UNIVERSITY OF CALIFORNIA

Los Angeles

Compiler Support for Customizable Domain-Specific Computing

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Hui Huang

2014

ABSTRACT OF THE DISSERTATION

Compiler Support for Customizable Domain-Specific Computing

by

Hui Huang

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Jason Cong, Chair

It is known that with the support of domain-specific customizable heterogeneous architecture, energy efficiency can be significantly improved by adapting architectures to match the requirements of a given application or application domain. One of the main challenges in this emerging trend is how to efficiently take the advantage of the *heterogeneity* and *customization* features in those architectures. This research investigates developing efficient compiler support to automate the platform mapping and code transformation process.

First, considering customizable computing engines, we have investigated both tightly-coupled and loosely-coupled computing elements. In terms of tightly-coupled computing engine customization, customizable vector ISA supports are explored to better exploit data-level parallelism in the high performance applications. We identify the needs

and opportunities to explore customized vector instructions and quantify their benefits. We build an automatic compilation flow in LLVM-2.7 compiler infrastructure to efficiently identify customized vector instructions from a given set of applications. The memory alignment overhead, which is known to be critical for vector processing efficiency, has been optimized in our customized vector ISA identification flow. To support efficient vector ISA customization, we design a composable vector unit (CVU), which can be used both separately and in a chained mode, to support a large number of virtualized custom vector instructions with minimal area overhead. The results show that our approach achieves an average 27% speedup over the state-of-art vector ISA.

Second, in terms of loosely-coupled computing elements, it is known that on-chip accelerators are combined with general-purpose cores in an effort to amortize the cost of the design across many application domains. In recent days programmable accelerators (PA) are widely investigated in the design of domain-specific architectures to improve the system performance and power. Micro-architectures with a series of PAs have been explored to provide more general supports for customization. One important feature in the PA-rich systems is that the target computational kernels are compiled with a set of pre-defined PA templates and dynamically mapped to real PAs at runtime. This imposes a demanding challenge on the compiler side regarding how to generate high-quality PA mapping code. We present an efficient PA compilation flow, which is fairly scalable in mapping large computation kernels into PA-rich architectures and provides support for full pipelined execution to achieve the highest energy efficiency. A concept called *maximal* PA *candidate* is proposed to drastically reduce the number of input PA candidates in the mapping phase without influencing the overall mapping optimality. Efficient pre-selection and pruning techniques are employed to further speedup the maximal PA mapping process. Our experimental results show that for 12 computation-intensive standard benchmarks, the proposed approach achieves a significant improvement on the compilation time comparing to the state-of-art PA

compilation approaches. The average mapping quality is improved by 23.8% and 32.5% for connected PA candidates and disjoint ones, respectively.

Third, in domain-specific computing multi-level software-controlled memories have been commonly used to better utilize domain-specific knowledge of particular applications and achieve high performance/energy efficiency. At the level of L1 memory, while conventional cache works well for general workloads, some recent works explore the idea of using a hybrid cache, which can be flexibly partitioned into a traditional cache and an SCM. In the hybrid cache architecture, first-level SCM has been utilized as prefetch buffer to hide memory access latency. We quantify the impact of data reuse on SCM prefetching efficiency and propose a reuse-aware SCM prefetching (RASP) scheme, which shows 31.2% performance gain over previous work. On the other hand, SCM has also been widely used in last level on-board memory to reduce the data movements between computing cores (i.e. host processor and accelerator cores), which is usually transferred through low-bandwidth bus and known to be one of the major performance bottlenecks in modern heterogeneous systems. To efficiently manage LL-SCM, we propose a task-level-reuse-graph (TLRM) based LL-SCM data movement scheme to minimize the amount of data transfers between heterogeneous computing cores through the slow PCIe bus. With the introduction of TLRM, the data movement optimization between host and accelerator cores can be approximated using a linear programming based solution, and an average 25% reduction of host-accelerator data transfers is observed from previous work.

The dissertation of Hui Huang is approved.

Jens Palsberg

Glenn Reinman

Lieven Vandenberghe

Jason Cong, Committee Chair

University of California, Los Angeles

2014

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am deeply grateful to my advisor Professor Jason Cong for his guidance, support and vision throughout my Ph.D. study. During the past five years Jason offers persistent support, constructive suggestions, criticisms and encouragements when guide my graduate research, and has given me the valuable opportunity to turn research ideas into real products. Without his continuous guidance and support, this dissertation would not have been possible.

I would also like to express my appreciation to my doctoral committee members, Professor Jens Palsberg, Professor Glenn Reinman and Professor Lieven Vandenbergh. Their comments provides deep insight and greatly improve the quality of this dissertation.

In addition, I would like to thank all my colleagues in the UCLA VAST lab, especially Wei Jiang, Chunyue Liu, Yi Zou and Bin Liu. Their insight in research, optimistic life attitude and generous spirit largely encourage me to overcome the difficulties encountered from the beginning until the end of my Ph.D. study. I really appreciate and cherish the time working closely with these talented colleagues.

Finally, I would like to take this opportunity to thank my parents Hongmin Huang and Chunhua Zhao, my fiancé and best friend Maoqi Wang. Although the separation from them is painful, their love consistently support me to pursue the Ph.D. degree and higher goals in my life. I dedicate this dissertation to them.

This research was partially supported by the MARCO Gigascale System Research Center (GSRC) and the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127 and the NSF grant CCF-0903541.

VITA

2004-2008 B.S. Department of Computer Science

Peking University, Beijing, P.R. China

PUBLICATIONS

Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou, "A Fully Pipelined and Dynamically Composable Architecture of CGRA", International Symposium on Field-Programmable Custom Computing Machines (FCCM 2014), pp. 9-16, May 2014.

Hui Huang, Taemin Kim and Yatin Hoskote, "Edit Distance Based Instruction Merging Technique to Improve Flexibility of Custom Instructions Toward Flexible Accelerator Design", Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC 2014), Jan. 2014.

Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang and Glenn Reinman, "Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity", Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED 2013), pp. 305-310, Sep.2013.

Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu and Glenn Reinman, "Combined Static and Dynamic Optimizations for Hybrid SRAM and STT-RAM Caches", Proceedings of International Symposium on Low Power Electronics and Design (ISLPED 2012), August 2012.

Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu and Glenn Reinman, "Reconfigurable Hybrid Cache: An Energy-Efficient Last-Level Cache Design", Proceedings of Design, Automation and Test Europe (DATE 2012), March 2012.

Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Hui Huang, Bin Liu, Raghu Prabhakar and Glenn Reinman, "Compilation and Architecture Support for Custom Vector Instruction Extension", Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASPDAC 2012), pp. 652-657, Jan. 2012.

Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman and Yi Zou, "An Energy-Efficient Adaptive Hybrid Cache", in the Proceedings of International Symposium on Low Power Electronics and Design (ISLPED 2011), pp. 67-72, August 2011.

Jason Cong, Hui Huang, Chunyue Liu and Yi Zou, "A Reuse-Aware Prefetching Algorithm for Scratchpad Memory", in Proceedings of 48th Design Automation Conference (DAC 2011), pp. 960-965, June 2011.

Jason Cong, Hui Huang and Wei Jiang, "Pattern-Mining for Behavioral Synthesis", IEEE Transactions on Computer-Aided Design (TCAD 2011), Volume 30, Issue 6, pp. 939-944, June 2011.

Jason Cong, Hui Huang and Wei Jiang, "A Generalized Control-Flow-Aware Pattern Recognition Algorithm for Behavior Synthesis," in Proceedings of Design, Automation and Test Europe (DATE 2010), pp. 1255-1260, March 2010.

Chapter 1. Introduction

1.1 Customizable Heterogeneous Architecture

As discussed in [2], in order to meet ever-increasing computing needs and overcome power density limitations, the computing industry has halted simple processor frequency scaling and entered the era of parallelization, with tens to hundreds of computing cores integrated in a single processor, and hundreds to thousands of computing servers connected in a warehouse-scale data center. However, such highly parallel, general-purpose computing systems still face serious challenges in terms of performance, power, heat dissipation, space, and cost. Recently the research focus has moved from parallelization to domain-specific customization in which computing engines and interconnects can be specialized to a particular application domain to gain significant improvement in power-performance efficiency comparing to general-purpose architecture.

The motivation of domain-specific customizable computing platform is derived on three observations:

1) Each user typically has a high computing demand only in one or a few selected application domains (e.g., graphics for game developers, circuit simulation for integrated circuit design houses, financial analytics for investment banks) [2], which makes developing a customizable computing platform where computing engines and memories can be specialized to a particular application domain possible. Taking the advantage of the domain-specific knowledge, these architectures normally can gain significant improvements in power-performance efficiency comparing to a general-purpose architecture.

2) The power-performance gap between a fully customized platform, such as application-specific integrated circuit (ASIC), and a general-purpose platform can be very large. A case study of the 128-bit key AES encryption algorithm is discussed in [2]. An ASIC implementation in 0.18um CMOS achieves 3.86Gbits/second at 350mW, while the same algorithm coded in Java and executed on an embedded SPARC processor yields 450bits/second at 120mW. This difference implies a performance/power efficiency gap of approximately 3 million Gbits/seconds/Watts.

3) It is very costly to implement a fully customized ASIC architecture for each application, due to the fact that the non-recurring engineering cost of an ASIC design at the current 45nm CMOS technology is over $50M [3] and the design cycle can easily exceed a year. The large ASIC cost also imposes a strong need for an architecture platform to be efficiently customized to a wide range of applications in one domain or a set of domains, which can bridge the huge performance/power gap between ASICs and general-purpose processors with moderate hardware costs.

To realize the order-of-magnitude performance/power efficiency improvement via customization with reasonable cost, both industry and academia have been turning their attention on developing customizable heterogeneous platforms. For example, NVIDIA's Fermi GPU introduces memory customization capability, in which the shared memory space can be reconfigured into either cache or scratchpad memory with multiple possible sizes.

A more general customizable architecture is presented in [2], which includes: 1) integration of customizable cores and co-processors that will enable power-efficient performance tuned to the specific needs of an application domain; and 2) reconfigurable high-bandwidth and low-latency on- and off-chip interconnects, which can be customized to specific applications. Figure 1-1 illustrates an example of such customizable

domain-specific architecture, in which a set of fixed cores coexist with customizable cores, programmable fabric, and a set of distributed cache banks ($).

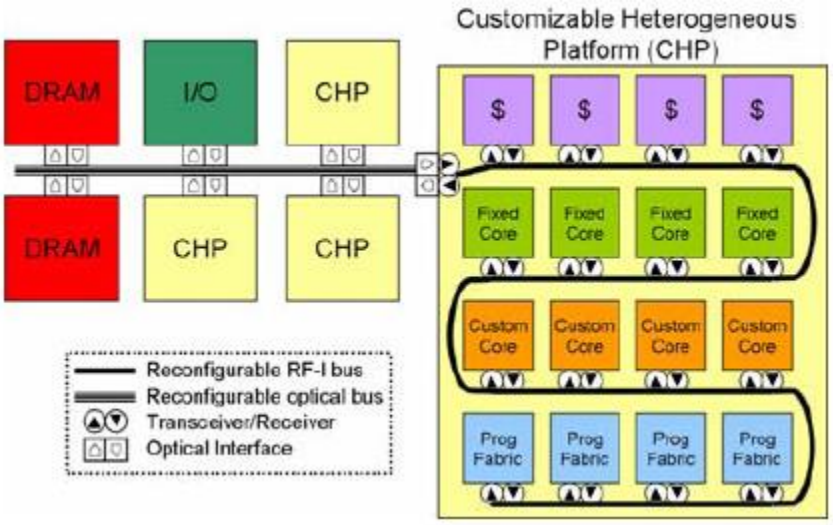


Figure 1-1. Customizable heterogeneous platform for domain-specific computing.

As we know, fixed cores can vary dramatically in their energy efficiency, computational power, and area, but have limited reconfigurability. One example of this kind of architecture is the IBM Cell, with one general-purpose PPE core and the more numerous, but simpler, SPE cores. On the other hands, customizable cores provide coarse-grained adaptation to application demand, offering a number of discrete, tunable options that can be set, with flexibility somewhere between FPGAs and fixed cores. It is possible to design cores with a rich set of tunable characteristics to enable significant performance/power efficiency, such as customizable vectorization support or computing accelerator support.

With the emergency of the customizable domain-specific platform, one of the main challenges is how to efficiently take the advantage of the heterogeneity and customization features in those architectures.

This problem can be recapped as how to map one or a set of applications to a customizable heterogeneous platform with high performance/power efficiency. Considering that manual optimization is time-consuming and also not scalable as the design space increases, it is very important to develop efficient compiler support to automate this platform mapping.

1.2 Compiler Support for Customizable Domain-Specific Computing

As we discussed, with the support of customizable domain-specific platform, performance/power efficiency can be significantly improved by adapting architectures to match the requirements of a given application or application domain. On the other hand, this also imposes challenges on the compiler size to provide high-quality mapping solution on such reconfigurable architectures. The existence of heterogeneity greatly increases the complexity of its programming model. For example, the code executed on host processors cannot be directly used on hardware accelerators. In addition, explicit data transfers are required for host-accelerator communication. In this section, we will briefly look through three customizable heterogeneous platforms, including tightly-coupled customized vector unit and loosely-coupled programmable accelerators and customizable memories.

1.2.1 Customizable Vector Unit

It has been discussed that customization can achieve significant power-performance efficiency improvement [21], and this is also the case with the vector or SIMD applications. Recently increasing attention has been given to customized vector ISA support from both academia and industry. For example, Convey system [39] supports application-specific vector instruction set, with which users are allowed to reconfigure the vector ISA to match different application features. The authors of [16] propose a

SystemC-based support for customized vector instruction. The work in [15] introduces a customized vector instruction set for multimedia applications and the work in [36] explores customized SIMD units with high-level synthesis techniques.

This trend presents new challenges to both compiler and architecture designs to provide efficient customized vector ISA support. At compiler side, the challenge will be how to efficiently identify application/domain-specific vector instruction and perform automatic customized vectorization.

A crucial step to achieve high performance in a customized vector design is the identification of frequently executed instructions. There already exist extensive work on customized *scalar* instruction exploration [12][17][38]. However, a naïve employment of the existing techniques without considering the vector features will result in inefficient customized vector instruction generation. For example, one important feature of vector processing is the existence of memory alignment. For example, AltiVec requires memory accesses to be aligned at a 16-byte boundary and it cannot handle unaligned vector loads and stores; In AVX mis-aligned memory accesses are supported with a large performance penalty.

We introduce an automatic LLVM-based compilation flow to extract customized vector instructions from one or a set of applications. Pattern recognition approaches have been used here to identify frequently appeared customized vector instruction candidates and an optimal alignment insertion scheme has been developed to reduce the memory alignment overhead. This flow is tested on the composable vector processing units (CVUs), which can be chained together to create customized vector instructions. This design allows programmable customized vector extensions and can achieve up to 52% speedup over standard vector ISA and 14.6X area gain over the dedicated ASIC-based design.

1.2.2 Customizable Computing Accelerator

Programmable accelerator (*PA*) has been proposed to enable varying degrees of customization together with general-purpose cores [51] [54] [55] [56] [57]. In a standard PA architecture, a programmable accelerator template is implemented inside each PA unit to support a selected set of computation tasks with reasonable hardware design costs. The entire pre-defined PA template may support a relatively complicated computation task, while it can be reconfigured dynamically to perform a set of simpler but more general sub-tasks. Therefore, each accelerator unit in a PA-rich system can be customized to computation tasks with different granularity, which enables efficient switching among varying degrees of customization at runtime.

With more flexible customization support, the PA-rich design has been raised as a promising solution to improve the system performance-power efficiency. However, this design trend imposes a demanding challenge on the compiler side – how to generate high-quality PA mapping code which can efficiently utilize the programmable execution units existing in a PA-rich architecture.

In general, the PA compilation flow can be divided into two phases - PA candidate identification and PA template mapping. Given the data flow graph (DFG) of application kernels, the PA candidate identification phase extracts all the data flow subgraphs which are executable on the PA units. To decide whether a subgraph is executable on PA units, subgraph isomorphism checking is performed between the subgraph and the PA templates. After that, the identified candidates will be fed into the mapping phase, in which a subset of candidates will be mapped to PA templates to accelerate the target kernels.

As discussed in [60], since the PA candidates identification and PA mapping problems are both difficult to solve, scalability has been considered as a main problem in the existing PA compilation flows. Considering that the number of PA candidates grows exponentially with the size of input DFG and PA template, the mapping problem may become

intractable for large DFG blocks. When disjoint PA candidates are considered, the mapping problem size is even larger after including all the legal combinations of connected PA candidates.

The other challenge comes with the pipelined PA execution. In a fully pipelined PA design, input data comes in at every clock cycles, buffers or dummy PAs [99] need to be inserted to guarantee the correctness of pipeline behavior. This serves as a new demanding resource requirement, which is not considered in previous work.

Targeting scalable PA compilation of fully pipelined execution, we build an automatic PA compilation flow, which supports both connected and disjoint PA candidates. Delay units are inserted in the PA mapping graph to balance the path delays in a pipelined execution. Comparing to the scalable PA compilation approaches proposed in [60] and [55], our approach achieves a significant reduction on the overall compilation time. The corresponding mapping quality has been improved by 23.8% and 32.5% on average for mapping the connected-only and disjoint PA candidates, respectively. We also investigate the impact of a given throughput target on resource usage in accelerator pipelines. Here resource usage includes not only PAs, but also delay units required to balance path delay. An optimal PA mapping algorithm is used to efficiently map on-chip accelerator resources to a pipelined execution. Compared to the PA compilation approaches proposed in [60], our approach achieves a significant reduction on mapping size and up to 33.8% improvement on system performance.

1.2.3 Customizable Memory

Modern high performance processors are known to be abundant in processing elements, e.g. general-purpose cores or customized hardware accelerators (FPGA, GPU, etc.). Memory accesses become an increasing performance bottleneck, preventing applications from fully exploiting the computing power. To alleviate the memory bottleneck, communication optimizations, including memory latency reduction and efficient

bandwidth utilization, turns out to be crucial for system performance and energy efficiency. Traditional hardware-controlled cache suffers from 'blind' data movement decisions, which are made independent with program behavior. As an alternative, software-controlled memories (SCM) have been employed either as an independent storage unit or sitting together with D-cache/I-cache to effectively enhance performance and power. This trend has already been reflected on real designs, e.g., NVIDIA's latest Fermi GPU has software controlled scratchpad memories (SPM) called "shared memory" which can be partitioned into cache and SPM at configuration points 1:3 or 3:1, with SPM and L1 cache sitting on top of L2 cache. Similarly, the local store in IBM's Cell broadband can be managed as a combination of direct buffers to store access with regular patterns and software-controlled cache as a fall-back solution [73].

Comparing to single-level SCM, multi-level SCM designs provides better tradeoff the access speed difference between different memory levels, therefore has been widely explored. In typical embedded processors, the L1 SCM normally consists with fast SRAM memories (e.g. scratchpad memories) and last level (LL) SCM can be either SRAM or DRAM (e.g. FPGA's off-chip memory and GPU's global memory). The optimization target of SCMs sitting at different memory level also differs.

L1 SCM normally is a small piece of fast memory, which sits closest to the computing core and is responsible to feed data in time. Targeting low memory access latency, L1 SCM has been utilized as prefetch buffers in embedded systems and parallel architectures to hide memory access latency [1]. This is motivated by the fact that conventional cache prefetching suffers from the problem that the data evicted from cache by the newly prefetched data is still "alive," i.e., will be accessed frequently in the near future. An extreme case is that N prefetched elements are mapped to the same set in a direct-mapping cache. Therefore, only the last element will be kept in the cache after prefetching, while the previous $N-1$ data transfers are useless with additional energy

overhead. On the other hand, SCM-based prefetching can make a “smart” eviction decision, thereby avoiding such cache prefetch inefficiency.

On the other hand, shared last level SCM has been widely utilized in heterogeneous parallel architectures to tradeoff the low bandwidth from main memory. For example, the performance of PCIe bus connecting host memory is ~10GB/s, which turns out to be an important bottleneck of modern heterogeneous systems. Accordingly, how to efficiently reuse the data stored in LL-SCM becomes one of the major compiler challenges in a heterogeneous system where workloads distributed on different computing cores. Compared to hardware-controlled memories, the introduction of SCM as a last level buffer offers optimization potential on cross-core data transfers by taking the advantage of the knowledge of target applications.

To fully utilize the multi-level SCM memory space, we have investigated prefetching and reuse capability for L1 and LL SCM, respectively. We propose a reuse-aware SCM prefetching scheme, called RASP, to hide memory access latency and minimize the number of data transfers from lower-level memory; To efficiently manage LL-SCM, we propose a task-level-reuse-graph based LL-SCM data movement scheme to minimize the amount of data transfers between heterogeneous computing cores through the slow PCIe bus. An average 25% reduction of host-accelerator data transfers is observed from previous work.

Chapter 2. Compiler Support for Customizable Vector Instruction Extension

Vectorization has been commonly employed in the high performance computing domain to exploit data-level parallelism in those applications. In this chapter we analyze the needs and opportunities to explore customized vector instructions and quantify their benefits. We build an automatic compilation flow in LLVM-2.7 compiler infrastructure to efficiently identify customized vector instructions from a given set of applications. The memory alignment overhead, which is known to be critical for vector processing efficiency, has been optimized in our customized vector ISA exploration flow. This flow is tested on the composable vector units, which can be used separately or in a chained mode to support a large number of (virtual) customized vector instruction units with minimal area overhead. The results show that our approach achieves an average 27% speedup over the state-of-art vector ISA. We also observe a large area (around 11.6X) gain over the dedicated ASIC-based design.

2.1 Introduction

SIMD vector processors are very effective in executing programs with extensive data-level parallelism, such as multimedia processing, graphics and scientific computing. In recent years, vector extension has become one of the most common additions to both general purpose microprocessors and super computers, due to the growing demands on high-performance computing. There are several state-of-art vector ISAs in the market, such as Intel's AVX [40], Motorola/IBM's AltiVec [23].

It has been recognized that customization can achieve significant performance improvement [15] and this is also the case with the vector or SIMD applications.

Recently increasing attention has been paid to customized vector ISA support. The authors of [12] propose SystemC-based support for customized vector ISA. The work in [11] introduces a customized vector instruction set for multimedia applications. The work in [23] designs customized SIMD units with high-level synthesis techniques. The newly developed Convey system [26] provides supports for application-specific vector instruction sets, with which users are allowed to reconfigure the vector ISAs to match different application domains. This trend presents new challenges to both compiler and architecture design to provide efficient customized vector ISA support with small hardware cost.

At compiler side, the main challenge is how to efficiently identify application/domain-specific vector instructions and perform automatic customized vectorization. A crucial step to achieve high performance in a customized vector design is the identification of frequently executed vector instructions. There already exist extensive work on customized *scalar* instruction exploration (e.g. [8][13]. However, a naïve employment of the existing techniques on the input program without considering the vector features will result in inefficient customized vector instruction generation. One important feature of vector processing is the existence of memory alignment problem raised by the vector architecture [18]. For example, AltiVec requires memory accesses to be aligned at a 16-byte boundary and it cannot handle unaligned vector loads and stores; In AVX mis-aligned memory accesses are supported with a large performance penalty.

Here a mis-aligned memory reference means that the address of the data is not a multiple of the vector register size [26]. Let's look at the example in Figure 2-1 (without loss of generality, we assume for array reference $A[i_1][i_2]\ldots[i_N]$, the starting address of each array dimension, namely $A[0][0]\ldots[0]$, $A[i_1][0]\ldots[0]$, $A[i_1][i_2][0]\ldots[0]$, …, $A[i_1][i_2]..[i_{N-1}][0]$, has been aligned to memory boundary).

As shown in Figure 2-1(b), arrays A, B and C are loaded into vector register *VR1-3* in a mis-aligned manner. If we directly perform a vector add on the 3 vector registers, it will generate incorrect results, where A[i] = B[i] + C[i]; to resolve it, vector register *VR1* and *VR2* are shifted to left by 3 elements and 2 elements, respectively in Figure 2-1(c).

Therefore, in order to ensure the functionality correctness, the "shifted" alignment of the input nodes needs to match that of the output node. This alignment constraint imposes challenges on automatic vectorization process due to its sizable impact on the power-performance efficiency in vector processing.

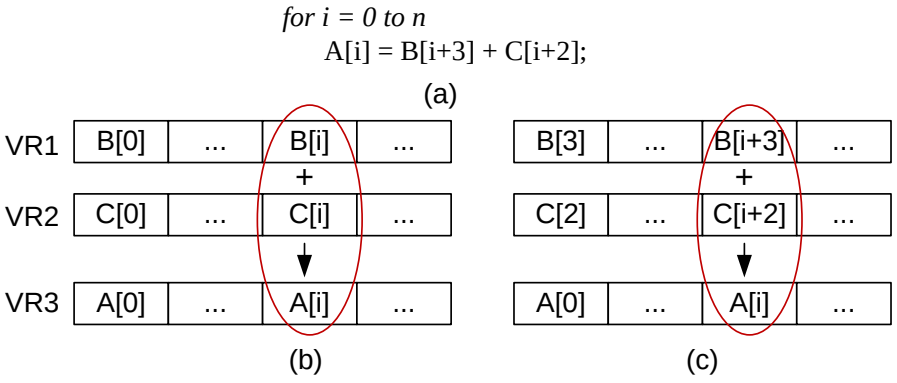


Figure 2-1. (a) One example loop (b) Mis-aligned vector addition (c) Aligned vector addition.

In the customized vector ISA exploration phase, if the memory alignment issue has not been resolved properly, it may result in undesired overhead on performance. Let's consider a vectorizable loop shown below (without loss of generality. Here we assume A[0], B[0], etc. are aligned to memory boundary).

for i = 0 to n

$A[i] \mathrel{+}= B[i+1]*C[i];$

The scalar customized instruction candidates inside this loop only contain one *multiply-add* (MAC) operation. While for vector exploration, since additional alignment instructions are required to resolve the unaligned array reference $B[i+1]$, both MAC and aligned MAC should be considered as customized vector instruction candidates, as shown in Figure 2-1(b). If we simply replace the sequential loop with unaligned vector MAC operations, it may result in either incorrect execution or pay considerable performance penalty.

Earlier implementations of vector processor [7] [18]re all based on non-customized vector instructions. The vector instructions in VIRAM are designed to vectorize

embedded system applications by adding support for narrower data-type and different styles of permutation. VESPA [26] is a flexible FPGA-based vector engine. However it only supports integer vector operations.

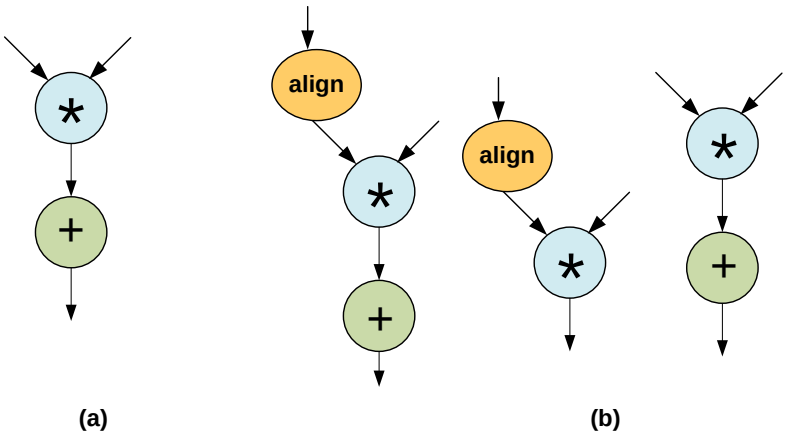


Figure 2-2. (a) Customized scalar instruction candidates (b) Customized vector instruction candidates.

In this chapter we introduce an automatic compilation flow to perform alignment-efficient customized vector instruction identification, and the architectural support for area-efficient customized vector operations.

(1) We identify the existing opportunities to derive customized vector instructions. A *boundary-extension* technique and an *operation-based* vectorizability checking technique are developed to fully investigate customized vector instruction exploration space.

(2) We propose an LLVM-based compilation flow to extract customized vector instructions from one or a set of applications. Pattern-based approaches have been used here to identify beneficial customized instruction candidate

(3) We propose an optimal memory alignment scheme that minimizes the total *shifting distance* to generate alignment-efficient vector patterns.

This flow is tested on the composable vector processing units (CVUs), which can be chained together to create customized vector instructions. This design allows programmable customized vector extensions and can achieve up to 52% speedup over standard vector ISA and 14.6X area gain over the dedicated ASIC-based design.

2.2 Motivational Example

In this section, we illustrate the existence of application-specific or domain-specific customized vector patterns with real-life applications.

Let's first consider one computation kernel in *jacobi rician-denoise* [41] which is a double precision floating point application in the medical imaging domain. Figure 2-3 shows the kernel loop in this application, which performs five-point stencil computation on a 2D image. Seven arrays are involved in the computation kernel as inputs and the loop body can be vectorized without violating data dependencies. (here we only consider vectorization through the innermost loop)

```
for m = 1 to M - 1
    for n = 1 to N - 1
        u[m][n] = (ulast[m][n] + DT * (ug[m][n+1] +
                                       ug[m][n-1]  +
                                       ug[m+1][n] +
                                       ug[m-1][n]  +
                                       GM*f[m][n]))
                  / (c[m][n] + DT * (g[m][n+1] +
                                     g[m][n-1]  +
                                     g[m+1][n] +
                                     g[m-1][n]  +
                                     r[m][n]))
```

Figure 2-3. . One kernel loop in *jacobi rician-denoise.*

Figure 2-4 shows the corresponding data flow graph for the vectorizable loop, in which each node represents a vector instruction, such as vector-add or vector-multiply. Each vector input is denoted by the array name followed by alignment value normalized to output $u[m][n]$. For example, the relevant alignment offset between $u[m][n]$ and

ug[*m*][*n*-1] is 1, therefore "*ug:1*"has been used to represent vector input *ug*[*m*][*n*-1].

From Figure 2-4, we can see the two branches of *div* operation are similar to each other in terms of both operation counts and data path. The left branch contains nine operation nodes and the right one contains eight nodes – only differ from each other by one *mul* operation, which exposes the opportunity to extract repeatedly executed customized vector instructions, sizing from one operation to eight operations. Two vector pattern candidates with occurrence equaling two have been highlighted in Figure 2-4.

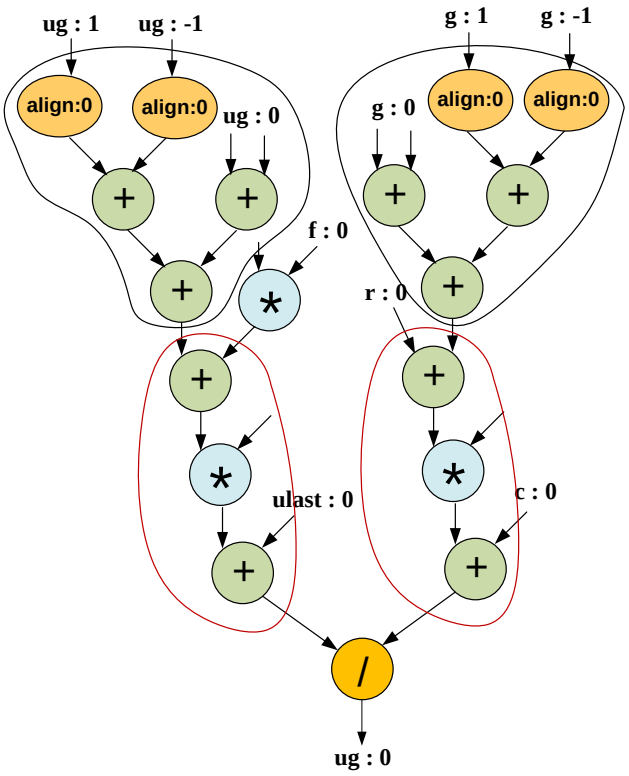
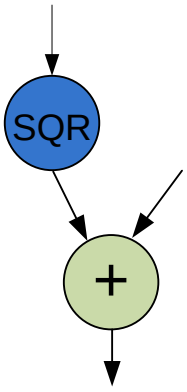


Figure 2-4. Data flow graph of kernel loop in *jacobi rician-denoise*.

In this example, note that since *ug*[*m*][*n*+1] and *ug*[*m*][*n*-1] both serve as inputs to the same vector add operation and they are mis-aligned array references, two alignment nodes are inserted to match them to the alignment offset of *ug*[*m*][*n*], namely 0. The same scenario also applies to array reference *g*[*m*][*n*-1] and *g*[*m*][*n*+1]. Our optimization on

alignment node insertion will be discussed in Section 2.3.

In addition to application-specific customized vector patterns, there also exist common vector patterns inside a specific application domain due to the similarity in computing models or algorithms, as shown in Figure 2-5. Figure 2-5(a) shows a double precision vectorizable accumulation of SQR operation, e.g., *a*[*i*]***a*[*i*]. The kernel code pieces in Figure 2-5(b) and (c) are extracted from *rician-denoise* and *level-set segmentation* [41] in the medical imaging domain. The vectorizable SQR-accumulation operation appears in both applications (4 times in *rician-denoise* and 3 times in *segmentation*), thus they can benefit from the same customized vector ISA extension.



(a)

```
for m = 1 to M-1
    for n = 1 to N-1
        g[m][n] = 1.0/sqrt( EPSILON +
           SQR(u[m][n] - u[m][n+1]) +
           SQR(u[m][n] - u[m][n-1])  +
           SQR(u[m][n] - u[m+1][n]) +
           SQR(u[m][n] - u[m-1][n]) );
```

(b)

```
for m = 1 to M-1
   Grad[n] = coeff*( SQR(Dx_0[m]) +
                     SQR(Dy_0[m]) +
                     SQR(Dz_0[m]) );
```

(c)

Figure 2-5. (a) Data flow graph of SQR-accumulate (b) Kernel code piece in *jacobi rician-denoise* (c) Kernel code piece in *level-set segmentation*.

2.3 Customized Vectorization Flow

Figure 2-6 shows the components of our customized vector instruction identification framework. This framework is implemented in LLVM-2.7 compiler infrastructure [42] with Omega Library [43] for dependency analysis. The flow is invoked as a back-end pass on the optimized LLVM intermediate representation (IR) code. As discussed in [9], automatic vectorization performed at source-level is usually decoupled from standard back-end optimization, comparing to lower-level IR, which is closer to the machine-level

code and can take the advantage of operating on optimized code. In our case, the optimized LLVM IR is used as top-level input in the customized vector instruction identification framework.

Figure 2-7 shows the kernel loop in *rician-denoise* application (to better illustrate each step in the framework, *gauss-seidel* implementation [41] is used here, which contains loop-carried true dependency).

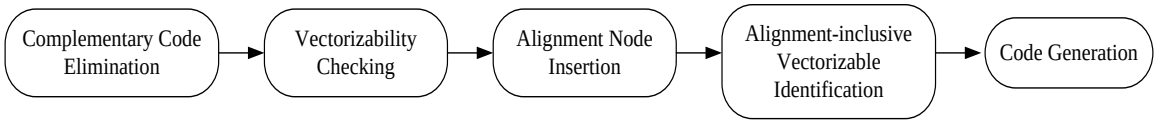


Figure 2-6. Customized vector instruction identification flow.

The data flow graph of the corresponding LLVM intermediate representation is shown in Figure 2-8. Each node in the data flow graph is labeled with the operation it performs and each edge represents the data flow dependency between two nodes.

```
for m = 1 to M - 1
    for n = 1 to N - 1
        u[m][n] = (u[m][n] + DT * (u[m][n+1] +
                                   u[m][n-1]  +
                                   u[m+1][n] +
                                   u[m-1][n]  +
                                   GM*f[m][n]))
                / (c[m][n] + DT * (g[m][n+1] +
                                   g[m][n-1]  +
                                   g[m+1][n] +
                                   g[m-1][n]  +
                                   r[m][n]))
```

Figure 2-7. Kernel code of *gauss-seidel rician-denoise*.

In this example, the node phi generates the value of loop induction variable n for the inner loop. If the value is obtained from the outer loop body, n equals 1; otherwise, n equals its current value plus one, namely the output of the add node under phi. To

calculate the address at the second array dimension for references u(g)[m][n-1] and u(g)[m][n+1], another two add nodes below phi accept the output of phi node, namely the value of n, and perform the corresponding array subscript calculation. The calculated array indices are sent to the getelementptr (gep) node to generate the address for the corresponding array element, which is followed by a load (ld) operation to access memory.

Algorithm 2-1. Vectorizable Code Region Extraction

1. G : LLVM-IR-based data flow graph of a given loop nest
2. BI : a set of boundary array / scalar input nodes in G
3. BO : a set of boundary output nodes in G
4.
5. for each node v in G,
6. if there exist a path from v to one node in BO and a path from one node in BI to v,
7. add v to V;
8. for each node v in V sorted in topological order,
9. if for any input t to v, t BI ∪ V_vec,
10. add v to V_vec;
11. else
12. if there exists an edge t'→v such that t' BI ∪ V_vec,
13. add v to BI;

2.3.1 Vectorizable Code Region Extraction

To extract the customized vector patterns which perform real computations, we need to remove the complementary nodes existing in the original LLVM IR, such as the loop invariant and branch instructions. Note that this cannot be achieved by simply removing operations in those classes and then performing customized vector instruction exploration on the reduced data flow graph. For example, in Figure 2-8 the three *add* operations below the *phi* node perform address calculation instead of real computation, thus should not be explored as customized vector instruction candidates. In our flow, we propose a *boundary-node-directed* vectorizable code region extraction approach.

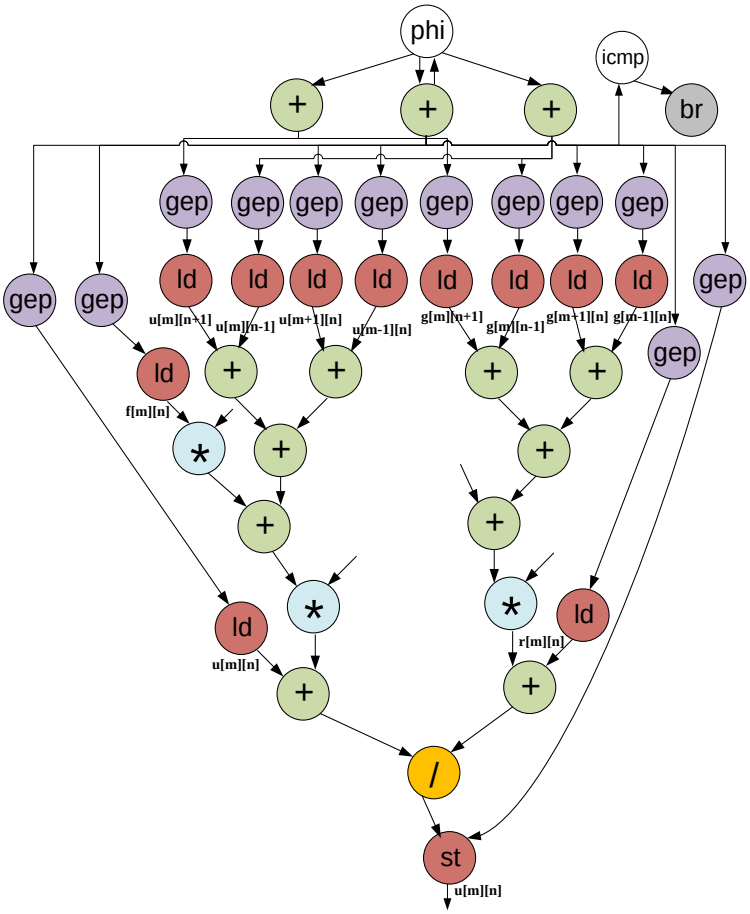


Figure 2-8. Data flow graph of gauss-seidel rician-denoise.

Definition 2-1. *The boundary nodes of the vectorizable code region in loop L are defined to be the legal inputs and outputs to a vector instruction, including memory load/store operations to continuous memory space (here we only consider array subscript as a linear combination of loop induction variable and constant), constants and scalar variables with value fixed inside L.*

In Algorithm 2-1, any node that locates outside the subgraph enclosed by the boundary nodes will be removed, as shown in Figure 2-9(a).

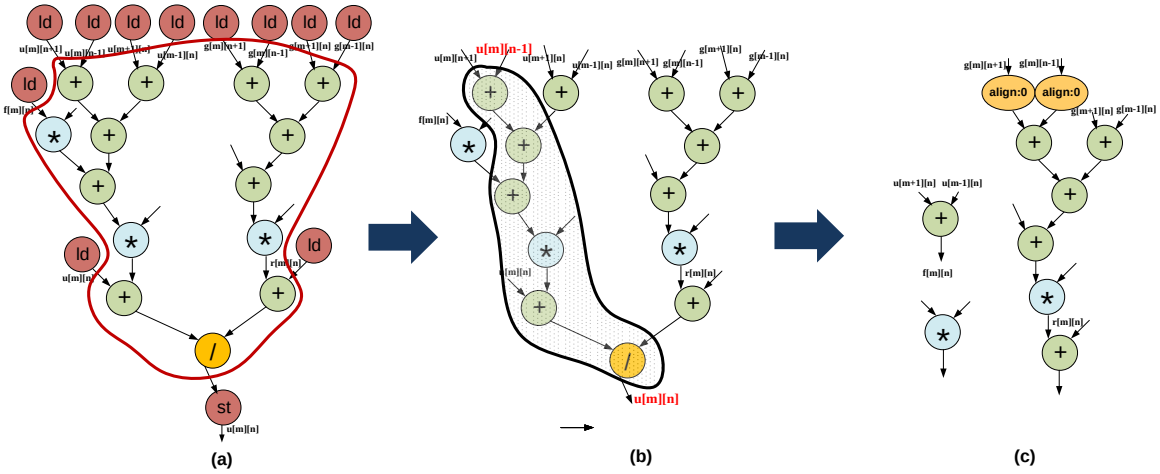


Figure 2-9. Complementary code elimination, vectorizability checking and alignment node insertion in the kernel code of *gauss-seidel rician-denoise*.

The nodes in set *V* but not *V_vec* are called "prohibited nodes", Those nodes cannot be vectorized due to illegal inputs. For example, the third addition in Figure 2-10(a) accepts random array access as input, which makes it in-vectorizable. However, we can introduce temporary arrays and apply loop distribution techniques to make it become a new "*boundary node*" (lines 11-12). In this case the operations following the in-vectorizable node will not be prohibited from vectorization, which helps to enlarge the customized vector pattern exploration space. As shown in Figure 2-10(b), *L add* operations in second inner loop can be performed in parallel. (Assume *L* equals the vector register length) This technique is referred as *boundary extension* in the remaining part of this chapter.

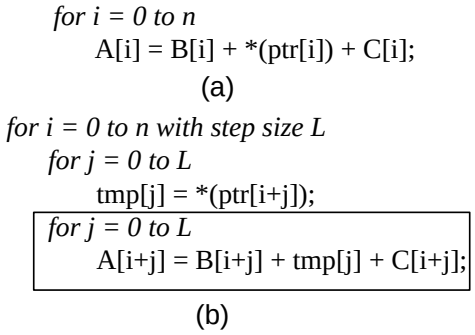


Figure 2-10. (a) Original code (b) Transformed code.

2.3.2 Operation-based Vectorizability Checking

Theorem 2-1. [29] *A statement contained in at least one loop can be vectorized if the statement is not included in any cycle of dependences.*

In the traditional loop vectorization techniques [29], one statement contains a set of operations and a corresponding memory store. For example, the statement in Figure 2-7 contains 13 operations. In those techniques, each statement is treated as the basic vectorization unit. However, this may lead to the loss of existing vectorization opportunities. For example, the loop body in Figure 2-7 contains one statement with self-dependency cycle. Based on Theorem 2-1, if the traditional approach is applied, all of the 13 operations inside that statement will be treated as in-vectorizable nodes. However, since the loop-carried true dependency only exists between $u[m][n]$ and $u[m][n\text{-}1]$, the nodes operated on the other array references can be executed in parallel without violating the existing dependency. Those vectorization opportunities will be missing in the statement-based approach.

To fully investigate the customized vector pattern space, we perform an *operation-based* dependency checking after applying the conventional statement-based approach. In the proposed operation-based dependency checking, each operation node in the vectorizable code region is treated as the basic vectorization unit to allow partial vectorization inside

one statement.

Algorithm 2-2. Vectorizability Checking

---

1. V_vec : vectorizable nodes obtained from Algorithm 2-1.
2. S : a set of statements inside the given loop nest
3.
4. Perform statement-based vectorization algorithm [28] on *S*,
   add the nodes adjacent to the violated dependency edges to *N*;
5. **for** each node *n* in *N*,
6. remove *n* from *V_vec* and add it to *N*;
7. **for** each node *v* in *V_vec* sorted in topological order,
8. **if** there exists an edge $v \rightarrow n$ or $n \rightarrow v$ such that $n \in N$,
9. remove *v* from *V_vec* and add it to *N*;

In Algorithm 2-2, the statement-based vectorization algorithm is first applied to differentiate vectorizable and in-vectorizable statements. For the in-vectorizable statements, further dependency checking is performed in an operation-based manner. Assume the statement is not vectorizable due to a set of violated dependency edges, lines 8-9 separate the operation nodes carrying the violated dependencies from those not. As shown in Figure 2-9(b), the violated dependency is between *u*[*m*][*n*-1] and *u*[*m*][*n*], the *add* operation associated with *u*[*m*][*n*-1] has been excluded from the vectorizable code region after dependency checking, as well as the downstream nodes reachable from *u*[*m*][*n*-1]. The statement can be partially vectorized by executing the unshaded nodes in parallel. By applying the operation-based vectorization check, the exploration space for customized vector pattern is further enlarged and exposes more opportunity to extract

beneficial instruction candidates. Comparing to statement-based approach, the extra complexity overhead of Algorithm 2-2 is *O(|V_vec|)* where |*V_vec*| is the number of nodes in the vectorizable code region.

2.3.3 Vectorizable Data Flow Graph Expansion

In this section we describe techniques to insert *alignment instructions* explicitly into the original data flow graph in presence of mis-alignment. The *alignment offset* of a memory access is defined as the byte-offset to the memory boundary of the array elements to be accessed at the first iteration in a normalized loop. For example, in the loop in Figure 2-1(a), the alignment offsets for accessing array *X* and *Y* are 3 and 2, respectively.

An alignment instruction is one that combines results of two neighboring vector load instructions and logically performs a shift on the vector registers. Note that there are different ways to insert alignment instructions. In [24] several heuristics are described, including zero-shift, eager-shift, lazy-shift and dominant-shift. A typical strategy is shown in Figure 2-11(a).

```
B'[i] = B[i+3] ;   //shift by 3
C'[i] = C[i+2] ;   //shift by 2
A[i] = B'[i] + C'[i];
```

(a)

```
B'[i+2] = B[i+3] ;   //shift by1
A'[i+2] = B'[i+2] + C[i+2];
A[i] = A'[i+2] ;      //shift by 2
```

(b)

Figure 2-11. (a) Shifting scheme 1 (b) Shifting scheme 2.

Here the total shift distance is 3+2=5. Yet, when each alignment instruction shifts a vector register by only one, the solution is suboptimal. A better solution is in Figure 2-11(b), where the total shifting distance is 1+2=3.

In this chapter, we introduce an optimal shift scheme, with the goal of minimizing the total shifting distance; as in our architecture, we shift by one for each alignment instruction to reduce hardware cost. Our method is based on a mathematical

programming formulation, and is able to obtain the optimal solution efficiently by taking advantage of the total unimodularity of the constraint matrix.

Without loss of generality, we consider a data flow graph where each node is either a memory load/store or an arithmetic instruction that takes two inputs produces one output. For each arithmetic instruction I, a pair of integers (β_1, β_2) is introduced to model its alignment property, where β_1 / β_2 is the relative offset of the first/second input operand with regard to the output of I. In the previous example, the addition operation has an alignment vector (3, 2). For each arithmetic instruction I, we associate a label on each of its port o_I^1, o_I^2, o_I^3 to indicate the actual alignment offset on its input operand o_I^1, o_I^2 or its output o_I^3. To ensure correctness after alignment, we need to make sure

$$o_I^1 - o_I^3 = \beta_I^1 \qquad (1)$$

$$o_I^2 - o_I^3 = \beta_I^2 \qquad (2)$$

This constraint means that the relative alignment offset between each input operand and the output value is fixed, and that they can be changed simultaneously when alignment instructions are inserted properly. In the above example, we have $o_+^3 = o_+^1 - 3 = o_+^2 - 2$.

For a memory access instruction m, its alignment offset is always zero, as we only do load/store in aligned fashion. We have constraint

$$o_m = 0 \qquad (3)$$

When the output of an instruction s is used by another instruction t as its first (or any other) operand, alignment instructions may be needed to shift the result of s. Let d_s denote the shifting distance, and we have

$$o_s^3 - o_t^1 \le d_s \qquad (4)$$

$$o_t^1 - o_s^3 \le d_s \qquad (5)$$

he above constraints means that we need to shift at least $|o_s^3 - o_t^1|$.

When instruction s is used by multiple instructions $\{t_1, t_2, \ldots, t_m\}$, the total shifting distance is at least $o_{ti}^1 - o_{tj}^1$. This is because any alignment offset between o_{ti}^1 and o_{tj}^1 will be covered during the alignment. We have

$$o_{ti}^1 - o_{tj}^1 \leq d_s, \text{ for all i,j} \qquad (6)$$

Combining the above constraints, we have the following formulation as an integer-linear programming.

minimize $\sum d_s$

subject to

$o_I^1 - o_I^3 = \beta_I^1$ for all instruction I

$o_I^2 - o_I^3 = \beta_I^2$ for all instruction I

$o_m = 0$ for all memory access m

$o_s^3 - o_t^1 \leq d_s$ for all s used by t

$o_t^1 - o_s^3 \leq d_s$ for all s used by t

$o_{ti}^1 - o_{tj}^1 \leq d_s$ for all s used by t_1 and t_2

All variables are integers (7)

The above formulation tries to minimize the total shifting distance for all values. For the aforementioned example, denote the instructions as 1(load x), 2(load y), 3(the addition), 4(store to z), the formulation is

minimize $d_1 + d_2 + d_3$

subject to $o_1 = 0$

$o_2 = 0$

$o_4 = 0$

$o_3^1 - o_3^3 = 3$

$o_3^2 - o_3^3 = 2$

$o_1 - o_3^1 \leq d_1$

$o_3^1 - o_1 \leq d_1$

$o_2 - o_3^2 \leq d_2$

$o_3^2 - o_2 \leq d_2$

$o_3^3 - o_4 \leq d_3$

$o_4 - o_3^3 \leq d_3$

All variables are integers

Integer-linear programming formulations are known as a general-purpose tool for modeling combinational optimization problems, including those notoriously hard ones. An typical ILP solver, even if equipped with sophisticated algorithms (like cutting planes, dual decomposition), will still rely on enumerative approaches such as branch-and-bound, and thus still runs in exponential time in many practical cases. Therefore, the problems that can be solved by ILP are limited in practice. Fortunately, for the above formulation, we can show that the integrality constraints are unnecessary. That is, the problem formulation can be solved as a linear programming problem without the integrality constraints, while still guaranteeing integral solutions. This is because we can take advantage of the special structure in the constraint matrix for this specific problem. In the following, we show mathematically why the above formulation can be solved optimally in polynomial time.

Definition 2-2. *(Total unimodularity). A matrix A is totally unimodular if every square submatrix of A has a determinant either 0, 1 or -1.*

Clearly, a totally unimodular matrix can only have entries 0, 1, or -1. Total unimodularity plays an important role in combinatorial optimization, due to the result in Lemma 1.

Lemma 2-1 [28]. *If A is totally unimodular and b is a vector of integers, every extreme point of polyhedron {x: Ax ≤ b} is integral.*

Lemma 2-1 implies that an integer linear programming problem can be solved without the integrality constraints when its constraint matrix is totally unimodular and the right-hand side is integral.

Many previous work have taken advantage of the total unimodularity of the constraint matrix in a number of applications [28]. In fact, our formulation leads to a constraint matrix that has exactly the same structure as that of [19]. Thus the following theorem can be derived:

Theorem 2-2. *The problem is tractable and can be solved in polynomial time with linear programming algorithm [19]***.**

After the optimal alignment scheme is derived from the unimodularity of this formuation, the original data flow graph will be expanded to include the corresponding alignment nodes, as shown in Figure 2-9(c).

2.3.4 Pattern-Based Customized Vector Instruction Identification

This section presents the pattern-based approach to efficiently identify the vector pattern candidates from the data flow graph expanded by inserting optimized alignment nodes.

The pattern recognition approach we use is based on [18] work which is very scalable in benefit of subgraph enumeration and similarity checking technique. A breadth-first (HPR) search strategy is adopted in our flow to discover frequent pattern candidates in practice.

HPR, as suggested by its name, is a complete search algorithm which discovers patterns with a breath-first-search approach. At step $k + 1$, all the convex patterns with k nodes are

extended by one neighbor node using the proposed subgraph enumeration techniques. After a new subgraph is generated, it is compared to the existing patterns to perform graph isomorphism checking. If a subgraph is isomorphic to an existing pattern *P*, we call it a *pattern instance* of *P*. A *characteristic-vector* based filtering scheme is adopted to reduce the number of the graph isomorphism checking. The characteristic vector captures important properties of the original subgraph such that if the signature of a subgraph is significantly different than the signature of a given pattern, this subgraph is not needed for matching with the pattern, which avoids the graph isomorphism checking.

By applying the pattern-based approaches to the expanded data flow graph from Section 2.2, we can extract all the frequently executed vector pattern candidates with associated alignment information.

To measure the gain of a given customized vector pattern, we have used a model to estimate the energy-performance-product improvement in our flow. To simplify the model, we do not consider the boundary cases in a loop.

$$gain(p) = \#inst - (|P_{critical}| / L) + (1 / L \cdot \#inst - \alpha \cdot dist_i)_g$$

Here #inst represents the estimated execution time with scalar instruction support; L is the length of vector register, namely the higher level of data parallelism supported by vector architecture. The length of critical path ($|P_{critical}|$) divided by the data parallelism factor L is used to estimate the vector execution time. Considering that with the complex customized vector patterns, the instruction counts can be reduced accordingly, which corresponds to less energy consumption on instruction decoding logic, as well as the potential reduction in the L1 instruction cache misses. Another benefit comes from the reduced number of branch prediction operation in the transformed vector code. The estimated instruction count ratio between customized vector instruction and scalar instruction equals $1/L \cdot \#inst$, which is used to measure the difference in power consumption. Note that additional alignment instructions are introduced in the vectorized

code, which should also be taken into consideration in the energy-performance model. Here α is the scaled energy cost by shifting one element and $dist_{align}$ is the overall shifting distance introduced by the alignment instructions in the customized vector pattern.

2.4 Experiment Results

2.4.1 Evaluation Methodology

We have considered nine applications from widely known standard benchmarks suite like Parsec [10] (*streamcluster* and *swaptions*), Rodinia [13][14] (*cutcp*, *mri-q* and *mri-gridding*) and four applications from the medical imaging domain [41] (*denoise*, *deblur*, *registration* and *segmentation*).

We evaluate the proposed customized vector instruction extraction flow by running full system simulations on each benchmark. The overall simulator framework is implemented upon Simics [31] and the GEMS toolset [32] in the single core configuration. Normal vector engine support has been added to this framework. Figure 2-12 shows our architectural support for composable vector units (CVUs). It consists of a series of CVUs, a programmable crossbar and a sequencer. They are all tightly-coupled connected to the core. The composable vector units are connected together through the programmable crossbar. The inputs to the programmable crossbar are from the outputs of CVUs and the core's register file. The outputs of the crossbar are connected to CVUs. The sequencer, which is programmed by the core, is responsible for reprograming the crossbar in every scheduling step. In this way different connection patterns between CVUs can be supported. Internally the crossbar is a series of multiplexers.

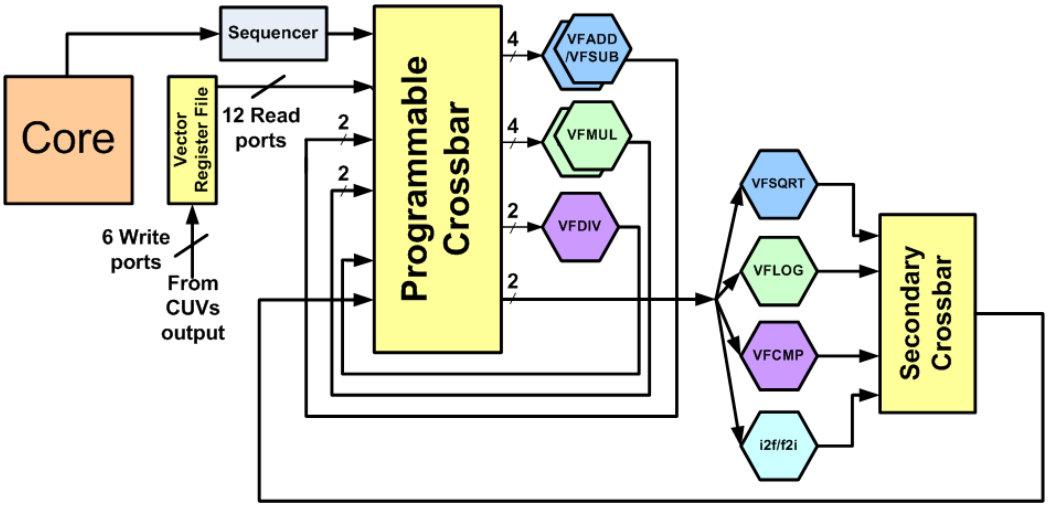


Figure 2-12. The CVU architecture.

2.4.2 Pattern Recognition Results

Table 3-1 shows the customized vector pattern recognition results for the nine benchmark kernels. At each row, columns 2-6 represents lines of kernel code, the number of pattern found, the number of pattern instances and runtime, respectively. For example, test bench *streamcluster*, the code in its kernel contains 96 lines of C code, and the 92 vector patterns are found with 240 pattern instances. The overall runtime is less than one second. From Table 3-1 we can see, the average number of instances for each pattern inside the kernel is around 4. The repeated occurrence of the same vector pattern in program kernels exposes the opportunity of program execution speedup by providing the customized vector support for the corresponding pattern.

The last column in Table 3-1 shows the area synthesis result for patterns in each benchmark, in total the area equals 5574062 um^2.

Table 2-1. Pattern recognition results on 9 computation-intensive benchmarks and their synthesized area on ASIC.

	#line	#pattern	#inst	time (s)	Area (um^2)

streamcluster	96	92	240	0.13	117193
swaptions	152	69	292	0.21	153371
cutcp	67	78	285	0.19	471836
mri-q	79	71	100	0.33	62424
mri-gridding	119	119	385	0.49	904371
denoise	274	187	650	0.52	1357131
deblur	202	29	151	0.24	227410
registration	222	1499	3122	1.42	506124
segmentation	179	2211	4172	1.72	1774202

2.4.3 Alignment Optimization Results

To illustrate the of the proposed alignment insertion scheme, we have compared our solution to four alignment policies proposed in [24], in terms of the overall shifting distance and number of inserted alignment nodes (normalized to our solution). The four reference points in this evaluation include:

(i) *Zero-shift policy* (Z)

Shift each mis-aligned array reference to the alignment offset of 0 immediately after it is loaded from memory.

(ii) *Eager-shift policy* (E)

Shift each mis-aligned array reference directly to the alignment of the store.

(iii) *Lazy-shift policy* (L)

Based on the eager-shift policy, but delay shifting as long as the alignment offset matches between nodes with the same output node.

(iv) *Dominant-shift policy* (D)

Shift each mis-aligned array reference to the most dominant alignment offset in the graph.

Table 2-2. Comparison on overall shifting distance.

	Z	**E**	**L**	**D**
fft	3.4	2	2	1
streamcluster	2.8	2.4	1	2.4
rician-denoise	1	1	1	1
registration	1.5	1.5	1	1.5
segmentation	1	2	1	1

We list the comparisons among the five alignment policies, as shown in Table 3-2. Here only the results of five applications with complicated mis-aligned patterns, such as stencil computations in the medical imaging domain and the mis-alignment introduced by sum reduction technique in *streamcluster*; The shifting distance in Table 3-2 has been normalized to the proposed solution, in which we can see our solution can generate the minimal shifting distance for all the five applications. Among the four schemes proposed in [24], for a few applications the lazy-shift or dominant-shift solution also equals the optimal one, and outperforms zero/eager solutions. While the intrinsic heuristic feature in the two policies lead to less efficient solutions in other cases. The overhead to calculate the optimal solution is less than 7% of the overall compilation time.

2.4.4 Performance Comparison Results

We consider three reference points in the experiments:

(i) Normal vector (NV): Execution of the program using standard state-of-art vector instructions (Intel AVX).

(ii) Dedicated custom vector (DCV): Execution of the program using dedicated ASIC-based customized vector instructions.

(iii) Composable vector (CCV): Execution of the program using CVU-based customized vector instructions.

Figure 2-13 shows the normalized speedup on each individual benchmark. Speedups have been normalized to the normal vector version. We make the following observations:

(i) Benchmarks such as *mri-gridding*, *mri-q* and *deblur* achieve a very large speedup. This is because the kernels in these benchmarks i.e., the critical functions have a structured pattern which is suitable for our architecture. Our compilation flow successfully captured such vectorizable patterns.

(ii) Benchmarks *denoise, registration* and *segmentation* achieve moderately good speedups. We find the patterns in those benchmarks contains two or three parallel *add* and *mul* operations. Due to the available CVU resource constraint we have, such parallelism cannot be fully supported in the CVU-based design.

(iii) The execution time difference between CCV and DCV is very small. Though the latter design does not need to consider the resource constraint. On average CCV is 5% slower than the DCV design on all the benchmarks, which further illustrate the efficiency of our CVU configuration selection strategy.

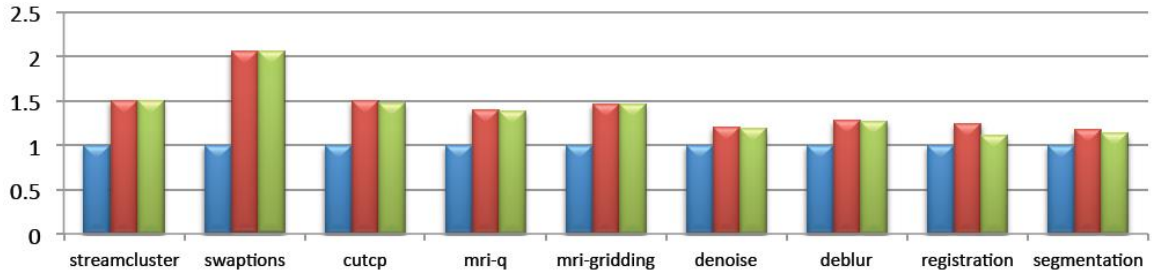


Figure 2-13. Normalized speedup.

2.5 Conclusion and Future Work

Customized vector domain has attracted increasing attention from both academia and industry. To provide efficient customization support, in this chapter we introduce an

LLVM-based compilation flow to perform automatic customized vector ISA extension. A composable vector unit (CVU) is proposed to support a large number of customized vector instruction by allowing chaining among vector units. Our future direction is to extend the composable vector unit design to a multi-core environment such that the CVUs can be shared among multiple requesting cores.

Chapter 3. Compilation for Programmable Accelerators

In recent days programmable accelerators (PA) are widely investigated in the design of domain-specific architectures to improve the system performance and power. Micro-architectures with a series of PA have been proposed to provide more general supports for customization. One important feature in the PA-rich systems is that the target computational kernels can be compiled with pre-defined PA templates and dynamically mapped onto real PAs at runtime. This imposes a demanding challenge on the compiler side regarding how to generate high-quality PA mapping code. In this chapter, we present an efficient PA compilation flow which is fairly scalable in mapping large computation kernels into PA-rich architectures. A concept called *maximal* PA *candidate* is proposed to drastically reduce the number of input PA candidates in the mapping phase without influencing the mapping optimality. Efficient pre-selection and pruning techniques are employed to further speedup the maximal PA mapping process. Our experimental results show that for 12 computation-intensive standard benchmarks, the proposed approach achieves a significant improvement on the compilation time comparing to the state-of-art PA compilation approaches. The average mapping quality is improved by 23.8% and 32.5% for connected PA candidates and disjoint ones, respectively.

3.1 Introduction

Customization is an appealing solution to increase performance power efficiency, which is one of the primary design concerns in the era of many-core systems. A recent industry trend to address it is introducing computation accelerators in many-core designs. The Convey system [51], Intel's Larrabee [52] and Nallatech [53] are example of this. The accelerators, which are normally designed as specialized hardware blocks in the general-purpose processors, can be implemented to support a wide variety of tasks, ranging from fairly simple ones (e.g., a multiply accumulate operation) to more complex

ones (e.g., FFT, encryption/decryption or video encoding/ decoding). However, very complicated accelerators will suffer from the same non-recurring hardware cost as most ASIP work does. On the other hand, a simple accelerator design, which may be general enough for most applications, cannot achieve significant power-performance gains with limited hardware specialization.

To solve this problem, programmable accelerator (*PA*) has been proposed to enable varying degrees of customization in an accelerator-rich systems [51] [54] [55] [56] [57]. In a standard PA architecture, a programmable accelerator template is implemented inside each PA unit to support a selected set of computation tasks with reasonable hardware design costs. The entire pre-defined PA template may support a relatively complicated computation task, while it can be reconfigured dynamically to perform a set of simpler but more general sub-tasks. Therefore, each accelerator unit in a PA-rich system can be customized to computation tasks with different granularity, which enables efficient switching among varying degrees of customization at runtime.

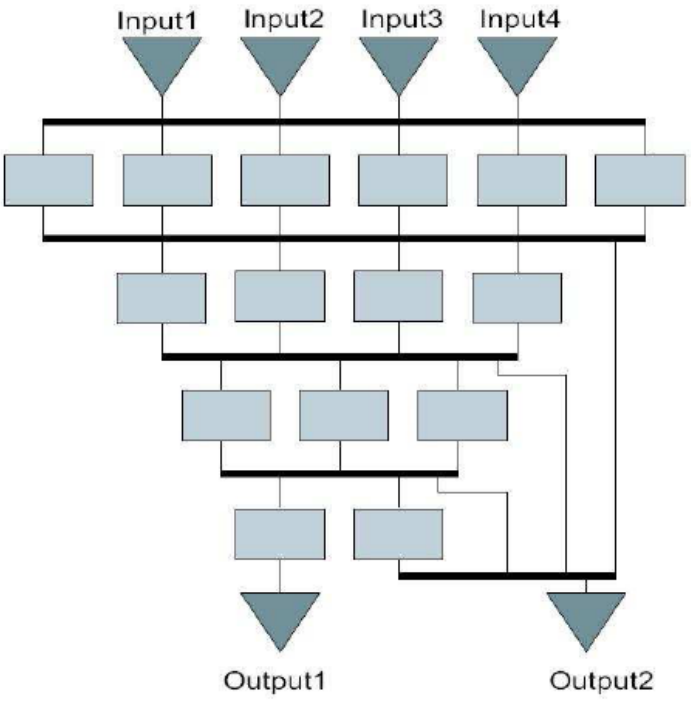


Figure 3-1. Example of a CCA implementation [4].

Figure 3-1 shows the accelerator template used in a configurable computation accelerator (*CCA*) design proposed in [54]. The CCA is built with 15 functional units arranged in a two-dimensional array, in which each functional unit can perform arithmetic or logical operations. Four input data are fed to the top row and processed by each row to generate two outputs at maximal. Functional units in different rows are connected with full interconnects, so that data can be transferred between any two computation units from two adjacent rows. At runtime, the interconnects in this CCA implementation can be configured by hardware control signal to support any 4-input 2-output computation patterns with dependency depth less than 5, namely all the legal computation subgraphs in the given template. In this case, the programmable or configurable accelerators can efficiently accelerate a wide range of applications with flexible customization support. In a later work [55] the built-in full interconnect in this CCA template is simplified to further reduce the area cost while still provides enough customization diversity.

As we discussed, with more flexible customization support, the PA-rich design has been raised as a promising solution to improve the system performance-power efficiency. However, this design trend imposes a demanding challenge on the compiler side – how to generate high-quality PA mapping code which can efficiently utilize the programmable execution units existing in a PA-rich architecture.

Different from most of the traditional instruction-set-extension work [58] [59], which aims at designing a highly customized instruction set for a given set of applications, the PA compilation flow targets at efficiently partitioning application kernels into PA-executable code pieces, or *PA candidate*, and mapping them with the pre-defined PA template.

In general, the PA compilation flow can be divided into two phases - PA candidates identification and PA template mapping. Given the data flow graph (DFG) of application kernels, the PA candidates identification phase extracts all the data flow subgraphs which are executable on the PA units. To decide whether a subgraph is executable on PA units,

subgraph isomorphism checking is performed between the subgraph and the PA templates. After that, the identified candidates will be fed into the mapping phase, in which a subset of candidates will be mapped to PA templates to accelerate the target kernels.

As discussed in [60], since the PA candidates identification and PA mapping problems are both difficult to solve, scalability has been considered as a main problem in the existing PA compilation flows. Considering that the number of PA candidates grows exponentially with the size of input DFG and PA template, the mapping problem may become intractable for large DFG blocks. When disjoint PA candidates are considered, the mapping problem size is even larger after including all the legal combinations of connected PA candidates.

There already exist a few relevant work investigating developing scalable mapping methods to obtain optimal solutions for moderate-size application kernels. For example, in [60], a scalable subgraph mapping algorithm is proposed to generate optimal PA mapping solutions with connected PA candidates. The limitation of this work is the lacking support of disjoint PA candidates due to scalability problem, thus it cannot fully utilize the existing parallelism in a PA template. An extension of this work is discussed in [55], in which the optimally selected connected accelerator patterns are greedily grouped into disjoint ones. However, there is no guarantee on the optimality of the corresponding PA mapping solution, which has been generated in a heuristic way.

To better illustrate the scalability problem in PA compilation, here we use an application from the medical imaging domain as an example, which is called *segmentation* [61]. Following the flow proposed in [60], the PA candidates identification phase extracts 1147 connected PA candidates from the kernel block in segmentation, which contains 115 nodes. To map those PA candidates optimally, the PA mapping algorithm in [60] takes more than 30 minutes to complete. From this example we can see, the exponentially increased number of PA candidates in the identification phase imposes fast-growing pressure on the mapping phase which itself is an NP-complete problem [60]. In this case,

purely optimizing the PA mapping phase, such as the work in [60] and [55], cannot release the scalability problem efficiently when large data blocks or PA templates are considered.

To alleviate the scalability problem in PA compilation, in this chapter contributes we introduce a concept called *maximal* PA *candidate* to efficiently reduce problem size in the ensuing PA mapping phase while maintains the mapping optimality. Our experiment results show that on average the number of input PA candidates in the PA mapping phase has been reduced by 210X and 82X for mapping connected candidates and disjoint ones, respectively. We also show a scalable PA compilation flow with the support of disjoint PA candidates. Comparing to the scalable PA compilation approaches proposed in [60] and [55], our approach achieves a significant reduction on the overall compilation time. The corresponding mapping quality has been improved by 23.8% and 32.5% on average for mapping the connected-only and disjoint PA candidates, respectively.

3.2 Related Work

In this section, we briefly discuss the existing accelerator mapping flows, which can be categorized into the heuristic approach and the exact approach.

As discussed in Section 3.1, both the PA candidates identification and PA mapping problems are difficult to solve. Heuristic approaches have been employed as a standard flow to reduce the mapping complexity. A widely-employed heuristic method is to perform greedy enumeration and immediate selection [62]. In this flow, a seed node is picked and grows by gradually adding neighbor nodes until the expanded subgraph is no longer executable on the PA template. Then the corresponding subgraph is immediately mapped to a PA unit. This process will be repeatedly applied in the remaining data flow graph until it is empty. Heuristics normally can generate feasible solutions with reasonable compilation time. However, there is no guarantee on the optimality of the

compilation results.

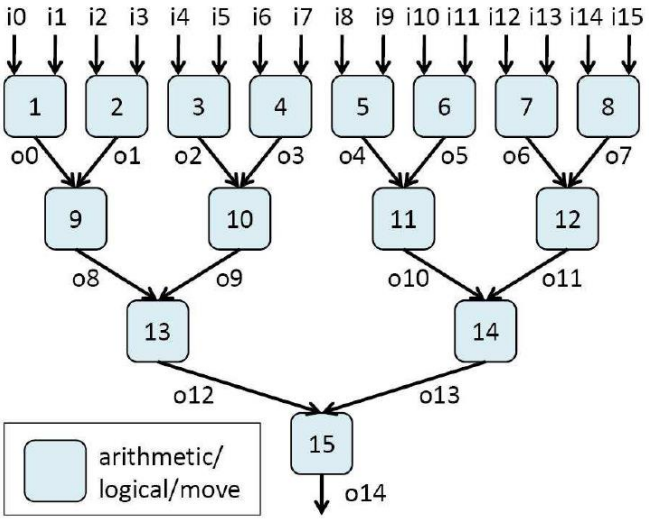


Figure 3-2. A sample PA template.

On the other hand, a standard exact mapping flow can be described as full enumeration followed by optimal mapping [60] [63]. Here full enumeration means enumerating all the possible PA candidates in the target kernels. Then the optimal mapping algorithm, such as ILP-based or branch-and-bound approach, will be applied on the full set of PA candidates. Efficient pruning techniques have been proposed in [60] and [63] to reduce the mapping time. However, due to the fact that the number of PA candidates may grow exponentially with the size of the data flow graph and PA template, the problem size for the optimal mapping algorithm may still be very large and thus make the optimal approach intractable [55].

When disjoint subgraphs are considered, the number of PA candidates further explodes. Therefore a mixture of heuristic and exact approaches has been employed to balance the time complexity and mapping optimality. For example, the authors of [55] first generate optimal PAT mapping solution for connected candidates, then greedily group the selected connected PA candidates into possible disjoint one, which may not be optimal.

There also exists another set of works focus on generating the maximal multiple-input

single-output patterns (MAXMISO or SUBMAXMISO) without limiting the number of inputs or size [64][65]. In [66] and [67], immediate selection is applied to the connected MAXMISO subgraph and an ILP-based approach has been used to optimally packing the selected connected subgraphs into disjoint ones. The generated solution also has the problem of sub-optimality, since the connected subgraphs are selected in a heuristic way and both work assume that no overlapping exist among MAXMISOs. Another limitation in the these work is that the MAXMISO subgraphs are defined to be the subgraphs with maximal size and are generated without considering the micro-architectural constraints, such as size and data flow structure. In this case these work cannot be applied to the PA compilation problem, since the selected subgraphs may not be executable on the PAs.

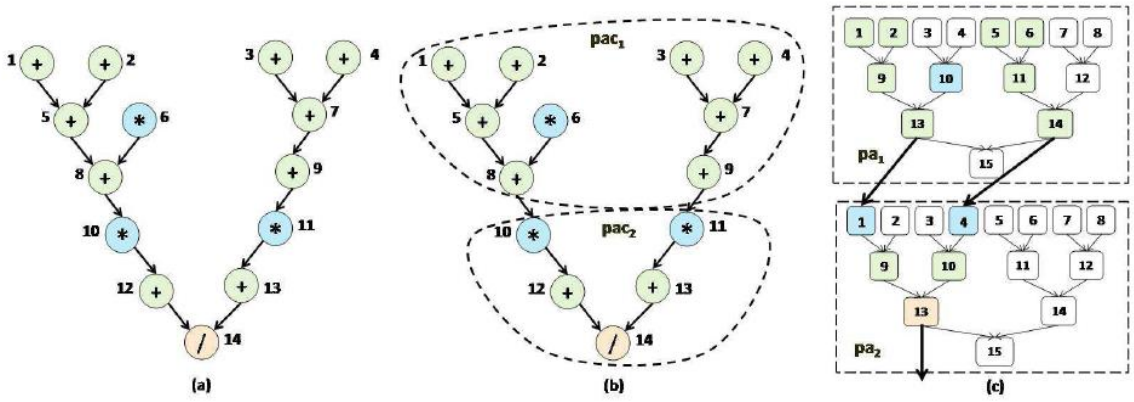


Figure 3-3. (a) DFG of the kernel loop in rician-denoise. (b) One PA mapping solution. (c) Runtime PA configuration of (b).

As discussed above, most of the previous PA compilation works either suffer from limited scalability or sub-optimality of the PAT mapping results, which may largely reduce the possible performance-power efficiency gain when run general applications on the PA-rich designs.

3.3 PA Compilation Example

In this section, we use a real-life medical imaging application, rician-denoise [41], to illustrate the PA compilation results on a sample PA template.

As shown in Figure 3-2, the sample PA template used here is a 4-level binary-tree structure, in which each node can either perform arithmetic operations or forward the input value to its output. The interconnect between two rows is designed in a way that each data can be transferred to next row or be directly accessed as a PA output. In this case at maximal 15 outputs are supported in this PA template, while the PA template in Figure 3-1 only allows two PA outputs at maximum, which largely restricted the possible support for disjoint PA candidates. Therefore it is not very suitable to test the PA compilation flow supporting disjoint PA candidates. Note that our proposed flow can be applied to any predefined PA template and the template in Figure 3-2 is only used here as one example.

Figure 3-3(a) shows the simplified data flow graph of the kernel loop in rician-denoise, which contains 14 arithmetic operation nodes (*add, multiply and divide*). Figure 3-3(b) shows two selected PA candidates pac_1 and pac_2 which covers the entire DFG. Let's first look at the connected candidate pac_2, it is isomorphic to subgraph {1, 4, 9, 10, 13} in the given PA template, therefore is identified as a PA candidate. The corresponding runtime PA configuration is shown in Figure 3-3(c), in which the 15-node PA unit pa_2 will dynamically reconfigured to match the 5 computations in pac_2. The remaining nodes in the PA template will not perform real computations in this mapping.

Comparing to the connected-only case, PA compilation with disjoint PA candidates can better take the advantage of the data-level parallelism inside PA template. For example, pac1 in Figure 3-3(b) contains two connected subgraphs. Since both subgraphs in pac_1 are PA-executable and the outputs of nodes 13 and 14 in pa_1 of Figure 3-3(c) can be calculated without violating data dependency constraint, pac_1 itself is also a PA candidate and can be mapped to one PA template. As shown in Figure 3-3(c), nodes 1, 2, 5, 6, 8 are mapped to template nodes 1, 2, 9, 10, 13; nodes 3, 4, 7, 9 are mapped to template nodes 5, 6, 11, 14.

3.4 Preliminaries and Problem Formulation

To formally formulate the proposed maximal PA compilation problem, in this section we first introduce the necessary definitions and theorems.

Here we assume the input data flow graph is a DAG called G<V, E>, in which each node in V represents an operation it performs and each edge in E represents the data dependency between two nodes. The pre-defined acyclic PA template graph is called $T<V_T, E_T>$, in which V_T and E_T include a set of operation nodes and data dependency edges in the PA template, respectively. The operation nodes that are not supported in the given PA template are called *forbidden nodes*. Without loss of generality, in the rest of this chapter we assume G is the data flow graph after removing all the forbidden nodes.

Definition 3-1. *Given an input data flow graph G <V, E >, a subgraph G*<V*, E*>* $\subset$ *G<V, E > is* ***convex*** *if there exists no path between any two nodes in G* which involves a node* $\in V - V^*$

Definition 3-2. *If G*<V* E*> is a subgraph of G<V, E >, which consists of* $K \geq 1$ *connected components* $G_1^* < V_1^*, E_1^* >$, $G_2^* < V_2^*, E_2^* >$, ..., $G_K^* < V_K^*, E_K^* >$. $G^* < V^*, E^* >$ *is called a legal subgraph of G if (i)* $\forall$ i, $G_i^* < V_i^*, E_i^* >$ *is convex (*$i \in [1, K]$*) (ii) there exists no data dependency between any pair of connected components in G*, when* $K > 1$.

Definition 3-2 can be applied to both connected (when K = 1) and disjoint subgraphs (when K > 1). It ensures that the enumerated legal subgraphs can be scheduled on the PA template without violating data dependency constraints.

Definition 3-3. *Given a PA template* $T < V_T, E_T >$ *and an input data flow graph G < V, E >, a subgraph G*< V*, E*>* $\subset$ *G < V, E > is called an* ***PA candidate*** *if there exist a legal subgraph T** $\subset$ *T, which is isomorphic to G*.*

Based on Definition 3-3, we can further define the concept of *maximal PA candidate* as follows:

Definition 3-4. *A PA candidate G*< V* E*> is called a* ***maximal PA candidate****, if* $\forall V_i \in V - V^*$*, the expanded subgraph* $G^+ <V^* \cup \{vi\}, E^*>$ *is not a PA candidate.*

For example, PA candidate pac_2 in Figure 3-3(b) is not a maximal PA candidate, since it can be expanded by adding nodes 8 or 9 and the expanded graph is still a PA candidate. pac_1 is a maximal PA candidates since it cannot be further expanded and mapped to the PA candidate without violating the data dependency constraints.

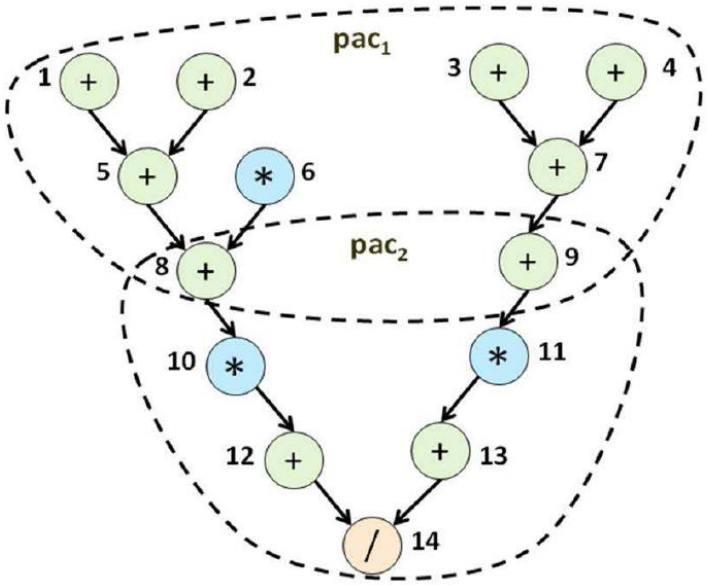


Figure 3-4. Two compatible maximal PA candidates.

Note that PA candidates may overlap with each other at a certain set of nodes. If it is possible to distribute each overlapping node to exactly one maximal PA candidate and the transformed subgraphs are still PA-executable, the overlapping candidates are called *compatible PA candidates* and the corresponding transformed subgraphs without overlapping nodes are called *overlapping- free subgraphs*. Figure 3-4 shows two maximal PA candidates pac_1 and pac_2, which overlap with each other at nodes 8 and 9. They are

compatible with each other since we can remove nodes 8 and 9 from pac2 and the remaining graph is still a PA candidate.

Given a mapping solution consisting of N PA candidates, the PA utilization efficiency can be evaluated by $\frac{|V|}{N\times|V^T|}$, in which |V| and $|V^T|$ refers to the size of target DFG and PA template. A higher utilization efficiency implies the computational resources present in the PA template have been fully utilized. Considering that both |V| and $|V^T|$ are given, we need to minimize the number of selected PA candidates in a mapping solution to improve PA utilization. In this case, we can formulate the PA compilation problem into two sub-problems:

Problem 3-1. ***PA candidate identification.*** *Given an input data flow graph G and PA template T, identify all the PA candidates in G, which can run on the PA units.*

Problem 3-2. ***PA mapping.*** *Given an input data flow graph G and a set of identified PA candidates, select a minimal number of non-overlapping PA candidates which can cover the entire G and map each selected PA candidate to a PA unit.*

We can see that the enumerated PA candidates can be either connected or disjoint, therefore the mapping phase may involve a very large number of PA candidates and becomes difficult to obtain the optimal solution. To make this problem more tractable, we propose a maximal PA compilation flow, which can be partitioned into the two sub-problems as blow:

Problem 3-3. ***Maximal PA candidates identification.*** *Given an input data flow graph G and PA template T, enumerate all the maximal PA candidates in G, which can run on the PA units.*

Problem 3-4. ***Maximal PA mapping.*** *Given an input data flow graph G and a set of enumerated maximal PA candidates, select a minimal number of compatible maximal PA candidates which can cover the entire G.*

Note that the number of maximal PA candidates is much smaller than total number of PA

candidates. For example, the PA candidates {14}, {12, 14}, {13, 14}, {12, 13, 14} share the same maximal PA candidate pac2 in Figure 3-4. Hence the problem size in the maximal PA mapping phase can be largely reduced.

Theorem 3-1. *The optimal solution for the original PA mapping problem defined in Problem 3-2 equals the optimal solution for the maximal PA mapping problem defined in Problem 3-4.*

Proof. Assume the optimal solution for the maximal PA mapping problem contains N maximal PA candidates and the optimal solution for the original PA mapping problem contains M PA candidates.

(i) Assume that the optimal PA mapping solution contains M non-overlapping PA candidates $\{P_1, P_2, ..., P_M\}$. For each P_i in the optimal solution, if Pi is a maximal PA candidate, let $P_i^* = P_i$; If P_i not a maximal PA candidate, expand it by adding neighboring nodes until a corresponding maximal PA candidate P_i^m is generated, let $P_i^* = P_i^m$. The derived set of PA candidate $\{P_1^*, P_2^*, ..., P_M^*\}$ contains M maximal PA candidates, which covers the entire data flow graph and are compatible with each other (the corresponding overlapping-free subgraphs are $\{P_1, P_2, ..., P_M\}$). Therefore $\{P_1^*, P_2^*, ..., P_M^*\}$ is one feasible solution for the maximal PA mapping problem, and we have $M \geq N$.

(ii) Assume that the optimal maximal PA mapping solution contains N compatible maximal PA candidates $\{P_1, P_2, ..., P_N\}$. For each P_i in the optimal solution, $P_i^*$is defined to be the corresponding overlapping-free subgraph of P_i. The derived set of PA candidates $\{P_1^*, P_2^*, ..., P_N^*\}$ contains N PA candidates which do not overlap with each other. Therefore $\{P_1^*, P_2^*, ..., P_N^*\}$ is one feasible solution for the original PA mapping problem, and we have $N \geq M$.

From (i) and (ii), we can get N = M.

Theorem 3-1 demonstrate the optimality of the proposed maximal compilation flow, in which the original PA mapping problem is transformed to the maximal PA mapping

problem with a much smaller problem size.

3.5 Maximal PA Compilation Flow

In this section we discuss the proposed maximal PA compilation flow, which is performed in two steps: maximal PA candidate identification and maximal PA mapping.

3.5.1 Maximal PA Candidates Identification

Efficient pattern identification techniques have been investigated in a wide range of work [67] [68] [69]. In our flow, the subgraph identification and isomorphism checking techniques proposed in [18] are employed to generate connected PA candidates efficiently. At step k + 1, all the PA candidates with k nodes are extended by adding one neighbor in topological order to reduce duplicate identification. After a new subgraph G* is generated, it will be compared to the subgraphs of the given PA template for graph isomorphism checking. A filtering scheme based on characteristic vector (CV) [67] is applied here to reduce the number of expensive graph isomorphism checking operations.

If G* with k + 1 nodes is a PA candidate, all the subgraphs of G* with k nodes will be marked as non-maximal. In this case, when k increases to the maximal PA size, all the maximal connected PA candidates have been generated.

The work in [69] proves that any connected component of a disjoint PA candidate must be a connected PA candidate. Therefore the disjoint PA candidates can be generated by grouping a set of connected ones together. In our flow, at step l + 1, all the non-maximal PA candidates with l connected components are extended by adding one connected component and all the subgraphs of G* will be marked as non-maximal.

Note that instead of generating all the disjoint PA candidates in an input data flow graph, we only target at those which can be mapped to the pre-given PA template. Therefore the micro-architectural constraints in the PA template, such as depth, size, number of inputs/outputs, can be applied to prune the identification space. For example, after covering the disjoint PA candidate pac_1 with pa_1 in Figure 3-3(c), the entire PA template

will be occupied and no more nodes can be mapped to the remaining nodes {3, 4, 7, 8, 12, 15} in the PA template without violating the data dependency constraint. Therefore the disjoint PA candidate pac1 will not be grouped with any new connected component to form a larger PA candidate and can be removed from the set of PA candidates to be further expanded. Similarly, size and number of ports for each newly-generated disjoint PA candidates are also collected during the identification process for early pruning. Another example is the PA template designed in [54], in which only two outputs are supported. This imposes a fairly strict constraint on the disjoint PA candidate generation and hence efficient output-port-directed pruning strategy presented in [69] can be applied to reduce the exploration complexity.

3.5.2 Maximal PA Mapping

Now that we have a set of maximal PA candidates, a subset of those candidates need to be selected and mapped to PA units.

Here we present a branch-and-bound algorithm to generate the optimal covering solution with maximal PA candidates. Taking the advantage of the features of maximal PA candidates, efficient preprocessing and pruning techniques have been developed to reduce the algorithm runtime. The proposed maximal PA mapping approach is shown in Algorithm 1. The algorithm inputs include a data flow graph G and a set of maximal PA candidates MP. The final output is a subset of MP which covers G with the minimal number of maximal PA candidates. As shown in lines 8-15, the entire mapping flow can be divided into three stages: pre-selection, greedy-sol-gen and max-cover, as discussed below:

pre-selection As we discussed in Section 3.2, the exact algorithms for the PA mapping problem accept a full set of identified PA candidates as inputs. Therefore at most cases a node in G is covered only by one PA candidate, unless it is disconnected from other nodes in G. For example, node 14 in Figure 3-4 can be covered by possible PA candidates such

as {14}, {12, 14}, {13, 14}, etc. On the other hand, when we only include maximal PA candidates in the mapping phase, node 14 is only contained in one maximal PA candidate pac_2 in Figure 3-4. In this case, we can directly conclude that PA candidate pac_2 will be selected in the optimal covering solution, and remove all the nodes covered by pac_2 from G. Then the mapping process only needs to be applied to the remaining data flow graph with less PA candidates. For example, after pac_2 is selected, the remaining graph contains 7 nodes, which is only half of the original size.

greedy-sol-gen Note that in a branch-and-bound approach, the current optimal solution is usually used to prune the searching space and speedup the covering process. For example, if we know that the best possible solution generated from the current step is worse than the current optimal solution, we can immediately stop branching at the current direction and save the corresponding algorithm runtime. In this case, the initial solution should be set as close as possible to the real optimal one for fast pruning. In Algorithm 3-1, a greedy covering solution with disjoint PA candidates is used to enable fast initial pruning, as shown in line 9.

max-cover Lines 17-40 show the branch-and-bound based covering algorithm, which is applied to the reduced data flow graph generated after the pre-selection stage. For each maximal PA candidate, it can be either included or excluded in a feasible solution, as shown in lines 14-15 and lines 36-37. Here if the second parameter *decision* equals *true*, the corresponding PA candidate will be selected in the current solution, otherwise not. In this case all the possible combinations will be evaluated to obtain the optimal solution.

When the entire graph is covered after adding a new PA candidate, the corresponding covering solution will be compared to the current optimal solution at line 21. If the newly-generated solution turns out to be better, compatibility checking will be performed on the selected PA candidates. The current optimal solution will be updated if the selected PA candidates are compatible with each other, as shown in lines 22-25.

Assume each overlapping node v_i is covered in n_i PA candidates, therefore in the worst

case $\prod n_i$ non-overlapping node assignment schemes need to be evaluated to decide whether a set of overlapping PA candidates are compatible or not. To perform fast compatibility checking, tight nodes are first removed from the overlapping node set. Here an overlapping node v is a called tight node of PA candidate P if all the overlapping-free subgraphs of P will contain v. For example, the overlapping nodes locating in a path between two nodes in P are tight nodes, if the corresponding two nodes do not belong to the overlapping set. Therefore it should be directly assigned to P, otherwise the convexity of P cannot be maintained. In this case, if a node v is the tight node of more than one selected PA candidates, we can directly conclude that no overlapping-free node assignment scheme exists and the covering solution is not compatible. After removing the tight nodes, all the possible node assignment schemes in the remaining overlapping set will be evaluated in which the same pruning technique using tight nodes can be applied. The overlapping removal problem itself is computational demanding, but it will only be performed when a better solution has been found. In practice, with efficient pre-selection as well as the initial greedy solution obtained at line 9, the number of overlapping PA candidates and nodes during compatibility checking process is small. As shown in Section 3.6, the total algorithm runtime, including compatibility checking, is fairly affordable.

In order to efficiently prune the searching space, at line 13, all the PA candidates are sorted in the decreasing order of its size in G^-, namely $|s_i|$, which ensures that the size of the currently added PA candidate is always greater than (or equal to) the PA candidates added later. With this observation, we can conclude that after the i^{th} PA candidate has been selected, at least $\left\lceil \frac{N}{s_i} \right\rceil$ PA candidates are needed to cover the remaining graph, where N equals the number of uncovered nodes. Therefore, at lines 33-34, an early optimality checking is performed to evaluate the current partial covering solution. If the best possible covering solution by continuing growing the currently selected PA candidate set is worse than the optimal solution we have obtained so far, no further searching from the

current state will be performed and the algorithm will directly return to an earlier covering state.

Algorithm 3-1. Maximal PA Mapping Algorithm

1: $G(V, E)$: input data flow graph

2: MP: a set of maximal PA candidates $MP_1(V_1, E_1), ..., MP_N(V_N, E_N)$

3: MP^s: the set of pre-selected maximal PA candidates

4: MP*: a subset of MP which covers G optimally

5: V^s: the nodes covered by MP^s

6:

7: Procedure *max-PA-mapping()*

8: *pre-selection();*

9: optimal_sol = *greedy-sol-gen*();

10: for each PA candidate $MP_i \in MP\text{-}MP^s$ do

11: si = $|\{v|v \in V_i \&\& v \notin V^s\}|$

12: end for

13: sort PA candidates in MP - MP^s in decreasing order of s_i

14: *max-cover*(1, true);

15: *max-cover*(1, false);

16:

17: Procedure *max-cover(i, decision)*

18: if decision = true then

19: add $MP_i \in MP - MP^s$ in MP*

20: if MP* covers V -V^s then

21: if |MP*|+|MP^s|< optimal_sol then

22: if the PA candidates in MP*+MP^s are compatible with each other then

23: optimal_sol = |MP*|+|MP^s|;

24: optimal_set = MP*+MP^s;

25: end if

26: end if

27: return;

28: end if

29: end if

30: if i+1 > |MP-MP^s| then

31: return;

32: end if

33: if $|MP^*|+\left\lceil \frac{\#uncovered\ nodes}{s_i} \right\rceil \geq$ optimal_sol then

34: return;

35: end if

36: *max-cover*(i+1, true);

37: *max-cover*(i+1, false);

38: if decision = true then

39: remove MP_i from MP*

40: end if

3.6 Experimental Results

3.6.1 Experiment Setup

We evaluate the proposed maximal PA compilation flow on 12 computation-intensive applications from widely-known benchmark suites and computing domains, with the kernel DFG size ranging from moderate to large. The test cases include five benchmarks from the SPEC2006 suite [70] (*calculix, leslie3d, povray, bwaves* and *lbm*), four applications from the medical imaging domain [41] (*compressive sensing*, *registration*, *rician-denoise* and *segmentation*), and three applications from the Rodinia benchmark suite [13][14] (*heartwall, leukocyte* and *cfd*), which is designed for heterogeneous computer systems with accelerators. The applications inside each benchmark suite are listed in the increasing order of application kernel size.

Here we consider two scenarios: (1) PA compilation only with connected PA candidates (2) PA compilation with disjoint PA candidates. The proposed PA compilation flow is evaluated in both cases, and compared with the representative previous work [60] [55] targeting scalable PA compilation.

Our PA compilation flow is implemented with the LLVM compiler infrastructure [42]. In the experiments, the tested benchmarks are compiled with all the standard optimizations in O3 turned on. The compilation time is obtained on a 4-core Intel Xeon CPU (E5404) running at 2 GHZ.

3.6.2 Comparison Results

In this section, we show the comparison results of four PA compilation flows - scalable connected PA compilation (SC-PAC) [60], the proposed maximal connected PA compilation (MC-PAC), scalable disjoint subgraph mapping(SD-PAC) [55] and the proposed maximal disjoint PA compilation (MD-PAC), in which the first two approaches only target connected PA candidates and the last two consider disjoint candidates.

Compilation time. Table 4-1 shows the comparison results on the PA compilation time. Following [60] and [55], 600 second is used as a maximum time limit, upon which the PA compilation will be terminated and the best solution generated by this time point will be reported. Note that 1 sec. in Table 4-1 means that the compilation can complete in one second.

From Table 4-1, we can make the following observations:

(1) The compilation time of SC-PAC and SD-PAC are very close to each other. The reason is that SC-PAC is a sub-routine of SDPAC. In SD-PAC, the optimal connected PA mapping solution is first generated with SC-PAC. After that, a greedy grouping operation is performed on the selected PA candidates with negligible time overhead, as shown in Table 4-1.

(2) In the connected compilation case, the maximal PA compilation algorithm can complete in less than 10 seconds for all the benchmarks, while the SC-PAC flow fails to complete for six test cases, and its compilation time increases quickly when the compilation problem size grows.

(3) In the disjoint compilation case, the results are similar, in which MD-PAC can complete in no more than 300 seconds for all test cases.

The large gap of algorithm runtime between SC(D)-PAC and MC(D)-PAC can be explained with Table 4-2 and Table 4-3. The problem size of PA mapping is related to two factors - the target DFG size and the total number of PA candidates which can be selected into a mapping solution. As we discussed, with the proposed concept of maximal PA candidates, both factors can be efficiently reduced. From Table 4-2 we can see, by only including the maximal ones, the total number of PA candidates in the mapping phase can be reduced by 210X and 82X on average in the connected and disjoint case, respectively. Table 4-3 shows the reduction on the number of nodes to be covered in the kernel DFG, after the pre-selection discussed in Section 3.5.2 is applied. SC-PAC and SD-PAC normally need to cover the size of the entire DFG, since most DFG nodes belong to more

than one PA candidate and cannot be selected directly. While in MC-PAC and MD-PAC, the number of nodes to cover can be reduced by 20% and 25% on average, as shown in Table 4-3.

Note that for *lbm*, *segmentation* and *cfd*, a greedy MD-PAC approach has been applied after generating all the maximal connected PA candidates, which results in a much smaller runtime. The reason is that the disjoint pattern enumeration is too costly in those benchmarks, e.g., in *segmentation*, the number of PA candidates with two connected component already exceeds 20,000. To solve this problem, once all the connected PA candidates have been identified, the mapping problem size of PA compilation is roughly estimated as the product of target DFG size and the number of identified PA candidates. In our disjoint PA compilation flow, if this product is larger than a given value (e.g. 9000), a greedy compilation algorithm will be invoked, in which all the maximal connected PA candidates are further grouped into disjoint PA candidates and fed into the mapping phase. Since the maximal connected candidates will not be removed after the grouping operation, this can ensure the solution generated in our flow is always better than or equal to that in [55].

Algorithm Scalability. To illustrate the scalability of the proposed maximal PA compilation flow, we plot compilation time with the corresponding problem size for the 12 benchmarks. Here the compilation problem size is estimated as the product of target DFG size and the number of identified PA candidates.

As shown in Figure 3-5, SC-PAC and SD-PAC runs fairly fast for moderate-size applications, while exhibits limited scalability when the problem size grows. Note that *leukocyle* is one application which can be compiled within 600 seconds even with a large problem size. This is because the real runtime will also be influenced by other factors, such as subgraph overlapping and the efficiency of the initial greedy solution. The estimated problem size is used here to provide an insight of the overall trend.

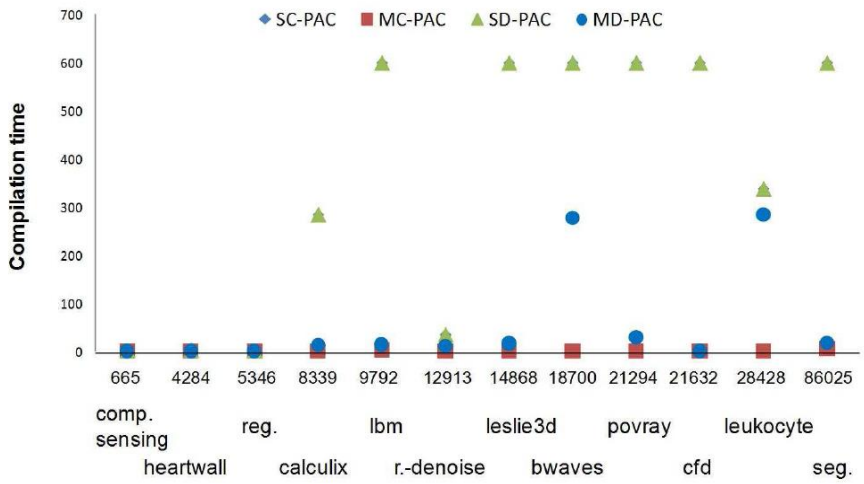


Figure 3-5. Algorithm runtime vs. input problem size.

Considering the maximal PA compilation flow, the increased problem size has a small effect on the MC-PAC runtime and it can finish quickly for all the 12 benchmarks. When disjoint PC candidates are included, the corresponding compilation flow MDPAC gradually slows down as the problem size increases, but it still can finish in less than 300 seconds for all the benchmarks tested. As we discussed, when the estimated problem size exceed a given threshold, a greedy MD-PAC process will be invoked and the corresponding compilation time falls drastically while still can generate reasonable mapping quality, which will be shown later in this section. This further demonstrates the scalability of the proposed maximal compilation flow to deal with large benchmarks or PA templates, even including the disjoint PA candidates.

Table 3-1. Comparisons on PA compilation time (sec)

	calculix	Leslie.	povray	bwaves	lbm	c.s.	reg.	denoise	seg.	h.w.	leuk.	cfd
SC	284.3	-	-	-	-	1	1	34.2	-	2.1	338	-
MC	1	1	1	1	4.1	1	1	1	5.2	1	1	1
SD	284.9	-	-	-	-	1	1	34.6	-	2.2	339	-
MD	14	17	29	277	14.6	1	1	11	18.9	1	284.1	1

Table 3-2. Comparisons on the number of PA candidates

	calculix	Leslie.	povray	bwaves	lbm	c.s.	reg.	denoise	seg.	h.w.	leuk.	cfd
SC(D)	269	413	507	425	204	35	198	349	1147	252	618	416
MC	12	16	29	21	36	11	14	19	44	11	33	27
MD	98	31	378	215	54	28	103	162	57	11	270	46

Table 3-3. Kernel size reduction with *pre-selection*

	calculix	Leslie.	povray	bwaves	lbm	c.s.	reg.	denoise	seg.	h.w.	leuk.	cfd
Orig.	31	36	42	44	48	19	27	37	75	17	46	52
MC	9	10	10	5	3	6	8	7	20	4	11	5
MD	10	13	10	7	0	9	10	7	0	8	18	0

Mapping optimality. Figure 3-6 shows the comparison results on the final mapping solution, which equals the number of selected PA candidates to cover the target DFG.

From the results we can see, comparing to the optimal approach SC-PAC, MC-PAC generates better mapping solution at 6 applications with relatively large kernel size. This is due to the fact that with those large test cases, SC-PAC cannot finish within 600 seconds and thus cannot obtain the actual optimal result even though the approach itself is optimal. On average, MC-PAC can achieve 14% improvements over SC-MAC in terms of the mapping quality, and MD-PAC can achieve 23.8% improvements over the heuristic approach SD-PAC and 32.5% improvements comparing to the results of SC-PAC with connected PA candidates.

3.7 Algorithm Generalization

So far we have discussed the maximal PA compilation flow to cover the target applications with a minimum number of non-overlapping maximal PA candidates. This flow can be further generalized in two directions.

Longest PA path length optimization In PA mapping work, there exist another set of work targeting at finding the PA mapping solution with minimal longest PA path length, which can be called as min-length optimization. Here the length of a PA path is calculated by the number of PA units locating on the path. Algorithm 1 can be easily modified to generate the optimal min-length solution. In this case, the maximal PA candidates are sorted in decreasing order of longest path length (in terms of DFG nodes) at line 13 in Algorithm 1, instead of size. Therefore we can ensure the longest node-wise path length of the currently added subgraph, denoted by l, is always no less than the ones added later. In this case, to cover the remaining DFG with longest node-wise path length equaling L, at least $\left\lceil \frac{L}{l} \right\rceil$ PA units are needed to cover the L nodes. Assume the current shortest path length is l_s, which equals the smallest number of selected PA candidates locating in the same path. At lines 33-34, we can apply similar pruning strategy - if $\left\lceil \frac{L}{l} \right\rceil + l_s$ is no less than the current optimal solution, the algorithm will return to an earlier covering stage, since the best possible solution if we continue adding new PA candidates into the current partial solution cannot be better than the best solutions we have obtained so far.

Overlapping By allowing overlapping, duplicate computations may be performed and have a side effect on performance/power efficiency. Therefore most previous work does not allow selected PA candidates to be overlapped with each other. However, the existence of overlapping PA candidates may result in a better mapping solution, as discussed in [58]. To generalize Algorithm 1, we can prove the optimal solution for a general PA mapping problem, which need not to be non-overlapping, equals the optimal solution for a general maximal PA mapping problem without compatibility constraint.

The proof is similar to the proof of Theorem 3-1.

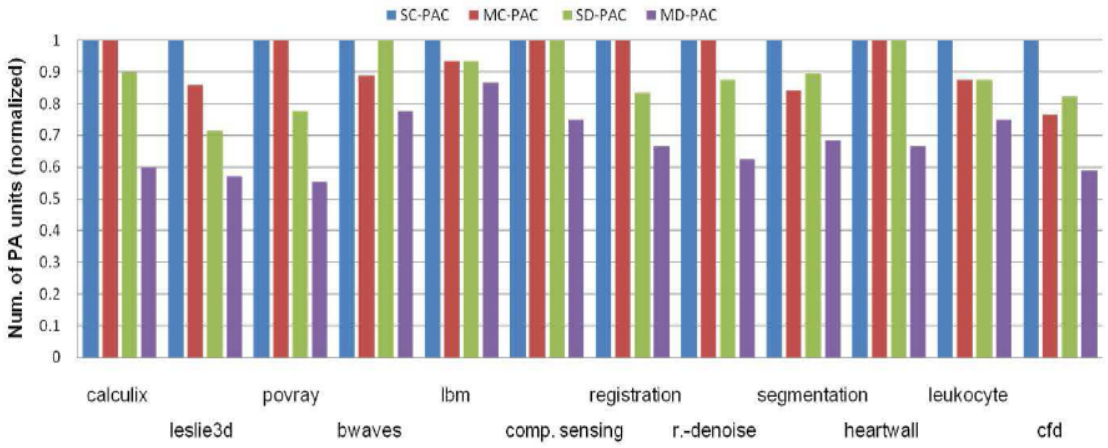


Figure 3-6. Comparisons on PA compilation result.

Therefore the modified Algorithm 1 after removing the compatibility checking at line 22 can be directly applied to the general maximal PA mapping problem.

3.8 Conclusions

In this chapter we introduce a new PA compilation flow based on maximal PA candidates. The proposed flow shows significant improvements in terms of compilation time, result quality as well as scalability. One thing to note here - currently PA candidates are defined with subgraph isomorphism, while in general full equivalence checking techniques can be applied to check whether the two computational subgraphs generate the same results, which will be investigated in the future work.

Chapter 4. Compilation for Fully Pipelined Accelerators

Programmable accelerators (PA) are widely investigated in the design of domain-specific architectures to improve system performance and power. In PA-rich systems, target computational kernels are compiled with pre-defined PA templates and dynamically mapped onto real PAs. To secure highest energy efficiency, full pipelining has become a critical factor in PA design. This imposes demanding challenges on compiler regarding how to generate high quality mapping code. In this chapter we propose an optimal PA mapping algorithm to efficiently map computation kernels onto a series of fully pipelined accelerators. The proposed approach achieves an average 1.24X speedup comparing to previous work.

4.1 Introduction

Customization is an appealing solution to increase performance-power efficiency, which is one of the primary design concerns in the era of many-core systems. A recent industry trend to address it is by designing and integrating fixed-function computation accelerators on the die, targeting application domains demanding high performance and power-efficient execution. Graphs, media, audio and imaging are example domains of this [96][97]. Although fixed-function accelerators can be designed to provide the best performance/energy efficiency for a specific domain, it suffers from poor flexibility problem, hence are not suitable for the domains with constantly changing use protocols.

To address this problem, programmable accelerator (PA) has been proposed to enable varying degrees of customization in accelerator-rich systems [39][54][55][22][98]. In a standard PA architecture, a programmable accelerator template is implemented in each PA unit to support a selected set of computation tasks with reasonable hardware design costs. The entire pre-defined PA template can be dynamically reconfigured to perform a set of simpler but more general subtasks. Therefore, each accelerator unit in a PA-rich

system can be customized to computation tasks with different granularity, which enables efficient switching among varying degrees of customization at runtime. One example is the PA template used in [54], which can be configured by hardware control signals at runtime to support all the 4-input 2-output computation patterns with dependency depth less than 5.

Prior PA-flavor designs were proposed in the era when transistor count is a limited kind of resources. Taking CGRA (coarse-grained reconfigurable architecture) as an example, each processing element is placed with multiple instructions through modulo scheduling, thus needs to switch among different modes when execute these instructions. This kind of time multiplexing incurs extra control logics and is necessary only when the transistor resource is limited. In the era of dark silicon, the system performance is no longer limited by transistor count, but mainly constrained by energy consumption. Motivated by this trend, fully pipelined programmable accelerator without unnecessary time multiplexing has been designed to achieve the highest energy efficiency, such as [99].

On the other hand, the emergence of those PA-based designs imposes a demanding challenge on the compiler side − how to generate high-quality PA mapping code to achieve the highest energy efficiency. The first challenge is how to fully utilize on-chip resources. Note that the computation carried by a PA at runtime, which can be called active region, is one subtask supported by its PA template. The total number of PAs used to cover a given input kernel highly depends on the active region size of each PA instance, and a number of PA compilation work [55][60] targets finding an optimal PA mapping solution with least PA usage. The second challenge comes with the pipelined PA execution. In a fully pipelined PA design, input data comes in at every clock cycles, buffers or dummy PAs [99] need to be inserted to guarantee the correctness of pipeline behavior. This serves as a new demanding resource requirement, which is not considered in previous work.

In this chapter, we investigate and model the impact of throughput target on resource

usage in accelerator pipelines. Here resource usage includes not only PAs, but also buffers required to balance path delay. We also propose an optimal PA mapping algorithm to efficiently map on-chip accelerator resources to pipelined execution. Compared to the PA compilation approaches proposed in [60], our approach achieves a significant reduction on mapping size and up to 33.8% improvement on system performance

4.2 Overview of Fully Pipelined PA

In this section, we use a real world example to illustrate the execution model of one recently-design fully pipelined PA architecture called CHARM [99].

Figure 4-1 shows the basic architecture of CHARM. It consists of a set of PAs as computation elements, with dedicated interconnects in a chain. But most of the data ports of computation elements will go through a pipelined permutation data network to support arbitrary topology of the data flow graph in user applications. There are also a small number of delay units connected to the data network to provide temporary storage if any data element will be used by different modules in different time slots. The local SPM banks in the memory complex iterate the regular access patterns of load/store operations in user applications and read/write data under the control of address generations. While one side of SPMs is connected to the data network, the other side is connected to global data unit for data from external memory. There is also a synchronization unit for pipeline management, and a configuration unit to provide all the modules with constant configuration bits generated by an accelerator/buffer controller (ABC) from compiler binaries. The controller is a module that directly talks with ABC and monitors the status of all the modules in an island.

Figure 4-2 shows a sample PA template containing four computation nodes in total, which can be configured to all the one-node, two-node, three-node subgraphs of itself.

This PA template is designed following the Xilinx DSP48E structure [100], which is frequently used in a variety of applications. When the kernel loop in Figure 4-3 (a) is mapped to the underlying hardware, each load/store operation will be mapped to pipelined address generators coupled with scratch-pad memories. Each add/subtract/multiply operation will be be mapped to a pipelined PA. As shown in Figure 4-3 (b), the kernel loop is partitioned into 4 PAs, which contains nodes {2, 6}, {3, 7}, {4, 8, 10} and {1, 5, 9, 11} respectively. All the edges in the data flow graph of the kernel loop will be mapped to a pipelined data network. Every clock cycle, five data elements associated with the five input data array references will be loaded for computation. In the next cycle, while the last five data elements are still wandering at the intermediate stages, the other five data elements for the next loop iterations are sent to the network. This full pipeline guarantees high energy efficiency of computation and full exploitation the benefits of regular computation/access patterns of the target application.

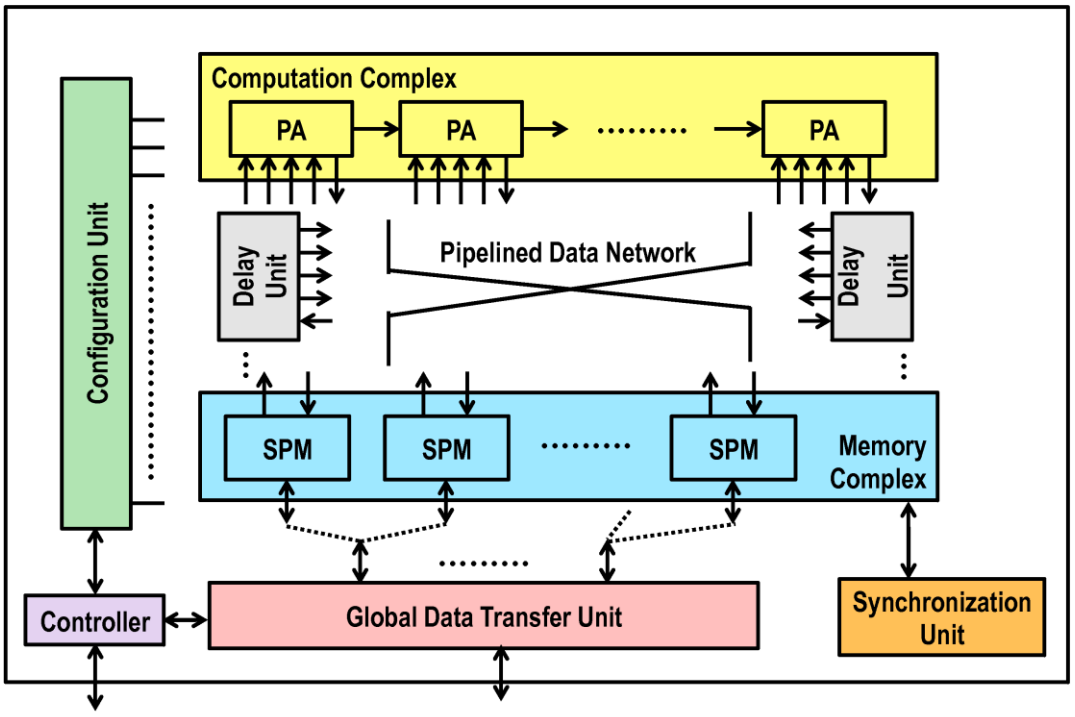


Figure 4-1. Architecture of CHARM

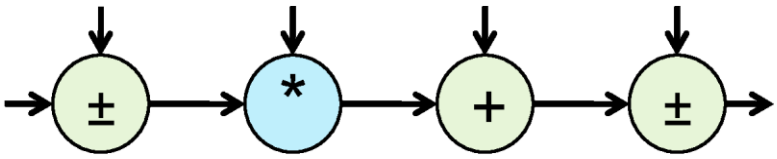


Figure 4-2. A sample PA template

One thing to note is that during pipelined execution some data elements in the data network will be used by multiple modules in different cycles. For example, the data element generated by u[i][j] will be used by PA_0-PA_3 in Figure 4-3 (b). This result in four different paths which start from u[i][j] and end at PA_3 - {u[i][j], PA_3}, {u[i][j], PA_0, PA_3}, {u[i][j], PA_2, PA_3} and {u[i][j], PA_1, PA_2, PA_3}. Therefore we need to insert delay units to temporarily store these data elements between its first use and its last use. As shown in Figure 4-3 (c), two delay units are inserted to balance the delay along those four paths. The challenge is that due to the full pipelining feature of our underlying hardware, each delay unit will receive a new data element every clock cycle. To reduce the amount of delay units used in a pipelined execution, we need to reduce the lifetime of each data element as much as possible. This motivates us to do path balancing for resource and energy savings.

4.3 Preliminaries

Definition 1 *Given an input data flow graph G<V,E>, a set of PA candidates* $\{G_1(V_1,E_1),G_2(V_2,E_2),...G_M(V_M,E_M)\}$, *we define* $.MG_M(V_M,E_M)$ *as a PA mapping graph, if it satisfies: (1) there exist an injective function* $.f:G_i \to v\{i \in [1,K], v \in V_M\}$ *(2)* $.f(G_1) \cup .f(G_2) \cup ... \cup f(G_2) \cup V_{io} = V_M$ *; (3)* $.V_1 \cup .V_2 \cup ... \cup V_K \cup V_{io} = V$ *; (4)* $.E_1 \cup .E_2 \cup ... \cup E_K \cup E_M = E$.

In other words, each PA candidate or memory reference node in the G is mapped to one

node in MG (constraint (1&2)). MG covers the original data flow graph (constraint (3&4)). Figure 4-3 (b) shows a PA mapping graph in which each node either corresponds to a PA candidate or a memory reference. Since it takes four PA templates to cover the input data flow graph, the size of MG equals four times the template size.

Definition 2 *A digraph G<V,E> is called a balanced graph, if it satisfies: for* $\forall u \in V$, $\forall v \in V$, *if there exist at least one path from u to v, then all the paths from u to v have the same path length. Here the length of a path p is defined to be the number of vertices in p.*

Based on the definition above, we can see Figure 4-3 (c) is a balanced mapping graph, since all the paths from node $u[i][j]$ and PA_3 have length 4.

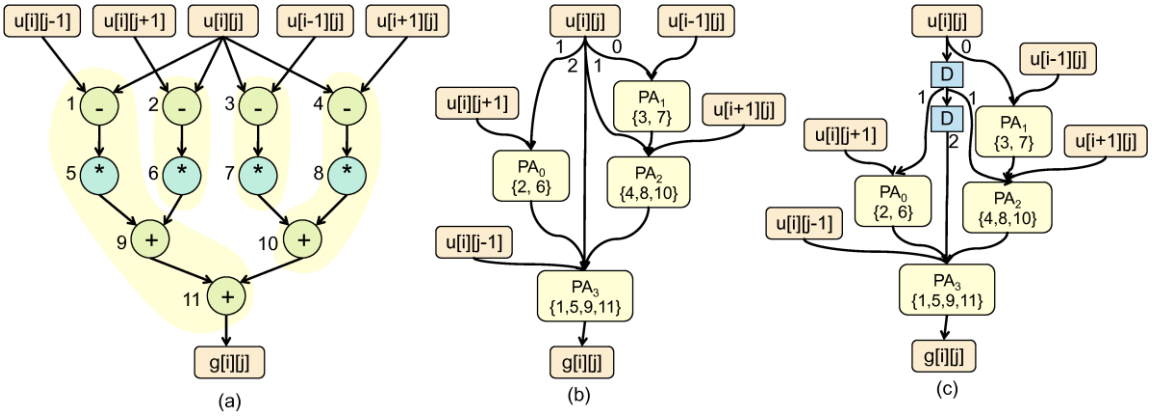


Figure 4-3. (a) Mapping solution I of rician-denoise. (b) Mapping graph of (a). (c) Balanced mapping graph of (a).

4.4 Throughput-Aware Path Balancing

In this section, we discuss a general delay unit insertion scheme under a given accelerator pipeline throughput, represented by a initial interval (II). Suppose there exist K paths $P_1, P_2, \ldots, P_K$ from vertex u to v with length l_i (0 < i <= K). Without loss of generality, we can assume $l_1 \leq l_2 \leq \cdots \leq l_K$. Then during pipelined execution with one input data coming in every II cycles, $\lceil (l_K - l_1)/II \rceil$ delay units need to be added to path p_i to guarantee the arrival time of vertex u's *K* inputs are equal.

From the discussion above we can see when II equals 1, namely a full pipelined execution, the system exhibits the most demanding requirement on the number of delay units.

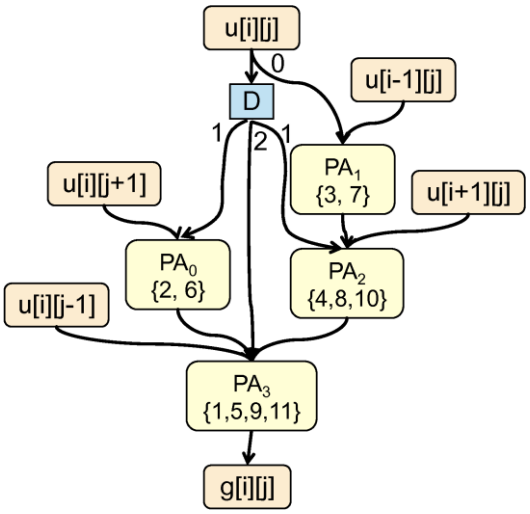


Figure 4-4. Delay unit insertion (II = 2).

To further illustrate the impact of II on delay unit insertion, we use Figure 4-4 to show the balanced mapping graph of Figure 4-3 (b) when II equals 2. In this case, only 1 delay unit need to be inserted at paths from $u[i][i]$ to PA_3. Assuming data u_T is stored in delay unit D at cycle T, the value of u_T are sent to PA_1, PA_0, PA_2 and PA_3 at cycle T, T+1 and T+2, respectively. At cycle T+2, a new data u_{T+1} comes in and overwrites u_T in delay unit D. Figure 4-5 further illustrates the behavior of delay units in both Figure 4-3(b) (2 delay units, II=1) and Figure 4-4 (1 delay unit, II=2). As we can see, when II equals 1, a new data will come in every clock cycle. In order to inject delay 2, two chained delay units are inserted at the output of reference $u[i][j]$. The value of u_T is sent to both PA_1 and the first delay unit at cycle T; at cycle T+1, u_T will be sent to PA_0, PA_2 and the second delay unit, and the first delay unit is overwritten by a new data u_{T+1}; at cycle T+2, u_T is sent to PA_3 from the second delay unit, as shown in Figure 4-5(b). When II

equals 2, a new data will come in every two cycles. In this case, the delay unit in Figure 4-5(a) holds the value of u_T until T+2 - during this period, u_T is accessed by PA_1 at time T, by PA_0 and PA_1 at time T+1, finally consumed by PA_2 at time T+2. At the same time, the next data u_{T+1} overwrites the value of u_T in the delay unit.

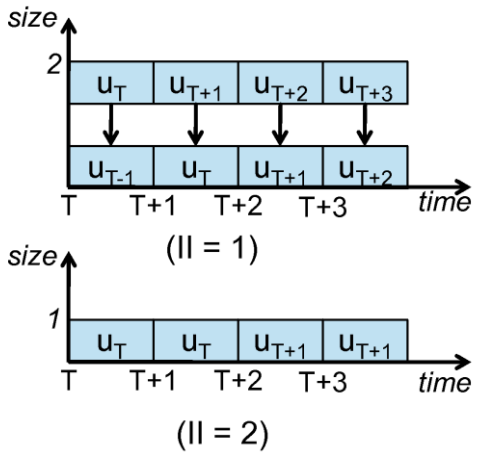


Figure 4-5. Chained delay units for a target II.

Figure 4-6 shows the chained delay unit insertion scheme under a given II. Delay unit D_i holds each data for *II* cycles, then pass it to D_{i+1}. As we can see, in total $\lceil (l_K - l_1)/II \rceil$ delay units are inserted in the chain to provide delay $(l_K - l_1)$.

We've shown that given a PA mapping graph, it can be transformed into a balanced graph by adding delay in the unbalanced paths. Note that if there is no cycle in the PA mapping graph, delay can be added to the primary input of each path by postponing its access from on-chip memory. In this case no area overhead will be incurred from delay unit insertion.

As one can see from Figure 4-7, different mapping solutions may lead to different path balancing results. For example, mapping solution in Figure 4-7(a) contains 5 PA nodes. It is larger than the size of Figure 4-3(b), which covers the data flow graph with only 4 PA nodes. However, if we look at the path balancing between u[i][j] and its output PA nodes, the maximal path length difference equals 1. This means only 1 delay unit need to be

inserted to enable a pipelined execution, as shown in Figure 4-7(c). Therefore the total mapping size of Figure 4-3(c) is *4*area(T) + 2*area(delay_unit)*, and the total mapping size of Figure 4-7(c) equals *5*area(T) + 1*area(delay_unit)*. If the PA system is equipped with a smaller number of delay units comparing to the amount of PAs, which is usually the case considering interconnect design complexity, mapping solution in Figure 4-7 will be selected since it consumes less scarce resource in the system. On the other hand, this further enlarges the search space when looking for an optimal mapping solution.

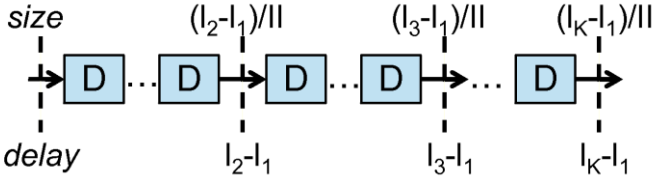


Figure 4-6. Delay propagation when II = 1 and 2.

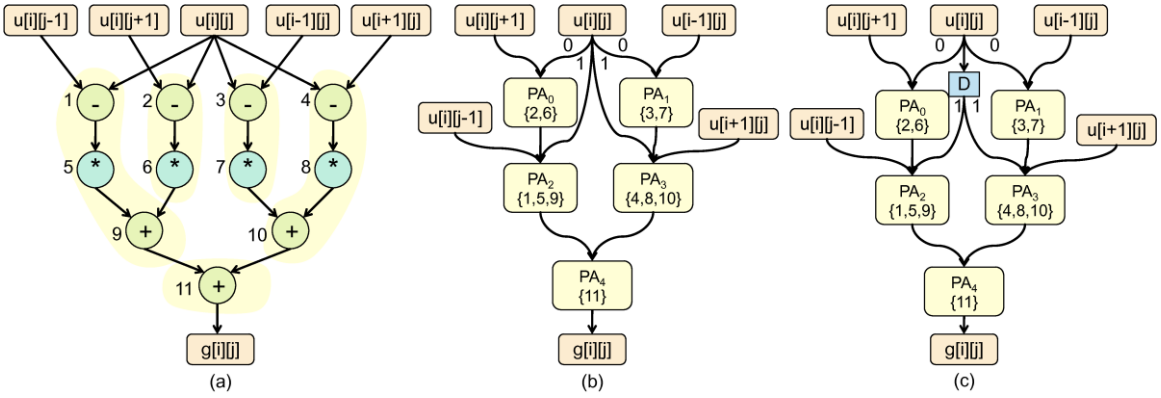


Figure 4-7. (a) Mapping solution II of rician-denoise. (b) Mapping graph of (a). (c) Balanced mapping graph of (a).

4.5 Pipelined PA Mapping

In this section, we introduce an optimal delay unit insertion approach and a corresponding balanced PA mapping algorithm to efficiently map on-chip accelerator resources to a pipelined execution.

4.5.1 Delay Unit Insertion

We consider a PA mapping graph which contains one or multiple undirected cycles. As shown in Figure 4-8, node *s* is the entry node, or *split node*, of a cycle; node m is the exit node, or *merge node*, of a cycle. After delay unit insertion, the lengths of any two paths between a split node and a merge node are the same. Without loss of generality, here we only consider the nodes covered by cycles, denoted by V_C.

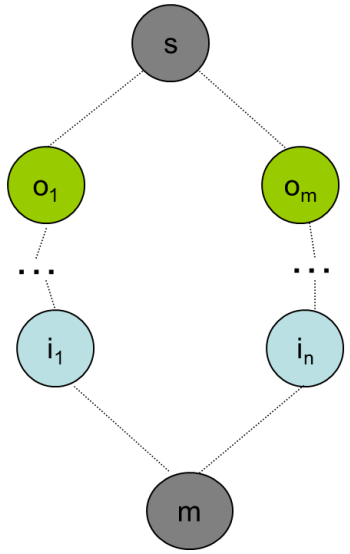


Figure 4-8. An undirected cycle in a data flow graph.

As discussed in Section 4.4, the goal of delay unit insertion is to make the lengths of all paths in a cycle (between the split node and the merge node) equal. One simple but greedy solution is to add delay units at the output ports of the split node to make the length of each path the same. However, this simple heuristic cannot guarantee the optimality of the final solution. As shown in Figure 4-9(a), there are two neighboring undirected cycles – the left one contains node 1, 2, 3, 5, 7 and the right one contains node 1, 3, 4, 5, 6, 8, 9. Following the heuristic approach, one delay unit will be inserted between split node 1 and node 3 to balance the path length in the right cycle. In addition, two delay units need to be inserted between node 1 and node 2 to balance the path lengths in the left cycle. In total three delay units are added to the original data flow graph. This

scheme is not optimal, considering that after introducing one additional delay at path 1->3->5->9 of the right cycle, the length of the longest path (1->3->5->7) in the left cycle also increases. Figure 4-9(b) shows an optimal solution with only two delay units needed to balance the path length in both cycles.

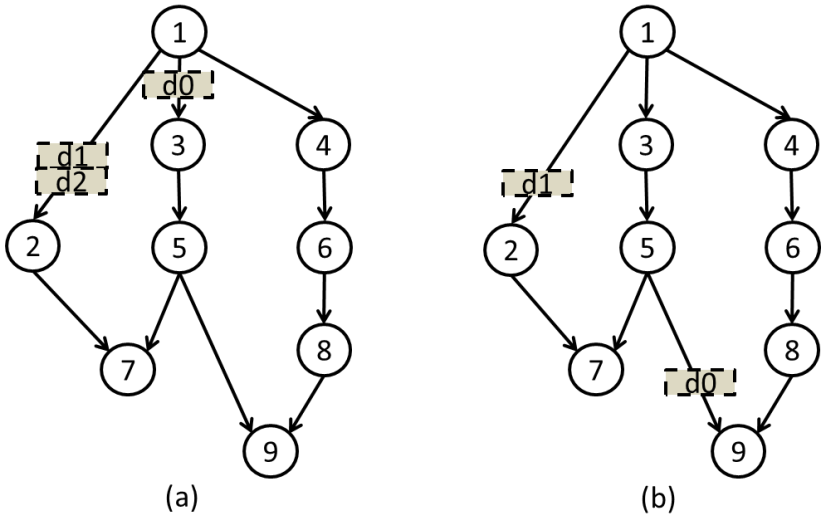


Figure 4-9. (a) A greedy delay unit insertion scheme. (b) An optimal delay unit insertion scheme.

For each cycle node *v*, we associate a label (d_v, d_1, d_2, ... d_n) on *v* and each of its output nodes o_1, o_2, ... o_n (only consider output nodes covered by at least one cycle) to indicate the depth of *v* and its output nodes. Each solution provides a way to organize the delay unit insertion. For each node, the basic constraint is $d_i - d_v > 0$, which implies node depth will increase from an input node to an output node. When node *v* is used by its output node o_i, the result of *v* may need be delayed before it is fed to o_i, and the actually delay offset between *v* and o_i can be calculated by $d_i - d_v$.

When node *v* is used by multiple nodes o_1, o_2, ... o_n, we construct a set $S_v = \{d_1 - d_v, d_2 - d_v, \ldots d_n - d_v\}$, and claim the minimum number of inserted delay units at the output ports of *v* equals $max(S_v) - min(S_v)$, namely the difference between the biggest delay offset and the smallest offset.

Below is the mathematical programming formulation for the delay unit insertion problem,

$$minimize \sum_{v \in V_C} m_v$$

$subject\ to$

$$d_u - d_v - 1 \leq m_v, \qquad v \in V_C \ u \in V_{output(i)} \ \& \ u \in V_C \quad (1)$$

$$d_u - d_w \leq m_v, \qquad v \in V_C \ u,w \in V_{output(i)} \ \& \ u,w \in V_C \quad (2)$$

$$d_i, \ m_i \geq 0, \qquad i \in V_C \ (3)$$

Here we introduce a variable m_v for node *v*, which represents the maximal delay offset difference among *v*'s output nodes. Using the above model, the problem is solved using a linear programming solver as the underlying constraint matrix is totally unimodular. It can be solved optimally in polynomial time [107].

4.5.2 Balanced PA Mapping

In this section, we introduce a branch-and-bound based optimal PA mapping algorithm with balanced path delay and smallest mapping size. Here the size of a mapping solution is defined to be the total area of the balanced mapping graph, including the area of both PA and delay units. The mapping size metric can also be defined as *num(PA) + α*num(delay_unit)* Here the value α is set to *unit_area(delay_unit)/unit_area(PA)*. We can also set α to be *io(delay_unit)/io(PA)*, which equals the ratio between the number of IO ports of each delay unit and each PA. In this case, this metric measures the pressure on the interconnect design imposed by the corresponding PA mapping solution.

As shown in Algorithm 4-1, if the second parameter decision is true, the corresponding PA candidate will be selected in the current solution, otherwise not (as shown in lines 13-15). In this case all the possible combinations will be evaluated to obtain the optimal solution. When the entire data flow graph is covered after adding a new PA candidate, the newly generated mapping solution will be compared to the current optimal solution. If it turns out to be better, the current optimal solution will be updated in line 18.

In order to efficiently prune the searching space, we've developed a pruning technique

combining two metrics (line 26): (1) PA candidates are added in the decreasing order of their size. In this case, after the i^{th} PA candidate with size s_i has been selected, at least $\lceil N/s_i \rceil$ PA candidates are needed to cover the remaining data flow graph, where N equals the number of uncovered nodes. (2) With d delay units generated in the current partial mapping solution (Figure 4-10), at least d delay units are needed to balance the path delay difference in the corresponding complete mapping graph.

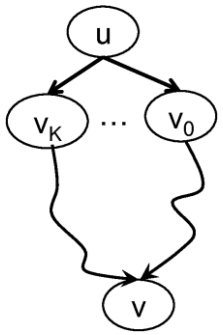


Figure 4-10. A partial mapping graph.

Algorithm 4-1 Balanced PA Mapping Algorithm

1: $G(V, E)$: input data flow graph

2: P: a set of PA candidates $P_1(V_1, E_1)$, ..., $P_N(V_N, E_N)$

3: S: 0

4: II: target throughput

5:

6: Procedure *balanced-PA-mapping()*

7: optimal_sol = *greedy-sol-gen*();

8: sort PA candidates in P in decreasing order of size

9: *balanced-cover*(0, true);

10: *balanced-cover*(0, false);

11:

12: Procedure *balance-cover(i, decision)*

13: if decision = true then

14: add P_i in S

15: if S covers G then

16: if |S| + alpha*total_delay_unit < optimal_sol then

17: optimal_sol = |S| + alpha*total_delay_unit;

18: end if

19: return;

20: end if

21: end if

22: if i+1 > |P| then

23: return;

24: end if

25: if $|S|+\left\lceil \frac{\#uncovered\ nodes}{s_i} \right\rceil$ + alpha*total_delay_unit $\geq$ optimal_sol then

26: return;

27: end if

28: *balanced-cover*(i+1, true);

29: *balanced-cover*(i+1, false);

30: if decision = true then

31: remove P_i from S

32: end if

4.6 Experimental Results

4.6.1 Experiment Setup

We evaluate the proposed maximal PA compilation flow on 10 computation-intensive applications from widely known benchmark suites and computing domains. The testcases include three benchmarks from the *SPEC*2006 suite [70] (*calculix*, *povray* and *bwaves*), five applications from the image processing domain [41] (*gradient*, *registration*, *rician-denoise*, *segmentation* and *edge sobel*), and two applications from the *Rodinia* benchmark suite [13] (*leukocyte* and *cfd*), which is designed for heterogeneous computer systems with accelerators.

Our PA compilation flow is implemented with the LLVM compiler infrastructure [42]. To further evaluate our compilation flow, we have extended Simics [31] and GEMS[32] and conduct cycle-accurate simulations on a recently-developed pipeflined PA architecture called *CHARM*.

4.6.2 Comparison Results

In this section, we evaluate the proposed balanced PA mapping algorithm (*BPM*) with an optimal PA mapping flow in [60]. In [60] the optimization objective is to minimize PA usage when covering the input kernel graph. We apply Algorithm 4-1 to generate a corresponding balanced map for [60].

Mapping optimality. Figure 4-11(a) shows the comparison results when mapping to a fully pipelined execution (II = 1), including the usage of both PA and delay units. *y*-axis shows the number of PAs and delay units occupied in the corresponding mapping solution.

From the results we can see, by applying balanced PA mapping, the delay unit usage has been significantly reduced in all the benchmarks. On average, BPM requires 32.6% less delay units comparing to [60]. On the other hand, the PA usage in both BPM and [60] are

similar. This means when we set the value of α small, BPM will converge to a solution close to PA-optimal solution. Note that the PA usage in BPM may be larger than that in [60] but with more balanced mapping graph (*gradient*, *povary* and *leuocyte*).

Figure 4-11 (b) shows mapping size of BPM when mapped to a pipelined execution with different II (normalized to corresponding mapping size of [60]). As we can see from the figure, when II increases, the pressure on delay unit usage will be released, therefore the mapping size gap between BPM and [60] becomes smaller. When II is small, such as in a fully pipelined execution, the mapping quality difference between BPM and [60] is the most significant. This demonstrates the usage of BPM targeting pipelined accelerator execution. On average, the proposed flow achieves a 17.7% reduction on the total mapping size, with the maximal reduction up to 27.2% when II = 1.

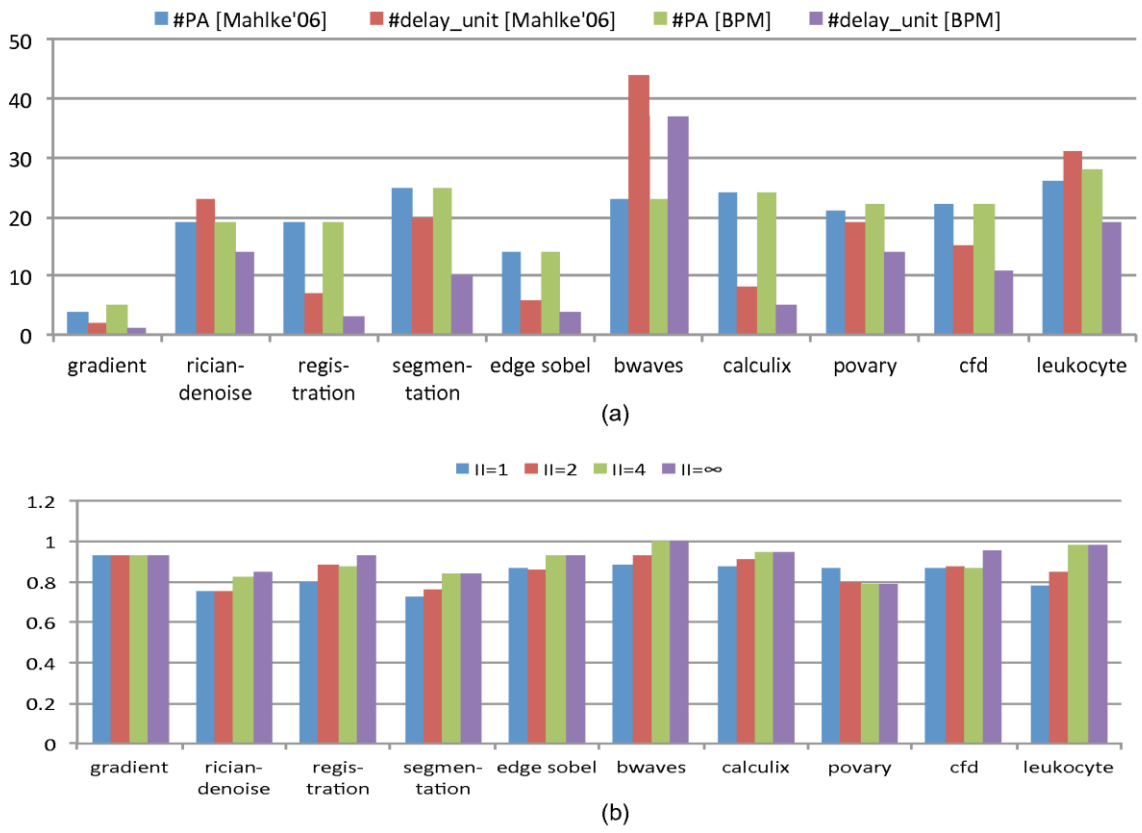


Figure 4-11. (a) Mapping size comparison of BPM and [9] (II = 1) (b) Mapping size comparison of BPM under different II.

Performance. Figure 4-12 shows the comparison of execution time on CHARM platform (II=1). The performance gain comes from improved data level parallelism when the overall mapping size is reduced. Given a limited number of accelerators and delay units, a smaller mapping solution implies higher degree of accelerator duplication to support parallel execution, so that multiple copies of accelerators can execute at independent loop iteration space. In CHARM, *dummy* PAs which are used to route data and do not perform any computation are inserted as delay units. On average, the overall performance gain of *BPM* is 23.6% over [60].

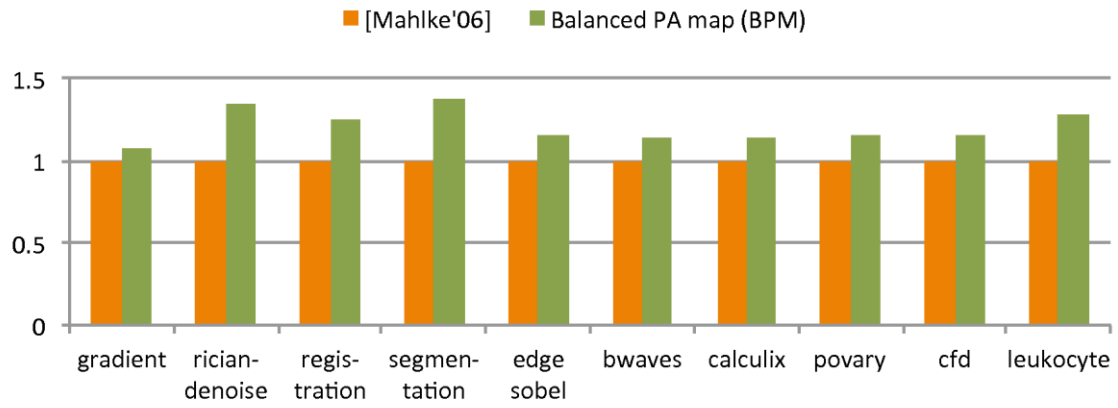


Figure 4-12. Performance comparison.

4.7 Conclusion

PA-rich platforms and full pipelining have been considered closely to provide high performance and power efficiency. On the other hand, it also brings a number of challenges on compilers to generate high-quality acceleration codes. In this chapter we discuss the impact of pipeline II on the resource usage in a pipelined PA system. An optimal PA mapping algorithm is proposed to map input programs onto a target pipelined execution. The proposed flow shows significant improvements in terms of mapping quality and system performance.

Chapter 5. Communication Optimization for Software-Controlled Memories

Multi-level software-controlled memories (SCM) have been extensively utilized in heterogeneous embedded systems. The knowledge of data access pattern enables the opportunity for compile-time communication optimizations, which can be applied to different SCM levels to hide memory access latency and improve bandwidth utilization. In this chapter we quantify the impact of data reuse pattern on both L1 and shared last-level (LL) SCM management. We propose a reuse-aware data movement scheme for multi-level SCMs. 31.2% performance and energy improvements are observed with L1 SCM prefetching. The host-accelerator data transfers are reduced by 25% comparing to previous work [102].

5.1 Introduction

As discussed in Chapter 3, modern high-performance platforms are equipped with abundant computing elements. In order to fully utilize the available computing resources, communication optimizations become major challenges for designers. The inclusion of cache is one traditional way for general-purpose core to manage data movement. However, the hardware-controlled feature of cache makes it difficult to make a customized decision based on program behavior. As an alternative, software controlled memories (e.g. scratchpad memory or SPM) have been widely used in embedded systems and commercial high-performance processors, such as IBM's Cell processor and NVIDIA's GPUs. Programmers can tune the software manually or through special compiler support to manage SCM explicitly and control data movement in a more predictable way. Figure 5-1 shows a heterogeneous parallel architecture with general-purpose processor core (e.g. Intel Xeon) and fast hardware accelerator cores (e.g. FPGA or GPU). The accelerator cores sit beyond two levels of SCMs (private L1 SCM

and shared L2 SCM).

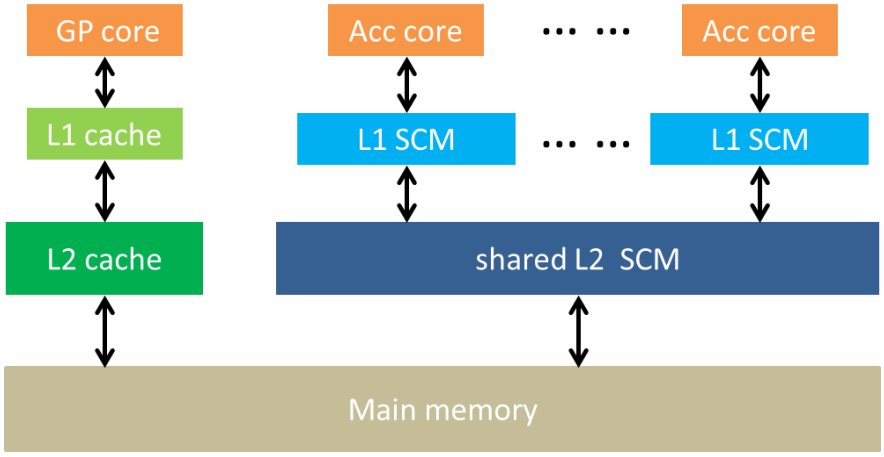


Figure 5-1. Two-level SCM-based heterogeneous platform.

A number of work addressing the compiler support for efficient SCM management have been developed. The allocation scheme in SCM can be divided into two categories. The first category targets L1 (on-chip) SCM allocation. To hide access latency to lower level memories, L1 SCM can be utilized as a prefetch buffer with explicit control over data replacement policy. Compared with conventional cache prefetching, SCM-based prefetching can avoid the scenario in which the data evicted from cache by the newly prefetched data is still "alive," i.e., will be accessed frequently in the near future. However, limited attention has been given to SCM prefetching in the literature. For example, the management scheme in [72] only includes initial prefetching operations with no further analysis of prefetching for dynamic data transfers, and program execution may stall due to late SPM buffer update. In fact, prefetching too late to hide the memory access latency will harm the overall performance, while prefetching too early will put stress on the required SPM size to accommodate those data before their first access. The work in [80] prefetches the entire array into scratchpad memory (or SPM, one type of L1 SCM) before its access with the assumption that the entire array can fit into SPM. However, this is usually not the case, such as scientific applications with large input array. In [73], the direct buffers in Cell's local store (SPM) are utilized to support data

prefetching with runtime library support. In [81], array prefetching in SPM is managed through Markov-chain-based prediction. In [82], SPM is used as a prefetch buffer for video applications by gradually overwriting old data with new data. One common limitation of those works is that SPM prefetching decisions are made independently without considering possible data reuse pattern. For example, [73] and [81] only focus on applications without regular memory reuse, and the scheme proposed in [82] works only for streaming applications. There exist some unified prefetching and reuse schemes for cache. For example, prefetching instructions in [83] are issued only for the memory references with high probability to be a miss. However, since the work targets a normal cache, the compiler does not have explicit control over data eviction, and cache pollution may still occur. Besides, since the cache block to store the prefetched data is determined by hardware, data layout and eviction set selection are not considered in this chapter. The same problem also exists in other cache prefetching work such as [84] and [85]; hence, those works cannot be directly applied to SCM prefetching.

The second category targets last level SCM management to balance the low bandwidth from main memory, which can be further divided into static allocation and dynamic allocation. In static allocation schemes, data layout in SCM is determined at compile time and will remain fixed throughout program execution. Examples of static SCM allocation schemes include [74], [75], [76] and [77]. Compared with a static scheme, dynamic allocation allows SPM data transfers during execution and hence can better accommodate run-time program requirements. For example, the work in [78] applies loop and data transformation to efficiently reduce the number of data transfers between SCM and main memory. In [79] a compiler-driven approach is presented which partitions the program into code regions and the bring-in/swap-out sets for each region are determined heuristically. In [80] the authors propose a dynamic compiler-directed approach to manage SPM through array live-range partitioning and graph coloring. The SPM buffer allocation approach in [72] is based on memory access pattern analysis to improve data

reuse. In [101], a dynamic programming-based data allocation approach is proposed for one program region. Then the optimal solution for each program region is heuristically combined as a global solution. In [102], a highly complex, integer LP formulation is proposed to calculate the global optimal SCM allocation. Those previous works rely on the assumption that all the compute kernels will be executed on the hardware accelerated cores. However, in reality commodity processor cores also serves as a competitive computing resource. For example, data transfers between host memory and global SCM are quite common in CUDA-based GPU programming when two consecutive tasks are allocated to GPU core and processor core, respectively. In addition, neither work supports partial data transfers, namely the entire array has to be treated as one data item. This restriction will lead to inefficient memory utilization at run-time, which also differentiates the last level SCM management problem from the traditional register allocation problem, as discussed below.

Register allocation is one of the most widely studies topics in computer science [103][106]. Its goal is to assign unbounded number of variables to a finite number of machine registers without interfering the lifetime of each variable. Variables which cannot be assigned to machine registers need to be moved to memory, which is called *spilling*. Register allocation has been proved to be NP-complete [104][105], which can be reduced to a graph-coloring problem. The major difference between registration and the last level SCM management discussed here is whether the variable in the problem formulation is divisible. With partial transfer supported, last level SCM management can be approximated as an LP problem and solved in polynomial time. We can also prove the near-optimality of the LP solution, when the total number of candidate arrays is smaller than to the total number of accesses. This is usually the case in most scientific applications.

Targeting the multi-level SCM hierarchy, we propose a reuse-aware L1 SCM prefetching scheme to hide memory access latency and minimize the amount of data transfers from

lower-level memory. The concept of *reuse candidate graph* is introduced to guide prefetching decisions. The proposed scheme is evaluated with cache prefetching, prefetch-only SPM management and a DRDU-generated SPM management scheme [72]. We also develop a task-level-reuse-graph based LL-SCM data movement scheme to minimize the amount of data transfers between heterogeneous computing cores through the slow PCIe bus. Partial array transfers are supported in our approach. An average 25% reduction of host-accelerator data transfers is observed from previous work.

5.2 L1-SCM Management

5.2.1 Impact of Reuse Pattern on SCM Prefetching Efficiency

With explicit control on data movement, we need to identify the prefetched and evicted data sets, which is essential for an SCM prefetching scheme. The basic implementation is to prefetch data *P* iterations earlier than its actual access to hide the load latency, where *P* is the estimated prefetch latency from lower-level memory in terms of loop iteration [86]. In other words, in order to hide the memory access latency, the prefetching instruction of the memory reference set at iteration $i+P+1$ will be issued at iteration i and will replace iteration i's data access set. In this scheme SCM is mainly used as prefetch buffer with size $P+1$. Figure 5-2 shows a simplified loop kernel code of 429.*mcf* from the *SPEC*2006 suite. At iteration i the newly prefetched data for iteration $i+P+1$, replace *cost*[i], *head*[i] and *tail_potential*[i] which will not be re-accessed later.

For programs with regular data reuse patterns, the naive prefetching scheme that simply replaces the data accessed at the current iteration with the data set to be accessed after *P* iterations is not efficient. More specifically, the data to be prefetched or brought in may already reside in SCM. In this case, duplicate prefetching for the same data from conventional memory will increase the number of issued prefetching instructions as well as the total energy consumption. On the other hand, if the data to be re-accessed in the near future is moved out of SCM, those data need to be re-prefetched into SCM before

the next accesses, which also will introduce additional overhead. Figure 5-3(a) shows the kernel code of 401.*bzip*2 from the *SPEC*2006 benchmark. We can see that iteration $i = 8$ is the dividing point where reuse occurs and the prefetching set shrinks by half since the data to be prefetched have already been brought into SCM at an earlier iteration. For example, *fmap*[4] is brought into SCM as *fmap*[i] at iteration $i = 4$, and is re-accessed as *fmap*[$i-4$] at iteration $i = 8$. The iteration space after iteration 8 can be seen as a "stable" region, and the prefetching set for any iteration in that region only contains *fmap*[i].

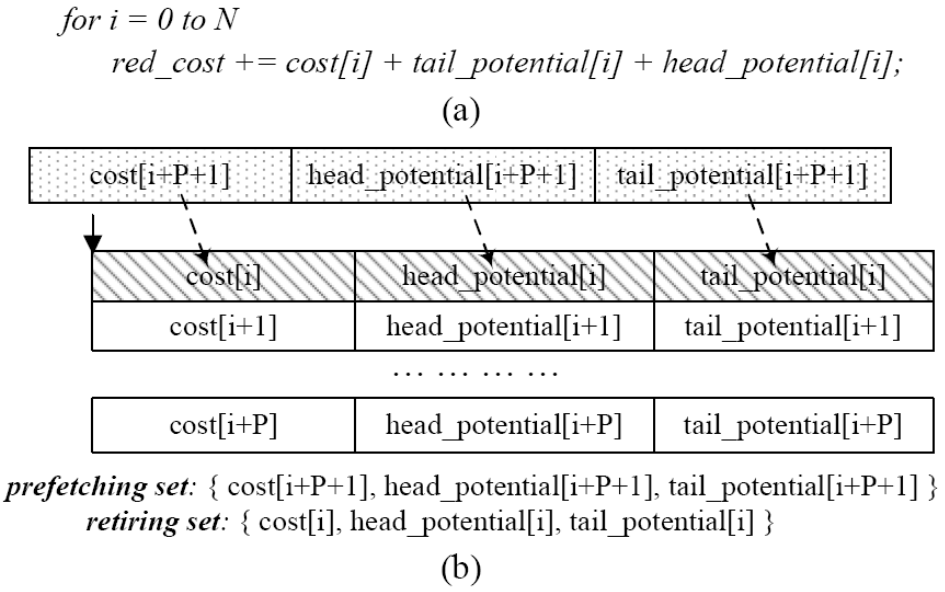


Figure 5-2. (a) Simplified kernel of 429.mcf. (b) SCM management of 429.mcf.

Figure 5-3 further illustrates the difference between prefetching schemes with/without considering reuse patterns. The prefetch-only and reuse-aware SCM prefetching schemes are shown in Figure 5-3 (b)(c). In the reuse-aware scheme, the prefetch set at iteration i only contains one element *fmap*[$i+P+1$], as *fmap*[$i+P-3$] has already been prefetched at iteration $i-4$. The corresponding retiring set only contains *fmap*[$i-4$]. Compared with the prefetch-only scheme, the number of prefetch instructions issued at each iteration is reduced by 2X, and the associated access to lower-level memory will also be reduced accordingly. In Section 5.4 we show that compared to the prefetch-only scheme, the reuse-aware prefetch strategy can achieve up to a 42.6% reduction on energy

consumption and a 39.3% reduction on execution time.

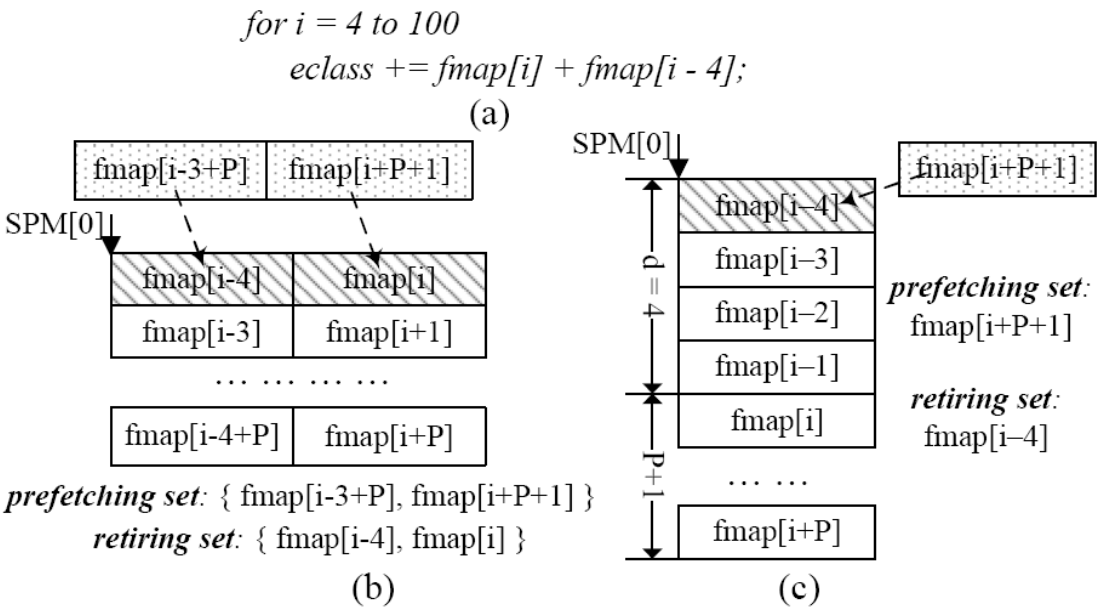


Figure 5-3. (a) Simplified kernel of 401.bzip2. (b) Prefetch-only SCM management of 401.bzip2. (c) Reuse-aware SCM prefetching scheme of 401.bzip2.

5.2.2 RASP: Reuse-Aware SCM Management

5.2.2.1 Preliminaries

Definition 5-1. [16] *Given a normalized n-level loop nest, suppose there is data dependence between memory reference R_1 at iteration $\vec{i}$ and reference R_2 at iteration $\vec{j}$, then the reuse distance vector $\vec{d}$ is defined as a vector of length n such that $\_d(R_1,R_2) = \vec{i} - \vec{j}$.*

Definition 5-2. *A reuse candidate graph is a directed graph $G(V_G,E_G)$ where V_G are array references in a uniformly generated set (UGS)*[1] *and each reuse edge $V_s \rightarrow V_d$ (V_s, $V_d \in V_G$) in EG represents the data dependence between references V_s and V_d with reuse*

[1] A uniformly generated set is a set of affine references of the same array with the same access matrix.

distance vector $\vec{d}(V_s,V_d)$. *Assume* $\vec{d}(V_s,V_d) = (d_1, d_2, \ldots, d_n)$, *the length of reuse edge* $V_s \rightarrow V_d$, *denoted by* $l(V_s,V_d)$, *is defined to be* $\sum_{i=n}^{1}(d_i \cdot \prod_{j=i}^{n} U_{j+1})$, where U_{j+1} *is the upperbound of* j^{th}*-level loop nest* ($U_{n+1} = 1$).

One example of a reuse candidate graph built for the kernel code in the *rician-denoise* [17] application is shown in Figure 5-4. Each vertex in the reuse candidate graph represents one array reference. The directed edge from $u[i+1][j+1]$ to $u[i][j+1]$ with reuse distance vector $\vec{d} = (1,0)$ implies that $u[i][j+1]$ at iteration $\vec{k} + \vec{d}$. will reuse the array element accessed by $u[i+1][j+1]$ at iteration $\vec{k}$, and the length of $V_i \rightarrow V_j$ equals M. If SCM size is large enough to hold data until the next access at $\vec{d}(V_i,V_j)$ iterations later, the corresponding reuse edge $V_i \rightarrow V_j$ will be marked as an ***active*** edge. Notice that the reuse candidate graph is constructed for UGS references; a loop may have more than one reuse candidate graphs. The reuse candidate graph of irregular or non-affine references only contains one vertex, namely the reference itself.

In order to analyze reuse possibility and calculate the number of required data transfers into SCM, *local region* and *reuse region* are defined for each vertex in the reuse candidate graph.

Definition 5-3. *Given reuse candidate graph G with iteration space U, for each reference* V_k *in G, we define the local region of* V_k *to be the iteration subspace in which data accessed by reference* V_k *is prefetched from lower-level memory, denoted by* L_{Vk} *;* V_k*'s reuse region is defined to be the iteration subspace in which access to* V_k *can reuse data stored in SCM for other references, denoted by* R_{Vk} *and* $R_{Vk} = U - L_{Vk}$.

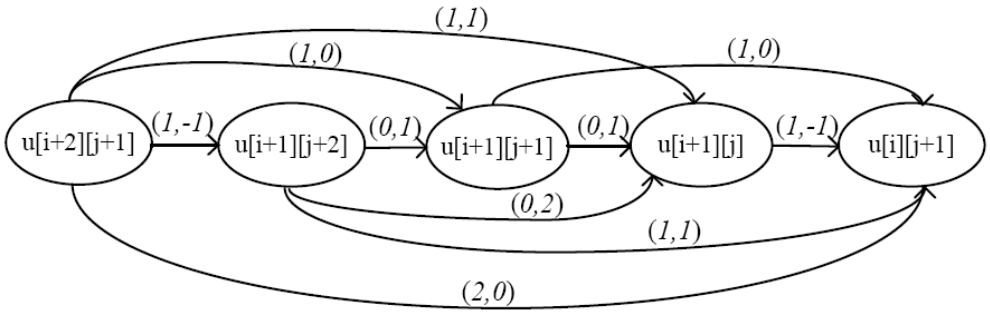
From Definition 5-3 we can conclude that the total size of *local region* of all the vertices in *G* equals the amount of data needed to be brought into SCM, since the data accessed in the *local region* of a given memory reference is prefetched directly from lower-level

memory instead of reusing its parent memory reference in *G*.

Based on the discussion above, we can formulate the *reuse − aware SCM prefetching problem* as follows:

```
for i = 0 to N
    for j = 0 to M
        v[i][j] = u[i+1][j+1] + DT*(u[i][j+1] + u[i+2][j+1] + u[i+1][j] + u[i+1][j+2]);
```

(a)



(b)

Figure 5-4. (a) Normalized kernel loop of rician-denoise. (b) Reuse candidate graph built on (a).

Problem 5-1. *Given the reuse candidate graphs* G *constructed for a loop nest, the maximal SCM size S and the estimated prefetch latency P, select a set of reuse edges to be active and create a SCM buffer for each vertex in G accordingly to hide the access latency P, so that the number of required data transfers from conventional memory hierarchy is minimized, under the constraint that the total size of the allocated SCM buffers cannot exceed S.*

5.2.2.2 SCM Buffer Allocation

In the proposed scheme, one single SCM is seen as a one-dimensional address space and shared among all the inner-loop memory references. In previous work, affine address transformation has been used to map original data addresses to the corresponding address in SCM [88] [89]. The transformed SCM address space is not compact, which will lead to a waste of the limited SCM memory resource.

In the proposed management scheme, each vertex V_k in the reuse candidate graph will be allocated a SCM buffer buf_{Vk} of size L. In order to hide the memory access latency, L has to be larger than the estimated prefetch latency P, as discussed in Section 5.2. In this case array access $A[i]$ in a normalized loop nest will be mapped to an SCM address at $SCM[pos_A+i\%L]$, where *SCM* represents the entire one-dimensional SCM memory space and *posA* is the starting address of *A*'s buffer.

Given reuse candidate graph G and a set of selected active reuse edges E, the SCM buffer size L allocated for each reference V_k in G is set as follows:

$$\mathrm{L} = min\{P + 1 + l(V_k, V_m), |L_{vk}|\} \qquad (1)$$

In Equation 1, $|L_{Vk}|$ is the size of V_k's local region and $V_k \rightarrow V_m$ represents the longest active outgoing edge of V_k. Equation 1 can be derived from the following two cases:

Case 1 : There is no active outgoing edge of V_k in E, namely $l(V_k,V_m)$ equals 0. In this case, V_k will not be reused by any other vertex, hence SCM is merely used as a prefetch buffer of size P+1. However, if the amount of data needed to be brought into SCM, namely $|L_{Vk}|$, is smaller than P+1, SCM buffer size is set to $|L_{Vk}|$.

Case 2 : There exist active outgoing edges of V_k in E, which means V_k will be reused later by other vertices. If $|L_{Vk}| \geq P + 1 + l(V_k,V_m)$, data accessed at iteration $\_i$ of vertex V_k will bestored in SCM until its next access at iteration $\tilde{\imath} + \tilde{d}$ (V_k,V_m) and be replaced with data prefetched at iteration $\tilde{\imath} + \tilde{d}$ (V_k,V_m). Hence the required SCM buffer size equals P+1+$|\tilde{d}$ $(Vk,Vm)|$, namely P+1+$l(V_k,V_m)$; Otherwise, only data in the local region of V_k need to be prefetched into SCM, the allocated SCM buffer size equals $|L_{Vk}|$.

As shown in Figure 2-2(c), the reuse distance vector between $fmap[i-4]$ and $fmap[i]$ is (4); therefore the SCM buffer size for $fmap[i]$ equals P+5. Buffer size for $fmap[i-4]$ is set to 4 (assume $P \geq 4$) since the size of its local region equals 4.

5.2.2.3 Data Transfer Measurement

Theorem 5-1. *Assume that V_i in reuse candidate graph G has no active incoming edges, the number of reduced data transfers by activating reuse edge $V_i \rightarrow V_j$ with reuse distance vector $\tilde{d}(V_i,V_j) = (d_1, d_2, \ldots, d_n)$, equals $\prod_{k=1}^{n}(U_k - |d_k|)$*

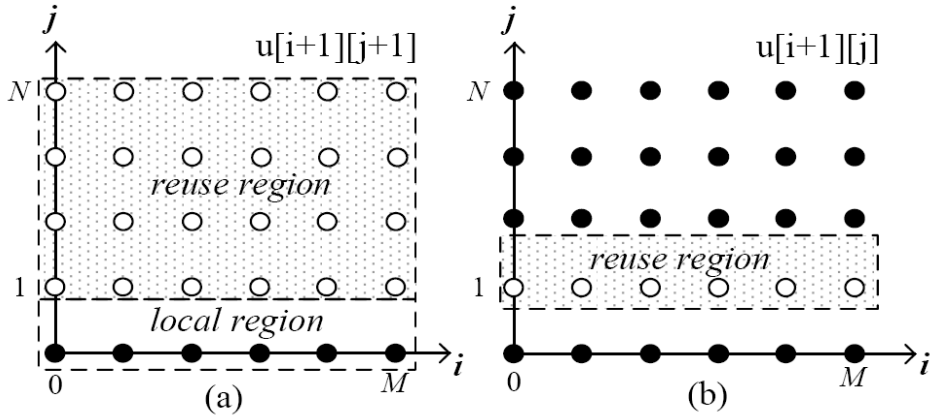


Figure 5-5. (a) Iteration space partition of reference u[i+1][j+1]. (b) Iteration space partition of reference u[i+1][j].

Proof. Two conditions need to be satisfied to ensure that memory reference V_j at iteration $(t_1, t_2,\ldots,t_n)$ can reuse V_i at $(d_1,d_2,\ldots,d_n)$ iterations before: (1) $0 \le t_k \le U_k, \ \forall k \in [1, n]$ (U_i is the upperbound of the k^{th}-level loop); (2) $0 \le t_k - d_k \le U_k, \ \forall k \in [1, n]$. The two conditions are derived from the fact that both the first and second accesses fall into the iteration space. The number of iterations satisfying (1) and (2) is the total number of reuses that occur.

Theorem 5-1 can be used to calculate the number of remaining data transfers given a set of active reuse edges. However, in Figure 2-3(b), suppose edge $u[i+1][j+2] \rightarrow u[i+1][j+1]$ and edge $u[i+1][j+1] \rightarrow u[i+1][j]$ are both selected as active edges, Theorem 5-1 still works for vertex $u[i+1][j+1]$ since $u[i+1][j+2]$ has no active incoming edge, while it is not the case for vertex $u[i+1][j]$. The iteration subspace R, in which access to vertex $u[i+1][j+1]$ can reuse earlier $u[i+1][j+2]$, is shown in Figure 2-4(a) with soft dots.

Vertex $u[i+1][j]$ at iteration $\vec{t}$ will reuse $u[i+1][j+1]$ at iteration $\vec{t}-1$. If iteration $\vec{t}-1$ locates in region *R*, $u[i+1][j]$ needs to go upwards to visit vertex $u[i+1][j+2]$ at iteration $\vec{t}-2$. However, the allocated SCM buffer for vertex $u[i+1][j+2]$ is only $P+2$ which only can hold data of one more iteration; hence the reuse attempt of $u[i+1][j]$ will fail in this case.

For a vertex with active incoming edges, the size of its local region equals the number of required data transfers into SCM. Figure 2-4(b) shows the local and reuse region of $u[i+1][j]$ where reuse along edge $u[i+1][j+1] \rightarrow u[i+1][j]$ is enabled. In general, the local and reuse region of vertex *Vk* can be derived as follows:

Theorem 5-2. *Given reuse candidate graph G with iteration space U, assume the active incoming edge set of vertex* V_k *is* $\{V_{i1} \rightarrow V_k, V_{i2} \rightarrow V_k, ..., V_{in} \rightarrow V_k\}$, V_k*'s reuse region* $R_{Vk} = \{t \mid t \in U \wedge ((t-d(V_{i1}, V_k) \in LV_{i1} \vee t-d(V_{i2}, V_k) \in LV_{i2} ... \vee t-d(V_{in}, V_k) \in LV_{in})\}$; V_k*'s local region* $L_{Vk} = U - R_{Vk}$.

Proof. V_k can reuse the data from the local region of its inputs V_i within the corresponding reuse distance, namely $R_{Vk,Vi} = \{ t \mid t \in U \wedge (t-d(V_i, V_k) \in LV_i\}$. When consider all the inputs $\{V_{i1}, V_{i2}, ..., V_{in}\}$, R_{Vk} *equals the union of each region region* $\{t \mid t \in U \wedge ((t-d(V_{i1}, V_k) \in LV_{i1} \vee t-d(V_{i2}, V_k) \in LV_{i2} ... \vee t-d(V_{in}, V_k) \in LV_{in})\}$.

Theorem 5-2 can be applied to vertices of a given reuse candidate graph in topological order to identify their local and reuse regions, i.e., starting from the root vertex which has no active incoming edge and its local region is the entire iteration space *U*.

5.2.2.4 Reuse-Aware SCM Prefetching Algorithm

In this section we present a reuse-aware SCM prefetching algorithm, namely *RASP*, aimed at hiding memory access latency and minimizing data transfers from lower-level memory. In general, a SCM buffer is allocated to each vertex in the reuse candidate graph, either for pure prefetching or unified prefetching and reuse.

Algorithm 5-1 Reuse-Aware SCM Prefetching (RASP) Algorithm

1: *U :* iteration space of loop nest *l*

2: *G :* reuse candidate graph set constructed for loop nest *l*

3: *E :* set of activated edges in $\mathbb{G}$

4: *S :* maximal size of SCM storage

5: *P :* estimated prefetch latency

6:

7: For all the vertices *v* in *G*, set initial SCM buffer size to be *P*+1 with local region *U*

8: while 1 do

9: traverse all the unactivated edges in *G* and calculate their SCM utilization ratio

10: activate the edge (*u*,*v*) with largest positive utilization ratio under SCM size constraint

11: add edge (*u*,*v*) to *E*

12: add all the edges (*u*,*v'*) to *E* if $l(u, v') \leq l(u,v)$

13: update SCM buffer size of *u*

14: update local/reuse region of *v*, *{v'}* and vertices reachable from *v* and *{ v'}* along edges in $\mathbb{E}$

15: if size of $\mathbb{E}$ remains the same then

16: break;

17: end if

18: end while

To find the active reuse edge sets with minimum required data transfers under SCM size constraint is NP-hard, as one can reduce a Knapsack problem to it. To balance the runtime overhead, we propose a heuristic algorithm to approximate the optimal solution.

As shown in the RASP algorithm, edges in the reuse candidate graphs are activated one by one under the maximal SCM size constraint. Here *activate* edge $u \rightarrow v$ means allocate a SCM buffer for u which is large enough for the corresponding reuse to occur. Hence, when edge $u \rightarrow v$ is activated, all the edges starting at u with a smaller required SCM size, namely a smaller edge length, should also be activated accordingly, as shown in line 12.

The metric used to indicate SCM utilization efficiency is called *SCM utilization ratio*, which equals the ratio of data transfer reduction to the buffer size increment by activating a given reuse edge (u,v). The amount of reduced data transfers after activating edge (u,v) equals the size difference of the local region of vertex v and all the vertices reachable from v along selected active edges, which can be obtained with Theorem 5-2.

Lines 10-13 show that after an edge ending at vertex v has been activated, the local/reuse region updates are applied to vertex v and the vertices reachable from v along edges in the current active set E. The updates for v's downstream vertices are necessary since the reduction of v's local region after activating edge (u, v) has further impact to v's descendants.

Note that the worst-case time complexity of the RASP algorithm is $O(n^4)$, where n is the number of vertices in the reuse candidate graph set.

After the finalization of the activated edge set, the prefetching scheme for each vertex can be determined accordingly. v will be removed from the prefetching set for iterations in its reuse region.

5.3 LL-SCM Management

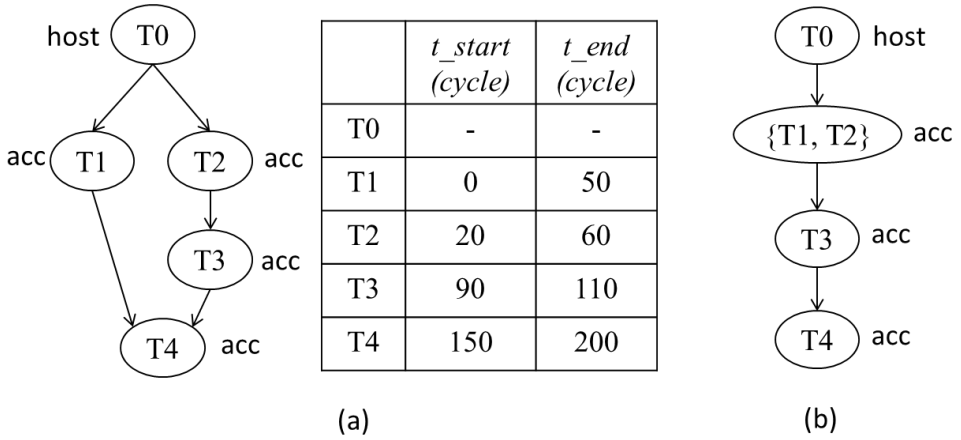
5.3.1 Architecture Model

In modern heterogeneous multi-core systems, hardware accelerator cores usually sit beyond multi-level SCMs. This multi-level SCM design helps to better tradeoff the size and bandwidth differences between different memory levels. In typical embedded processors, the L1 SCM normally consists of fast SRAM memories (or SPM) and

LL-SCM can be either SRAM or DRAM (e.g. FPGA's off-chip memory and GPU's global memory). Figure 5-1 is one example of a two-level SCM-based architecture, in which each hardware core has a private L1 SCM and a shared LL (or L2) SCM. The general purpose core, which is treated as host processor, communicates with hardware accelerator cores through PCIe bus connecting to the shared LL-SCM.

5.3.2 Application Execution Model

Applications are represented with a directed acyclic graph (DAG) *G(V, E)*. Different from the data flow graph depicted in Chapter 2, each node in *V* represents a task which is executed either on the host processor or on one of the hardware accelerator cores. Here we restrict each task to be executed on exactly one core. A directed edge $u_0 \rightarrow u_1$ in *E* represents a precedence dependency between two tasks represents by the u_0 and u_1. Each task is also associated with three parameters *t_start*, *t_exec* and *core_type* – *t_start(v)* represents the scheduled starting time for task *v*; *t_exec(v, k)* represents the execution time of task *v* on core *k*; *core_type(v)* represents the processor or accelerator core task *v* is mapped to. Here we call the tasks mapped to host cores *host tasks*, and the tasks mapped to accelerator cores *accelerator tasks*. With *t_start* and *t_exec*, the completion time *t_end* for task *v* can be easily calculated by *t_start(v) + t_exec(v, core_type(v))*.



	t_start (cycle)	*t_end (cycle)*
T0	-	-
T1	0	50
T2	20	60
T3	90	110
T4	150	200

Figure 5-6. (a) Example of task graph. (b) Merged task graph.

In the remaining discussion, we assume that the parallel execution is not allowed between

host and accelerator cores, which is common in practice. Figure 5-6(a) shows an example task graph with 5 task nodes, in which *T0* is mapped to host core, and *T1~T4* are mapped to accelerator cores. When two tasks mapped to the same type of cores have overlapping lifetime [*t_start*, *t_end*], a merged task node will be generated and replace the original two nodes. As shown in Figure 5-6, node *T1* and *T2* are merged into one node in task graph Figure 5-6(b) since they have overlapping lifetime on accelerator cores. This guarantees the sequential execution order in the final task graph.

5.3.3 Task-Level-Reuse-Graph Based LL-SCM Management

5.3.3.1 Task-Level Reuse Graph (TLRG)

Given the merged task graph and pre-determined task mapping decisions, a data structure called *task-level reuse graph* (TLRG) is created for each array accessed in the program.

Definition 5-3. *A task level reuse graph for array d is a directed graph G*(*V,E*) *where* $v_i \in V$ *represents an accelerator task* v_i *which accesses array d; the directed edge* $v_i \rightarrow v_j$ (v_i, $v_j \in V$) *in E represents a RAR or RAW dependence on array d between neighboring accelerator tasks* v_i *and* v_j.

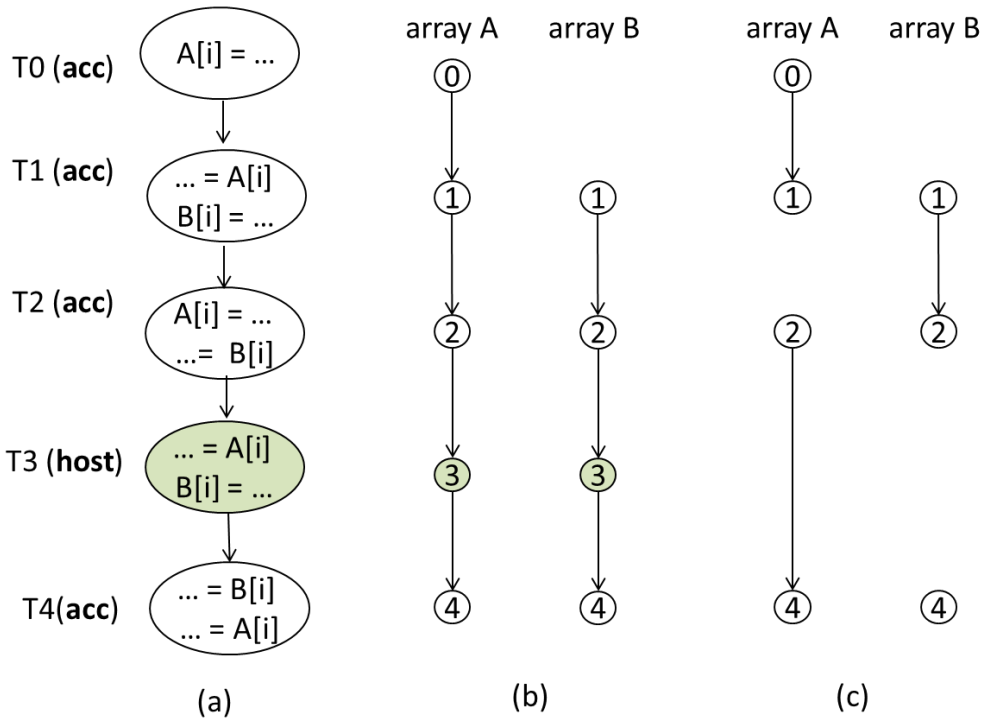


Figure 5-7. (a) Example task graph. (b) Task level data dependency graphs for array A and B. (c) Task level reuse graphs for array A and B.

Figure 5-7(a) is an example of the merged task graph, in which each node is labeled with its topological id. Figure 5-7(b) shows the corresponding data dependency graphs for array *A* and *B*. As one can see, array *A* is accessed by tasks *T0~T4* with RAW, WAR, RAW and RAR dependencies associated with each edge. Similarly, array *B* is accessed by tasks *T2~T5* with RAW, WAR and RAW dependencies. By the definition of TLRG, Figure 5-7(b) can be reduced to Figure 5-7(c), in which only accelerator task nodes and read-after-read/write dependencies are maintained. The other set of dependency edges (WAW and WAR) are removed in TLRG since there is no need to hold an array in LL-SCM if it will be updated in its next access. One thing to note is that since array *A* is not modified by host task *T3*, the original RAR dependency edge from *T2* to *T3* is propagated to *T4* in Figure 5-7 (c), namely the next accelerator task node reading *A*.

5.3.3.2 Problem Statement

When executing accelerator task *k*, there are two sets of data or array items D_k residing in LL-SCM: (i) arrays which are accessed by task *k*, denoted by D_k^1; (ii) arrays which will be read by another accelerator task executed later, denoted by D_k^2. Here we assume for each accelerator task, D_k^1 can always fits in LL-SCM. In general, program transformations (e.g. loop distribution or tiling) can be applied to partition the task node into smaller sub-tasks, which is beyond the scope of this discussion.

If we look at the sources of host-accelerator communication, the first one is due to the limited LL-SCM size. There is no guarantee that the second set of arrays D_k^2 can be completely kept in LL-SCM until its next reuse. If D_k exceeds the LL-SCM size limit, PCIe-based data transfers will be incurred to migrate a subset of D_k to host memory; the second source of host-accelerator data exchange is the existence of RAW dependency between accelerator and host tasks (e.g. an array read by an accelerator task is overwritten by an earlier host task). In this case, explicit data transfers between host memory and LL-SCM are needed in order to keep memory coherency between the two cores, which is an unavoidable overhead associated with the a specific task mapping result.

Theoreom 5-1. *Given a task graph G(V, E) associated with a task mapping decision, the amount of host-accelerator data transfers incurred by true dependency between host and accelerator tasks remains the same when LL-SCM management scheme changes.*

Proof is obvious. Since the task mapping has been fixed, no matter how LL-SCM management changes, RAW dependency must be satisfied. Therefore the data migration due to RAW dependencies cannot be eliminated or saved. In other words, only the second set D_k^2 can be considered for communication overhead reduction.

Based on the discussion above, the LL-SCM management problem can be formulated as follows:

Problem 5-2. *Given task level reuse graphs {$TLRG_i$} for a set of arrays {d_i} and the size constraint for LL-SCM, construct data transfer scheme for each array d_i such that: (1) when d_i is accessed by an accelerator task, it is in LL-SCM. (2) the total amount of data stored in LL-SCM at each task does not exceed the LL-SCM size constraint. (3) the total amount of data exchange between LL-SCM and host memory is minimized.*

5.3.3.3 Mathematical Formulation

For each array *i* at task node v_j in $TLRG_i$, we introduce two variables $mi_{i,vj}$ and $mo_{i,vj}$. $mi_{i,vj}$ represents the portion of data which are moved from host memory to LL-SCM for array *i* at task v_j. Similarly, $mo_{i,vj}$ represents the portion of array *i* which are moved from LL-SCM to host memory at task v_j. The total amount of data moved in and out at task v_j can be formulated as $\sum_{i\in D}(mi_{i,v_j} + mo_{i,v_j}) \cdot size_i$, where $size_i$ represents the size of array *I*, and *D* represents the entire set of arrays accessed in the program. Below is the mathematical programming formulation for the data transfer minimization problem:

$$minimize \sum_{i\in D} \sum_{v_j \in TLRG_i} \left(mi_{i,v_j} + mo_{i,v_j}\right) \cdot size_i$$

subject to

$$mi_{i,v_j} \geq 0, \quad i \in D,\ v_j \in TLRG_i \quad (1)$$

$$mo_{i,v_j} \geq 0, \quad i \in D,\ v_j \in TLRG_i \quad (2)$$

$$mo_{i,v_j} - mi_{i,v_{j+1}} = 0, \quad i \in D,\ v_j \to v_{j+1} \in TLRG_i \quad (3)$$

$$\sum_{i\in D_k^1} size_i + \sum_{i\in D_k^2} \left(1 - mo_{i,v_j}\right) \cdot size_i \leq S_{LLSCM}, \quad v_j \to v_{j+1} \in TLRG_i \text{ and } k \in (v_j, v_{j+1}) \quad (4)$$

$$mi_{i,v_j} = 1, \quad if\ v_j \in TLRG_i\ has\ no\ incoming\ edge. \quad (5)$$

$$mi_{i,v_j} = 0, \quad if\ v_{j-1} \to v_j \in TLRG_i\ and\ v_j = v_{j-1} + 1 \quad (6)$$

$$mo_{i,v_j} = 0, \quad if\ v_j \in TLRG_i\ has\ no\ outgoing\ edge,$$

$$or\ if\ v_j \rightarrow v_{j+1} \in TLRG_i\ \ and\ v_{j+1} =\ v_j + 1 \quad (7)$$

The first two constraints impose non-negativity of the amount of data transfers at each task. The third constraint imposes integrity of data accessed at a specific task, which can be utilized to further simplify the LP formulation by merging *mi* and *mo* variables along TLRG each edge into one. The fourth constraint imposes size constraint at each task *k*.

Theorem 5-3. *Given array set D and the corresponding task level reuse graph set {TLRG$_i$(V$_i$, E$_i$)}, there exists one optimal solution for Problem 5-2, in which for each array d$_i$, the host-accelerator data exchange only occurs at task node v ∈V$_i$.*

Proof: Assuming in there exists one optimal data transfer solution of Problem 5-2, in which host-accelerator data transfers occur for array d_i at a non-TLRG task node *k*. We use $trans_k$. to denote the data transfer at task node *k*. There are three possible scenarios:

(1) $k < v_0$, namely the data transfer happens before the first accelerator tasks starts. In this case, we can find an equivalent solution by delaying $trans_k$ to $trans_{v0}$.

(2) $v_j < k < v_{j+1}$, where $0 <= j < |V_i|-1$, namely the data transfer happens between two task level reuse nodes v_j and v_{j+1}. If $trans_k$ is host-to-accelerator data transfer, we can postpone it to node v_{j+1}; otherwise, $trans_k$ can be moved to task node v_j. In both cases, the data transfer incurred by the transformed scheme is equivalent to the original one.

(3) $k > v_{|Vi|-1}$, namely the data transfer happens after the last accelerator tasks ends. In this case, we can find an equivalent solution by making $trans_k$ happen immediately after task $v_{|Vi|-1}$ completes.

From Theorem 5-3 we can see that the host-to-accelerator and accelerator-to-host data transfer of the same array at two neighboring TLRG nodes always have the same amount, as depicted in the third set of constraints. On the other hand, by utilizing the concept of TLRG, the problem size of the LP formulation can be significantly reduced comparing to previous work [102] in which data movement is enabled at every task.

The fourth set of constraints can be explained by Figure 5-7. After executing task *T2*, the

amount of array *A* storing in LL-SCM equals $(1 - mo_{A,2}) \cdot size_A$ until its next reuse at task T4. Therefore during this period the LL-SCM space occupied by array *A* remains to be $(1 - mo_{A,2}) \cdot size_A$. In the fourth set of constraints, the first term calculates the memory space occupied by the arrays accessed at task *k*. The second term represents the set of arrays stored in LL-SCM for future reuse.

In addition to TLRG, the problem size of the LP formulation can be further reduced by removing variables created for the 'boundary' nodes, as shown in (5)(7) and consecutively-executed task nodes in TLRG in (6)(7). The associated *mi* and *mo* variables for those nodes can be eliminated safely, since the data transfer behavior is pre-determined and it has no influence on the solution optimality.

Constraint (5) means if task v_j is the first accelerator access of the most up-to-date array *i*, array *i* need to be migrated from host memory to LL-SCM. Similar case is for (7). If two consecutive tasks are both mapped to accelerator cores and access the same array, no host-accelerator migration of that array will happen between these two tasks, as depicted in (6) and (7).

5.3.3.4 Rounding and Optimality Discussion

Note that the amount of optimal data transfer obtained for the LP formulation in 5.3.3.3 may not be an integer value. For example, if at a specific task the value of move-out variable (*mo*) for an array of size 100 equals 0.881, the optimal data movement scheme cannot be strictly followed. It is impossible to transfer a non-integer amount in practice. To generate a practical solution, after an optimal solution is obtained for the LP formulation, rounding is conducted on the corresponding data transfer amount – For array *i*, the amount of outgoing data transfers at task node v_j in $TLRG_i$ is rounded to $\lceil mo_{i,v_j} \cdot size_i \rceil$, which guarantees the size constraint of LL-SCM will not be violated at node v_j. Enforced by constraint (3), the same amount of incoming data transfers will be incurred

at node v_{j+1}.

Theorem 5-4. *Assume the amount of data transfers in the rounding solution equals* $d_{rounding}$*, and the optimal solution equals* d_{opt}*, we have* $d_{rounding} \leq d_{opt} + 2 \cdot \sum_{i \in D} |V_{TLRG_i}|$*, where* $|V_{TLRG_i}|$ *is the number of nodes in array i's task level reuse graph.*

Proof: $2 \cdot \sum_{i \in D} |V_{TLRG_i}|$ equals the maximal possible amount of additional data transferred incurred by rounding.

Theorem 5-4 demonstrates the optimality of the rounded LP solution. When the number of arrays accessed is not large, which is usually the case in real time applications, the rounded LP solution is fairly close to the optimal solution.

5.4 Experiment Results

5.4.1 Experiment Setup

We evaluate the proposed L1-SCM management scheme on a simulation platform built upon Simics [90] with GEMS [91]. Omega library [22] is used for memory reuse analysis. The energy results are obtained with HP McPAT tool [92]. Table 5-1 shows the architecture parameters used in our model.

The first-level memory is partitioned into programmer-transparent cache and compiler-managed SCM memory space at the ratio of 1:3, 2:2 or 3:1, which is close to Fermi's 3:1 and 1:3 configuration points. An in-flight counter is added for each SCM block to check whether prefetching has finished or not. This can ensure the correctness of program functionality.

Table 5-1. Architecture parameters

Processor Core	Sun UltraSPARC III Cu processor core
L1 SCM	32KB, 64 byte block, 2 cycle access latency
L2 SCM	512MB, 64 byte block, 20 cycle access latency

Main Memory	4GB, 320 cycle access latency

The proposed L1-SCM management scheme is compared with three reference points in our experiments. The first reference point is cache prefetching with the entire L1 memory allocated to conventional cache [93]. The second reference point is a hybrid memory system in which SCM is used as a prefetch buffer, following the prefetch-only scheme discussed in Section 5.1. The third reference point is the DRDU generated SCM management scheme [72].

The LL-SCM management is compared with two reference points in our experiments. The first reference point is the heuristic algorithm proposed in [101] which generates local optimal solution for each task using dynamic programming. A greedy algorithm is developed to create the global LL-SCM management scheme. The second reference point is an ILP-based approach proposed in [102], which evaluates all the possible data movement decisions at each task. Neither of these two works supports fractional data movement.

5.4.2 Comparison Results

5.4.2.1 Comparison on L1-SCM management

Performance. Figure 5-8 shows the overall performance comparison results for the eight benchmark kernels, where the four bars correspond to the cache prefetching scheme, SCM prefetch-only scheme, DRDU and RASP scheme. As shown in the figure, RASP exhibits speedup ranging from 9.6% to 44.3% in six out of eight applications, compared to the cache-only scheme. In *hmmer* a slight performance degradation occurs. The reason is that the access pattern in *hmmer* has strong data locality which can be captured well by conventional cache architecture. In addition, cache pollution is less likely to happen here since the next access will occur soon. In this case the extra instruction overhead of calculating the SCM address cannot be offset by the small amount of reduced data transfers.

When compared to the SCM prefetch-only scheme, *mcf* and *lbm* are two special cases in which data access patterns are either random or streaming. In this case the generated SCM prefetch-only scheme is exactly the same as RASP. This also explains the small performance difference, when compared to DRDU in these two applications. For most of the remaining applications, the performance improvement over the SCM prefetch-only scheme is less than that over cache, since cache pollution is avoided in the SCM prefetching scheme.

On average RASP has achieved a 15.9%, 12.9% and 18.5% performance improvement over cache prefetching, prefetch-only SCM management and DRDU results. The corresponding maximal gains are 44.3%, 39.3% and 32.8%, respectively.

Memory Access Latency. The comparison of normalized memory access latency is shown in Figure 5-9. We can see that the memory access latency reduction in RASP is larger than the performance improvement for most of the test cases, when compared to cache prefetch and the SCM prefetch-only scheme. This can be explained by the instruction overhead introduced by explicit SCM management. In summary, RASP has shown an average 31.6% memory access latency reduction over the cache-only case, a 26.4% reduction over the SCM prefetch-only case and an average 19.5% reduction over DRDU result. The corresponding maximal gains are 59.6%, 46.2% and 50.3%, respectively.

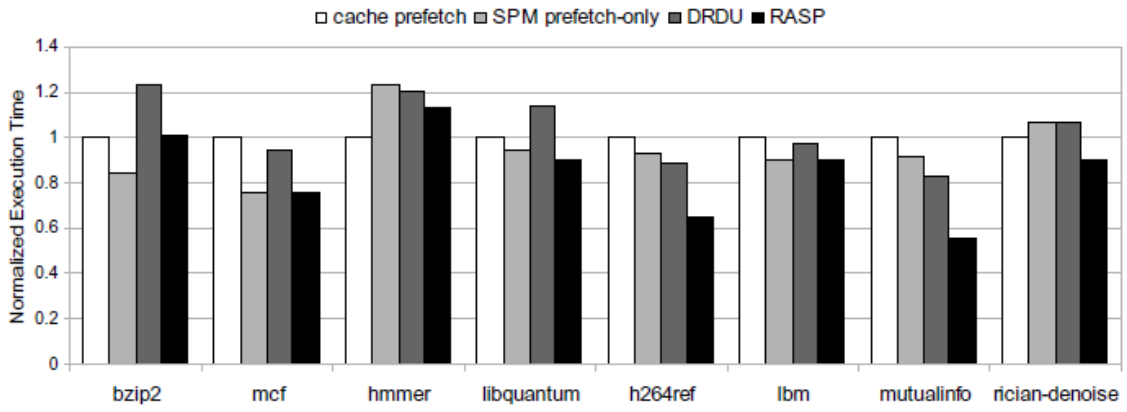


Figure 5-8. Comparison of execution time.

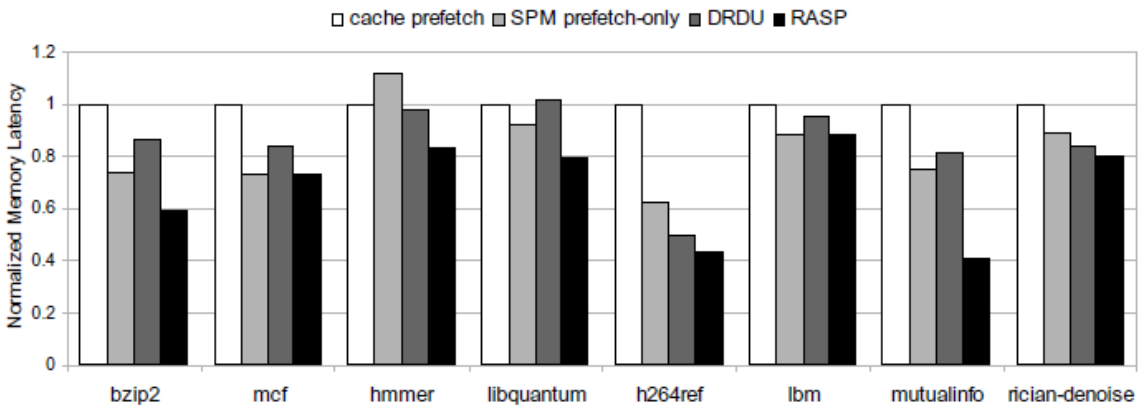


Figure 5-9. Comparison of memory access latency.

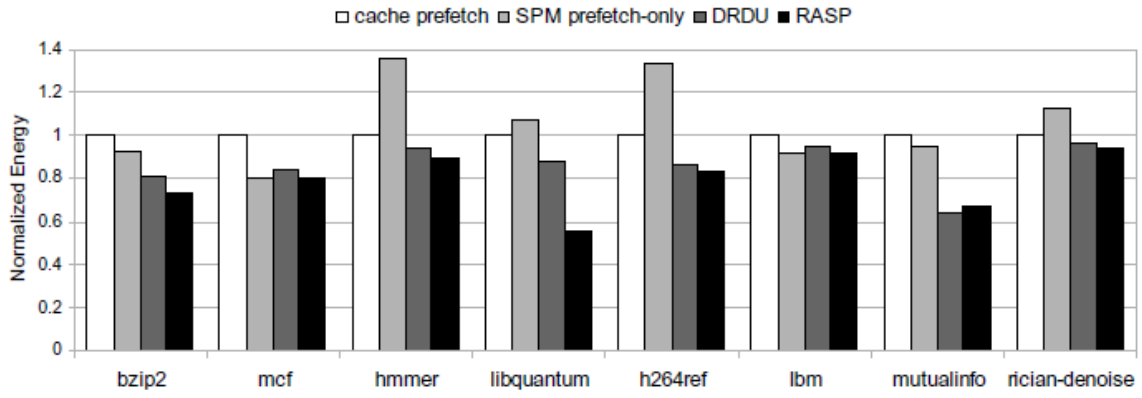


Figure 5-10. Comparison of energy consumption.

Energy Consumption. Figure 5-10 shows the energy consumption comparison among the four schemes. Since cache can take advantage of the existing data locality in the program and save further access to lower-level memory, a 6% energy decrease of cache prefetching over the SCM prefetch-only scheme is observed.

On the other hand, up to 44.7%, 42.6% and 27.7% energy gains are achieved by RASP over the other three schemes. The reasons include the intrinsic less energy consumption for SCM access as well as the reduced number of accesses to lower-level memory by efficiently utilizing the reuse pattern with SCM. The average energy consumption reduction of RASP is 22% and 31.2% over cache and the SCM prefetch-only case, respectively. The average 10% energy reduction over DRDU comes from the improved execution time, as well as the reduced SCM data transfers.

5.4.2.2 *Comparison on LL-SCM management*

The proposed LL-SCM management scheme is evaluated on a 3-phase medical imaging pipeline and a 11-phase in-house neuron network training benchmark. The medical imaging pipeline consists of *rician-denoise*, *registration* followed by *segmentation*. In total there are 10 tasks with 11 arrays involved, including two accelerator tasks in denoise, three accelerator tasks in registration, two host tasks and three accelerator tasks in segmentation. The neuron network training benchmark contains 10 accelerator tasks and 1 host task with 4 arrays as optimization candidates.

Table 5-2. Comparison on problem size.

	Medical image pipeline		Neuron network training	
	#variables	*#constraints*	*#variables*	*#constraints*
ILP formulation [102]	30	106	40	148
LP formulation	6	13	6	16

Problem size. In the ILP-based approach [102], three 0-1 variables are created for each array at every task phase, indicating whether an array is moved into LL-SCM before the current task, whether an array is evicted from LL-SCM after the current task, and whether an array resides in LL-SCM at current task, respectively. In total 30 binary variables are created for the medical pipeline and 40 variables are created for neuron network training. On the other hand, by utilizing the concept of TLRG, the total number of optimization variables in the proposed LP formulation is 6 in both cases, which show a 5X and a 6.8X reduction from [102], respectively. Table 5-2 also presents the number of constraints in the problem formulation, as we can see, the proposed LP formulation has a 6X and a 9.5X reduction comparing to [102], respectively.

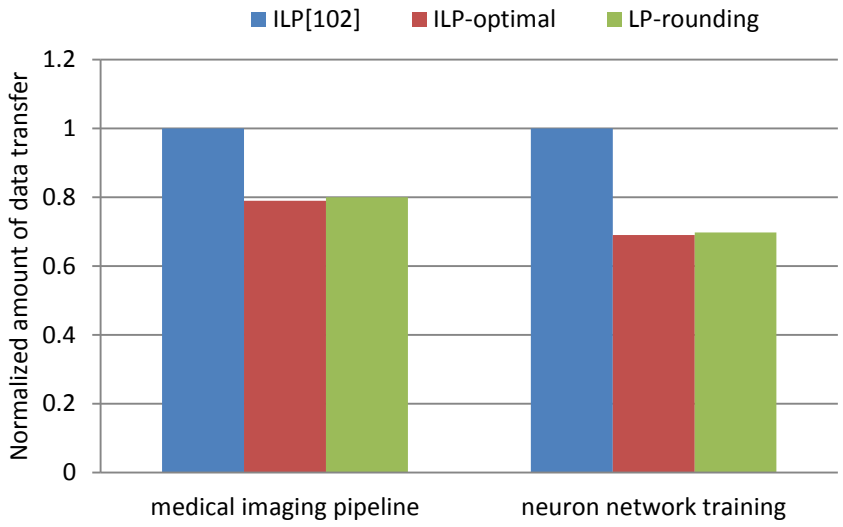


Figure 5-11. Comparison of host-accelerator communication.

Optimality. To illustrate the optimality of the proposed LP solution, we compare the amount of host-accelerator data transfer with [102] (ILP, no partial data transfer support) as well as an optimal ILP-based formulation considering partial data transfer. As shown in Figure 5-11, we observed a 20% and 30.2% data transfer reduction over [102] in the target imaging pipeline and neuron network training, respectively. The source of this reduction comes from the support of partial data transfer in the proposed approach, which can utilize the LL-SCM memory space in a more efficient way. In addition, the gap induced by rounding are less than 0.1% in both cases comparing to an optimal ILP solution, which further demonstrate the optimality of the proposed LP approach when the application TLRG graph is small (as shown in Table 5-2).

5.4.3 Discussion of L1-SCM Utilization Efficiency

Figure 5-12 shows the comparison between RASP and DRDU in terms of SCM buffer size and the number of data transfers from lower-level memory. We only include applications with regular reuse patterns in this comparison. The same SCM size constraint is applied to the two approaches and the prefetching scheme in RASP is disabled for fairness. In general, we can see that the number of data transfers from

lower-level memory of RASP is 7% smaller over the DRDU result and 41.2% smaller over the original program. The buffer size of RASP is 22.7% smaller than the DRDU buffer size. The smaller required SCM size in RASP scheme implies a higher SCM storage utilization ratio, which also provides more space to harmonize with SCM management techniques working on other program elements, e.g. [94].

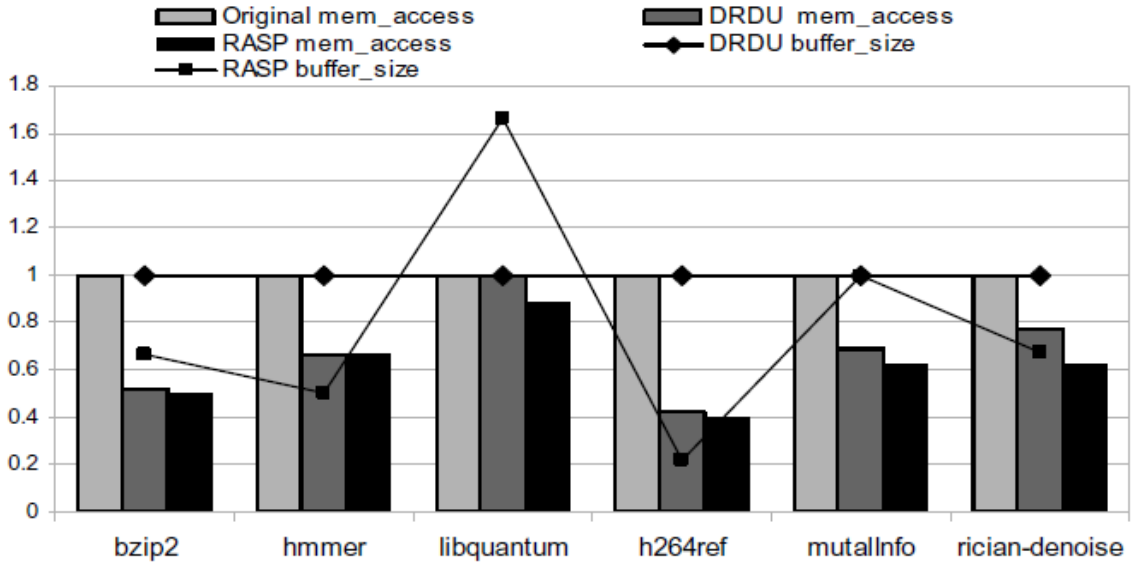


Figure 5-12. Comparison of buffer size and SCM data transfers.

5.5 Conclusions and Future Work

In this chapter we introduce a reuse-aware SCM prefetching scheme to efficiently utilize SPM memory space and a task-level-reuse-graph based LL-SCM data movement scheme to minimize the amount of data transfers between heterogeneous computing cores and host processors. The proposed L1-SCM prefetch scheme shows a significant performance/power improvement against previous SCM management techniques and an average 25% reduction of host-accelerator data transfers is observed over previous LL-SCM management work, which demonstrates the impact of reuse patterns on accelerator memory management efficiency. Note that the proposed schemes can be combined with traditional techniques of data locality optimization, i.e., loop interchange or tiling, to further improve the usage of SCM. The co-optimization effectiveness will be

investigated in our future work.

Chapter 6. Conclusion Remarks

In this research, we use customized vector units, programmable accelerators and hybrid memory to showcase the compilation for computing or memory components in a heterogeneous system. The experiments conducted on these platforms also demonstrate the computing power of multi-core heterogeneous architectures. We believe the next-generation compute engines will incorporate more heterogeneous processor cores or accelerators, which will make virtualization increasingly important.

Beyond component optimization, the next step ahead is system-level optimization, which is not just a simple addition of each module. For example, communication between different modules is a crucial design factor which has restricted the efficient utilization of the ample on-chip computing resources (i.e., accelerators). Note that system-level optimizations not only include compiler transformations, but also involve architecture designs and even algorithm design. This thesis presents a few building blocks for the system-level compiler optimization.

As one can see from the thesis, the introduction of heterogeneity and customization opens a door to improving energy/performance efficiency in SoC designs. A further step will be leveraging these two features in enterprise data centers, i.e., enabling more power-efficient management of enterprise workloads. Today's data centers are already equipped with a wide collection of heterogeneous technologies, i.e. operating systems, storage, hardware/tools from multiple vendors, applications with different business requirements. The scale of modern data centers also increases the complexity of management. We believe that it brings not only more challenges but also more interesting research topics to investigate along this path.

References

[1] J. Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman and Yi Zou, "An Energy-Efficient Adaptive Hybrid Cache," *in the Proceedings of International Symposium on Low Power Electronics and Design (ISLPED 2011)*, pp. 67-72, August 2011.

[2] J. Cong, V. Sarkar, G. Reinman and A. Bui. Customizable Domain-Specific Computing. UCLA Computer Science Department Technical Report TR# 100018, Los Angeles, California, 2010.

[3] Int'l technology roadmap for semiconductors, http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[4] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan and D. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In. *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp.81-92, 2003.

[5] B. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp.36-47, 2008.

[6] J. Cong, Y. Fan, G. Han, W. Jiang and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *Proceedings of the IEEE International SOC Conference*, pp. 199-202, 2006.

[7] K. Atasu, L. Pozzi, and P. Ienne, Automatic application-specific instruction-set extensions under micro-architectural constraints. In *Proc. DAC*, pp. 256-261, 2003.

[8] L. Bachega, S. Chatterjee, K. A. Dockserz, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowskix, G. K. Liu, M. P. Mendell, C. D. Wait, and T. J. C. Ward, A High-performance SIMD floating point unit for BlueGene/L: architecture, compilation, and algorithm design. In Proc. PACT, pp. 85–96, 2004.

[9] R. Barik, J. Zhao and V. Sarkar, Efficient selection of vector instructions using dynamic programming. In Proc. MICRO, pp. 201-212, 2010.

[10] C. Bienia, S. Kumar, J. P. Singh and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications, In Proc. PACT, 2008.

[11] A. J. C. Bik, M. Girkar, P.M. Grey and X. Tian, Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. Intel Technology, 2001.

[12] P. Brisk, A. Kaplan and M. Sarrafzadeh, Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In Proc. DAC, 2004.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In Proc. IISWC, pp. 44-54, 2009.

[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K.Skadron.A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads.In Proc. IISWC, 2010.

[15] M.O. Cheema and O. Hammami, Customized SIMD unit synthesis for system on programmable chip – a foundation for HW/SW partitioning with vectorization. In Proc. ASP-DAC, pp. 54-60, 2006.

[16] V.A. Chouliaras, K. Koutsomyti, T. Jacobs, S. Parr, D.J. Mulvaney and R.J. Thomson, SystemC-defined SIMD instructions for high performance SoC architectures. In Proc. ICECS, pp. 1-4, 2006.

[17] J. Cong, Y. Fan, G. Han, and Z. Zhang, Application-specific instruction generation for configurable processor architectures. In Proc. FPGA, pp. 183-189, 2004.

[18] J. Cong and W. Jiang, Pattern-based Behavior Synthesis for FPGA Resource Reduction, In *Proc. FPGA*, pp. 107-116, 2008.

[19] J. Cong, B. Liu and Z. Zhang. Scheduling with soft constraints. In Proc. ICCAD, pp. 47-54, 2009.

[20] J. Cong, B. Liu, S. Neuendor_er, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: from prototyping to deployment. In IEEE TCAD, 30(4):473-491, 2011.

[21] J. Cong, G. Reinman, A. Bui and V. Sarkar, Customizable domain-specific computing, In IEEE Design & Test, vol. 28, pp. 6-15, 2011.

[22] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, G. Reinman and Yi Zou, AXR-CMP: Architecture Support in Accelerator-Rich CMPs, In SAW-2, 2011.

[23] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to powerpc accelerates media processing. In IEEE Micro, pp. 85–95, 2000.

[24] A.E. Eichenberger, P. Wu, and K. O'Brien, Vectorization for SIMD architectures with alignment constraints. In Proc. PLDI, pp. 82–93, 2004.

[25] B. Fischer and J. Modersitzki, Curvature based image registration, J. Math. Imaging Vis., vol. 18, no. 1, pp. 81–85, 2003.

[26] F. Franchetti and M. PÄuschel, A SIMD Vectorizing compiler for digital signal processing algorithms," in Proc. IPDPS, pp. 20-26, 2002.

[27] T. Henretty, K. Stock, L.N. Pouchet, F. Franchetti, J. Ramanujam and P. Sadayappan, Data layout transformation for stencil computations on short-vector SIMD architecture. In Proc. CC, pp. 225-245, 2011.

[28] A.J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedra. In H. W. Kuhn and A. W. Tucker, editors, Linear Inequalities and Related Systems, pp. 22–46. Princeton University Press, 1956.

[29] K. Kennedy and John R. Allen. Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[30] C.E. Kozyrakis and D.A. Patterson, Scalable vector processors for embedded systems. IEEE Micro, pp. 36–45, 2003.

[31] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, Simics: A full system simulation platform. In IEEE Computer, vol. 35, pp. 50-58, 2002.

[32] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill and D. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, Computer Architecture News, pp. 92-99, Sept. 2005.

[33] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks, Vectorizing for a SIMD DSP architecture. In Proc. CASES, pp. 2–11, 2003.

[34] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In Proc. PLDI, pp. 132–143, 2006.

[35] S. Oberman et al, AMD 3DNow! technology and the K6-2 microprocessor. In HOTCHIPS10, 1998.

[36] V. Raghunathan, A. Raghunathan, M.B. Srivastava and M.D. Ercegovac, High-level synthesis with SIMD units. In Proc. VLSI Design. pp. 407-413, 2002.

[37] S.C. Woo, M.i Ohara, E. Torrie, J.P. Singh, and A. Gupta.The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, pp. 24-36, 1995.

[38] P. Yu and T. Mitra, Scalable custom instructions identification for instruction-set extensible processors, in Proc. ICCS, pp. 69-78, 2004.

[39] Convey system, http://www.conveycomputer.com/index. html

[40] Intel AVX, http://software.intel.com/en-us/avx/

[41] ITK software guide, http://www.itk.org/ItkSoftwareGuide .pdf

[42] LLVM Compiler Infrastructure, http://llvm.org/

[43] Omega library , http://www.cs.umd.edu/projects/omega/

[44] Parboil Benchmark suite, http://impact.chrc.illinois.edu/pa rboil.php.

[45] Synopsys design compiler. http://www.synopsys.com

[46] CACTI: http://www.hpl.hp.com/research/cacti/

[47] AutoPilot : http://www.xilinx.com/tools/autoesl.htm

[48] Asanovic K., Beck J., Irissou B., Kingsbury B., Morgan N., Wawrzynek J., "The T0 Vector Microprocessor" In Proceedings HOT Chips VII, Stanford, CA, August 1995.

[49] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A vector extension to the alpha architecture. In International Symposium on Computer Architecture, May 2002.

[50] Christos Kozyrakis and David Patterson., "Scalable Vector Processors for Embedded Systems," IEEE MICRO, vol. 23, no. 6, pages 36-45, November 2003.

[51] Convey computer, http://conveycomputer.com.

[52] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: A Many-Core x86 Architecture for Visual Computing,” in IEEE.Micro, vol. 29, no. 1, 2009, pp. 10-21.

[53] Nallatech development systems, http://www.nallatech.com.

[54] 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 Proc. ISCA, 2005.

[55] A. Hormati, N. Clark, and S. Mahlke, “Exploiting Narrow Accelerators with Data-Centric Subgraph Mapping,” in Proc. CGO, 2007, pp. 341-353.

[56] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture Support for Accelerator-Rich CMPs,” in Proc. DAC, 2012.

[57] J. Cong, M. Ghodrat, M. Gill, C. Liu, G. Reinman, and Y. Zou, “Architecture Support for Accelerator-Rich CMPs,” in 2nd Workshop on SoC Architecture, Accelerators and Workloads, 2011.

[58] J. Cong, Y. Fan, G. Han, and Z. Zhang, “Application-specific instruction generation for configurable processor architectures,” in Proc. FPGA, 2004, pp. 183-189.

[59] K. Atasu, L. Pozzi, and P. Ienne, “Automatic application-specific instruction-set extensions under microarchitectural constraints,” in Proc. DAC. New York, NY, USA: ACM Press, 2003, pp. 256-261.

[60] N. Clark, A. Hormati, S. Mahlke, and S. Yehia, “Scalable Subgraph Mapping for Acyclic Computation Accelerators,” in Proc. CASES, 2006, pp. 147-157.

[61] S. Hu, M. Lipasti, and J. Smith, “An Approach for Implementing Efficient Superscalar CISC Processors,” in Proc. HPCA, 2006, pp. 213-226.

[62] L. Pozzi, K. Atasu, and P. Ienne, “Exact and Approximate Algorithm for the Extension of Embedded Processor Instruction Sets,” in Proc. TCAD, vol. 25, no. 7, 2006, pp. 1209-1229.

[63] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, “A DAG-Based Design Approach for Reconfigurable VLSI Processors,” in Proc. DATE, 1999, pp. 778-779.

[64] C. Galuzzi, K. Bertels, and S. Vassiliadis, “A Linear Complexity Algorithm for the Generation of Multiple Input Single Output Instructions of Variable Size,” in Proc. SAMOS, 2007, pp. 283-293.

[65] C. Galuzzi, E. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis, “Automatic Selection of Application-Specific Instruction-Set Extensions,” in Proc. CODES+ISSS, 2006.

[66] C. Galuzzi, D. Theodoropoulos, and K. Bertels, “A Clustering Method for the Identification of Convex Disconnected Multiple Input Multiple Output Instructions,” in Proc. SAMOS, 2008, pp. 65-73.

[67] J. Cong and W. Jiang, “Pattern-based behavior synthesis for FPGA resource reduction,” in FPGA. New York, NY, USA: ACM, 2008, pp. 107-116.

[68] P. Bonzini and L. Pozzi, “Polynomial-time subgraph enumeration for automated instruction set extension,” in DATE. New York, NY, USA: ACM Press, 2007, pp. 1331-1336.

[69] P. Yu and T. Mitra, “Disjoint Pattern Enumeration for Custom Intruction Identification,” in Proc. FPL, 2007, pp. 273-278.

[70] SPEC CPU2006, http://pec.it.miami.edu/cpu2006/.

[71] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing,” in Proc. IISWC, 2009, pp. 44-54.

[72] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, “DRDU: A Data Reuse Analysis Technique for Efficient Scratch-Pad Memory Management,” in ACM Trans. Des. Autom. Electron. Syst., 2007.

[73] T. Chen, T. Zhang, Z. Sura, and M. Tallada, “Prefetching Irregular References for Software Cache on Cell,” in Proc. CGO, 2008, pp. 155–164.

[74] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems,” in Proc. CODES, 2002, pp. 73–78.

[75] J. Sjodin and C. Platen, “Storage Allocation for Embedded Processors,” in Proc. CASES, 2001, pp. 15–23.

[76] O. Avissar, R.Barua, and D.Stewart, “An Optimal Memory Allocation Scheme for Scratchpad-based Embedded Systems,” in ACM TRANS. Embed. Comput. Syst., 2002, pp. 6–26.

[77] M. Verma, S.Steinke, and P. Marwedel, “Data Partitioning for Maximal Scratchpad Usage,” in Proc. ASPDAC, 2003, pp. 77–83.

[78] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “Dynamic Management of Scratchpad Memory Space,” in Proc. DAC, 2001, pp. 690–695.

[79] S. Udayakumaran and R. Barua, “Compiler-decided Dynamic Memory Allocation for Scratchpad Based Embedded Systems,” in Proc. CASES, 2003, pp. 276–286.

[80] L. Li, H. Feng, and J. Xue, “Compiler-directed Scratchpad Memory Management via Graph Coloring,” in ACM Trans. Archit. Code Optim., 2009, pp. 1–17.

[81] T. Yemliha, S. Srikantaiah, M. Kandemir, and O. Ozturk, “SPM Management Using Markov Chain Based Data Access Prediction,” in Proc. ICCAD, 2008, pp. 565–569.

[82] A. Beric, R. Sethuraman, H. Peters, G. Veldman, J. Meerbergen, and G. Haan, “Streaming Scratchpad Memory Organization for Video Applications,” in Proc. Circuits, Signals and Systems, 2004, pp. 427–432.

[83] T. Mowry, M. Lam, and A. Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching,” in Proc. ASPLOS, 1992, pp. 62–73.

[84] S. Vanderwiel and D. Lilja, “Data Prefetch Mechanisms,” in ACM Computing Surveys, 2000, pp. 174–199.

[85] R. M. Rabbah, H. Sandanagobalane, M. Ekapanyapong, and W. Wong, “Compiler Orchestrated Prefetching via Speculation and Predication,” in Proc. ASPLOS, 2004, pp. 189–198.

[86] T. C. Mowry, “Tolerating latency through software-controlled data prefetching,” Ph.D. dissertation, Stanford University, 1994.

[87] K. Kennedy and J. Allen, Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers Inc., 2002.

[88] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Bountev, and P. Sadayappan, "Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories," in Proc. PPoPP, 2008, pp. 1–10.

[89] M. Kandemir and A. Choudhary, "Compiler-Directed Scratchpad Memory Hierarchy Design and Management," in Proc. DAC, 2002, pp. 628–633.

[90] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," in IEEE Computer, 2002, pp. 50–58.

[91] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator(GEMS) toolset," in Computer Architecture News, 2005, pp. 92–99.

[92] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-core and Many-core Architectures," in Proc. MICRO, 2009, pp. 469–480.

[93] Sun Microsystems, "UltraSPARC-II Enhancements: Support for Software Controlled Prefetch," White Paper, 1997.

[94] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU," in IEEE Trans. On Computers, vol. 59, no. 8, 2010, pp. 1047–1062.

[95] A. Bui, K. Cheng, J. Cong, L. Vese, Y. Wang, B. Yuan and Y. Zou, "Platform Characterization for Domain-Specific Computing", *Proceedings of the 17th Asia and South Pacific Design Automation Conferenc*e (ASPDAC 2012), 2012.

[96] Intel Moorestown, http://www.intel.com/pressroom/archive /reference/Mooresto wnbackgrounder.pdf.

[97] The OMAP5430 Platform, http://www.ti.com/ww/en/omap/omap5/omap5-OMAP5430.html.

[98] V. Govindaraju, C. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for Energy Efficient Computing," in Proc. HPCA, 2011, pp. 503–514.

[99] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A Composable Heterogeneous Accelerator-Rich Microprocessor," in ISLPED, 2012, pp. 379–384.

[100] XtremeDSP 48E, http://www.xilinx.com/technology/dsp/xtremedsp.htm.

[101] Q. Zhuge, Y. Guo, J. Hu, W-C Tseng, C. Xue and E H-M Sha, "Minimizing Access Cost for Multiple Types of Memory Units in Embedded Systems Through Data Allocation and Scheduling", in TSP, 2012, Vol. 60, No. 6.

[102] Y. Liu and W. Zhang, "Exploiting Multi-Level Scratchpad Memories for Time-Predictable Multicore Computing", in ICCD, 2012, pp. 61-66.

[103] F. Magno, Q. Pereira and J. Palsberg, "Register Allocation via Coloring of Chordal Graph", in ASPLAS, 2005, pp. 315-329.

[104] F. Magno, Q. Pereira and J. Palsberg, "Register Allocation after Classical SSA Elimination is NP-complete", in FOSSACS, 2006, pp. 79-93.

[105] G.J. Chartin, "Register Allocation and Spilling via Graph Coloring", in SCC, 1982, pp. 98-105.

[106] C. Andersson, "Register Allocation by Optimal Graph Coloring", in CC, 2003, pp. 34-45.

[107] J. Cong, B. Liu and Z. Zhang, "Scheduling with Soft Constraints", in ICCAD, 2009, pp. 47-54.