hot basic blocks. We adopt the profile tool

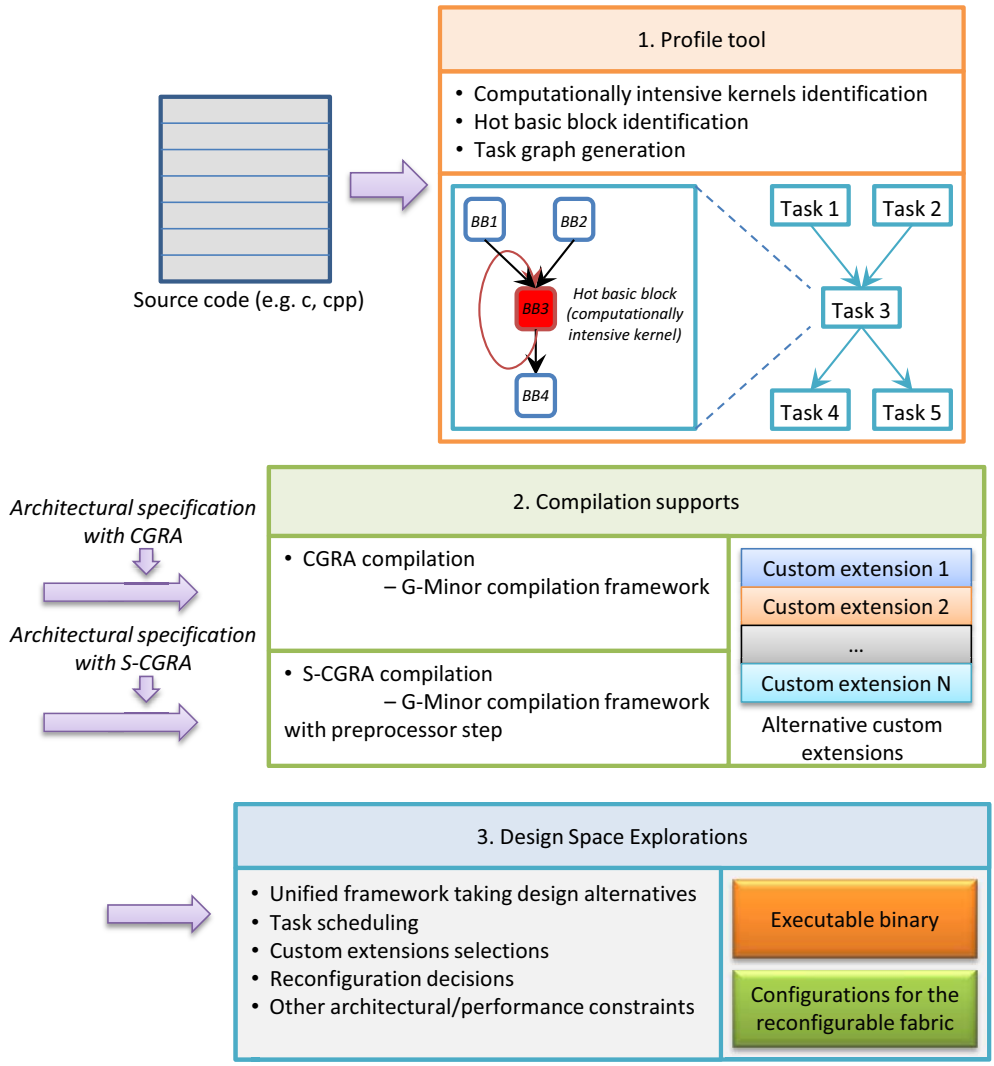


Figure 6.3: The whole design automation flow

from [132], which is originally used to identify custom instructions. Beside hot basic block identification, the application is also partitioned and represented as a task graph. Each task could either be a function or a set of functions. The task graph generation could also be done manually by experienced designer.

The generated task graphs together with hot basic blocks are then fed into compilers regarding the different architectural specifications. For shared CGRA, we use our G-Minor compilation framework to generate different configurations for computationally intensive loop kernels as custom extensions. If the S-CGRA is used as the shared reconfigurable fabric, then we use the modified G-Minor framework, which is integrated with a pre-processing step as detailed in Chapter 5.

The alternative custom extensions generated from the compilers will be used in our final design space exploration step. The unified design space exploration framework is able to take in all the architectural specifications and custom extensions to produce feasible customization solutions. The exploration process should be able to perform task scheduling, select custom extensions, and make reconfiguration decisions with the considerations of various architectural and performance constraints. The final outputs would be an executable binary to be run in the MPSoC system and a configurable file containing all the runtime configuration information for the reconfigurable fabric.

6.5.2 Experimental Evaluations for MPSoCS with CGRA and S-CGRA

We now study the dynamic MPSoC customization techniques. As S-CGRA is just a derivative of CGRA and the MPSoC architectures are very similar, we combine the experimental evaluations for the two architectures. The experiments are designed in the first place to verify the benefits brought by resource sharing. This is done through the comparisons between the architecture with a shared coprocessor and the architecture with private coprocessors. Experiments are also conducted to evaluate the advantages of using the proposed architecture with the S-CGRA comparing to the architecture with the normal CGRA.

Experimental Setup

Compiler modification The back-end of SimpleScalar-gcc cross compiler is modified to extract the DFGs for the computation intensive kernels. These DFGs are fed into G-Minor compilation framework with/without the proposed clustering algorithm as a pre-processing step. The G-Minor mapper will generate multiple versions of solutions or CEs by varying the number of available rows in the CGRA or S-CGRA.

Frequency The frequency of the baseline processor is set as 2GHz, and the frequency for S-CGRA achieves 606MHz according to the synthesis results presented in Chapter 4. For the normal CGRA, we assume each functional unit is comprehensive to support all the operations and consists of one basic functional unit and one complex functional unit as referred in Chapter 4. The frequencies of the S-CGRA and normal CGRA are, however, roughly the same, as they are both constrained by the critical path length of the complex functional unit.

Area Each functional unit in the normal CGRA consumes roughly 74% of the area consumed by the specialized functional unit in S-CGRA. Thus, using the area for one row consisting of 4 SFUs, we are able to create a row of 5 comprehensive FUs in the normal CGRA. Similarly, the area of 8 SFUs is similar to the area for creating 10 comprehensive FUs. Notice that in both S-CGRA and CGRA, about 35% of the total area is used for networking, which matches the results reported in [73, 44, 21].

Reconfiguration time The reconfiguration time is set according to the number of configuration bits and the DMA transfer rate. It is the time consumed to transfer all the II configurations for the target loop kernel from on-chip configuration memory to the configuration caches. Recall that each SFU needs 62 bits for functional configuration per cycle. Then, for a 4X4 S-CGRA that contains 16 SFUs arranged in a 2D mesh topology, we need 992 bits per cycle to configure the functionalities of the SFUs. As referred in 4, the network configuration requires 328 bits. If the maximum depth of the configuration cache is 10 (max II = 10), then the total number of the configuration bits is 13,200. Assuming a 2MB/s DMA transfer rate, the reconfiguration time in terms of baseline processor cycles is 1,573,563 cycles.

Comparisons between MPSoCs with CGRA and S-CGRA

We evaluate the efficiency of the proposed architecture using JPEG encoder and MP3 encoder applications. Our proposed architecture consists of two cores with a shared S-CGRA, named as ***Shared S-CGRA*** for the convenience. The architectures to compare with includes a two-core system with private S-CGRAs (***Private S-CGRA***), a two-core system with a shared CGRA (***Shared CGRA***), and a two-core system with private CGRAs (***Private CGRA***). Five hot kernels are identified for each of the applications. Each application is partitioned into two pipeline stages and each stage is mapped to one core. We will examine the efficiencies of the architectures by checking the execution time of the critical pipeline stage.

We first constrained the number of SFUs in one S-CGRA row to 4 and the number of FUs in one CGRA row to 5. The experimental results are shown in Figure 6.4. Clearly, sharing technique could bring significant speedup for both architectures with S-CGRA and normal CGRA when the area budget is limited. It is also shown that using S-CGRA as the shared coprocessor is more promising than the normal CGRA. In both the applications, around 5% speedup is observed for ***Shared S-CGRA*** comparing to ***Shared CGRA***. Finally, for the streaming

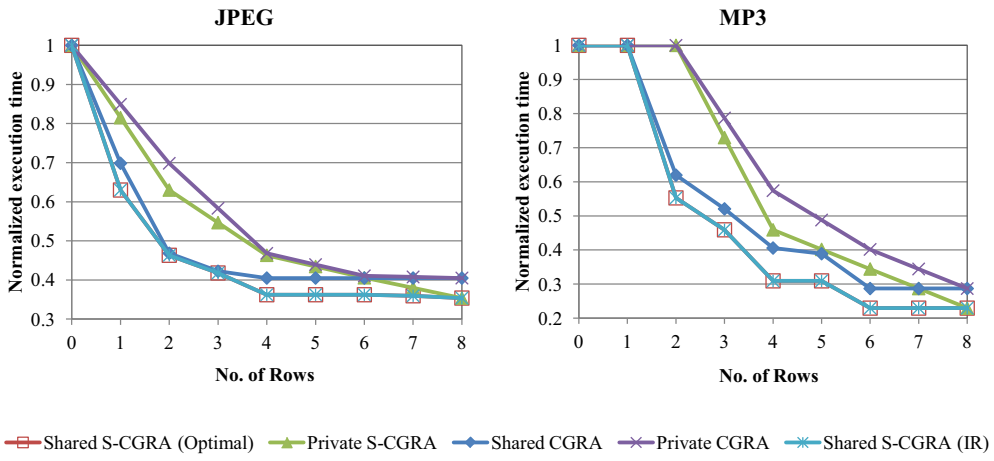


Figure 6.4: Experimental results for shared S-CGRA, private S-CGRA, shared CGRA and private CGRA, each row consists of 4 SFUs or 5 FUs

applications such as JPEG and MP3, we conclude that 4×6 S-CGRA is the optimal architecture, which has least number of rows and is sufficient to achieve most of the speedup. In the experiments, we also demonstrate the efficiency of our iterative refinement heuristic by comparing it to the optimal solution. The iterative refinement heuristic, denoted as **Shared S-CGRA (IR)**, consistently generates similar results as the optimal solution, shown as **Shared S-CGRA (optimal)** in the figure.

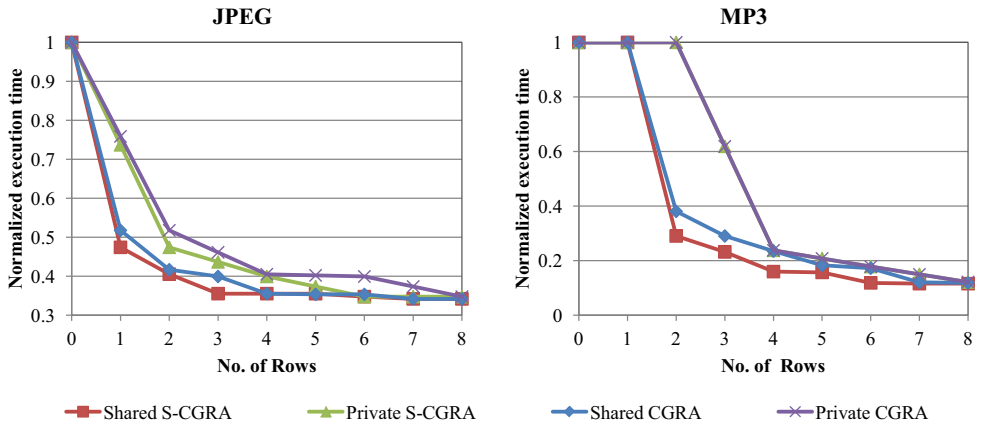


Figure 6.5: Experimental results for shared S-CGRA, private S-CGRA, shared CGRA and private CGRA, each row consists of 8 SFUs or 10 FUs

Another factor that impacts the final speedup is the number of functional

units in one row. We now change the number of SFUs in one S-CGRA row to 8 and the number of FUs in one CGRA row to 10. The experimental results in Figure 6.5 show that the extra accelerations brought by using S-CGRA become negligible when the number of available rows increases. This is true as a sufficiently large CGRA will give the same II as one S-CGRA for a particular loop kernel. However, the architectures with S-CGRAs outperform the architectures with CGRAs when the area budget is constrained. Obviously, sharing is preferred to achieve significant speedups. Again, we conclude that S-CGRA with 6 rows is also a good architecture for streaming applications.

6.6 Chapter Summary

In this chapter, we cover the design space exploration technique, which is the final step in the design automation tool for MPSoC customization. We formalize the design space exploration problem in dynamic MPSoC customization with the considerations of task scheduling, customization alternatives selections, resource sharing, reconfigurations and architectural/performance constraints. An optimal algorithm is then proposed to solve the problem. Considering the high complexity of the optimal algorithm, an iterative heuristic is designed to reduce the design space exploration time. With the efficient design space exploration technique, we have a full design automation tool. The design automation tool contains different architectural specifications, compiler supports to generate custom extension alternatives for the different architectures, and a unified design space exploration framework that can efficiently generate final MPSoC customization solutions. Through our experimental evaluations, we confirm the efficiency of the whole design automation tool and give feedbacks for the preferences of the architectures.

Chapter 7

Conclusion

7.1 Thesis Contribution

This thesis exposes and tackles the challenges in MPSoC customization problems. The thesis presents a unified framework for crafting a heterogenous MPSoC through customization techniques.

The main contributions of this thesis are as follow:

- We formalize the static MPSoC customization problem with the considerations of task scheduling, chip area sharing, alternative custom instruction sets selections and QoS constraints. An efficient hierarchical algorithm is proposed to locate the most resource-efficient customized MPSoC designs in the vast design space dealing with streaming applications.

- We propose a novel customizable MPSoC architecture with a shared coarse-grained reconfigurable fabric, S-CGRA. The heart of our innovation is a specialized functional unit (SFU) that can execute most application-specific instructions at ASIP-like efficiency through fast reconfiguration. Using SFU as the primary processing element of the S-CGRA, the S-CGRA is able to explore massive speedups of the computational intensive kernels.

- A graph minor approach is proposed by us to solve CGRA mapping problems. The graph minor formalization for the CGRA mapping problem serves as a bridge between the graph theory and the practical CGRA compilation problem. We design a customized and efficient graph minor search algorithm that employs aggressive pruning and acceleration strategies. Extensive experimental evaluations show that our approach achieves quality schedule with minimal compilation time.

- We formalize the problem of dynamic MPSoC customization with a shared reconfigurable fabric. With the considerations of reconfigurations and all the other challenges found in static MPSoC customization, we have successfully developed an efficient algorithm that can minimize the execution time for multi-threaded applications by selecting appropriate custom instructions and reconfiguration points. We demonstrate the benefits of sharing the reconfigurable fabric as opposed to independent reconfigurable fabric per core.

7.2 Future Work

MPSoC customization problem is highly complex. Despite our extensive design efforts, we only tackle a small portion of the whole MPSoC customization problem. Some of the possible future research directions include:

- *Power management for the customizable MPSoC.* As power consumption becomes a more and more important topic in embedded system design, it is valuable to evaluate the impacts of power consumption in MPSoC customizations. As different custom extensions could have different power consumptions, one potential topic could be efficient runtime MPSoC customization under the thermal constraints.

- *A combination of fine-grained and coarse-grained architectures.* We have investigated the MPSoC customization techniques individually for both the fine-grained and coarse-grained architectures. As different applications might require different customization granularity, a study on the hybrid architectures is desired.

- *Many-core system customization with clustered reconfigurable fabrics.* The many-core era will turn the processor customization problem into a prosperous research area. We can expect that the overhead of sharing a centralized reconfigurable fabric would be too expensive and clustered reconfigurable fabrics could be introduced to solve the scalability problem. However, the run-time application demands would complicate the architectural designs and scheduling mechanisms.

These are only some preliminary thoughts and they require comprehensive investigations. We believe that the multi-processor customization will benefit significantly from the continued research in this domain.

Bibliography

[1] The trimaran compiler infrastructure. `http://www.trimaran.org`.

[2] In conversation with tensilica ceo chris rowen. *IEEE Design Test of Computers*, 25(1):88–95, 2008.

[3] Shail Aditya, Vinod Kathail, and B Ramakrishna Rau. *Elcor's machine description system: Version 3.0*. Hewlett Packard Laboratories, 1998.

[4] Isolde Adler, Frederic Dorn, Fedor V Fomin, Ignasi Sau, and Dimitrios M Thilikos. Fast minor testing in planar graphs. *Algorithmica*, 64(1):69–84, 2012.

[5] Mythri Alle, Keshavan Varadarajan, Reddy C Ramesh, Joseph Nimmy, Alexander Fell, Adarsha Rao, SK Nandy, and Ranjani Narayan. Synthesis of application accelerators on runtime reconfigurable hardware. In *Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 13–18. IEEE, 2008.

[6] Federico Angiolini, Jianjiang Ceng, Rainer Leupers, Federico Ferrari, Cesare Ferri, and Luca Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *Proceedings of the 2006 conference on Design, Automation and Test in Europe*, pages 1–6. IEEE, 2006.

[7] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Design and architectural exploration of expression-grained reconfigurable arrays. In *Proceedings of the 2008 Symposium on Application Specific Processors*, pages 26–33. IEEE, 2008.

[8] Kubilay Atasu, Günhan Dündar, and Can Özturan. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 172–177. ACM, 2005.

[9] Kubilay Atasu, Oskar Mencer, Wayne Luk, Can Ozturan, and Gunhan Dundar. Fast custom instruction identification by convex subgraph enumeration. In *Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 1–6. IEEE, 2008.

[10] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th annual Design Automation Conference*, pages 256–261. ACM, 2003.

[11] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[12] Nikhil Bansal, Sumit Gupta, Nikil Dutt, and Alexandru Nicolau. Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations. In *Workshop on Application Specific Processors, held in conjunction with the International Symposium on Microarchitecture (MICRO)*, 2003.

[13] Lars Bauer, Muhammad Shafique, Simon Kramer, and Jörg Henkel. Rispp: rotating instruction set processing platform. In *Proceedings of the 44th annual Design Automation Conference*, pages 791–796. ACM, 2007.

[14] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 68(6):790–808, 2008.

[15] Paolo Bonzini, Giovanni Ansaloni, and Laura Pozzi. Compiling custom instructions onto expression-grained reconfigurable architectures. In *Proceedings of the 2008 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 51–60. ACM, 2008.

[16] Unmesh D Bordoloi, Huynh Phung Huynh, Tulika Mitra, and Samarjit Chakraborty. Design space exploration of instruction set customizable mpsocs for multimedia applications. In *Proceedings of the 2010 International Conference on Embedded Computer Systems*, pages 170–177. IEEE, 2010.

[17] Anne Bracy, Prashant Prahlad, and Amir Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 18–29. IEEE, 2004.

[18] Anne Bracy and Amir Roth. Serialization-aware mini-graphs: Performance with fewer resources. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 171–184. IEEE, 2006.

[19] Janina A Brenner, Sándor P Fekete, and Jan C van der Veen. A minimization version of a directed subgraph homeomorphism problem. *Mathematical Methods of Operations Research*, 69(2):281–296, 2009.

[20] Timothy J Callahan, John R Hauser, and John Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4):62–69, 2000.

[21] Liang Cao and Huang Xinming. SmartCell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP Journal on Embedded Systems*, 2009, 2009.

[22] Jorge E Carrillo and Paul Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 9th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 141–150. ACM, 2001.

[23] Liang Chen and Tulika Mitra. Shared Reconfigurable Fabric for Multi-core Customization. In *Proceedings of the 48th Design Automation Conference*, pages 830–835. ACM, 2011.

[24] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on CGRAs. In *Proceedings of the 2012 International Conference on Field Programmable Technology*, pages 285–292. IEEE, 2012.

[25] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on CGRAs. *ACM Transactions on Reconfigurable Technology and Systems*, 2014.

[26] Liang Chen, Joseph Tarango, Philip Brisk, and Tulika Mitra. A Just-in-Time Customizable Processor. In *Proceedings of the 48th Design Automation Conference*, pages 524–531. ACM, 2011.

[27] Linag Chen, Nicolas Boichat, and Tulika Mitra. Customized MPSoC Synthesis for Task Sequences. In *Proceedings of the 9th Symposium on Application Specific Processors*, pages 16–22. IEEE, 2011.

[28] Zhimin Chen, Richard Neil Pittman, and Alessandro Forin. Combining multicore and reconfigurable instruction set extensions. In *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–36. ACM, 2010.

[29] N Clark, J Blome, M Chu, S Mahlke, S Biles, and K Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *Computer Architecture, 2005. ISCA\'05. Proceedings. 32nd International Symposium on*, pages 272–283, 2005.

[30] Nathan Clark, Amir Hormati, Scott Mahlke, and Sami Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 147–157. ACM, 2006.

[31] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 30–40. IEEE, 2004.

[32] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[33] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[34] Michael Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *Proceedings of the 2003 conference on Design, Automation and Test in Europe*, pages 980–985. IEEE, 2003.

[35] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 151–160. ACM, 2008.

[36] André DeHon. Dpga utilization and application. In *Proceedings of the 4th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 115–121. ACM, 1996.

[37] JC DeSouza-Batista and Alice C Parker. Optimal synthesis of application specific heterogeneous pipelined multiprocessors. In *Proceedings of the 1994 International Conference on Application Specific Array Processors*, pages 99–110. IEEE, 1994.

[38] Muhammad K Dhodhi, Imtiaz Ahmad, Anwar Yatama, and Ishfaq Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of parallel and distributed computing*, 62(9):1338–1361, 2002.

[39] Giuseppe Di Battista, Maurizio Patrignani, and Francesco Vargiu. A split & push approach to 3d orthogonal drawing. *Journal of Graph Algorithms and Applications*, 4(3):105–133, 2000.

[40] Jack J Dongarra and Piotr Luszczek. Introduction to the HPCChallenge benchmark suite. Technical report, DTIC Document, 2004.

[41] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proceedings of the IFIP WG*, pages 264–267, 1995.

[42] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[43] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the 17th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 191–200. ACM, 2009.

[44] Anup Gangwar, M Balakrishnan, Preeti R Panda, and Anshul Kumar. Evaluation of bus based interconnect mechanisms in clustered VLIW architectures. In *Proceedings of the 2005 Conference on Design, Automation and Test in Europe*, pages 730–735. IEEE Computer Society, 2005.

[45] Philip Garcia and Katherine Compton. A reconfigurable hardware interface for a modern computing system. In *Proceedings of 15th annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 73–84. IEEE, 2007.

[46] Philip Garcia and Katherine Compton. Kernel sharing on reconfigurable multiprocessor systems. In *Proceedings of the 2008 International Conference on Field Programming Technology*, pages 225–232. IEEE, 2008.

[47] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.

[48] Rani Gnanaolivu, Theodore S Norvell, and Ramachandran Venkatesan. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *Proceedings of the 2010 International Conference on Soft Computing and Pattern Recognition*, pages 145–151. IEEE, 2010.

[49] Rani Gnanaolivu, Theodore S Norvell, and Ramachandran Venkatesan. Analysis of inner-loop mapping onto coarse-grained reconfigurable architectures using hybrid particle swarm optimization. *International Journal of Organizational and Collective Intelligence*, 2(2):17–35, 2011.

[50] David Edward Goldberg et al. *Genetic algorithms in search, optimization, and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989.

[51] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R Reed Taylor, and Ronald Laufer. Piperench: a coprocessor for streaming multimedia acceleration. *ACM SIGARCH Computer Architecture News*, 27(2):28–39, 1999.

[52] Ricardo E Gonzalez. Xtensa: A configurable and extensible processor. *IEEE micro*, 20(2):60–70, 2000.

[53] Ricardo E Gonzalez. A software-configurable processor architecture. *IEEE Micro*, 26(5):42–51, 2006.

[54] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 503–514. IEEE, 2011.

[55] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 International Workshop on Workload Characterization*, pages 3–14. IEEE, 2001.

[56] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: using epimorphism to map applications on CGRAs. In *Proceedings of the 49th annual Design Automation Conference*, pages 1284–1291. ACM, 2012.

[57] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. REGIMap: Register-Aware Application Mapping on Coarse-Grained Reconfigurable

Architectures (CGRAs). In *Proceedings of the 50th Annual Design Automation Conference*, pages 18:1–18:10. ACM, 2013.

[58] Pierre Hansen and Keh-Wei Lih. Improved Algorithms for Partitioning Problems in Parallel, Pipelined, and Distributed Computing. *IEEE Transactions on Computers*, 41(6), 1992.

[59] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the 2001 conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[60] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Proceedings of 2007 International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[61] Scott Hauck, Thomas W Fry, Matthew M Hosler, and Jeffrey P Kao. The chimaera reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration Systems*, 12(2):206–217, 2004.

[62] John R Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21. IEEE, 1997.

[63] Edwin SH Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994.

[64] Tensilica Inc. `http://www.tensilica.com`.

[65] Mohammad Ashraf Iqbal and Shahid H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Transactions Parallel and Distributed Systems*, 6(2):170–175, 1995.

[66] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[67] Haris Javaid and Sri Parameswaran. Synthesis of heterogeneous pipelined multiprocessor systems using ILP: JPEG case study. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–6. ACM, 2008.

[68] Muhammad Kafil and Ishfaq Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6(3):42–50, 1998.

[69] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[70] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[71] Vida Kianzad and Shuvra S Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):667–680, 2006.

[72] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee W Yoon, Doosan Cho, and Yunheung Paek. High throughput data mapping for coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1599–1609, 2011.

[73] Yoonjin Kim. Reconfigurable multi-array architecture for low-power and high-speed embedded systems. *Journal of Semiconductor Technology and Science*, 11(3):207–220, 2011.

[74] RK Kincaid, DM Nicol, DR Shier, and D Richards. A multistage linear array assignment problem. *Operations research*, 38(6):993–1005, 1990.

[75] Ralf Koenig, Lars Bauer, Timo Stripf, Muhammad Shafique, Waheed Ahmed, Juergen Becker, and Jörg Henkel. KAHRISMA: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture. In *Proceedings of the 2010 Conference on Design, Automation and Test in Europe*, pages 819–824. European Design and Automation Association, 2010.

[76] Shiann-Rong Kuang, Chin-Yang Chen, and Ren-Zheng Liao. Partitioning and pipelined scheduling of embedded system using integer linear programming. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, volume 2, pages 37–41. IEEE, 2005.

[77] Rakesh Kumar, Norman P Jouppi, and Dean M Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 195–206. IEEE, 2004.

[78] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[79] Yu-Kwong Kwok and Ishfaq Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47(1):58–77, 1997.

[80] Zion Kwok and Steven JE Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *Proceedings of the 13th annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 35–44. IEEE, 2005.

[81] James Lebak, Albert Reuther, and Edmund Wong. Polymorphous computing architecture (pca) kernel-level benchmarks. Technical report, DTIC Document, 2005.

[82] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335. IEEE, 1997.

[83] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. Flora: Coarse-grained reconfigurable architecture with floating-point operation capability. In *Proceedings of the 2009 International Conference on Field-Programmable Technology*, pages 376–379, 2009.

[84] Jong-eun Lee, Kiyoung Choi, and Nikil D Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design & Test of Computers*, 20(1):26–33, 2003.

[85] Rainer Leupers, Kingshuk Karuri, Stefan Kraemer, and M Pandey. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *Proceedings of the 2006 Conference on Design, Automation and Test in Europe*, volume 1, pages 6–pp. IEEE, 2006.

[86] Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341–352, 1973.

[87] Lindo System Inc. Lingo. `http://www.lindo.com`.

[88] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213. IEEE, 2006.

[89] Roman Lysecky, Greg Stitt, and Frank Vahid. Warp processors. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):659–681, 2004.

[90] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 7th annual ACM/SIGDA International Symposium on Field programmable Gate Arrays*, pages 135–143. ACM, 1999.

[91] Larry McMurchie and Carl Ebeling. PathFinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 3rd annual ACM/SIGDA International Symposium on Field programmable Gate Arrays*, pages 111–117. ACM, 1995.

[92] Bingfeng Mei, S Vernalde, D Verkest, H De Man, and R Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the 2003 Conference on Design, Automation and Test in Europe*, pages 296–301. IEEE, 2003.

[93] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proceedings of the 2003 International Conference on Field Programmable Logic and Application*, pages 61–70. Springer, 2003.

[94] T. Miyoshi et al. A coarse grain reconfigurable processor architecture for stream processing engine. In *FPL*, 2011.

[95] Andreas Moshovos, Zhi Alex Ye, Prithviraj Banerjee, and Scott Hauck. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 225–225. ACM Press, 2000.

[96] David M. Nicol and David R. O'Hallaron. Improved algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, 1991.

[97] Nils J Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982.

[98] Spec org. SPEC CPU Benchmark Suits. `http://www.spec.org/cpu`.

[99] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 136–146. ACM, 2006.

[100] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176. ACM, 2008.

[101] K. Patel et al. SYSCORE: a coarse grained reconfigurable array architecture for low energy biosignal processing. In *FCCM*, 2011.

[102] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[103] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1209–1229, 2006.

[104] Samantha Ranaweera and Dharma P Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, pages 445–450. IEEE, 2000.

[105] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[106] Rahul Razdan and Michael D Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 172–180. ACM, 1994.

[107] N Robertson and P Seymour Graph Minors. Graph minors. *Journal of Combinatorial Theory, Series B*, 77(1), 1999.

[108] Neil Robertson and Paul D Seymour. Graph minors. XX. Wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

[109] Charle R Rupp, Mark Landguth, Tim Garverick, Edson Gomersall, Harry Holt, Jeffrey M Arnold, and Maya Gokhale. The NAPA adaptive processing architecture. In *Proceedings of the 1998 IEEE Symposium on FPGAs for Custom Computing Machines*, pages 28–37. IEEE, 1998.

[110] Jeremy Kepner Ryan Haney, Theresa Meuse and James Lebak. The high performance embedded computing (HPEC) challenge benchmark suite. In *Proceedings of the 9th annual High-Performance Embedded Computing Workshop*, 2005.

[111] Peter G Sassone and D Scott Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proceedings of 37th International Symposium on Microarchitecture*, pages 7–17. IEEE, 2004.

[112] Peter G Sassone, D Scott Wills, and Gabriel H Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. *ACM SIGPLAN Notices*, 40(7):127–136, 2005.

[113] Markus Schwiegershausen and Peter Pirsch. A formal approach for the optimization of heterogeneous multiprocessors for complex image processing schemes. In *Proceedings of the 1995 European Design Automation Conference*, pages 8–13. IEEE, 1995.

[114] Seng Lin Shee and Sri Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *Proceedings of the 44th annual Design Automation Conference*, pages 811–816. ACM, 2007.

[115] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[116] Paul F Stelling and Vojin G Oklobdzija. Implementing multiply-accumulate operation in multiplication time. In *Proceedings of 13th IEEE Symposium on Computer Arithmetic*, pages 99–106. IEEE, 1997.

[117] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. Application-specific heterogeneous multiprocessor synthesis using exten-

sible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1589–1602, 2006.

[118] Timothy J Todman, George A Constantinides, Steven JE Wilton, Oskar Mencer, Wayne Luk, and Peter YK Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):193–207, 2005.

[119] Mohammed Ashraful Alam Tuhin and Theodore S Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *Proceedings of the 2008 Canadian Conference on Electrical and Computer Engineering*, pages 001723–001728. IEEE, 2008.

[120] Antonino Tumeo, Marco Branca, Lorenzo Camerini, Christian Pilato, Pier Luca Lanzi, Fabrizio Ferrandi, and Donatella Sciuto. Mapping pipelined applications onto heterogeneous embedded systems: a bayesian optimization algorithm based approach. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 443–452. ACM, 2009.

[121] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[122] Frank Vahid, Greg Stitt, and Roman L Lysecky. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. *IEEE Computer*, 41(7):40–46, 2008.

[123] K Van Rompaey, H de Man, D Verkest, and I Bolsens. CoWare-A design environment for heterogeneous hardware/software systems. In *Proceedings of the 1996 European Design Automation Conference*, pages 0252–0252. IEEE, 1996.

[124] Stamatis Vassiliadis, James Phillips, and Bart Blaner. Interlock collapsing ALU's. *IEEE Transactions on Computers*, 42(7):825–839, 1993.

[125] Lee Wang, Howard Jay Siegel, Vwani P Roychowdhury, and Anthony A Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[126] Matthew A Watkins and David H Albonesi. ReMAP: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 497–508. IEEE, 2010.

[127] Michael J Wirthlin and Brad L Hutchings. A dynamic instruction set computer. In *Proceedings of the 1995 IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107. IEEE, 1995.

[128] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[129] Sami Yehia and Olivier Temam. From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 238–249. IEEE, 2004.

[130] Jonghee W Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, and Yunheung Paek. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integrated Circuits*, 17(11):1565–1578, 2009.

[131] Pan Yu and Tulika Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the 41st annual Design Automation Conference*, pages 723–728. ACM, 2004.

[132] Pan Yu and Tulika Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 69–78. ACM, 2004.

[133] Pan Yu and Tulika Mitra. Disjoint pattern enumeration for custom instructions identification. In *Proceedings of the 2007 International Conference on Field Programmable Logic and Application*, pages 273–278. IEEE, 2007.

[134] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. MIRS: modulo scheduling with integrated register spilling. In *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, pages 239–253. Springer-Verlag, 2001.