

STATIC AND DYNAMIC INSTRUCTION MAPPING
FOR SPATIAL ARCHITECTURES

FENG LIU

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING

ADVISOR: PROFESSOR DAVID I. AUGUST

JUNE 2018

© Copyright by Feng Liu, 2018.

All Rights Reserved

Abstract

In response to the technology scaling trends, spatial architectures have emerged as a new style of processors for executing programs more efficiently. Unlike traditional out-of-order (OoO) processors, which time-share a small set of functional units, a spatial computer is composed of hundreds or even thousands of simple and replicated functional units. Spatial architectures avoid the overheads of time-sharing and of generating schedules repeatedly, by mapping instruction sequences onto the functional units explicitly and reusing the mapping across multiple invocations.

Currently, spatial architectures mainly use static methods to map and schedule instructions onto the arrays of functional units. The existing methods have several limitations: First, for programs with irregular memory accesses and control flows, they yield poor performance because the functional units need to be invoked sequentially to respect data and control dependences. Second, static methods cannot fully exploit speculation techniques, which are the dominant performance sources in OoO processors. Finally, static methods cannot adapt to changing workloads and are not compatible across hardware generations.

To address these issues and improve the applicability of spatial architectures, this dissertation proposes two techniques. The first, Coarse-Grained Pipelined Accelerators (CGPA), is a static compiling framework that exploits the hidden parallelism within irregular C/C++ loops and translates them into spatial hardware modules. The proposed technique has been implemented as a compiler pass and the experiment shows 3.3x speedup over the performance achieved by an open-source tool baseline.

The second technique, Dynamic Spatial Architecture Mapping (DYNASPAM), reuses the speculation system in the OoO processors to dynamically produce high performance scheduling and execution on a dedicated spatial fabric. The proposed technique is modeled by a cycle accurate simulator and the experiment shows the new technique can achieve 1.4x

geomean performance improvement and 23.9% energy consumption reduction, compared to an aggressive OoO processor baseline.

Acknowledgements

First, I would like to thank my advisor Professor David I. August for guiding me throughout my years in Princeton. I really appreciate the opportunities and challenges he gave me. During the years I worked in his research group, I learnt how to conduct research projects, write scientific papers and proposals. More importantly, I learnt from him never to give up and always to pursue big goals. I could not imagine that I could have made my way today without his encourage and support. I believe I will continue to benefit from the knowledge he taught me in the rest of my life.

I would like to thank Professor Sharad Malik and Professor David Walker for reading this dissertation and providing insightful comments. I also thank Professor Malik for giving me advice and help during my years in Princeton when I was really stressed and frustrated by study, research and language problems. I also thank Professor Walker for being always very generous when I was looking for advices. Additionally, I want to thank Professor Aarti Gupta and Professor David Wentzlaff for serving as my thesis committee member and their feedbacks that helped me to polish and refine my thesis. I really appreciate the time all the professors spent on helping me to finish the Ph.D. defence process.

This dissertation would not have existed without the help from everyone in the Liberty Research Group. I have found great friendship with everyone in the group. I would like to thank some senior group members, Yun Zhang, Arun Raman, Thomas B. Jablin, Jialu Huang, Prakash Prabhu, Hanjun Kim and Nick P. Johnson for their guide and help in my early years into my PhD study. Especially I really to thank Thomas B. Jablin, who really spent a lot of time teaching me domain knowledge. I also want to thank Soumyadeep Ghosh, who teaches me a good life attitude and how to handle complexities. I also want to say thank you to Stephen R. Beard, Taewook Oh and Heejin Ahn for helping me with my research and writing. I will remember those crazy paper deadlines we spent together.

I also thank the entire staff of the Department of Electrical Engineering and the Department of Computer Science in Princeton. Their professionalism really makes this such a great place to study and to do research. I also want to thank the staff of Davis International Center, who helped me to improve my language skills and also handled my endless visa issues.

There are a lot of friends I met over the years. I thank all of them for making my life in Princeton happy and unforgettable. Yungang Bao, Yida Wang, Aoxiang Tang, Tao Han, Shucheng Zhu, Xiaozhou Li, Xin Jin and Yunlai Zha, being my friends and roommates, helped me a lot with my daily life. They are my patient listeners whenever I had trouble in research or personal life. These friendship will become my largest fortune I have in my life.

I want to thank my parents for their unconditional love and support. They let me explore the world in the way I want. They taught me how to make balance between job and life, how to handle unhappiness and pressure, and how to love others. I cannot imagine how much they sacrificed to allow me to achieve what I have today. Everything I achieved today, is truly theirs.

Last, but not the least, I want to thank my wife Jie Feng, who makes me strong and happy. Without her encouragement, I could not survive the difficulties I had in the past years. She is really the source of my strength and the most important part of my life.

To my family.

Contents

Abstract	iii
Acknowledgements	i
List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Spatial Architectures	1
1.2 Instruction Scheduling for Spatial Architectures	3
1.3 Dissertation Contributions	5
1.4 Dissertation Organization	7
2 Background and Related Work	8
2.1 Instruction Mapping Techniques	8
2.2 Decoupled Software Pipelining Techniques	10
2.3 Existing Spatial Architectures	12
2.4 Issues with Existing Architectures and Their Mapping Techniques	16
3 Static Mapping with Coarse-Grained Decoupled Pipelining	18
3.1 Motivating CGPA	20
3.2 Coarse-Grained Pipeline Accelerators	24
3.2.1 CGPA Workflow	26

3.2.2	Pipeline Generation	27
3.2.3	CGPA Compiler Backend	30
3.3	Evaluation of CGPA	32
3.3.1	Methodology	32
3.3.2	Results	34
3.4	Applicability and Scalability of CGPA	36
3.4.1	Applicability	36
3.4.2	Scalability	41
4	Spatial Architecture Speculation with Hardware Reuse	44
4.1	Motivating SPAS	46
4.1.1	Dynamic Mapping for Spatial Architecture	46
4.1.2	Speculative Architecture Support for Dynamic Mapping	49
4.2	Hybrid Speculative Spatial Architecture Design	51
4.3	Spatial Fabric Implementation and Integration	55
4.3.1	Acyclically Connected Spatial Fabric	55
4.3.2	Integration into the Host OoO Pipeline	57
4.3.3	Intra- and Inter-Trace Memory Ordering	59
4.3.4	Configuration Datapath	60
4.3.5	Execution Example	60
5	Dynamic Mapping with Resource-Aware Instruction Scheduling	64
5.1	Motivating DYNASPAM	65
5.1.1	The Importance of Mapping Scope	65
5.1.2	Hardware Synthesis	67
5.2	DYNASPAM Design	68
5.2.1	Resource-Aware Scheduling	69

5.2.2	Priority Score Generation	73
5.2.3	Example	78
5.3	Evaluation	79
5.3.1	Methodology	79
5.3.2	Results	81
6	Conclusions and Future Directions	88
6.1	Conclusions	88
6.2	Future Directions	89

List of Tables

2.1	Comparison between DYNASPAM and other in-core reconfigurable computation engine.	14
3.1	New primitives added to LLVM IR to support worker invocation, dependence communication, and register value passing across hardware modules.	30
3.2	Descriptions of benchmarks used. P1: Pipeline Partition with Replicable Section in Sequential Stage; P2: Pipeline Partition with Replicable Section in Parallel Stage	33
3.3	Comparison between CGPA and related frameworks	35
4.1	Topologies of different spatial architecture examples.	47
4.2	Types of constraints for the mapping problem.	48
5.1	Priority Scores for different connection status of the producers.	73
5.2	Evaluation system parameters	80
5.3	Programs tested from the Rodinia Benchmark Suite.	82
5.4	Area Comparison for different components	83

5.5 Detected Traces and Average Configuration Lifetime. The “Mapped Traces” column shows how many hot traces are detected and translated to configurations by the hardware, and “Offloaded Traces” column shows among them how many traces are really offloaded to the fabric. The “Avg. Config. Lifetime” shows how many invocations the offloaded traces need to be evicted from the fabrics for the new offloaded traces. The larger the better. 84

List of Figures

2.1	Decoupled Software Pipelining schedules on a multi-core processor. (a) describes a loop which cannot be handled by other data-level parallelization techniques. (b) is the Program Dependence Graph(PDG) of loop (a). (c) is the DAG _{SCC} of the PDG. Solid lines represent data dependences while dotted lines represent control dependences. (d) shows the parallel execution schedules of the loop for DSWP on a 2-core processor, and how the technique can tolerate core-to-core communication latency. (e) shows how an enhanced version of DSWP (Parallel Stage PSWP) can scale to more cores.	11
2.2	A Design Space of Computer Architectures (Convolution Engine [66], DySER [34], BERET [35], and TPU [46]).	12
3.1	(a) Source code of the main loop of “em3d” program, with sections annotated as replicable or parallel; (b) Data-level parallelism is exploited by duplicating the replicable section; (c) Coarse-grained pipeline parallelism is exploited by separating the replicable section from the rest of the loop; (d) A simplified source-level Program Dependence Graph (PDG) of the main loop in em3d; (e) Pseudo-code of tasks for both hardware stages. The code in gray is the overhead generated by the CGPA compiler.	22

3.2	Different program patterns exploiting parallelism for hardware accelerator design.	25
3.3	A logical view of CGPA architecture within the dashed box. Each grey box contains circuit modules customized for the targeted loop and generated by CGPA compiler. In this figure, a Sequential–Parallel–Sequential (S-P-S) 3-stage pipeline is shown.	26
3.4	Workflow for CGPA HLS Framework	27
3.5	Speedup Results for Loop Kernels of the Benchmarks	34
3.1	The targeted loop for K-means algorithm, along with the identification of different sections.	37
3.2	Pipeline generated for K-means by CGPA.	38
3.3	Source code for targeted loop in 1D Gaussian Blur, along with the identification of different sections.	39
3.4	Pipeline generated for 1D Row Gaussian Blur by CGPA.	41
4.1	A generalized reconfigurable spatial architecture.	46
4.2	The overall view of the SPAS architecture. \$ is the abbreviation of cache. The blocks with gray color filled are the new components added to the conventional OoO pipeline or the existing components requires modification.	52

4.3	The functional units, pass registers, and interconnect of a stripe of Figure 4.1A generalized reconfigurable spatial architecture. figure.caption.29. Wires do not interact with each other except (1) ● is put on the intersection or (2) wires with different width intersect. Units A, B and C are pass registers; ALU represents an ALU unit; LDST represents a load/store unit; buffers on ALU input represent the FIFOs of input operands; the output buffer on the LDST unit represents the memory reservation buffer.	56
4.4	The design of Live-in and Live-out FIFOs for the spatial fabric and host processor integration in SPAS.	58
4.5	(a) Source code and assembly of the simple program example, and its data dependence graph; (b) An dynamic unrolling of (a) with OOO, which contains 2 ALUs and 1 MEM; (c) An instruction placement for the spatial fabric with the scheduling result from (b); (d) Activities of the functional units (0,0), (0,1), (1,0) and (1,1) in the first three cycles, indicating the pipelined execution.	61
5.1	(a) for two special architecture settings (without dotted line, and with dotted line to share operands), (b) and (c) show examples where naïve placement fails to create efficient schedules, and (d) show a resource-aware scheduling.	65
5.2	fabric functional units (FUs) and possible scheduling frontiers in (a) CCA; (b) 4x4 DySER; and (c) DYNASPAM fabric.	69
5.3	An example of logic to select ready instructions from the reservation stage for a function units. The priority can be changed by encoding more constraints in the priority encoder.	73

5.4	An example illustrating how instruction scheduling is impacted by the location information of the fabric.	77
5.5	Trace Coverage. “Normal” is the percentage of dynamic instructions executed on the host OoO pipeline, “Mapping” is the percentage of dynamic instructions which are detected as hot traces, but not offloaded to the spatial fabric yet, and “Accelerating” is the percentage of dynamic instructions which are offloaded to the fabric successfully.	83
5.6	Performance Comparison with Respect to Host OoO Pipeline.	86
5.7	Energy Comparison with Respect to Host OoO Pipeline.	87

Chapter 1

Introduction

1.1 Spatial Architectures

After being effective more than four decades, Moore's Law [60] is still dominating the scaling of micro-electronics industry, and the number of on-chip transistors are continuing to double every eighteen months. In the past, computer architects utilized these transistors to build complex micro-architectures, comprising more functional units, deeper pipelines, and more complex cache hierarchy, for more powerful out-of-order (OoO) execution, that led to better performance. The focus of these innovations was to explore the hidden *instruction-level parallelism* (ILP) dynamically in the programs and to reduce program execution time. However, exploiting ILP gave progressively diminishing returns due to the small instruction scopes that the computer architecture could optimize at runtime [86], while deep OoO pipeline can introduce unnecessary energy overhead. The consequence is that the power consumption becomes the critical bottleneck of improving program performance on both portable devices and data center servers. Since neither deep OoO pipeline, simultaneous

multi-threading nor multi-core improves energy efficiency, we need to re-examine the design of mainstream OoO processors and consider introducing more efficient paradigms to the computation.

OoO processors deliver high performance by their powerful *dynamic scheduling* mechanism. This mechanism actually relies on two subsystems: One subsystem comprises the *speculative* units such as branch and memory dependence predictors, that can be dynamically trained to predict the program behaviors for different workloads. The second subsystem implements the capability of regenerating instruction schedules according to *dynamic* information by using renaming logic, reservation stations and reorder buffers [65, 36, 42]. The former subsystem helps the latter by relaxing instruction scheduling constraints in order to adapt to different workloads and generate high efficient schedule at runtime.

However, this is not an efficient design for computationally intensive programs, which usually have repeating instruction sequences. With OoO execution, even if a program enters a relatively repeating execution pattern with same instruction sequences, the processor does not take full advantage of the predictable program behavior and unnecessarily spends energy on redundantly exercising the speculative/dynamic scheduling units and regenerating schedules, for the same instruction sequences [56]. Moreover, the processor unnecessarily spends energy on dynamically resolving data dependences and delivering operands to the functional units, even though the dataflow is well known for the repeating instruction sequences and could be realized by wiring functional units together through more efficient dedicated data paths [6, 34, 35, 93].

An ideal efficient processor architecture would utilize the repeating program schedules and map instructions to a set of spatially distributed functional units. This would reduce energy consumption of both instruction scheduling and data delivery. This dissertation refers to such a processor architecture as “spatial architectures” with spatially distributed functional units, a spatial fabric. Spatial architectures map computation across a grid of

functional units for processing and build specialized datapath connections between them to fulfill dependences. Fixing instruction assignments to processing elements obviates the need to separate instruction execution into multiple pipeline stages (fetch, decode, rename and issue). Direct communication from producers to consumers of the elements obviates the need for the bypass network and the register file [6, 34, 35]. This simplification of the processor design allows spatial architectures to be more energy efficient, compared to conventional OoO processors.

1.2 Instruction Scheduling for Spatial Architectures

However, spatial architectures usually require more effort to map program instructions to its spatially distributed functional units. This is because the hardware resources (functional units and datapath between them) on spatial architectures cannot be time-shared like in OoO processors. Thus, a successful instruction mapping for spatial architectures needs to use its computational resources in an efficient way.

One approach to build these customized mapping for spatial architectures is to manually design dedicated circuit modules for each application using hardware description languages (HDLs) [52, 66]. While this approach often yields the best result, it requires significant non-recurring engineering costs to convert algorithms to HDL specifications.

Alternatively, in prior works, most reconfigurable spatial architectures, including Field-Programmable Gate Arrays (FPGAs), Programmable Functional Units (PFUs) [5, 10, 14, 20, 34, 35, 71, 88, 92] and Coarse-Grained Reconfigurable Arrays (CGRAs) [33, 57, 58, 67, 76, 85], explore static compiling techniques, called High-Level Synthesis (HLS). HLS maps high-level language code sequence at compile time to hardware logic. HLS can provide a large scheduling scope (the whole loop or even the whole program) and allow the

mapping generator to consider more instructions simultaneously in the scope, thus producing mappings that find more parallelism and achieve more efficient resource utilization. HLS has demonstrated its applicability for generating customized hardware from programs written in C/C++ or other HDLs [8, 25, 28, 32, 80, 82, 83, 91].

A deep dive into these use-cases reveals that they are mainly used in the domains of scientific computation [80] and digital signal processing (DSP) applications [25, 28, 32, 83, 91]. One typical feature of these applications is that their hot spots consist of affine loop nests. Such loop nests enable a series of loop transformations that expose *loop-level parallelism* to overlap instruction execution [16, 70, 89]. Additionally, the affine loops may also enable special circuit modules, such as systolic arrays and shift registers, to reduce memory traffic and improve performance [18, 19, 46]. For example, the convolutional neural networks have the regular pattern that executes a matrix multiply between input data and weights and then apply an activation function. Thus the Matrix Multiplication Unit in the TPUs [46] is designed to have a systolic array mechanism that contains 65,536 ALUs, thus 65,536 multiple-and-adds for 8-bit integers can be processed every cycle.

However, in the presence of loops with complex control flows or irregular memory accesses, these static mapping tools do not extract loop-level parallelism well, leading to poorly performing hardware designs. In reality, the majority of programs are implemented with non-array data structures and imperfect loop nests. This restricts the scope for parallelism within one or a few loop iterations, and generates hardware modules with limited performance improvement [82]. A new technique is required to generate efficient accelerators for this class of programs, and to bridge the gap between the synthesized and manually generated results for these programs.

Besides its limited applicability domains, the HLS methods cannot make use of information gathered at runtime to optimize their mappings for changing workloads. The optimizations in static mapping methods are conservative and usually impose unnecessary

constraints which bring extra resource usage and performance overhead. For example, all statically mapped CGRAs need extra control dependences between memory operations if they are may-alias. Additionally, programs that are statically mapped to a particular reconfigurable fabric cannot run effectively on a processor without the fabric, and may not be compatible with different fabric generations. With this binary compatibility issue, an application compiled to a reconfigurable fabric with certain physical settings cannot run on a processor without the fabric, and also it is not forward and backward compatible for different hardware generations [14, 15].

Ideally, we want spatial architectures to be in lieu of traditional OoO processors, which can (1) handle programs with complex control and memory access patterns and (2) rely on a OoO-like dynamic scheduling mechanism to provision functional units and fulfill data dependences for the computation defined by the programs. These goals involve both improving the existing static mapping methods by handling irregular programs and also enabling dynamic mapping methods to utilize dynamic information and generate scheduling at runtime for spatial architectures.

1.3 Dissertation Contributions

This dissertation improves both the applicability and adaptability of static mapping methods for spatial architectures.

To improve the applicability of static mapping, this dissertation introduces Coarse-Grained Pipelined Accelerators (CGPA) [54]. CGPA leverages the insight that a single program loops might be composed by different code sections with different purposes, thus it is natural to split them into several coarse-grained sections and apply different optimizations to each section individually. This dissertation implements a prototype of CGPA as an open-source HLS tool plugin, and applied the new method on a set of algorithms.

To overcome the adaptability limitation of static mapping and to leverage the optimization potential of OoO, this dissertation also presents DYNASPAM [53]. DYNASPAM is a framework that tightly couples a spatial fabric with an OoO pipeline and re-uses the hardware resources of the OoO pipeline to support speculative execution on spatial fabric. The insight behind DYNASPAM is that OoO processors excel at utilizing speculation and contain large instruction windows to dynamically map repeating instruction sequences onto the fabric. Therefore, combining OoO techniques with the execution efficiency of spatial architectures may lead to a more effective system.

In summary, the contributions of this dissertation are as follows:

- Coarse-Grained Pipelined Accelerators (CGPA), an HLS framework that utilizes coarse-grained pipeline parallelism techniques to synthesize efficient specialized accelerator modules from irregular C/C++ programs without requiring any annotations;
- Spatial Architecture Speculation (SPAS), a design of tightly coupled CPU-Spatial Architecture systems that supports speculative execution by reusing hardware prediction units in the host CPU processor. Re-using the branch predictor allows instruction traces to span multiple basic blocks. Re-using the memory speculation unit reduces energy consumption of the datapath and increases performance;
- Dynamic Spatial Architecture Mapping (DYNASPAM), a framework that combines SPAS with a novel dynamic, resource-aware mapping technique for reconfigurable spatial fabrics. DYNASPAM leverages the existing scheduling logic in the host OoO processor to provide a large mapping scope with only a small hardware overhead.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 describes the prior work related to spatial architectures and their instruction mapping techniques. Chapter 3 describes a new static mapping framework which splits single loops to multiple hardware pipeline modules for extracting better performance from these loops. Chapter 4 proposes a novel hybrid OoO-spatial architecture design which leverages the existing OoO architecture's speculation system and enables efficient mapping for dynamic workloads. Chapter 5 shows the design of a hardware module that can be included in an OoO processor pipeline to generate configurations for spatial architectures. Chapter 6 summarizes the results and also discusses the future work.

Chapter 2

Background and Related Work

Research into spatial architecture has been an active area for quite a long time, and different techniques have been proposed. Roughly, the spatial architectures (fabrics) can be classified into two categories: fine-grained and coarse-grained. Field-programmable gate array (FPGA) is the most common fine-grained reconfigurable spatial architecture. FPGA contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together. FPGAs contain large number of logic gates and RAM blocks to implement complex digital computations. However they require tens of seconds to completely program their arrays. The functional units of Coarse-Grained Reconfigurable Arrays (CGRAs) can be reconfigured as fast as normal functional units in OoO processors. Currently the mapping for both fine-grained and coarsed-grained spatial architectures is mainly based on the static compiling techniques.

2.1 Instruction Mapping Techniques

All existing instruction mapping techniques for spatial architectures focus on exploiting parallelism by finding independent operations in the **inner loops** and overlapping their

execution to reduce overall execution time. The parallelism hidden in the programs can be instruction-level and/or loop-level [2, 87, 70, 37, 31]. These forms of parallelism have been utilized by the existing HLS tools. Among these HLS tools, Instruction-level parallelism (ILP) techniques are combined with some other techniques such as if-conversion and loop unrolling to increase instruction scheduling window for better performance [50]. Besides ILP, Loop-level parallelism in modern HLS tools also increases the scheduling scope across multiple loop iterations and can potentially yield higher performance.

Existing HLS tools target two main types of loop-level parallelism. One class of tools targets loops with *data-level parallelism* [4]. If each loop iteration operates on disjoint data, parallel hardware modules can be designed to *fully* interleave loop iterations. In extreme cases, each iteration executes the same sequence of operations, leading to Single Instruction Multiple Data (SIMD) style parallelism [4]. In reality, however, few outer loops fit this pattern without additional transformations, thus limiting the applicability of HLS tools that exploit this kind of parallelism.

A second class of HLS tools exploits loop pipelining to *partially* interleave loop iterations [87, 70, 31, 37, 2]. Different pipelining schemes such as *pipeline vectorization* [87] and *software pipelining* [70, 31, 37, 2] have been adapted to HLS for this purpose. Pipeline vectorization is applicable to loops without true loop-carried dependences or with only regular loop-carried dependences. Software pipelining has been widely used to overlap computations from different iterations. However, complicated control and data dependences existing in loops limit the number of independent operations found from different iterations.

A complementary approach for HLS tools is to use loop transformations such as loop unrolling, flattening, permutation interchange and tiling to expose loop-level parallelism for innermost loops [89, 90]. However, these transformations are not effective when specializing programs with either complex control flows or irregular memory accesses.

As the availability of on-chip resources grows due to increase in number of transistors, targeting inner loops only is not sufficient to gain higher performance for programs that are more complicated than scientific kernels. This necessitates the use of transformations such as DOALL and Decoupled Software Pipelining [64] that exploit parallelism in **outer loops** [54].

2.2 Decoupled Software Pipelining Techniques

All the traditional HLS tools implement the DOALL technique, which is a data-level parallelization technique and designed to exploit parallelism for loops without any data dependences. The DOALL technique cannot handle loop-carried dependences, such as the example shown in Figure 2.1(a). To overcome its applicability limitation, Decoupled Software Pipelining (DSWP) [64] is designed for the loops with even loop-carried dependences exist. DSWP divides the loop body into multiple stages and assigns each stage to a different thread to create a pipeline. It first builds a DAG_{SCC} of the Program Dependence Graph (PDG) [24] (Figure 2.1(b) and (c)) of the target loop by coalescing each strongly connected component (SCC) in the PDG into a single node, then assigns those nodes to stages to identify a pipeline schedule. Loop-carried dependences are modeled as back edges in the PDG, which form SCCs. DSWP's partitioning causes all loop carried dependences to be communicated locally, as instructions from an SCC in the PDG are scheduled to the same thread. However, since the loop body is spread across multiple threads, any intra-iteration dependence that flows between stages must be communicated across threads. Since the stages are arranged into a pipeline, the inter-thread communication of DSWP parallelized programs exhibits an acyclic, or unidirectional pattern, where communication only flows along the pipeline. This feature allows the scheduling to tolerant core-to-core communications. As shown in Figure 2.1(d), when the communication latency increases from 1 cycle

to 2 cycles, the execution takes one extra cycle to fill the communication pipeline, but afterwards, the same throughput can be achieved (2 cycles per iteration). DSWP method can also be scalable if there are parallel stages. As shown in Figure 2.1(e), multiple instances of Stage C can be executed in parallel, so they can be scheduled to different cores, enabling a higher execution performance.

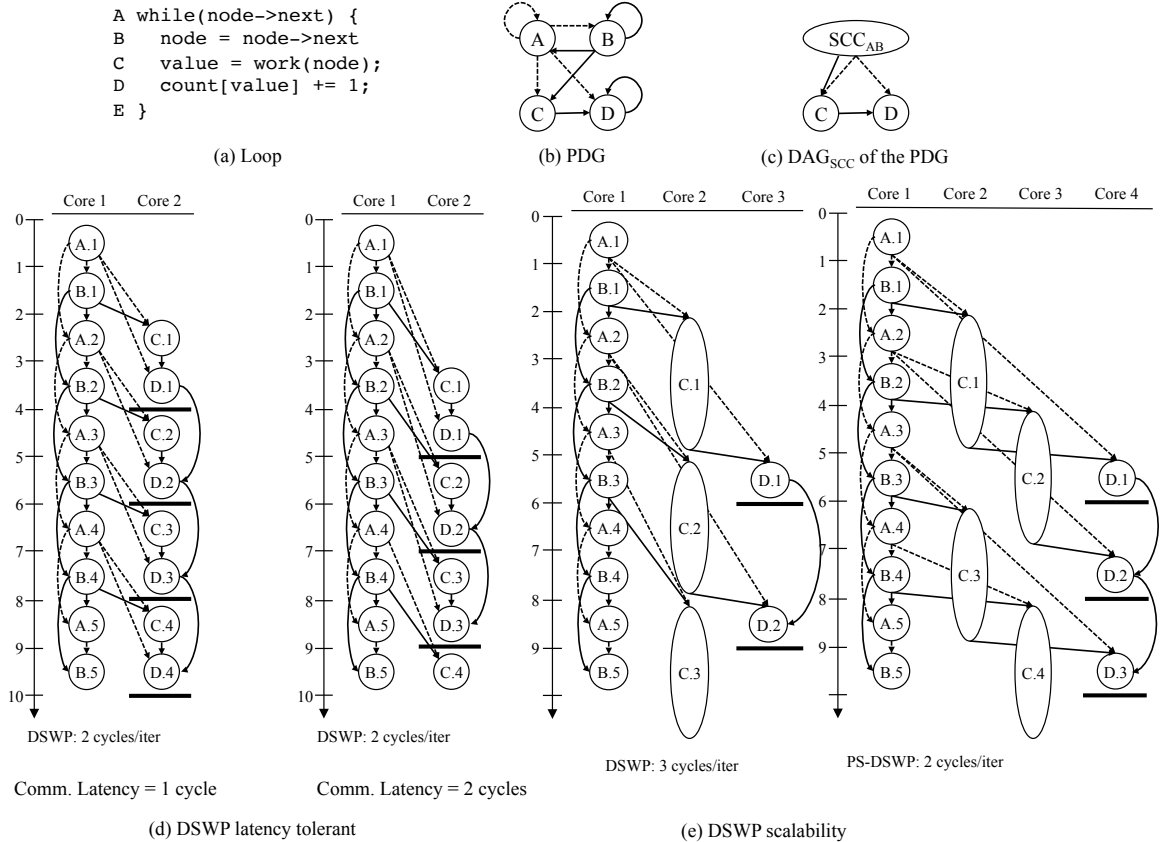


Figure 2.1: Decoupled Software Pipelining schedules on a multi-core processor. (a) describes a loop which cannot be handled by other data-level parallelization techniques. (b) is the Program Dependence Graph(PDG) of loop (a). (c) is the DAG_{SCC} of the PDG. Solid lines represent data dependences while dotted lines represent control dependences. (d) shows the parallel execution schedules of the loop for DSWP on a 2-core processor, and how the technique can tolerate core-to-core communication latency. (e) shows how an enhanced version of DSWP (Parallel Stage PSWP) can scale to more cores.

To utilize DSWP for mapping instructions in a HLS workflow, all the instructions scheduled into the same pipeline stage by DSWP are grouped together, and then can be

translated or mapped to separate hardware modules by the existing HLS methods. Moreover, the inter-thread communications between pipeline stages in DSWP execution can be implemented by First-In-First-Out (FIFO) buffers.

By introducing DSWP into HLS offers several advantages, compared to existing HLS techniques. First of all, due to its acyclic communication pattern, DSWP is generally tolerant to increases in communication latency between hardware modules [81]. Second, DSWP only pays the communication cost once to fill the pipeline. Third, the pipeline stages formed by DSWP can be optimized with other existing HLS techniques.

While DSWP provides an efficient way to enable existing HLS techniques for statically mapping irregular programs, we can also extend the architecture of spatial architecture to include the capability of dynamically mapping.

2.3 Existing Spatial Architectures

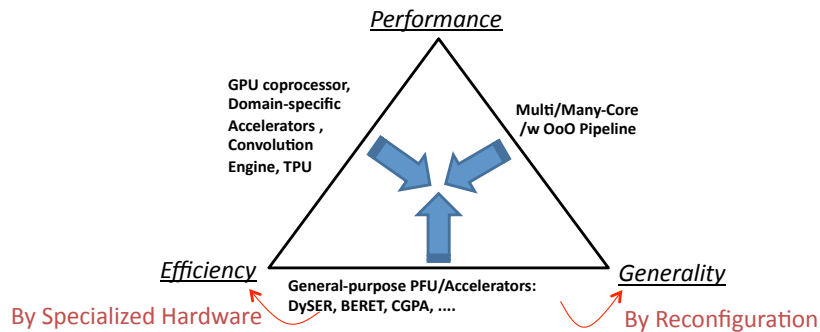


Figure 2.2: A Design Space of Computer Architectures (Convolution Engine [66], DySER [34], BERET [35], and TPU [46]).

In general, any computer architecture design attempts to balance performance, energy efficiency and generality, as shown in Figure 2.2. For example, OoO processors are designed to have limited number of functional units. These functional units are time-shared

using schedules generated at runtime. This type of design provides generality and performance, but wastes energy on generating instruction scheduling and delivering instruction/operands to these limited number of functional units. An alternative type of design, represented by Graphics Processing Units (GPUs), improves the energy efficiency by sharing one instruction pipeline (fetching, decoding, scheduling) with hundreds of ALUs. However this type of design usually is less general and is only applicable to scientific computation kernels with regular memory access.

Most existing spatial architectures are focusing on improving energy efficiency and generality. They usually contain a grid of functional units (or can be built from logic gates in FPGAs), such that the computational instructions can be distributed to these functional units spatially, without time-sharing. Because of this, the functionality of the functional units can be determined without complex pipeline logic, saving energy on instruction fetching, decoding and scheduling. Meanwhile this design simplifies the delivering of operands, saving a large portion of the energy cost associated with the datapath. The generality of spatial architecture follows from their ability to be reconfigured for different programs.

Table 2.3 summarizes the differences between the spatial architecture proposed by this dissertation and prior work. All the prior work can be classified into a few categories, according to their optimization targets:

Programmable Functional Units Programmable functional units [5, 10, 20, 34, 35, 71, 88, 92] such as OneChip, CHIMAERA and PRISC only consider short program traces or subgraphs (multiple dependent instructions in the program), and usually do not include memory operations. In these techniques, only energy consumed communicating intermediate results within the sequence can be reduced. BERET [35] classifies a set of common subgraph patterns for the superblocks (instructions across basic blocks [41]) in general

¹Note that BERET only supports control speculation by the assistance from compiler.

Reconfigurable Execution Engine	Compiler Effort		Hardware Feature				Target Instruction Range
	Placement Routing Not Required	Binary Compatible	Dynamic Mapping	Resource-aware Scheduling	Pipeline Execution	Dataflow	
PRISC [71]	x	x	x	x	x	x	HIMAERA Subgraph
CHIMAERA [92]	x	x	x	x	x	x	Subgraph
DySER [34]	x	x	x	x	✓	✓	Subgraph
ADRES [57]	x	x	x	x	✓	✓	Kernel
PipeRench [33]	x	x	x	x	✓	✓	Kernel
BERET [35]	✓	x	x	x	✓	✓	Subgraph
SGMF [85]	x	x	x	x	✓	✓	Kernel
Tartan [59]	x	x	x	x	✓	✓	Whole Program
WaveScalar[77]	x	x	x	x	✓	✓	Whole Program
CCA [14, 15]	✓	✓	✓	x	x	x	Subgraph
This work [53]	✓	✓	✓	✓	✓	✓	Kernel

Table 2.1: Comparison between DYNASpAM and other in-core reconfigurable computation engine.

purpose programs, and builds corresponding specialized hardware modules for each pattern. These works employ compiler techniques to extract and map subgraphs to the special functional units. CCA [14] also requires static subgraph extraction, but performs dynamic mapping.

Reconfigurable Spatial Co-Processors Another group of designs targets larger instruction sequences. Garp adapted the VLIW compilation technique to generate pipelined datapath on a fine-grained reconfigurable fabric [37]. ADRES [57] applies the same technique, but on a coarse-grained reconfigurable fabric with regular local connections between functional units.

General-Purpose Dataflow Architectures Tartan [6, 59] compiles entire programs onto spatially connected functional units, which operate completely asynchronously. Elastic CGRAs [38] uses a similar design but focuses more on gate-level implementations. The SGMF architecture [85] supports dynamic spatial dataflow execution and uses buffers in front of each functional unit to execute multiple invocations simultaneously. These techniques all require a static compilation to map instructions to the fabric, and their control edges for memory operations are conservative.

General-Purpose Spatial Processors In contrast to the dataflow architectures, RAW [79] supports both ILP and streaming instructions by routing operands between architecturally-exposed functional units over a point-to-point scalar operand network. TRIPS [74] and its successors such as TFlex [47] and T3 [72] use a compiler to find hyperblocks, and schedule each hyperblock in functional units individually. WaveScalar [77] uses a similar pipelined model as our work to execute *waves*, which are control flow graphs. It requires a new ISA to encode the global sequence of memory operations, which allows for dynamic reassembly to preserve program order.

Dynamic Trace Detection and Execution In addition to dynamic trace construction with trace cache [27, 73], many techniques optimize dynamically formed traces for high efficient backends. DIF [26, 62] dynamically compacts retired instructions for repeated execution on a Very-Long-Instruction-Word (VLIW) engine. HBA [23] and Yoga [84] select only hot traces and build VLIW/In-Order instruction streams for the retired instruction. CCA [14] dynamically maps instruction streams to spatial functional units with consideration given to placement but not resources. None of these techniques actively generate mappings during instruction scheduling. I-COP [12] builds a standalone coprocessor to complete binary optimization for incoming instruction streams, but it does not leverage existing micro-architecture features in OoO pipeline.

2.4 Issues with Existing Architectures and Their Mapping Techniques

According to the summary in the above sections, it is found that current spatial architectures either rely on software pipeline-based static mapping techniques to generate configurations or apply naïve dynamic mapping techniques to configure VLIW-like accelerators. Both these static and dynamic instruction mapping techniques can be improved, and the techniques proposed in this dissertation aim to solve these problems.

For the existing static instruction mapping techniques, the decoupled pipelining technique (DSWP) can be used to improve their applicability. This dissertation proposed CGPA, which splits the outer-loops of programs into sections with different execution patterns, thus existing static mapping methods can find more optimization opportunities in each code section (Chapter 3).

For the existing spatial architectures, they largely rely on compiler techniques to discover optimization opportunities statically and are lacking in adaptability. To improve this,

this dissertation proposed SPAS, which supports speculative execution by reusing hardware prediction units in the host OoO processors to detect hot traces for offloading to reconfigurable spatial fabric and speculatively control the execution on the fabric (Chapter 4).

Last but not least, to avoid the naïve instruction placement in the existing techniques, this dissertation also proposed DYNASPAM, a novel dynamic, resource-aware mapping technique for reconfigurable spatial fabrics (Chapter 5).

Chapter 3

Static Mapping with Coarse-Grained Decoupled Pipelining

The use of compiler techniques to map programs, which are written in high-level languages such as *C/C++*, to spatial fabrics statically is an active area of research. These fabrics can be fine-grained (bit-level reconfigurable) FPGAs or coarse-grained (word-level reconfigurable) reconfigurable arrays (CGRAs). This technique of mapping programs to hardware by compilation, is usually referred to High-Level Synthesis (HLS). HLS tools dramatically reduce the non-recurring engineering cost of creating specialized hardware from high-level languages. This enables fast prototyping and is also used for generating low power computations recently.

Existing HLS tools can successfully synthesize efficient accelerators for program kernels with regular memory accesses and simple control flows (mostly consisting of affine loop nests). Such loop nests enable a series of loop transformations that expose loop level parallelism to overlap instruction execution [16, 70, 89]. Additionally, the affine loops may also enable special circuit modules, such as systolic arrays and shift registers, to reduce memory traffic and improve performance [18, 19, 46]. However, in the presence of loops

with complex control flows or irregular memory accesses, these HLS tools can only invoke computational units of the fabric sequentially to execute instructions. Thus, no parallelism is exploited, which leads to poorly performing accelerator designs. In reality, a large portion of programs are implemented with non-array data structures and imperfect loop nests. This restricts the scope for parallelism within one or a few loop iterations, and generates hardware modules with limited performance improvement over general-purpose processors [82]. Therefore, a new technique is required to generate efficient accelerators for a large class of programs with irregular memory accesses and imperfect loop nests.

To solve this problem, this dissertation proposes Coarse-Grained Pipelined Accelerators (CGPA), an HLS framework which uses coarse-grained pipeline parallelism to generate efficient hardware accelerators for loops from unannotated C/C++ programs. CGPA leverages two distinct insights to improve efficiency and applicability of HLS. First, complex loop bodies with irregular memory accesses and imperfect loops usually contain coarse-grained code sections performing different tasks. HLS tools can separate and modularize these tasks to build an efficient system. Second, these complex loop bodies usually contain sections that are parallelizable. Coarse-grained decoupled software pipelining techniques [69, 68] can exploit the presence of these parallelizable sections to enable a type of parallelism not exploited by existing HLS tools.

CGPA automatically partitions individual loops into separate pipelined stages and generates buffer-connected hardware modules for these stages. Pipelining enables the overlapping of execution of an earlier iteration of the loop with a later iteration and also allows the synthesized hardware to tolerate variable memory latency. CGPA also utilizes hidden data-level parallelism within the pipelined stages to achieve high performance.

3.1 Motivating CGPA

Figure 3.1 shows the source code of the main loop in “em3d”, which simulates electron microscope tomography by constructing two linked lists to build a N -to- N bipartite graph [9]. This code has a number of features that motivate CGPA: recursive data structures, irregular memory accesses, and non-affine loop nests.

The nodes in one linked list contain an array of pointers to the nodes of the other linked list, i.e. the (read-only) “from” node is disjoint from the (updated) nodes of the traversed linked list. CGPA implements several static analysis that can determine these facts [21, 29, 30, 44].

The input to the core em3d algorithm consists of nodes in a linked list. Each node has four data members: `value`, `from_count`, an array of `from_nodes` which points to the nodes of the other linked list, and an array of `coeffs` (*coefficients*) for each `from_nodes`. The outermost loop *traverses* the linked list (line 1 in Figure 3.1(a)), and *updates* the value of each node by subtracting all the weighted values of its `from_nodes` using an inner loop (line 2-6 in Figure 3.1(a)). Even though this inner loop has some independent instructions across iterations, an attempt to exploit loop parallelism for this inner loop may fail because:

1. The iteration count, determined by `nodelist->from_count` (less than 10 for most cases), limits the amount of parallelism that can be exploited and also introduces the overhead of determining the control of loop exits. As a result, loop optimizations such as software pipelining cannot be applied.
2. The final weighted value reduction step in the loop induces a loop-carried dependence, which prevents a full application of data-level parallelism. Also the non-constant loop iteration numbers for each node disable the applicability of a circuit

structure like the *reduce* module, which is implemented as a tree and aggregates results from the the previous stage in [66].

Thus, the scheme for generating efficient accelerators for em3d should focus on its outer loop. At the algorithm level, we can divide the outer loop computations into two sections: *traversal* and *update*. The linked list traversal section (line 1 in Figure 3.1(a)) determines the address of the node used in an iteration and also the termination of the loop. We call this set of instructions a *Sequential Section* because fetching the node addresses must be serialized. Furthermore, because this section has no side-effects (for example, among other things, it does not store to the memory), we refer to this special sequential section as a *Replicated Section*, which means it is safe for multiple hardware modules to execute it redundantly. The *update* section for each node (lines 2-6 in Figure 3.1(a)) in one iteration is independent of updates to all other nodes, and can thus be executed in parallel (as long as the node addresses are known). We refer to this section as a *Parallel Section*. Existing HLS tools cannot optimize this outer loop, due to the existence of the Sequential Section, which introduces a loop-carried dependence, non-affine memory access and non-constant loop boundary.

CGPA can apply two novel loop parallelism techniques to build high performance hardware modules for this outer loop. One technique, called **replicated data-level parallelism** leverages the insight that computations from replicated sections can be safely executed as multiple parallel copies [43]. For example, CGPA could create four identical copies of the traversal section — one for each hardware module, as shown in Figure 3.1(b). During execution, each hardware module executes both fetch and update in one iteration. In the next three iterations, the module skips update and only executes fetch. By replicating fetch and distributing updates in a round-robin manner across the four hardware modules, CGPA can create replicated data parallel accelerators for em3d. The redundant fetching is useful

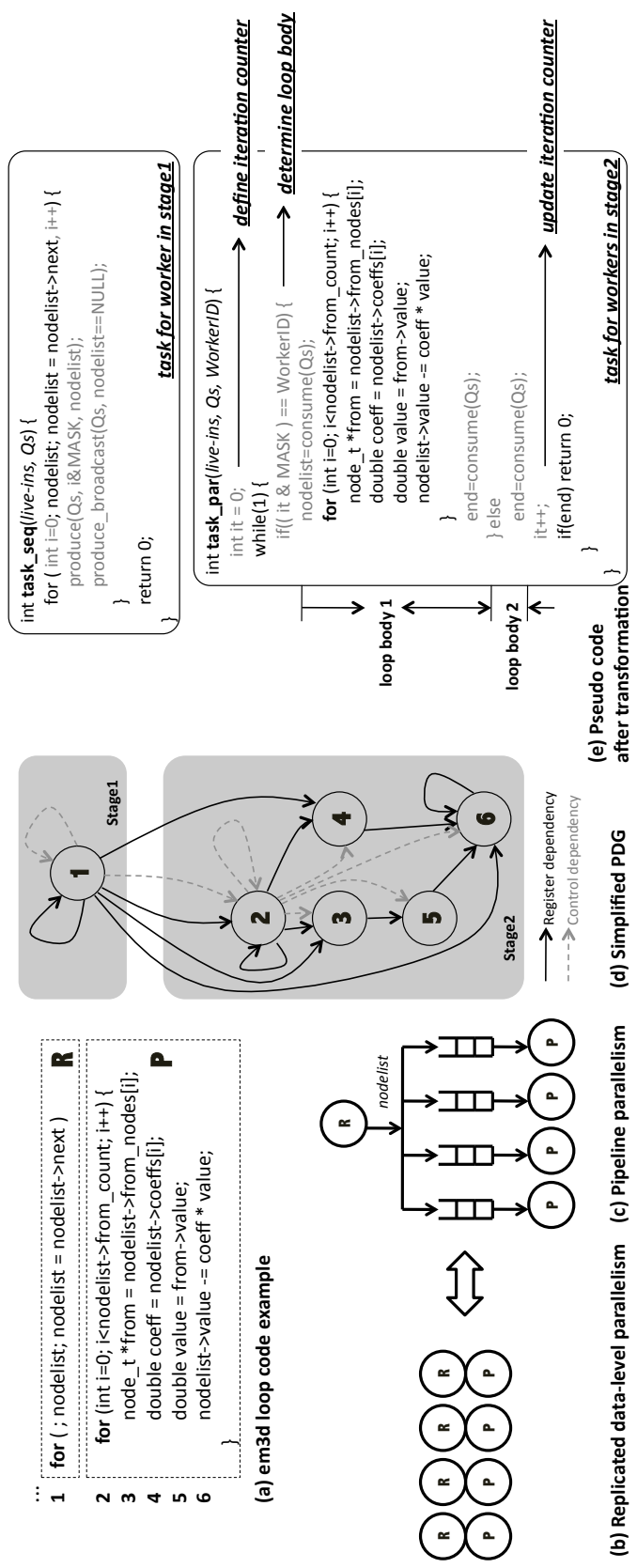


Figure 3.1: (a) Source code of the main loop of “em3d” program, with sections annotated as replicable or parallel; (b) Data-level parallelism is exploited by duplicating the replicable section; (c) Coarse-grained pipeline parallelism is exploited by separating the replicable section from the rest of the loop; (d) A simplified source-level Program Dependence Graph (PDG) of the main loop in em3d; (e) Pseudo-code of tasks for both hardware stages. The code in gray is the overhead generated by the CGPA compiler.

to calculate the correct node addresses for the corresponding updates. However, it causes unnecessary memory access overhead in each module.

CGPA can also adopt another approach, called **decoupled pipeline parallelism** to improve outer loop performance and generate accelerators similar to the results of manual accelerator designs [68]. This approach uses a set of FIFO buffers to separate the linked list traversal module from that for node updates. Since the traversal section only goes over the linked list and fetches node addresses, it can progress much faster than the update section. Thus one *sequential* traversal module can supply node addresses to multiple *parallel* update modules in another stage, as shown in Figure 3.1(c).

With this decoupled pipeline design, when control enters the loop, the hardware modules for both stages are invoked by the same start signal. The module in the first (sequential) stage begins fetching node addresses one by one, and assigns the node address values to the FIFO buffers of the parallel modules in the parallel stage in a round-robin fashion. The sequential stage stalls when there are cache misses or the corresponding buffers are full. Each module in the second (parallel) stage waits until there are node addresses in its buffer, and starts to process the update by fetching the node address directly from the buffer. After completing one iteration, the module in the parallel stage can get another pointer from the buffer, or stalls if the buffer is empty. This pipelining execution method brings the following two benefits:

1. **Tolerating Variable Latency:** In this example, memory accesses during the linked list traversal are irregular and might have variable latency due to cache misses. However, the buffers between stages ensure that the impact of variable latency is limited to one stage and does not cause stalls in the subsequent stages as long as the buffers are not empty.

2. **Enabling Higher Parallelism:** Since the sequential stage is split from the parallel stage by FIFO buffers, the updates of nodes from different iterations become completely independent of each other, thus enabling extra data-level parallelism within the parallel stage.

Besides the link list traversal example discussed in the context of em3d, many other programs can be mapped to decoupled hardware modules. Figure 3.2 makes a classification of some loop patterns existing in the programs, and the efficient hardware mappings for them. For example, the decoupled pipeline parallelism model (left bottom of Figure 3.2) has proved efficient for streaming applications [32], and has also been used to design accelerators for applications such as hash indexing [48]. In [48], the hash key generation and index traversal are decoupled to increase the throughput of the system.

CGPA is the first HLS tool to automatically extract this type of parallelism from a *single loop* and generate efficient pipelined hardware modules for it.

3.2 Coarse-Grained Pipeline Accelerators

Figure 3.3 shows a logical view of coarse-grained pipelined accelerators with a Sequential–Parallel–Sequential (S-P-S) pipeline. The number of stages for different applications is not fixed, and is determined automatically for each application by the CGPA compiler’s partition algorithm. The significant difference between the CGPA designs and existing accelerator designs is that there are multiple stages of the accelerator for one *single loop* and they are separated by FIFO buffers. Each hardware module with independent control that implements instructions from the original loop is called a *worker*. Each worker has its own independent control circuit and dedicated memory ports to the cache.

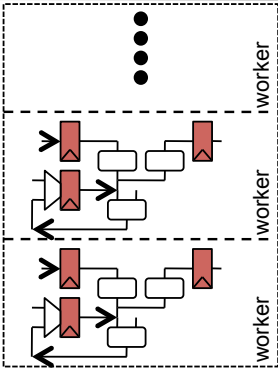
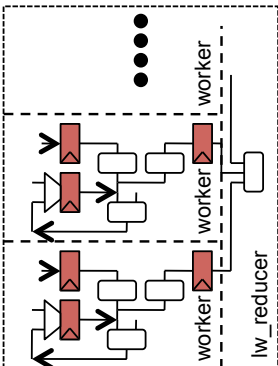
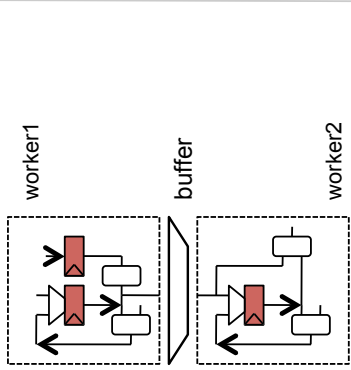
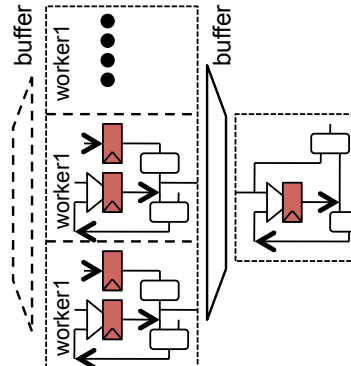
<p>Software Engineers' View</p> <pre>for(int i=0; i<N; ++i) { worker(A[i], B[i]) }</pre> <p>Example: - Vector Addition</p>	<p>Hardware Engineers' View</p> 	<p>Software Engineers' View</p> <pre>for(int i=0; i<N; ++i) { t = worker(A[i], B[i]) lw_reducer(t); }</pre> <p>Example: - Sum reduction - Convolution kernels</p>	<p>Hardware Engineers' View</p> 
<p>Software Engineers' View</p> <pre>for(int i=0; i<N; ++i) { t = worker1(ptr1(i), ptr2(i)) worker2(t, ptr3(i)); }</pre> <p>//ptr1(i) may be alias with ptr2(i) //ptr3(i) is alias with ptr1(i) & ptr2(i)</p> <p>Example: - Stream applications</p>	<p>Hardware Engineers' View</p> 	<p>Software Engineers' View</p> <pre>for(int i=0; i<N; ++i) { t = worker1(A[i], B[i]) worker2(t, ptr3(i)); }</pre> <p>//ptr1(i) maybe alias with ptr2(i) //ptr3(i) is alias with ptr1(i) & ptr2(i)</p> <p>Example: - K-means - em3d - hash-index</p>	<p>Hardware Engineers' View</p> 

Figure 3.2: Different program patterns exploiting parallelism for hardware accelerator design.

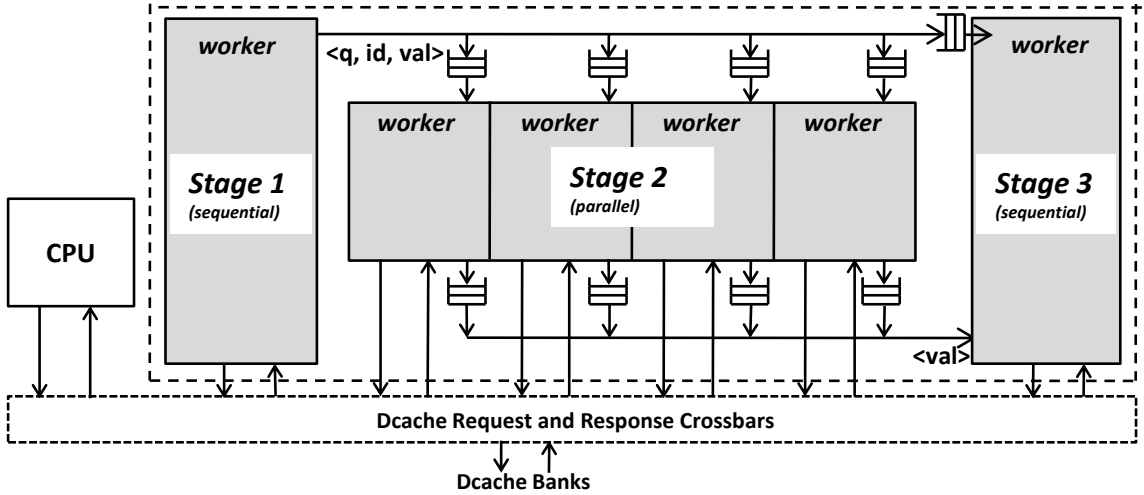


Figure 3.3: A logical view of CGPA architecture within the dashed box. Each grey box contains circuit modules customized for the targeted loop and generated by CGPA compiler. In this figure, a Sequential–Parallel–Sequential (S-P-S) 3-stage pipeline is shown.

The design of this pipelined accelerator can be embedded in a general-purpose co-processor such as conservation cores [82] and Legup accelerator [8] to improve the performance of these co-processors. It can alternatively be implemented as a standalone accelerator to improve the performance of targeted loops. This paper explores the former configuration.

3.2.1 CGPA Workflow

Figure 3.4 shows the workflow of the CGPA tool. The tool is built on top of the LLVM compiler infrastructure [49] (revision 164307). The tool accepts unannotated sequential C/C++ programs. The LLVM front-end (clang) translates the source code to intermediate representation (IR) for further analyses and optimizations. The compiler identifies hotspots in the code via a simple profiling step. Then, it applies a series of analyses and code transformations to create pipeline specifications from the IR. Subsequently, the compiler splits the program into two parts: one to be executed on the general-purpose processor

and the other (containing the transformed pipelines) to be implemented as accelerators. Additionally, the compiler generates wrapper functions to invoke the accelerators from the processor and pass them the necessary arguments. The code to be executed on the general-purpose processor is then compiled to binary, and a hardware backend translates the second part to Verilog descriptions and finally the device programming files.

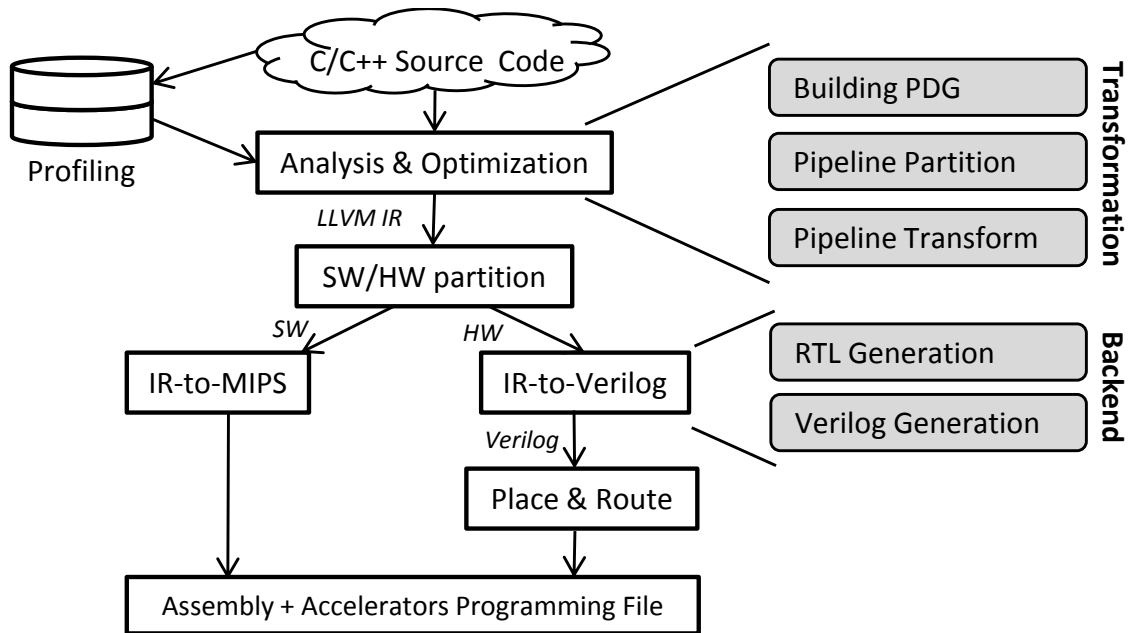


Figure 3.4: Workflow for CGPA HLS Framework

3.2.2 Pipeline Generation

The specification of the pipeline is generated during the analysis and optimization phase in the compiler. In this phase, a set of common optimization passes such as dead code elimination, strength reduction, and scalar optimizations are applied before generating the actual pipeline. There are three main steps in the pipeline generation stage: building the program dependence graph (PDG), pipeline partition, and pipeline transformation.

Building the PDG: The compiler first builds a program dependence graph (PDG) for each program. Each node in the PDG represents an instruction in LLVM IR and each

edge between the nodes represents a control or data dependence. During this process, a set of alias analyses can be applied to remove dependence edges between two memory instructions. For example, in the example of Section 5.1, several static analysis algorithms can determine that `from` and `nodelist` nodes are from different linked-lists and disjoint from each other [29]. After the PDG is built, the compiler consolidates all the strongly connected components (SCCs) in the PDG to create a directed acyclic graph (DAG) [24, 45]. Each component is classified as: parallel, replicable, or sequential [43]. Parallel SCCs contain no loop-carried dependences and thus, can be executed in parallel. The other SCCs are either replicable (do not contain an instructions with side-effects) or sequential (may contain instructions with side-effects).

Pipeline Partition: This is a coarse-grained instruction scheduling step that assigns instructions to different pipeline stages. The partitioning algorithm is adapted from the Parallel Stage Decoupled Software Pipelining (PS-DSWP) [68] algorithm. The main difference between CGPA and PS-DSWP lies in the identification of replicable sections, and deciding whether it should be inserted in the parallel stage as replicas or into the sequential stage. Generally, inserting replicable sections into a sequential stage will increase communication, because the results of the replicable sections must be sent explicitly to the following stages. Conversely, duplicating the replicable section in parallel stages will increase the amount of computations and memory accesses. The CGPA framework only duplicates lightweight replicable sections which do not contain load and multiply instructions. This is based on the intuition that time and resources required for the replicable section without these instructions are less than those for communicating results via FIFO buffers. Figure 3.1(d) shows that replicable sections with heavyweight load instructions are identified and inserted into a sequential stage (Stage 1), and the remaining instructions are grouped as one parallel stage (Stage 2). The required number and stage connections of the buffers are also determined at this step.

Pipeline Transform: This step forms a set of control-equivalent loops for the workers based on the results of the partition step. Control-equivalent means that all workers from all stages have the same loop iterations and exit points as the original loop, even though their bodies have been assigned different instructions. Then the compiler creates tasks for each stage, with the loop live-ins and buffers as arguments. Tasks for parallel stages have one additional argument to indicate the unique identification (ID) for the worker (relative to all other workers in the same stage). For the body of the tasks, the instructions of the original loop are distributed according to the result of the pipeline partition. Moreover, loop control branch instructions of the targeted loop are also duplicated across the tasks and the destination of original branch instructions are modified to recreate the original loop structure in each task.

Both register and control dependences between stages are communicated via FIFO buffers, and the compiler automatically inserts communication primitives into tasks. For data dependences, if the *definition* and *use* of a variable are in different stages, a *produce* primitive is inserted after the definition, and a *consume* primitive is inserted at the point in the later stage which corresponds to the definition. Also for control dependences, the compiler needs to *broadcast* the condition of the branch instructions to all the workers in the following stages as a data dependence, as shown in Figure 3.1(e).

One significant difference between CGPA partitioning and previous work [68] is the handling of the replicable section and loop termination. Duplicating lightweight replicable sections introduces loop-carried dependences in the parallel stage. To solve this, the CGPA compiler creates two copies of the loop body in the tasks for workers in a parallel stage, as shown in Figure. 3.1(e). One copy (loop body 1) has real computations which are the instructions (both parallel and replicable sections) assigned to this worker; another copy (loop body 2) is only for the replicable section. The compiler creates a new basic block to use the worker ID and iteration counter to decide which loop body the control should

enter at the beginning of each iteration. For the loop termination of workers in the parallel stage (when the parallel stage is not the first stage), the same strategy is adopted, and the branch exit condition from the previous stage is broadcasted to all the workers in the following stages and guarantee that they can exit when the previous stage finishes, as shown in Figure 3.1(e).

Once tasks are generated, the original loop in the parent function can be replaced by a set of calls to the generated tasks. This is done by inserting *parallel_fork* and *parallel_join* primitives with the corresponding arguments. The live-outs are communicated back to the original parent function by inserting *store_liveout* exactly before the exits of the tasks, and *retrieve_liveout* before the uses of the liveouts. Table 3.1 summarizes all the primitives inserted by the compiler during the pipeline transformation.

Class	Primitive	Arguments	Description
1	<i>parallel_fork</i>	<i>LoopID</i> , <i>Task</i> , <i>Liveins</i> , <i>Buffer</i> , <i>WorkerID</i>	In the current state, invoke a hardware module for <i>Task</i> (associated with <i>LoopID</i>) and read register values of <i>Liveins</i> as input. If this is a parallel worker, <i>WorkerID</i> is used as one extra input.
	<i>parallel_join</i>	<i>LoopID</i>	Stall in current state until all the workers related to <i>LoopID</i> have raised the finish signal
2	<i>produce</i>	<i>Buffer</i> , <i>WorkerID</i> , <i>Value</i>	Push <i>Value</i> to the FIFO <i>Buffer</i> with index <i>WorkerID</i>
	<i>produce_broadcast</i>	<i>Buffer</i> , <i>Value</i>	Push <i>Value</i> to all the workers connected to the FIFO <i>Buffer</i>
	<i>consume</i>	<i>Buffer</i>	Pop a value from the connected FIFO <i>Buffer</i>
3	<i>store_liveout</i>	<i>LiveoutID</i> , <i>Value</i>	Store a liveout value, <i>Value</i> with ID <i>LiveoutID</i> in a register
	<i>retrieve_liveout</i>	<i>LiveoutID</i>	Read value for a liveout with ID <i>LiveoutID</i> from the corresponding register

Table 3.1: New primitives added to LLVM IR to support worker invocation, dependence communication, and register value passing across hardware modules.

3.2.3 CGPA Compiler Backend

RTL Generation: The CGPA compiler first builds the RTL description of hardware modules from IR. Subsequently, it generates the Verilog code automatically from the RTL specification. For RTL generation, CGPA utilizes an open-source Verilog backend of LLVM [8]. In the instruction scheduling phase, it creates a control flow graph (CFG) of the offloaded

program. Then, the nodes of the CFG are split into multiple states of a finite state machine (FSM), after scheduling instructions at different clock cycles (represented by one state in the FSM). CGPA adopted the SDC (system of difference constraints) scheduling algorithm to assign instructions to the FSM states. The SDC algorithm solves the instruction scheduling problem by converting data dependence, control dependence, instruction latency, cycle time and available resources to a set of constraints in a linear programming (IP) problem. Since CGPA compiler generates pipelined parallel modules, scheduling constraints for the new primitives should be added to the basic dependence constraints and timing constraints to ensure correctness and performance. Here the notations from [17] is used to express the set of new constraints that must be preserved: $sv_{beg}(v)$ represents the scheduling variable associated with the starting state of instruction v ; $C_k(l)$ are primitives from Class k with LoopId l (Table 3.1), M is the set of memory access instructions, B represents branch instructions. Then the following additional scheduling constraints are introduced:

$$\forall f_i, f_j \in C_1(l) : sv_{beg}(f_i) - sv_{beg}(f_j) = 0 \quad (3.1)$$

$$\forall f_i \in C_1(l1), f_j \in C_1(l2) : |sv_{beg}(f_i) - sv_{beg}(f_j)| \geq 1 \quad (3.2)$$

$$\forall m \in M, \forall prodcons \in C_2 : |sv_{beg}(m) - sv_{beg}(prodcons)| \geq 1 \quad (3.3)$$

$$\forall b \in B, \forall lo \in C_3 : sv_{beg}(b) - sv_{beg}(lo) = 0 \quad (3.4)$$

Constraint 3.1 and Constraint 3.2 are for the Class 1 primitives in Table 3.1. f_i and f_j are two different primitive instances. Constraint 3.1 guarantees that the parallel invocation of hardware modules from the same loop will be within the same cycle, and only when all the modules finish, the state machine of the parent module will continue. Constraint 3.2 guarantees that the parallel invocation of hardware modules for different loops will not be invoked in the same cycle (must have at least one clock cycle difference).

Since memory operations may take multiple clock cycles, the Class 2 operations (*produce* and *consume*) are not side-effect free and should not be scheduled to the same state of memory operations. Otherwise, when memory operations stall the circuit, it causes multiple pops/pushes the same value to the FIFO buffers if they are scheduled within the same state. Constraint 3.3 makes sure the produce and consume primitive instances (*prodcons*) will not be scheduled with any memory operations (*m*) and have at least one clock cycle difference.

Finally, Constraint 3.4 allows stores of live-out values (*lo*) only when the loop exits (which is the branch instruction *b* in 3.4). The backend in [8] is fully utilized to translate the FSM with scheduled instructions to the RTL of datapath and control circuits.

Verilog Generation: Given the RTL specification, the backend of CGPA, which utilizes the Legup's implementation, generates the Verilog code automatically. However, to support all the primitives shown in Table 3.1, CGPA also includes a new hardware circuit library. During the Verilog generation phase, the wire connections between the generated modules and the hardware module in the library is completed automatically. Besides the Verilog code, the compiler also generates a test bench to verify the design. All the Verilog designs of the benchmarks passed the verification.

3.3 Evaluation of CGPA

3.3.1 Methodology

Evaluation Framework: The CGPA compiler is based on LLVM (revision 164307). The backend for CGPA is adapted from an open source Verilog backend [8], which generates both design and test bench files. The CPU/accelerators heterogeneous system from [8] is based on Altera DE4 which features a Stratix IV FPGA [1]. A 32-bit MIPS software core

executes the CPU part of the program. Both the instruction cache and the data cache are directly-mapped with 512 lines and 128 byte block size. The instruction cache and the data cache have 1 and 8 ports, respectively. For all the pipelined accelerators, the width of FIFO buffers is fixed to 32 bit, the depth is fixed to 16 entries and the number of workers in the parallel stage is fixed to 4. Quartus II 11.0 is used to synthesize, fitter and simulate the accelerator part of the program. The targeted synthesis frequency is set to 200MHz. The accelerators generated by CGPA have all been implemented and verified. For performance, the number of cycles for which the kernels of various programs were running is measured. The ALUT resource usage is obtained after place and route, and the power estimated is given by PowerPlay [1] with obtained activity files from a post-fitter simulation.

Benchmarks: Table 3.2 shows the descriptions of the set of kernels accelerated using CGPA and also the corresponding pipelined stages generated by CGPA. These benchmarks are chosen because they belong to different domains such as machine learning, graph partitioning, and image processing. Additionally, we are not aware of HLS synthesized accelerators for some of these kernels (hash indexing, em3d, and ks).

Benchmark	Domain	Description	P1	P2
K-means [11]	Machine Learning	Finding the nearest cluster for each node and updating its position	P-S	-
Hash-indexing [48]	Database	Computing hash key for each node and indexing it in a linked-list	S-P-S	-
ks	Graph Partition	Traversing doubly-nested linked-lists to find a max grain of swapping	S-P-S	-
em3d [9]	3D Simulation	Updating value for each node in a linked-list by subtracting weighted value in from_nodes	S-P	P
SIFT 1D-Gaussblur [55]	Image Processing	1D row Gaussian blurring; pipeline vectorization was applied to reduce memory access	S-P	P

Table 3.2: Descriptions of benchmarks used. P1: Pipeline Partition with Replicable Section in Sequential Stage; P2: Pipeline Partition with Replicable Section in Parallel Stage

3.3.2 Results

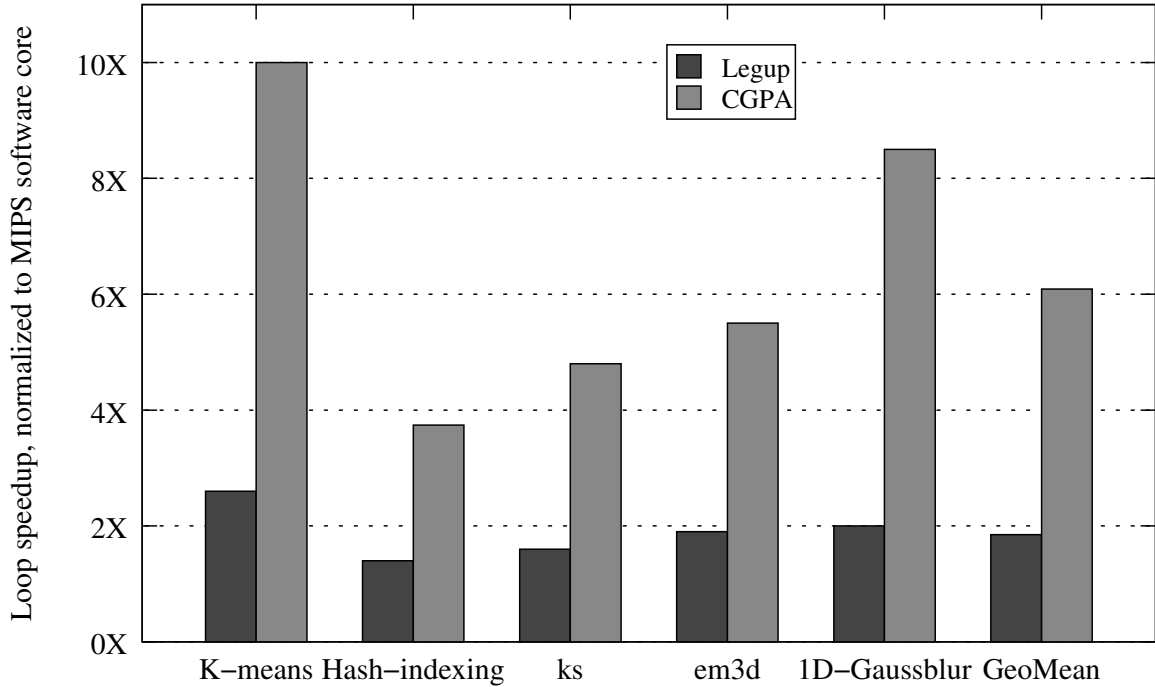


Figure 3.5: Speedup Results for Loop Kernels of the Benchmarks

Performance: In the experiment, three data points for each loop kernel are considered: (1) Performance on a MIPS software core; (2) Performance of hardware accelerators generated by Legup [8]; and (3) Performance of hardware accelerators generated by CGPA. Legup was chosen for comparison because it can generate accelerators for each kernel mentioned in Table 3.2. Additionally, other HLS tools targeting general-purpose programs use a framework similar to Legup [82]. Figure 3.5 shows the loop speedup numbers for the kernels, relative to the performance on the MIPS software core. The Legup HLS tool gives a 1.85x geomean speedup over the software core. CGPA gives a geomean of 3.3x speedup

over the performance achieved by Legup and a geomean of 6.0x speedup over the MIPS software core.

Area and Power: Table 3.3 shows the area and energy overheads for each kernel. For most benchmarks, the ALUT usage relative to Legup is approximately 4.1x. This is not surprising since CGPA creates four parallel workers in the parallel stage to increase performance. Another type of overhead comes from the usage of BRAM to build the FIFO buffers, which are not included in the ALUT usage. Table 3.3 also shows the power and energy dissipation for each benchmark. The results show that a geomean of 20% energy dissipation overhead is generated by CGPA, over the accelerators generated by Legup for the benchmarks. The sources of the energy overhead are passing values via the FIFO buffers, multi-port cache support and other computation overhead. On the other hand, the energy efficiency column, obtained by dividing the energy dissipation of MIPS software core by that of accelerators, shows CGPA can save significant energy compared to the MIPS software core.

Benchmark	Type	ALUT #	power (mW)	energy (uJ)	energy efficiency
K-means	Legup	1696	46	22.1	7.3
	CGPA (P1)	7197	162	22.9	6.9
Hash-indexing	Legup	421	47	12.1	6.7
	CGPA (P1)	2052	150	14.6	5.5
ks	Legup	1371	60	104.5	6.7
	CGPA (P1)	5741	233	131.7	5.3
em3d	Legup	623	72	1.66	6.4
	CGPA (P1)	2842	292	2.24	4.7
	CGPA (P2)	2624	305	2.49	4.2
SIFT 1D-Gaussblur	Legup	1319	53	1.27	7.4
	CGPA (P1)	3806	183	1.35	6.9
	CGPA (P2)	4168	194	1.55	6.0

Table 3.3: Comparison between CGPA and related frameworks

Tradeoff: To explore the tradeoff between computation and communication in the presence of replicable sections, replicated data-level parallelism for em3d and 1D-Gaussblur is also enabled by duplicating the replicated stage in the parallel workers (reported as P2 in Table 3.3). The pipelining method (P1) outperforms the duplicated replicable section method (P2) by 6% and 15% for em3d and 1D-Gaussblur, respectively. Moreover, as shown in Table 3.3, the pipelining method can reduce energy dissipation by 11% and 14% respectively for these two benchmarks. For the other benchmarks, replicated data-level parallelism was not found applicable.

3.4 Applicability and Scalability of CGPA

This section shows the applicability of CGPA by describing two interesting cases from the evaluated set of benchmarks, and also the scalability of the technique by exploring some design parameters.

3.4.1 Applicability

Two examples, K-means and 1D Row Gaussian Blur, described below show that CGPA understands the intention of different loop patterns and generates profitable hardware modules to improve the performance.

Example One: K-means

For K-means, the CGPA compiler builds a pipelined accelerator by targeting the loop for deciding cluster membership for each point and updating new cluster centers. Figure 3.1 shows the source code for the loop. This loop first finds the *membership* of each input data point by using the `findNearestPoint` function. This function calculates the Euclidean distance between a data point and the center of each cluster, then returns the *index* of

```

for (int i = 0; i < numNodes; ++i ) { R
    int index = findNearestPoint (nodes[i], nFeatures, clusters, nClusters); P
    if(membership[i] != index)
        delta += 1;
    membership[i] = index;
    new_centers_len[index] += 1;
    for(int j = 0; j < nFeatures; ++j)
        new_centers[index][j] += nodes[i][j];
} S

```

Figure 3.1: The targeted loop for K-means algorithm, along with the identification of different sections.

the cluster corresponding to the minimum distance. The returned *index* is used to assign membership to the data point and to update the positions of the corresponding cluster center.

The CGPA framework indicates that the targeted loop can be separated into three unique sections (shown in Figure 3.1). The induction variable calculation, which determines the index of data points used in an iteration and also the termination of the loop, is identified as a *Replicable Section*. The call to the `findNearestPoint` function can be executed independently for each data point, and thus is identified as a *Parallel Section*. The rest of the loop contains updates to three objects: `membership`, `new_center_len`, and `new_centers`. These updates are executed for each iteration, and cannot be overlapped with the updates from the other iterations. The CGPA framework identifies these updates as belonging to a *Sequential Section*.

In the pipeline transformation step of CGPA, the Parallel Section is deployed as the parallel stage in the pipeline, which then is translated into parallel hardware modules as shown in Figure 3.2. The Sequential Section is transformed to one hardware worker, which is connected to the parallel workers in the earlier stage via FIFO buffers. The compiler also identifies that the Replicable Section is lightweight, so it is duplicated across all workers.

Thus, each worker has its own induction variable calculation. One 4-channel FIFO buffer is generated to hold the index from the workers in the parallel stage. The sequential worker completes its task by fetching index values from the buffers on a round-robin basis. By separating the sequential worker from the main parallel computations in the algorithm, maximum parallelism can be exploited while ensuring correctness of execution.

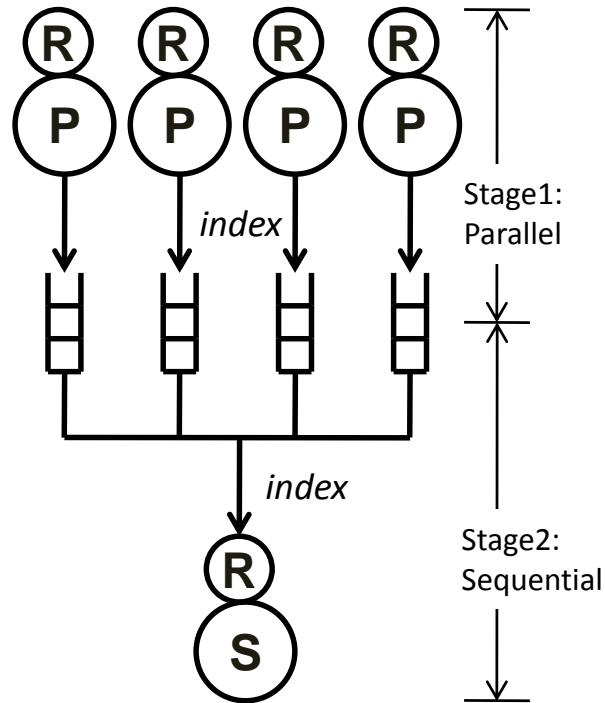


Figure 3.2: Pipeline generated for K-means by CGPA.

Some existing HLS tools have targeted the inner loop for the K-means kernel. For example, Gokhale et al. proposed the use of a systolic array structure for the calculation of Euclidean distances within the `findNearestPoint` function, based on an language extension of C, called Stream-C [32]. However, the CGPA framework differs from this approach in the following ways:

- The results of CGPA are more general (in terms of the number of different clusters and input data points), since CGPA does not assume a fixed number of clusters; and

- CGPA targets the outer loop, which potentially has a higher degree of parallelism, since the number of data points is typically orders of magnitude larger than the number of cluster centers.

CGPA is a coarse-grained pipeline parallelism technique for individual loops, and its transformation technique usually does not change the code structure of inner loops, which can be targeted by existing HLS techniques. In this example, the calculation of the Euclidean distances between the nodes and the centers can still be optimized by applying systolic array structures if the number of clusters is a constant. Thus, CGPA can be seen as complementary to existing work.

Example Two: 1D Row Gaussian Blur

```

for (int i = 0; i < height; ++i) {
    float img0 = img[i][0];
    float img1 = img[i][1];
    float img2 = img[i][2];
    float img3 = img[i][3];
    float img4 = img[i][4];
    for (int j = 0; j < width-4; ++j) {
        intermediate[i][j] = coef0*img0 + coef1*img1 + coef2*img2
            +coef3*img3+ coef4*img4;
        img0 = img1;
        img1 = img2;
        img2 = img3;
        img3 = img4;
        img4 = img[i][j+5];
    }
}

```

The code is annotated with dashed boxes and labels on the right side:

- R1**: Encloses the inner loop header `for (int j = 0; j < width-4; ++j) {`.
- P**: Encloses the calculation of `intermediate[i][j]`.
- R2**: Encloses the shift of image values: `img0 = img1; img1 = img2; img2 = img3; img3 = img4;`.
- R3**: Encloses the update of `img4`: `img4 = img[i][j+5];`.

Figure 3.3: Source code for targeted loop in 1D Gaussian Blur, along with the identification of different sections.

CGPA is able to generate parallel accelerator designs for both the 1D row and column Gaussian Blur kernels in Scale-Invariant Feature Transform (SIFT) program. This section only shows the result of the 1D row Gaussian Blur kernel, whose code is shown in Figure 3.3. For a certain row i , the loop has a window of size 5 moving from the left to the right of that row, and calculates a weighted sum reduction of all the image points within the window. In prior work, a series of optimizations, namely scalar replacement and pipeline vectorization [18], have been utilized to optimize the number of memory accesses for the loop. In our evaluation, these optimizations are applied for the CPU baseline, Legup and CGPA.

The CGPA framework is able to identify four different code sections for the targeted loop. Three of the four sections are identified as Replicable Sections. The first replicable section (labeled R1) performs induction variable calculations, which gives out the column index of the image data. The second replicable section (labeled R2) performs data swaps. The third replicable section (labeled R3) fetches new image data. The fourth section is a parallel section, which performs a weighted sum reduction of each window position. The sum reduction for each position can be performed independently of that for other positions, thus explaining why this section is identified as a parallel section.

CGPA handles the three replicable sections in different ways, according to their features. Since R1 and R2 are lightweight, they are replicated in the workers. R3 has a load instruction, thus it is inserted into a separated sequential pipeline stage. As a result, R1 is replicated in both the sequential stage and parallel stage, because it performs all the induction variable calculations. R2 is only replicated in the parallel stage, because its result is only used in the parallel stage. For each iteration, R3 fetches image data and broadcasts it to all four shift register chains (R2) in the second stage. A final pipeline partition and transformation generated by CGPA is shown in Figure 3.4.

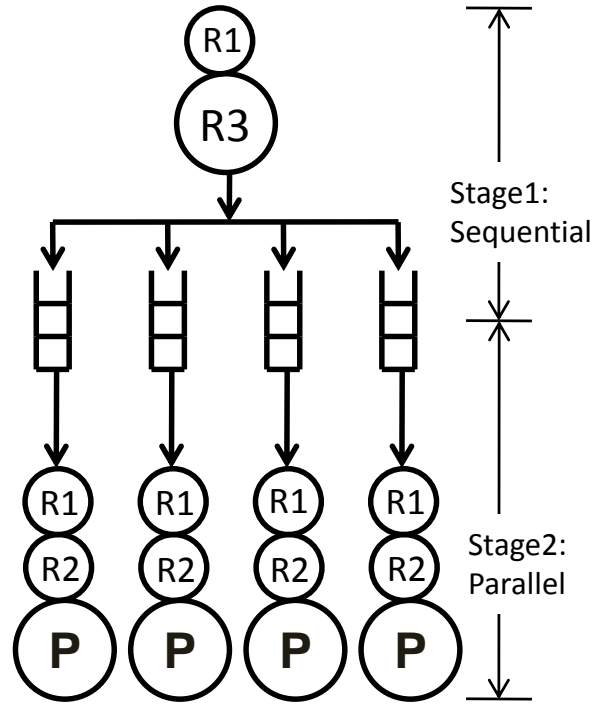


Figure 3.4: Pipeline generated for 1D Row Gaussian Blur by CGPA.

Compared to the pipeline vectorization technique which can only generate one reduction window [18], CGPA builds four parallel reduction windows to generate four intermediate data concurrently. This is not just a simple duplication of reduction windows that allows them to work independently (like the results generated by inserting R3 in a parallel stage). Experimental results show that decoupled pipelining technique (DSWP) improves the performance by 15% and reduces energy cost by 14%. Furthermore, the *map-reduce* style circuit in [66] can also be applied in the Parallel stage to improve the sum reduction performance. Again, this technique is complementary to CGPA.

3.4.2 Scalability

Due to the limitations of the experimental platform, this paper only shows cases with a maximum of 4 parallel workers in the parallel stage. The degree of parallelism that can be

potentially exploited by CGPA is larger than this number for all the benchmarks. In the ideal case, the scalability of CGPA depends on three issues:

Workload of the Sequential Stage: Increasing the workload on the sequential stages has two effects: (1) it may stall the parallel stage through the FIFO buffers, and (2) it may limit the overall speedup in accordance with Amdahl's law. The pipeline partition algorithm in CGPA tries to find maximum parallel section of the loop body, thus reducing the workloads of the sequential stages.

Workloads of the Replicable Sections in the Parallel Stage: If the number of workers in the parallel stage is increased, there is an increase in the chance that execution enters the loop body which contains only the replicable section. This could result in higher overheads for the whole accelerator system. Thus, it is important for efficiency and scalability to decide where the replicable section must be inserted. In CGPA, the partition algorithm can intelligently calculate the pipeline balance and decide which replicable sections should be inserted in the parallel stage.

Memory system support: As shown in the experiments, CGPA tries to reduce memory access by reusing input data from a sequential stage. However, in CGPA, since each worker in the parallel stage has its own memory ports, the overhead of building shared memory system becomes large if the number of parallel workers increases. To solve this problem, some existing memory optimizations, such as private cache and memory partition techniques can be applied. CGPA's pipeline partition design enforces an assignment of aliasing memory instructions to the same stage (by creating SCCs), and this indicates that there are no data access conflicts from different stages. Thus, this allows the application of existing memory optimizations, such as local memory.

On the other hand, the results of static mapping largely depend on the accuracy of compiler analysis. Without strong alias analysis techniques, CGPA would not be able to split aliasing memory instructions into different stages, and no partitions can be formed. To solve this issue, this dissertation also proposes a dynamic mapping technique.

Chapter 4

Spatial Architecture Speculation with Hardware Reuse

Static instruction mapping techniques are widely utilized to translate software programs to fine-grained reconfigurable fabrics, such as FPGAs. However, *dynamic mapping* techniques are critical for enabling the deployment of coarse-grained reconfigurable spatial architectures in mainstream computing devices. Some coarse-grained reconfigurable spatial architectures, such as Programmable Functional Units (PFUs) [5, 10, 14, 20, 34, 35, 71, 88, 92] and Coarse-Grained Reconfigurable Arrays (CGRAs) [33, 57, 58, 67, 76, 85] have the advantage of being amenable to reconfiguration, making them more general-purpose. However, most of these existing CGRAs rely on static mapping techniques to generate configurations. Executed a priori, static methods cannot make use of information gathered at runtime to optimize their mappings for changing workloads. Additionally, programs that are statically mapped to a particular reconfigurable fabric cannot run effectively on

a processor without the fabric, and may not be compatible with different fabric generations [14, 15]. This is a significant limitation because lots of programs, running on commercial processors, are not open source, but users frequently upgrade their hardware and computational infrastructure.

Dynamic mapping methods can overcome the adaptability and compatibility issues that static methods are facing. However, due to small instruction scopes and lack of speculation, current dynamic techniques fail to make the best use of routing resources in spatial architectures [14]. This may lead to increased execution time and energy consumption, as well as, in some cases, an inability to produce a feasible mapping.

To solve this problem, this dissertation proposes Dynamic Spatial Architecture Mapping (DYNASPAM), which includes both the speculative spatial architecture (SPAS) design and also a pure dynamic, pluggable, hardware mapping module. The insight of SPAS are that OoO processors excel at utilizing speculation and contain large instruction windows, thus combining OoO resources with the execution efficiency of spatial architectures leads to a more effective system. SPAS efficiently and transparently integrates with, and utilizes the resources of, an OoO processor to dynamically map large instruction traces to a reconfigurable fabric. SPAS reuses the prediction, speculation and scheduling techniques in advanced OoO processor pipelines for spatial architectures. Specifically, SPAS reuses the branch predictor and allows for instruction traces that spanning multiple basic blocks to increase the scheduling window. SPAS also reuses the memory speculation unit and increases mapping freedom by breaking memory dependences. The proposed dynamic resource-aware mapping technique for reconfigurable spatial fabrics from SPAS also leverages the existing scheduling logic in the host processor to select and map instructions from a large scope with only a small hardware cost.

4.1 Motivating SPAS

To understand the need for SPAS, let us first consider the limitation of existing dynamic mapping solutions for spatial architectures.

4.1.1 Dynamic Mapping for Spatial Architecture

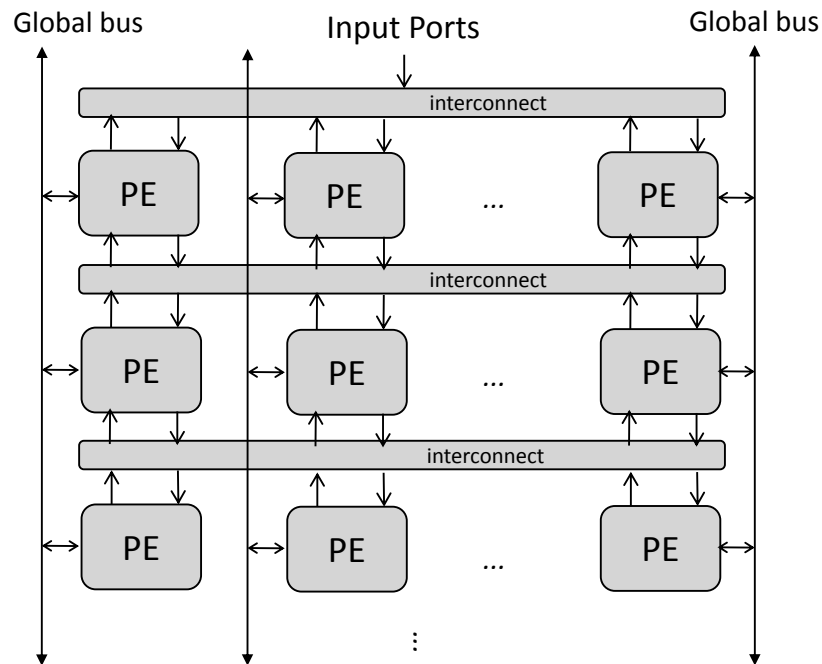


Figure 4.1: A generalized reconfigurable spatial architecture.

Figure 4.1 shows a generalized reconfigurable spatial architecture. Normally, the reconfigurable spatial architecture is organized as a grid of functional units, and each functional unit has connections to its neighbors (in the same row or different rows). Since different reconfigurable spatial architectures have different interconnection designs, we use a cross-bar like interconnect structure here to represent them. In real designs, this interconnect can be simplified by reducing the amount of connections. For example, in one row of PipeRench [33], a functional unit can only be connected to its adjacent functional units,

while the functional units in the same row of CCA [14] cannot communicate with each other at all.

Project	Internal Communication		External Communication
	same row	adjacent rows	
CCA [14]	-	cross-bar	top
DySER [34]	switch	switch	round
PipeRench [33]	switch	cross-bar	top & bus
This work [53]	-	cross-bar	top & bus

Table 4.1: Topologies of different spatial architecture examples.

Table 4.1 gives a summary of topologies for several different reconfigurable spatial architectures, which have different features for communicating data internally (between functional units) and externally (between functional units and the outside ports). Due to the two-dimensional physical arrangement of functional units in spatial architectures, the distribution of input and output ports is heterogeneous, i.e. the number of input and output ports each functional unit can access directly are very different. For example, only the functional units from the top row of CCA can directly access inputs [14], while only half of the functional units at the edge of DySER can directly access inputs [34]. Transferring input data to spatially internal functional units costs extra cycles and datapaths, or is even impossible. Obviously, this resource heterogeneity should be considered when mapping instructions to functional units to get good results. Applying global bus to connect spatially internal functional units can partially reduce the heterogeneity. The global bus can be shared by multiple producers and consumers, and provides flexibility in instruction mapping.

For both the static and dynamic mapping methods, the core problem is to satisfy the following three types of constraints when instructions are selected and placed to the functional units of a spatial fabric: *functionality constraint*, *resource constraint*, and *timing constraint* [63, 39, 40]. The description of each constraint and our methods to satisfy them

Constraint	Description	Solution or Heuristic
Functionality	Functional unit can provide the functionality	Instruction Selection Logic
Resource	Functional unit can provide the operands by routing or input ports	Resource-Aware Scheduling Policy
Timing	Start the instruction as soon as possible but respect all the dependences	Instruction Wake Up Logic

Table 4.2: Types of constraints for the mapping problem.

are elaborated in Table 4.2. The first two constraints actually decide the feasibility of the mapping, and the last one determines the mapping quality, which is usually expressed as optimization objectives by setting the upper bounds of the overall evaluations from all the mapped instructions. In an OoO processor, only functionality and timing constraints are considered by using heuristic instruction wake-up and selection logics, and the resource constraint is handled by moving data through centralized register file and bypass network automatically. In a spatial fabric, resource constraint becomes specially important as the datapaths and input port resources are limited. Thus the placement and routing decision by one instruction may conflict with one by another instruction. As a result, a mapping technique for spatial architectures should be *resource-aware*. The static mapping methods are effective and resource-aware since they can consider requirements from all the mapping instructions at the same time in a large mapping scope, then giving out a global mapping solution.

However, static mapping methods are not adaptable to changing workloads and compatible across hardware generations. Dynamic mapping for spatial architectures can overcome the limitations. Potentially, dynamic mapping techniques can be developed from existing dynamic optimization techniques, which use a method similar to trace caches [27, 73] to collect retired instructions and perform optimization. Unfortunately all existing dynamic techniques are naïve in the sense that they follow strict program order and consider only one instruction at a time and thus make locally optimal decisions for the constraints. For

example, DIF [26, 62] places retired instructions in the *first* VLIW instruction word that can access its ready operands. Meanwhile, CCA [14] dynamically maps in-order instructions from the writeback stage to the *first* available functional unit that can receive its operands from the cross-bar.

4.1.2 Speculative Architecture Support for Dynamic Mapping

Speculation implemented in OoO processors actually provides the bulk of the performance benefits of OoO processors [56]. Speculation enables better instruction scheduling results by relaxing the *timing* constraints of instructions. This fact inspires the idea of SPAS, which leverages the mature prediction and speculation design from OOO processors, and generates good instruction mapping for spatial architectures, especially for dynamic workloads. In a sequential program, there are three kinds of dependences which prohibit executing related instructions out of program order without speculation: register dependence, control dependence and memory dependence. In SPAS, register dependences are naturally handled by introducing the data-flow execution model in the spatial fabric. With data-flow execution model, one instruction starts to execute when its operands are ready and the register dependences are fulfilled by the physical datapath connections from producer functional units to consumer functional units. However, no execution model can inherently break control dependences and memory dependences without speculation.

Control Dependence Speculation With assistance from compilers, control speculation has been exploited for spatial architectures by forming enlarged basic blocks statically after profiling and checking their validity at runtime [35]. Since these techniques are based on profiling specific workloads, they cannot be generalized for a large class of unprofiled workloads. Also, if-conversion based techniques, such as hyperblocks, can be used to

convert control dependences to register dependences. However, this would increase the usage of hardware resources on the spatial fabric.

As a pure dynamic method, SPAS utilizes the branch predictor of OoO processors to dynamic select code sequences across multiple basic blocks speculatively and execute them on the spatial architecture as fat atomic instructions.

Memory Dependence Speculation Handling the delivery of direct *register* dependences in spatial architectures is a matter of routing from the producer functional unit to the consumer functional unit. However, properly handling *memory* dependences is more complex, because the memory order needs to be constructed through a centralized load/store (LDST) units. Static mapping techniques for spatial architectures, such as Tartan [59], CASH [75] and SGMF [85], add explicit control edges, by converting memory dependences to register dependences, between aliasing memory instructions to ensure that dependences are respected. Another similar technique is used in WaveScalar [77], a dataflow technique. In WaveScalar, all memory instructions are statically identified by two IDs: the sequence number of the instruction within the wave (trace), and a wave number indicating the wave (trace) invocation. All the issued memory instructions from the fabric are reassembled in the memory system, and executed in “total load-store order”. This method requires new LDST units and is overly conservative.

OoO pipelines intelligently break memory dependences using high confidence speculative techniques, such as Store-Sets [13]. Confidence is built by recording alias information during the misspeculation. SPAS reuses this memory speculation to increase freedom in mapping and executing memory instructions.

Misspeculation Handling All the sides effects of speculation need to be kept from the architectural state of the host pipelines. Usually output buffers need to be inserted between

the host pipeline and the fabrics to hold the live-outs and store values from the spatial fabric. These buffers actually become the synchronization points for starting new computation on the host pipeline or fabric, and enforces in-order executions between them.

OoO pipelines verify the validity of speculation at the end of the execution by committing the instructions from re-order buffers (ROB) in order. To fully exploit the control and memory dependence speculation of OoO processors and enable really out-of-order execution between the host pipeline and the spatial fabric, the spatial fabric in SPAS and the host pipeline are coupled by sharing the same ROB.

Summary As discussed previously, a lack of speculation presents one of the main bottlenecks to achieve a highly efficient dynamic mapping for spatial architectures. However, building an independent speculation system is expensive. Inspired by the application of prediction and speculation in OoO processors, this dissertation proposes a hybrid model, which tightly couples the OoO pipeline and the specialized spatial fabric, to benefit one other. In this hybrid architecture, the spatial fabric can execute instructions simultaneously with the OOO pipeline with higher performance, and at the same time, with the support of speculation system from OoO pipeline, the design and configuration of spatial fabric can be simplified. Meanwhile, the instruction scheduling logic of the OOO pipeline can also provide a large mapping scope and manifest logic for instruction dependences to enable a pure hardware dynamic instruction mapping.

4.2 Hybrid Speculative Spatial Architecture Design

Figure 4.2 shows the overall architecture of SPAS, a tightly coupled accelerator architecture that can automatically accelerate repeated execution traces from the host OoO pipeline

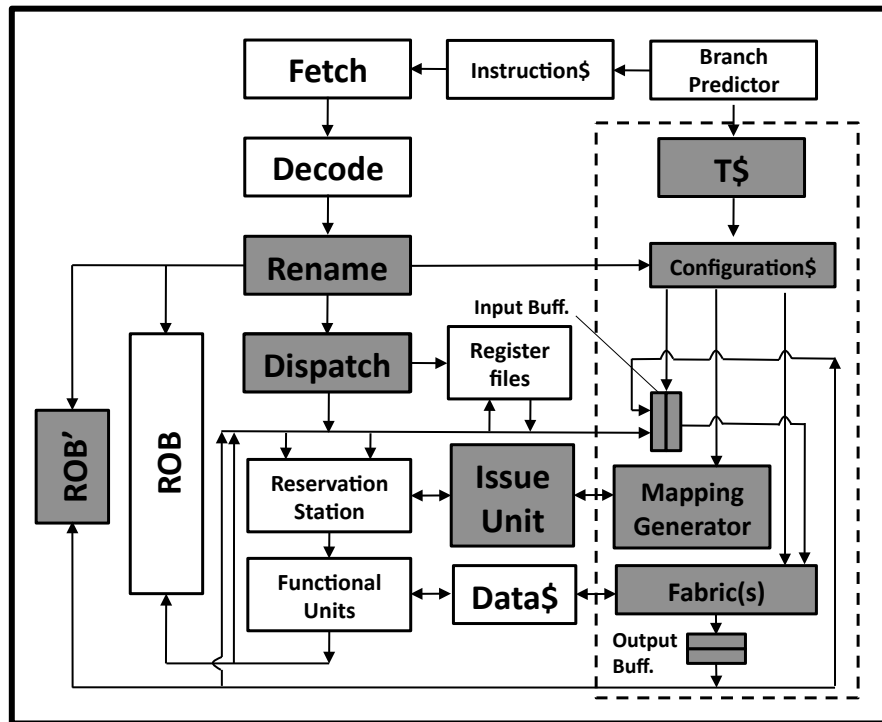


Figure 4.2: The overall view of the SPAS architecture. \$ is the abbreviation of cache. The blocks with gray color filled are the new components added to the conventional OoO pipeline or the existing components requires modification.

on a spatial fabric for efficient execution. SPAS is designed as an enhancement of high performance OoO processors and does not fundamentally change the structure and execution of the original OoO pipeline.

SPAS consists of three main components: a mapping cache, a mapping generator and a reconfigurable spatial fabric. The mapping cache contains both trace detection (T-Cache) and configuration storage (Configuration Cache). The mapping generator guides instruction issue in the host OoO pipeline and uses issue information to generate the mapping configuration for the fabric. The reconfigurable spatial fabric enables dataflow execution and acceleration of traces.

Trace acceleration in SPAS can be divided into three phases:

Trace Detection The detection phase identifies hot traces for acceleration. *T-Cache*, a trace cache like structure, recognizes recurring instruction sequences across multiple basic blocks. Upon commit of a branch instruction, T-Cache consults an internal history buffer that tracks the previous three branch results¹. T-Cache then builds an index consisting of the PC of the earliest branch instruction and the three results stored in the buffer and increments a saturation counter using this index. If the counter for this trace is larger than a preset threshold value, a flag in T-Cache for the entry is set to indicate the trace is hot.

Trace Mapping Upon receipt of a branch instruction, the fetch unit retrieves the predictions for the next three branches from the *branch predictor* to build an index into the T-Cache in order to determine if the predicted coming trace is hot. If the trace is hot, the fetch stage grabs instructions until it reaches the fourth branch instruction, as SPAS only tracks three branch instructions, or a preset instruction count limit and marks instructions in the sequence as *trace instructions*. These trace instructions are processed as normal instructions in the fetch, decode, and rename stages. When they arrive at the dispatch stage, the following process occurs :

1. The first trace instruction stops in the dispatch stage until all on-the-fly instructions that have been dispatched to the host OOO pipeline drain through the pipeline back-end;
2. The *mapping generator* interacts with the issue unit to generate configurations for the spatial fabric. It can be a hardware module, like the design proposed in Chapter 5, or even a co-processor [12]. The job of this mapping generator is to place the trace instructions onto the fabric and route their operands. This generates a *configuration* for the fabric with this mapping. When the issue unit schedules an instruction to an OoO functional unit, it simultaneously maps this instruction to a functional unit on

¹as suggested by [84]

the fabric, routes operands from the producers and thus generates a *configuration*. Note that no instructions execute on the fabric during this phase;

3. The mapping process finishes when the last trace instruction completes and writes back in the OoO pipeline. The configuration for the spatial architecture is stored in the *Configuration Cache*. The Configuration Cache is indexed similarly to the T-Cache, but contains less entries to save space. Any mis-predicted branches or pipeline squash will abort the mapping process.

With the completion of the trace mapping, the entry in Configuration Cache is marked as mapped. Also, the saturation counter for the entry is set to zero and incremented if the trace is predicted again by the fetch stage. The counter filters out traces that only appear a few times but could trigger reconfiguration of the fabric and its associated overhead. Once the saturation counter reaches a threshold value, the entry is marked ready and begins. To prevent frequent reconfiguration, the saturation counters in both the T-Cache and Configuration Cache are periodically cleared to prevent traces that execute infrequently from occupying the spatial fabric.

Trace Offloading Before offloading begins, the fabric is configured using the mapping result stored in the Configuration Cache. Concurrently, the rename stage renames the live-ins and live-outs of the trace and sends the ready register values to the fabric. After offloading begins, the fetch unit is directed to the end of the trace and begins normal operation. This allows the host OoO processor to execute concurrently with SPAS's fabric. If the same trace is executed back-to-back, the dependent live-outs from one invocation are forwarded directly using the global bus to the input ports of the fabric for future invocations, which allows for pipelined execution. During execution, the fabric sends live-out values to the ROB and bypass network of the host OOO pipeline.

4.3 Spatial Fabric Implementation and Integration

Prior work contains various fabric designs for reconfigurable spatial architectures; however, none of them are designed to dynamically map and offload hot traces from OoO processors. To enable this feature, we both optimize the complex internal interconnect design in existing reconfigurable spatial fabrics for dynamically detected traces and integrate the new hardware structures of SPAS with the existing OoO pipeline.

4.3.1 Acyclically Connected Spatial Fabric

To tailor the reconfigurable spatial fabric for dynamically detected traces, SPAS adopts the high level design of the reconfigurable fabric shown in Figure 4.1. Internally, the SPAS fabric is organized as stripes. Each stripe contains an array of functional units, which are connected by a circuit wired interconnect, as shown in Figure 4.3. Each functional unit contains one functional unit, a set of pass registers and multiplexers. Each functional unit can contain a different type of functional unit, and the set of functional units in a stripe can differ from other stripes. Each functional unit gets its operands from either the pass registers of the previous stripe (for dependences within a trace) or the global bus (for dependences between trace invocations).

Given that the data dependences found within traces are acyclic, the communication allowed within the SPAS fabric is also acyclic. There are three types of connections provided by SPAS: intra-stripe, local inter-stripe and global inter-stripe. Intra-stripe connections (as shown in ① and ② in Figure 4.3) are completed by the interconnect, and route intermediate results to the next stripes; local inter-stripe connections receive inputs from the previous stripe (such as ③) and can bypass values to the next stripe by storing results in a pass register (as shown in ④). Global inter-stripe connections are used to communicate live-ins and live-outs with the external OoO pipeline and between trace invocations.

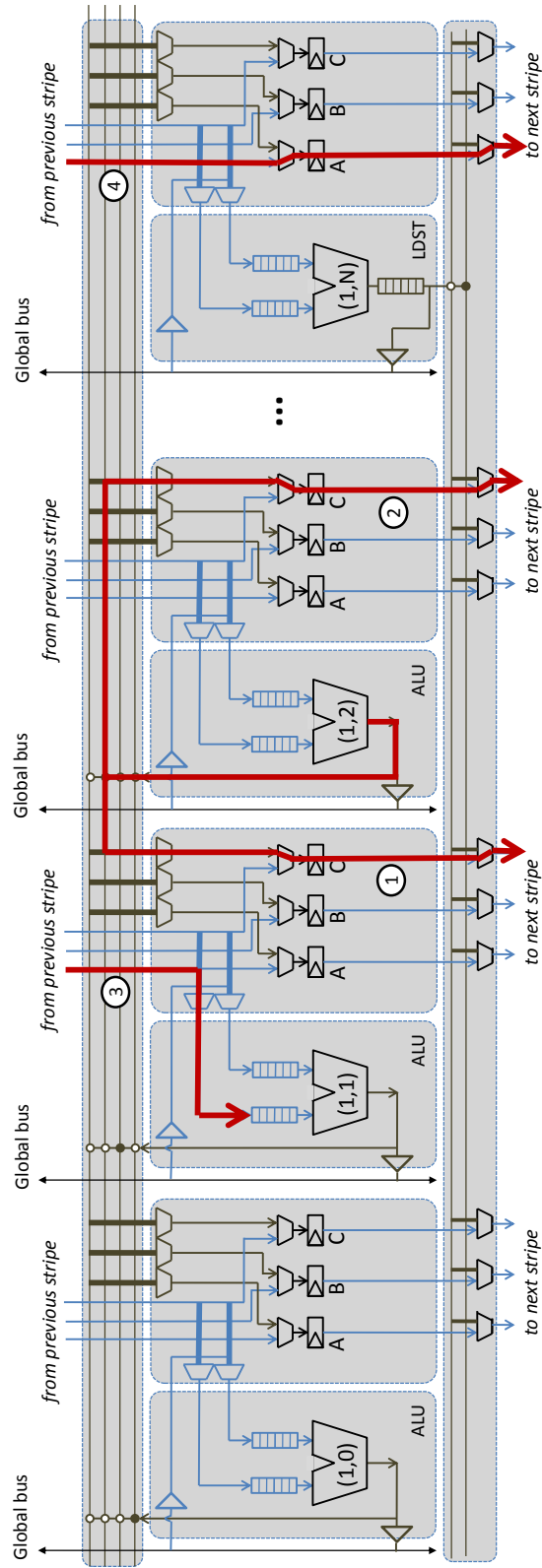


Figure 4.3: The functional units, pass registers, and interconnect of a stripe of Figure 4.1. Wires do not interact with each other except (1) ● is put on the intersection or (2) wires with different width intersect. Units A, B and C are pass registers; ALU represents an ALU unit; LDST represents a load/store unit; buffers on ALU input represent the FIFOs of input operands; the output buffer on the LDST unit represents the memory reservation buffer.

Given that the data dependences found in traces are totally acyclic, the communication allowed within the SPAS fabric is also acyclic. This design will over-provision resources, but actually simplifies data routing. With technology scaling, on-chip transistors will be plentiful and a large portion of the transistors need to be shut down or gated regularly to keep the power consumption within an on-chip budget, i.e. the so called “dark silicon” issue [22]. SPAS trades silicon area for energy efficiency and routing complexity.

Each functional unit in SPAS can be power-gated independently depending on the configuration to save both dynamic and static power consumption during execution. In Section 5.3, evaluation of SPAS shows that the configuration of functional units are constant for long periods of time, which is sufficient to tolerate the latency between transistor sleep and wake-up across multiple configurations.

4.3.2 Integration into the Host OoO Pipeline

To support seamless and speculative execution on the spatial fabric, the rename, dispatch, and reorder-buffer (ROB) units in the host OoO pipeline require some augmentation.

First of all, the rename unit renames live-in and live-out registers for the trace and reserves entries in input and output FIFOs in the spatial fabric. Separate entries in the FIFOs represent separate invocations of the trace. The oldest entries in the input FIFOs work like reservation station entries to receive live-in values from the OoO pipeline. Live-out FIFOs broadcast live-out values from the fabric to the OoO pipeline.

After trace renaming, the renamed result will be sent to the fabric too. The renaming result of live-ins and live-outs will be pushed to the input buffers and output buffers, as shown in Fig. 4.4. The input buffers are a set of FIFO queues, which are corresponding to each live-in. Only the heads of the buffers are active, and receive data from the external common data bus. An entry can also actively receive data from the external register files if it becomes the head. Entries are deleted once the data is obtained. In this method, only a

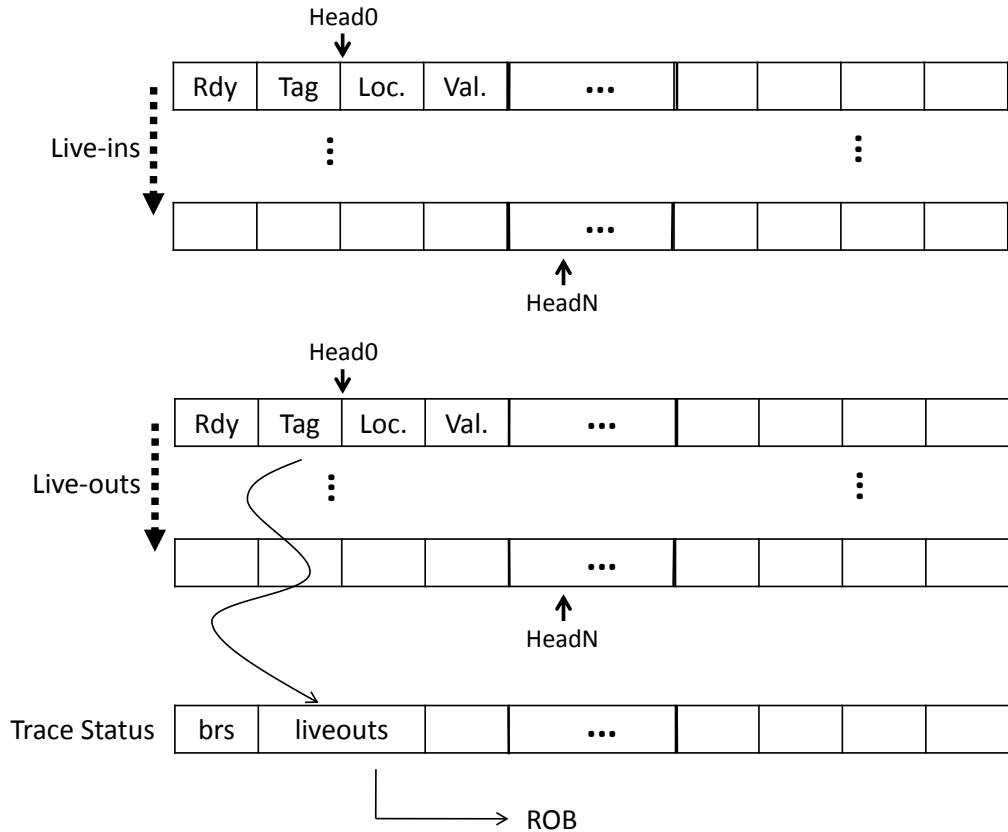


Figure 4.4: The design of Live-in and Live-out FIFOs for the spatial fabric and host processor integration in SPAS.

few entries need to be connected to the bypass network of the original OoO and the effects are like adding some entries in the instruction queue.

The output buffers work in a similar way, receiving data from the internal global data bus. Again only the header of the FIFOs needs to snoop on the bus since the live-outs from the same functional units never come out out of order. When output buffers are retired, their contents must be written to a trace status buffer.

As the trace may contain multiple side exits and experience misspeculation, the speculated result of branch instructions must also be logged by the trace status. A trace is considered as having successfully completed if all the liveouts are obtained and the branches prediction results are correct.

An extra index field is added in the main ROB to allow entries to point to a side ROB (ROB'), which contains the renamed live-out values, branch results, and memory stores of a trace invocation. Such an entry can only commit when all live-outs and branch results are obtained from the output FIFOs. This essentially means that the trace is treated as a fat atomic instruction by the host OOO pipeline. The ROB' commits or squashes (if there is a branch mis-speculation or memory order violation) the entry when it reaches the top of ROB. When the trace is committed or squashed, the ROB' broadcasts the information to all pipeline stages. The number of live-ins and live-outs that the rename and ROB' can handle are encoded as constraints in the dynamic mapping phase. If the number of live-ins or live-outs for a trace exceeds either number, a valid mapping cannot be completed.

4.3.3 Intra- and Inter-Trace Memory Ordering

SPAS utilizes the aggressive speculative LDST issue techniques in OOO processors and allows certain LDST instructions from the fabric to be executed out-of-order. To achieve this, SPAS keeps a simplified version of memory instructions, consisting of only their program counters (PC), instruction types, and their relative ordering, in the configuration. When a trace invocation is dispatched, the simplified memory instructions are sent to a memory dependence prediction unit similar to the Store-Sets [13], which is used by modern processors to speculatively predict aliasing memory instructions that alias. Memory operations that execute in the fabric consult the unit to determine if they can execute, or if they must stall in order to respect a memory dependence. If the dynamic memory dependence prediction unit mis-speculates and causes a memory violation, the trace is squashed in the ROB and re-executed after updating the offending dependence in the prediction unit.

Additionally, as load operations from a LDST unit on the fabric can receive responses out of order, SPAS adds a reservation buffer to each LDST unit to hold all the in-flight loads, as shown in the LDST unit in Figure 4.3.

4.3.4 Configuration Datapath

Due to the acyclic design of the data path, the reconfiguration of SPAS can be pipelined and overlapped with the computation. After the configuration of the whole fabric is generated by the configuration generation, the fabric can be configured stripe by stripe in a pipeline fashion. As shown in Figure 4.1, each configuration word is sufficient to configure all the functional units in the fabric. In the first cycle, the FUs in the first stripe receive the configuration fields, store part of the configuration words in the FU's configuration register, and at the same time, pass the reconfiguration words to the next stripe. In the second cycle, the same work can be done by the second stripe. At the same time, the first stripe can start to receive operands and output computation results. In this way, SPAS can configure and compute in pipelined fashion. The pipelined design largely reduces the overhead of online configuration, and avoids suspending the execution for reconfiguration.

4.3.5 Execution Example

Fig. 4.5 shows an example of pipelined trace execution with the SPAS reconfigurable fabric. In Fig. 4.5(a), the C source code, assembly, and the data dependence graph are shown. The OoO instruction issue units can apply dynamic unrolling to fully utilize the available functional units and execute instructions from different iterations simultaneously. In Fig. 4.5(b) shows that with 2 ALUs and 1 MEM unit, instructions from two iterations are executed concurrently, exploiting instruction level parallelism. However, the dynamically unrolling capability of OoO pipeline is limited by hardware resources, and the functional units need to be time-shared by instructions from different iterations. In this example, since each iteration contains 6 integer instructions, the 2 ALUs need at least 3 cycles to complete them. For each cycle, the intermediate results need to be fetched and sent back to the register file and bypass network. For computation intensive programs, which are usually

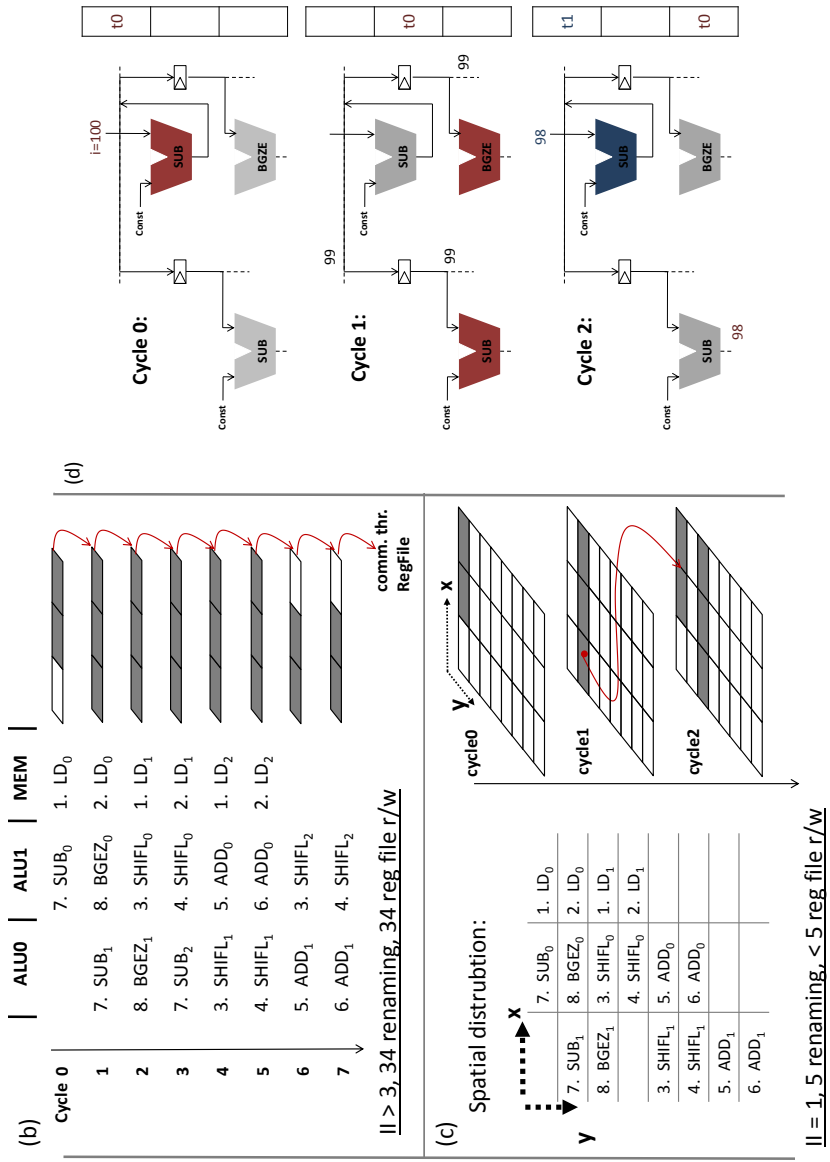


Figure 4.5: (a) Source code and assembly of the simple program example, and its data dependence graph; (b) An dynamic unrolling of (a) with OOO, which contains 2 ALUs and 1 MEM; (c) An instruction placement for the spatial fabric with the scheduling result from (b); (d) Activities of the functional units (0,0), (0,1), (1,0) and (1,1) in the first three cycles, indicating the pipelined execution.

the target of high efficient acceleration, the hardware resources are still the performance bottleneck, especially for traces with high confident branch prediction results.

The trace acceleration of spatial architectures targets this bottleneck in two steps: First, static trace instructions are assigned to the spatially connected functional units with correct data connections. For our fabric, due to the acyclic connection pattern, the observation of instruction scheduling on the original OoO pipeline can be used to assign instructions to the fabric. We refer this as *Schedule Reuse*. By combining schedule reuse with instruction placement and correct data path routing, the fabric can actually have higher throughput than original OOO pipeline, as shown in the right hand side of Fig. 4.5(c), which highlights the active functional units in each cycle after the execution. In Cycle 1, the *SUB* instruction in location (1,0) is executed and the result can be sent out by the global bus. Thus, the two instructions in (0,1) and (0,2) can receive this result, and starts the second trace instance, indicating pipelined execution of the trace instance for the same trace configuration.

An explanation with more details is shown in Fig. 4.5(d). In this figure, only the functional units in the left-up corner of the fabric is shown. In cycle 0, the live ins for the first trace instance are ready for functional unit (0,1) in the first row. Thus, it is ready to execute and produce results at the end of the cycle. The functional unit has 1 cycle latency, and at the end of cycle 0, the result is available on the result lines, and sent to the consumers through the interconnect and pass registers. This actually sets the two functional units in the second row as ready to execute since their operands become ready. In cycle 1, the functional units in the second row executes. The functional unit (0,1) is waiting for its operand which is dependent the previous trace instance. At the end of cycle 1, one live-out of the trace, which is also the result from functional unit (1,0) will be placed on the global bus and forwarded to the input of functional unit (0,0) to enable its execution in the next cycle. With this method, two back-to-back trace instances are actually executed on the fabric at the same time, with an initiation interval of 2 for a trace with 2 original loop iterations.

Besides the performance benefits, an equally important metric, energy consumption can also be estimated: First, after the first configuration of the fabric, the following trace instance execution does not need any energy for configuration. In contrast, the original OoO pipeline needs to fetch the instructions and decode them again; Second, since the intra-trace intermediate results are hardcoded in the hardware, only the live-ins and live-outs need to get a tag to communicate correctly. Also the access of centralized register files and bypass networks are largely reduced and replaced by independent registers.

Continuous sequences of hot basic blocks can form a trace. In our case, we limit the length of the continuous sequences to three basic blocks. Thus, the original basic block loop can be unrolled as a trace loop, in which each trace contains three basic blocks. When this trace is predicted as hot, the two side branches can be speculatively removed by the branch prediction results. On the other hand, the instruction sequence of a loop iteration can exceed the hardware resource limitations. In this case, only the trace portion can be accelerated on the fabric and the remaining part needs to be offloaded to another fabric or must use the host OoO pipeline.

An end-to-end evaluation is presented in the next Chapter to combine with the design of dynamic instruction scheduling component.

Chapter 5

Dynamic Mapping with Resource-Aware Instruction Scheduling

The speculation support from OoO pipeline and acyclic spatial fabric design from SPAS creates opportunities to achieve a pure hardware-based instruction mapping for spatial architectures. A dynamic hardware-based instruction mapping offers several benefits, compared to static mapping and other software-based dynamic solutions. A dynamic hardware-based mapping is fully automatic and binary compatible. It also does not preempt the normal software execution to regenerate configurations.

However, designing a hardware module that can map instruction traces to the targeted spatial fabric is a complex and difficult task. It requires a mechanism to build and store the data dependences between instructions of the scheduled trace. Meanwhile, it also requires a weight-based algorithm to select instructions from all the available instructions, to fulfill the scheduling target (mainly execution time constraints).

This dissertation proposes DYNASPAM, which includes an instruction mapping algorithm and a hardware module to select, place and route instructions for spatial fabric. DYNASPAM is light-weight because it largely leverages the existing instruction wakeup and issue logic in OoO processor pipeline.

5.1 Motivating DYNASPAM

5.1.1 The Importance of Mapping Scope

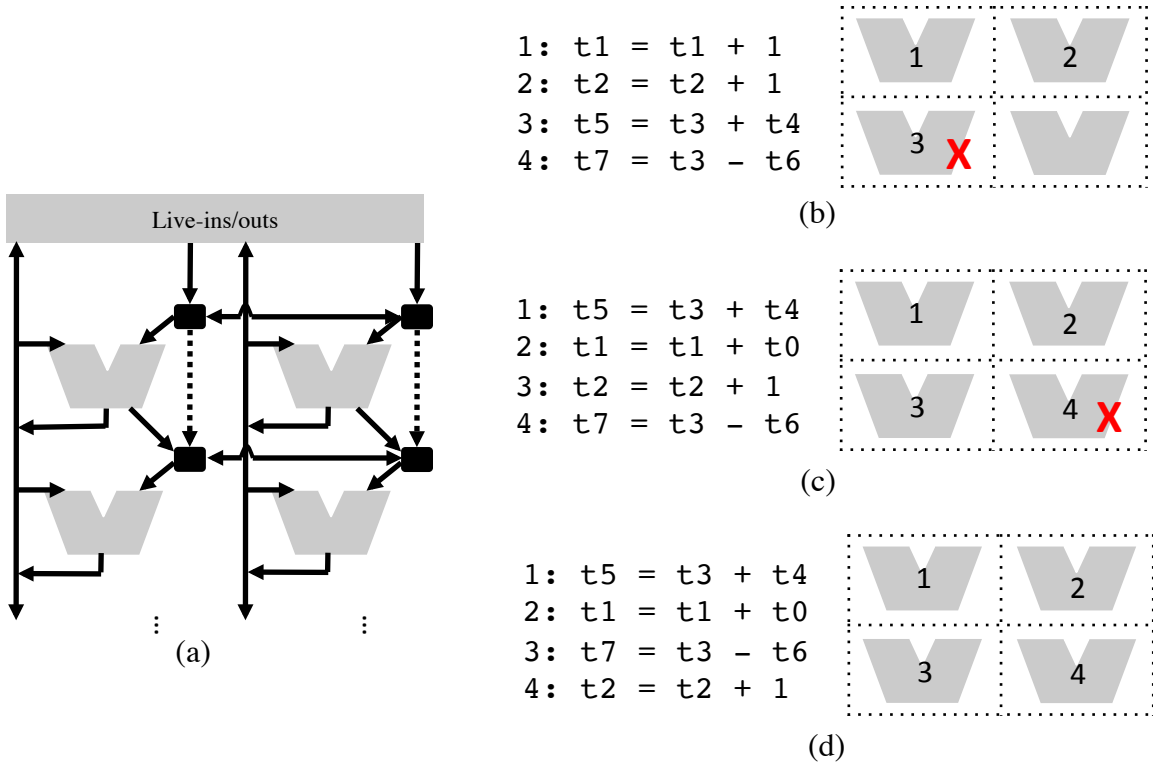


Figure 5.1: (a) for two special architecture settings (without dotted line, and with dotted line to share operands), (b) and (c) show examples where naïve placement fails to create efficient schedules, and (d) show a resource-aware scheduling.

The limited number and heterogeneity of hardware resources in spatial architectures complicate the procedure of mapping instructions to the spatial fabric. Due to the limited mapping scope, the naïve placements can only greedily satisfy the constraints from each

instruction one at a time, and may cause failure or inefficiency for the overall instruction trace.

Figure 5.1 shows the importance of a large mapping scope when generating a mapping for reconfigurable spatial architectures. Figure 5.1(a) shows an example spatial architecture. While all rows of functional units have the same capability, each of them has a unique input connection setting. Functional units in the first row can get two operands from live-ins at the same time, while functional units in the second row can get only one operand directly. This reflects an architecture with special input ports connected to the first row and a global bus to forward inputs to subsequent rows such as in PipeRench [33].

Figure 5.1(b) shows how one naïve mapping results in a schedule failure. In the example code, the first two instructions have one live-in operand, and the following two instructions have two live-in operands. There is no dependence between these four instructions, so they can be scheduled independently. If a mapping generator can see all four instructions, it would map instruction 3 and 4 to the functional units in the first row and instruction 1 and 2 to the second row, thereby allowing four instruction to be executed in a single cycle. However, the naïve mapping will place instruction 1 and 2 in the first row, resulting in a scheduling failure for instruction 3 and 4 because of *resource constraint*.

The *resource constraint* failure can be resolved by adding more resources to the fabric. For example, in Figure 5.1, if two extra data paths (by adding dotted lines) are applied to forwarding operands from one row to the next row, the placement shown in Figure 5.1 is feasible. However, this forwarding needs to take extra cycles, and leads to lower performance. In this example, it needs two cycles to complete the computation. Figure 5.1(c) is another example demonstrating the deficiencies of another naïve mapping. In this example, placing instruction 1 and 2 naïvely on the first row is reasonable as both of them require two source operands from the live-ins. Instruction 4 also takes two source operands

from live-ins, but there are no unallocated functional units in the first row any more. Fortunately, instruction 1 and 4 share the same source operand, t_3 , thus routing resources for t_3 can be reused by both instruction 1 and 4. However, in the naïve schedule shown in Figure 5.1(c), the functional units adjacent to instruction 1 are occupied, causing one extra datapath usage and two clock cycles to deliver the operand. With a larger mapping scope, the mapping generator will swap instruction 3 and 4 to make bypassing t_3 take only one cycle, as shown in Figure 5.1(d).

5.1.2 Hardware Synthesis

Although increasing the mapping scope may help in finding more efficient mappings, adding separate hardware logic to hold this scope for optimizing mapping of reconfigurable spatial architecture is expensive. DYNASPAM avoids this cost by leveraging the scheduling logic of the host OoO processor. As an OoO processor's scheduling logic is already equipped with a large instruction window and dependence analysis features, reusing its results can allow a reconfigurable spatial architecture to generate efficient mappings with little to no additional hardware cost.

With large instruction scheduling windows, the requirements of different instructions can be considered and this can largely reduce the failure ratio of instruction placement. However, in general, mapping large range instruction streams for reconfigurable spatial fabric with sparse connections is a NP hard problem. *Static, software-based* compilers are capable of generating relatively optimal configurations since they have a global view of the dependences in the instruction streams. However, building such an independent instruction window for dynamic instruction placement in *hardware* is very expensive. First of all, such a structure would support parallel checking of data dependences of all the scheduled instructions. Second, there should be associated hardware modules to record

the data dependences. The insight of our work is that we can reuse the instruction window of host OoO processor which are designed and optimized with high efficiency, to help the placement and routing with a large instruction window. It is a system which is totally transparent to the users, and online detects hot repeated execution long trace, generating spatial fabric configuration and also automatically offloading the computations. There are many technical challenges to be addressed in the proposed framework. First of all, to surpass the performance of modern OoO processors with less energy, the reconfigurable fabric should explore more parallelism than instruction level parallelism. These extra parallelism should be easily detected by the hardware at runtime, rather than programmers' efforts or compiler hints. This is essentially the requirement from accelerating close-source binaries. Second, the co-designed online configuration synthesis process should not change the design of the host OoO pipeline much, and have small impact of the program execution. In the following section, we show DYNASPAM solves the problems by incorporating the acyclically connected reconfigurable fabric from the previous chapter and plug-and-play hardware synthesis module.

5.2 DYNASPAM Design

To enable the dynamic mapping of a trace onto a reconfigurable spatial fabric, this dissertation proposes the *resource-aware* instruction scheduling technique. This technique is used to synthesize the configuration for the spatial fabric at runtime.

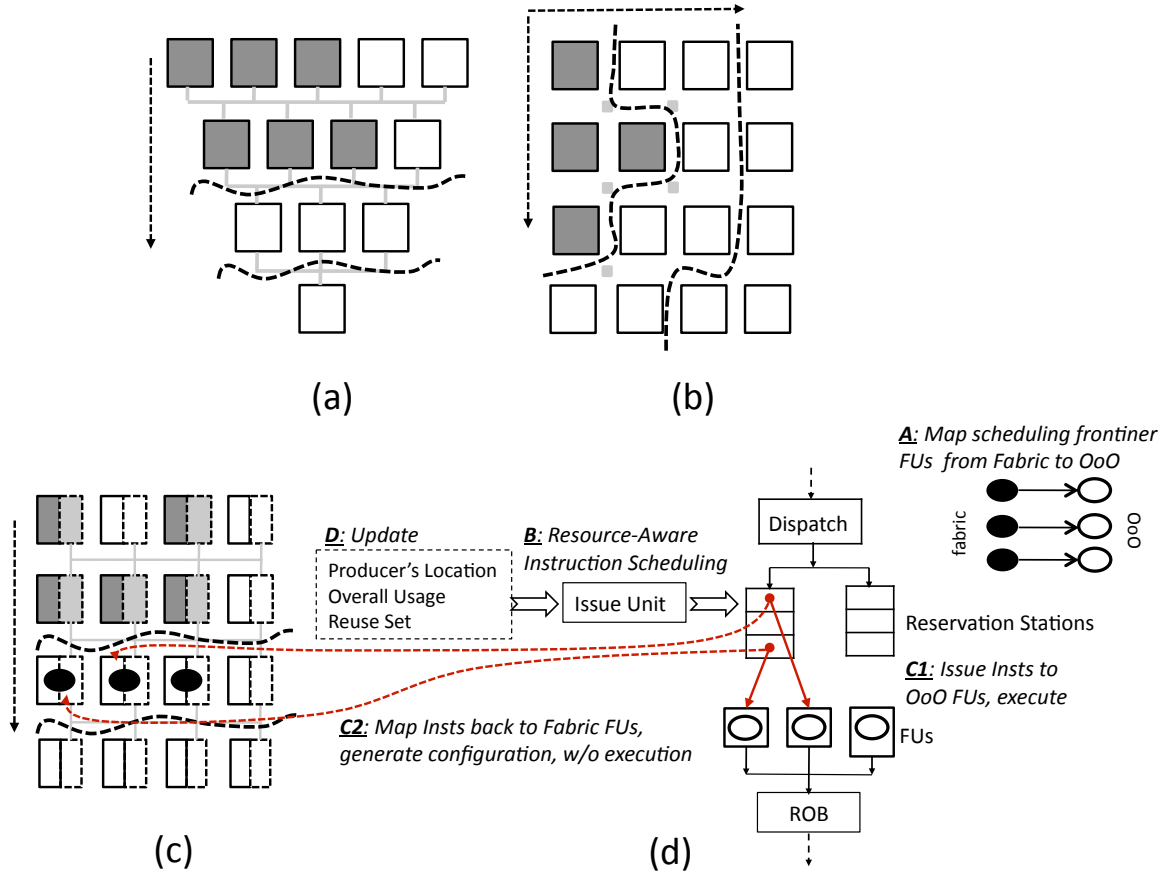


Figure 5.2: fabric functional units (FUs) and possible scheduling frontiers in (a) CCA; (b) 4x4 DySER; and (c) DYNASPAM fabric.

5.2.1 Resource-Aware Scheduling

Scheduling Frontier The *scheduling frontier* is the set of unallocated functional units that are directly connected to those that have been allocated. Initially, the *scheduling frontier* consists of functional units that can access live-ins directly. At the end of each scheduling step, the *scheduling frontier* moves along the data paths of the allocated functional units. Figure 5.2 shows the arrangements of functional units in three different spatial fabrics and possible positions of the *scheduling frontier* before the scheduling step. CCA [14] and DYNASPAM fabrics have no cyclic data paths between rows (or stripes), thus their *scheduling*

Algorithm 1: Resource-Aware Scheduling Algorithm

INPUT : $scheduleCycle$
OUTPUT A schedule of ready instructions to functional units: $Selected[:]$
:
1 $rowIdx \leftarrow SchedulingFrontierIdx(scheduleCycle)$;
2 **if** $rowIdx$ is *Invalid* **then**
3 | **SCHEDULE_FAIL**;
4 **end**
5 $FabricFUsVec \leftarrow SchedulingFrontierFUs(scheduleCycle)$;
6 $OOOFUsVec \leftarrow FabricToOOO(FabricFUsVec)$;
7 $ReadyInstsVec \leftarrow ReservationStation(scheduleCycle)$;
8 **foreach** $FU \in OOOFUsVec$ **do**
9 | **foreach** $Inst \in ReadyInstsVec$ **do**
10 | | $PriorityScore[FU, Inst] \leftarrow PriorityGen(FU, Inst, rowIdx)$;
11 | **end**
12 **end**
13 **foreach** $FU \in OOOFUsVec$ **do**
14 | $selectedInst =$
14 | **PriorityEncoder**($PriorityScore[FU, :], HostPriorityPolicy$);
15 | $Selected[FU] = selectedInst$;
16 | **UpdateTables**($FU, selectedInst$);
17 **end**

frontiers are straight. Meanwhile, DySER [34] has a complex data path network and its *scheduling frontier* can be irregular.

Scheduling Insights As discussed in Section 5.1, naïve mapping techniques are not globally resource-aware, and thus do not generate efficient mappings. Building a standalone scheduling unit for dynamic spatial fabric mapping would be prohibitively expensive, thus leveraging the existing micro-architecture of the OoO pipeline is desirable. To support its own scheduling, the OoO reservation station provides the following capabilities:

- *Instruction Buffering*: A large instruction window that can easily contain instructions from a large trace;

- *Data Dependence Analysis*: Instructions marked as *ready* in the instruction window are known to have their operands available, and are independent of all other ready instructions;
- *Instruction Assignment*: An issue unit can select ready instructions for multiple functional units by using priority rules, (referred to as *HostPriorityRule*), such as oldest-first, in its *Priority Encoder*.

One of the key insights of DYNASPAM is that these are the same set of features needed to support dynamic mapping for spatial fabrics, and thus we can equate the placement of trace instructions to functional units in the *scheduling frontier* with the instruction scheduling for the OoO functional units. However, the mapping of instructions to the fabric has additional resource constraints, such as the location of producers, data path availability, and the cost of allocating new paths. These constraints can be represented as a *priority score* that indicates both the feasibility and efficiency of mapping an instruction to a functional unit on the fabric.

If there is a one-to-one mapping between the functional units in the OoO pipeline and the functional units in the *scheduling frontier* (Step A in Figure 5.2), then selecting an instruction with the highest priority score (Step B in Figure 5.2) for a functional unit on the OoO pipeline (Step C1 in Figure 5.2) also maps it to a functional unit in the *scheduling frontier* (Step C2 in Figure 5.2). After mapping, the resource information, which is contained in a set of status tables, can be updated (Step D in Figure 5.2) as discussed in Section 5.2.2. Using this information allows DYNASPAM to perform *resource-aware* instruction scheduling.

Scheduling Algorithm The high-level Resource-Aware Scheduling Algorithm is shown in Algorithm 1. The input of the algorithm is the current *clock cycle*. First, the *scheduling*

frontier is identified as *rowIdx* (Line 1) and the functional units in the frontier are identified as *FabricFUsVec* (Line 5). Since the *scheduling frontier* in DYNASPAM is always aligned to a stripe, the *scheduling frontier* identifier is the stripe index of the fabric that is currently being mapped. Line 6 maps these functional units to the OoO functional units. Line 7 selects the ready instructions from the reservation station, and Lines 14 and 15 use the original select logic to assign instructions to OoO functional units. This reuses the instruction wakeup (data dependence checking) and the instruction select logic from the host OoO pipeline. The new instruction schedule can differ from the schedule generated by the original host priority rule; however, we expect that this kind of priority change does not cause a significant performance change [7].

Special Issues There are two special issues that can occur during scheduling. First, the number of functional units in the current *scheduling frontier* can be greater than that in the OoO pipeline or the issue width of the OoO pipeline, thus some functional units cannot be mapped to in the current scheduling cycle. This problem can be overcome by dividing one scheduling step into multiple cycles. An extra field must be added to each entry of the reservation station to identify the ready instructions that become ready in the middle of a scheduling step, but cannot be issued until the current scheduling step ends. Second, the issue unit must pause if there are OoO functional units that have not finished execution at the start of a scheduling cycle. Otherwise, the *scheduling frontier* could proceed before all functional units in a stripe have been scheduled.

Lines 10 and 16 generate the priority scores for each pair of instruction and functional unit by consulting to a set of status tables (Line 10), and then update these tables after the instructions are scheduled (Line 16).

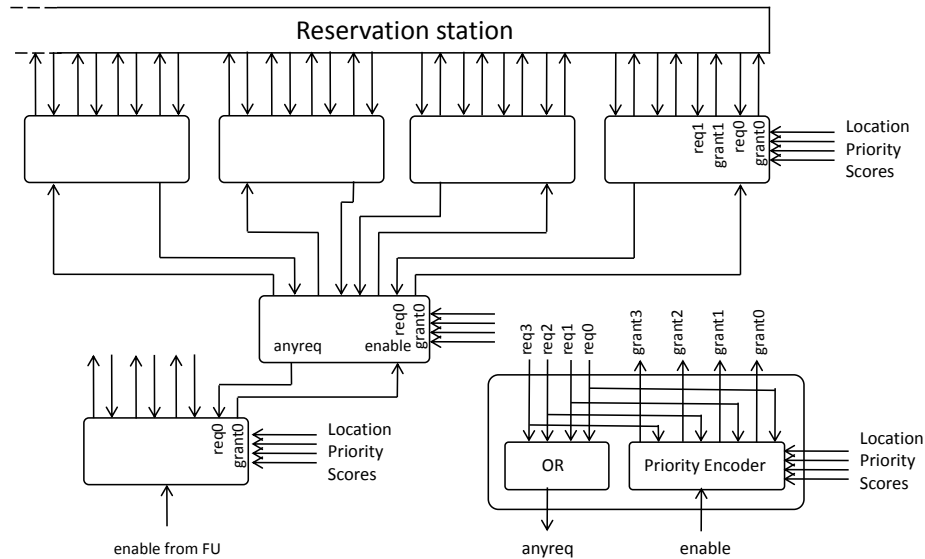


Figure 5.3: An example of logic to select ready instructions from the reservation stage for a function units. The priority can be changed by encoding more constraints in the priority encoder.

Category	Priority Level	Description
Feasibility	3	Two operands are live-ins thus requiring two input ports.
Routing Score	2	Two operands are not live-ins, and can be provided by <i>ReuseSet</i> .
	1	Only one operand can be provided by <i>ReuseSet</i> , while the other can be routed.
	0	None of the operands can be provided by <i>ReuseSet</i> , but they can be routed.
	-1	One of the operands can not be provided by <i>ReuseSet</i> and can not be routed.

Table 5.1: Priority Scores for different connection status of the producers.

5.2.2 Priority Score Generation

Priority Score A *priority score* is a ranking for the placement of an instruction onto a functional unit in the *scheduling frontier*. This priority can be used by the *Priority Encoder* in the issue unit to grant an issue request from the ready instructions [65]. Figure 5.3 shows

an example design of the hardware module to utilize the priority scores to select and assign instructions to functional units.

The *priority score* of a fabric can be customizable. DYNASPAM contains five priority scores (shown in Table 5.1) to represent levels of mapping feasibility and routing score (a higher score means lower routing cost).

Instructions that require two live-ins to be feasibly placed have the highest priority with functional units in the first row, as only these units have two input ports. If the instruction does not require two live-ins, the mapping algorithm prefers to put instructions where they can enjoy more data path reuse, i.e. the producer's value has been routed nearby, with priority levels 0-2 representing the amount of data of available for reuse. Priority level -1 represents a functional unit that cannot provide enough resources to route its operands. Thus, this instruction should not be scheduled to this functional unit.

The scheduling algorithm is not tied to any particular priority scoring mechanism; the scheduler should use a scoring mechanism that takes into account the resource constraints of the particular spatial architecture that is being mapped to. For example, in CCA [14] data used in one row cannot be reused in the same row. Thus, there is no routing cost preference. In DySER [34], there are multiple possible data paths that can route the same data for one functional unit, thus the routing latency should be considered.

Generation *PriorityGen*, a hardware module within the mapping generator, generates priority scores by consulting the current state of mapping, which is stored in three lookup tables: Producer Table (*ProdTable*), Overall Datapath Usage Table (*OverallUsage*), and Datapath Reuse Set Table (*ReuseSet*). The priority generation algorithm is shown in Algorithm 2. For each operand of an incoming instruction the algorithm checks:

1. *ProdTable* to obtain the location of the operands producers (Line 5);

Algorithm 2: MODULE PriorityGen

INPUT : $OOOFU, Inst, rowIdx$
OUTPUT $PriorityScore[:, :]$
:
1 $FabricFU \leftarrow OOOToFabric(OOOFU)$
2 $canReuse \leftarrow 0; canRoute \leftarrow 0; needInputs \leftarrow 0;$
3 **foreach** $op \in Inst.ops$ **do**
4 | $livein \leftarrow False;$
5 | $ProdLoc \leftarrow ProdTable(op);$
6 | **if** $ProdLoc$ *does not exist* **then**
7 | | $livein \leftarrow True;$
8 | | $needInputs ++;$
9 | **else if** $op \in ReuseSet(FabricFU)$ **then**
10 | | $canReuse ++;$
11 | **else if** $OverallUsage(ProdLoc, FabricFU) \neq \emptyset$ **then**
12 | | $canRoute ++;$
13 | **end**
14 **end**
15 **if** $needInputs == 2$ **then**
16 | **if** $FabricFU$ *can provide two InputPorts* **then**
17 | | $PriorityScore[OOOFU, Inst] \leftarrow 3;$
18 | **else**
19 | | $PriorityScore[OOOFU, Inst] \leftarrow -1;$
20 | **end**
21 **else**
22 | **if** $Inst.ops_num == canReuse == 2$ **then**
23 | | $PriorityScore[OOOFU, Inst] \leftarrow 2;$
24 | **else if** $Inst.ops_num == canRoute$ **then**
25 | | $ScorePriority[OOOFU, Inst] \leftarrow 0;$
26 | **else if** $Inst.ops_num == canReuse + canRoute$ **then**
27 | | $PriorityScore[OOOFU, Inst] \leftarrow 1;$
28 | **else**
29 | | $PriorityScore[OOOFU, Inst] \leftarrow -1;$
30 | **end**
31 **end**

Algorithm 3: MODULE UpdateTables

INPUT : *OOOFU, Inst*
INOUT : *ProdTable, ReuseSet, OverallUsage*

- 1 *FabricFU* \leftarrow *OOOToFabric(OOOFU)*;
- 2 *ProdTable(Inst.dest)* \leftarrow *FabricFU*;
- 3 **foreach** *op* \in *Inst.ops* **do**
- 4 *ProdLoc* \leftarrow *ProdTable(op)*;
- 5 **if** *ProdLoc* exists & *op* \notin *ReuseSet(FabricFU)* **then**
- 6 *newDatapath* \leftarrow **SELECT** *OverallUsage(ProdLoc, FabricFU)*;
- 7 **foreach** *FU* \in *newDatapath* **do**
- 8 add *op* to *ReuseSet[FU]*;
- 9 *OverallUsage(FU, newDatapath)* \leftarrow **USED**;
- 10 **end**
- 11 **end**
- 12 **end**

2. *ReuseSet* to determine if the required operand can be obtained directly from the pass registers from the previous stripe, in which case it does not need to add a new route from the producers (Line 9)¹;
3. Otherwise, *OverallUsage* to determine if there are available data paths to route the required data (Line 11);
4. Otherwise, the instruction cannot be assigned to the corresponding location, since there are no enough data path resources to route its operands (Lines 19 and 29).

Lines 22-27 summarize the scores from different operands, and give corresponding priority scores. The priority scores are stored in a two dimensional table, called *PriorityScore*, for each pair of ready instructions and functional unit. After one instruction is selected for a functional unit and issued, all the status tables are updated as shown in Algorithm 3.

Additionally, a *Live-Out Table* (LOT) is used to track functional units that produce live-outs and their corresponding architectural registers, and to configure the output ports of the

¹In the current implementation, live-in values are not added to the *ReuseSet* and must be acquired from the global bus on each use.

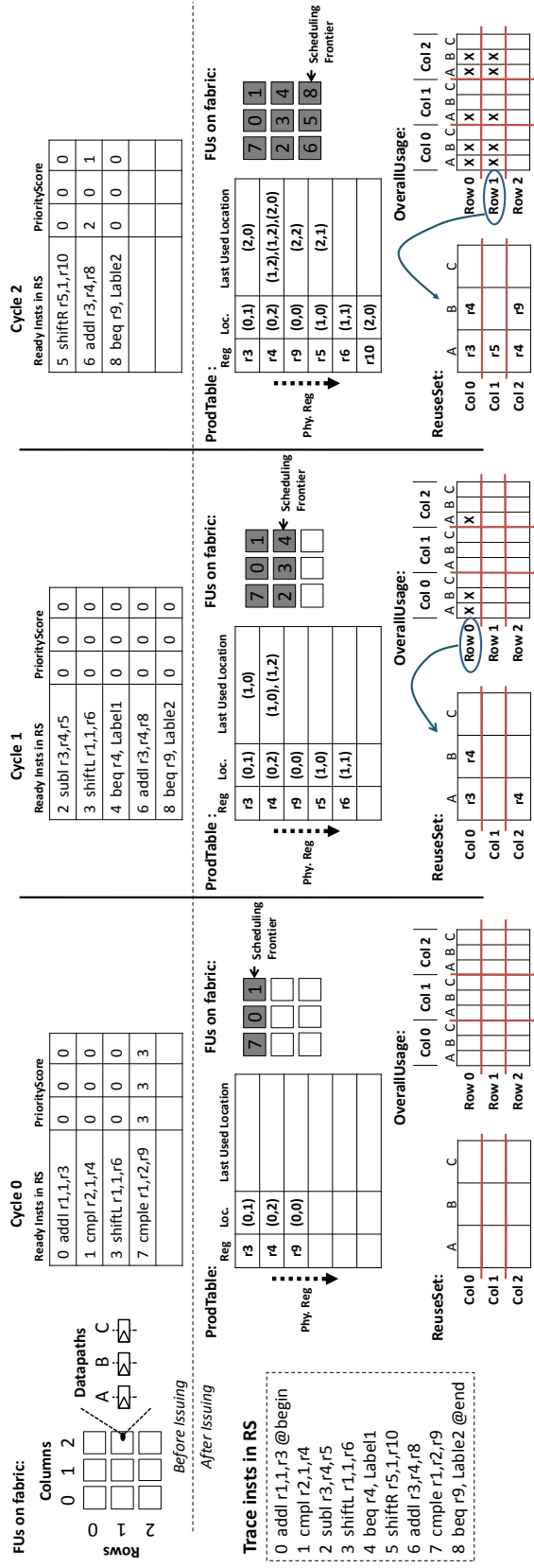


Figure 5.4: An example illustrating how instruction scheduling is impacted by the location information of the fabric.

fabric [14]. Upon advancing the *scheduling frontier*, a value is considered a potential live-out if its architectural register is not re-defined within the stripe, and will be automatically routed to the next stripe to increase the probability of reuse. A table called *Last Used Location* is used to track this information. If a potential live-out value is killed, the *Last Used Location* table is consulted, and any routing that was unnecessarily propagated for the killed live-out is removed.

5.2.3 Example

Figure 5.4 demonstrates mapping a short trace onto the DYNASpAM fabric and shows the additional hardware logic needed to support the mapping process. In this example, a trace with 9 instructions needs to be mapped to a fabric, as shown in Figure 4.3. *ProdTable* is a content addressable memory, or CAM, that maps the physical registers to locations on the fabric. *ReuseSet*, contains the physical registers which have values been stored in the pass registers of the previous stripe. It is also a CAM. *OverallUsage* tracks the overall data path usage across the whole fabric and is used to determine if there are enough resources to allocate a new data path to route the data. It can be implemented as a bitmap.

At the start of mapping, the trace instructions have been renamed and placed in the reservation station as per normal program execution. In cycle 0, four instructions are ready within the reservation station, and all status tables are empty. Three of the instructions (0, 1, and 3) generate Priority 0 for all functional units, indicating none of them can reuse the pass registers in the previous stripe to receive their operands. However, instruction 7 requires two input ports and thus has Priority 3 for all functional units. Instruction 0, 1, 7 are selected and placed in the corresponding functional units. Three entries of *ProdTable* are updated using the renamed destination registers of all instructions.

Since there are no available functional units in the *scheduling frontier*, the issue unit moves the *scheduling frontier* forward and begins the placement in cycle 1. In this cycle,

instructions 2, 4, 6, and 8 become ready. No instruction can reuse data from the pass registers in the previous stripe and are assigned Priority 0 for all functional units. Thus the original oldest first priority policy selects instruction 2, 3, and 4. The data tables are updated as follows: *ProdTable* adds new destination registers with the producer location; *ReuseSet* records the data in the current pass registers; and *OverallUsage* records the data path usage after this cycle of mapping.

In cycle 2, the final three instructions are ready. Instruction 5 and 8 have Priority 0 for all functional units since they cannot reuse any available data. Instruction 6 has Priority 2 for functional unit 0 as the pass registers of the previous row hold both of its operand values, r3 and r4. If Instruction 6 is placed there, no new data path routing is required. Mapping ends with instructions 6, 5, 8 being placed respectively.

The final mapping could not be obtained by the naïve method due to its limited instruction window. For example, if the instructions were placed in program order, Instruction 7 would not be placed in the first row, resulting in an infeasible schedule, and Instruction 6 would not be able to reuse the data path from Instruction 2, resulting in an inefficient schedule.

5.3 Evaluation

The full DYNASPAM system evaluated in this dissertation incorporates the SPAS design mentioned in Chapter 4 as well as the dynamic mapping techniques presented in this chapter.

5.3.1 Methodology

Area/Performance/Energy Simulation To evaluate the area overhead of the spatial architecture, functional units from OpenSparc T1 are used and the datapath for the SPAS

fabric is implemented in Verilog. The fabric design was synthesized using Synopsys Design Compiler [78] with a 32nm generic cell library. In order to evaluate the performance of the complete DYNASPAM system, this experiment incorporates both the innovations from the SPAS and the dynamic mapping technique from this Chapter, with an end-to-end experiment. The gem5 [3] simulation framework was used for performance evaluation. All I/O and initialization phases were skipped in the kernels to capture the main computation in the simulator. The baseline was an OoO processor with system configuration shown in Table 5.2. We also implemented the DYNASPAM subsystem in conjunction with the baseline OoO pipeline with same configurations. We only measured and compared the kernel performance for the benchmarks. We used McPAT v1.2 [51] to model the power/energy of DYNASPAM by using the performance statistics gem5 as input. We also estimated power for the configuration cache was estimated separately using CACTI [61].

Parameter	Setting
Fetch Unit	16-entry return stack; 4K-entry BTB Branch Predictor
Caches	64KB, 2-way, 2-cycle ICache; 64KB, 2-way, 2-cycle L1D; 2MB, 8-way, 20-cycle L2D (64-byte blocks for all caches)
Window Size	192-entry ROB; 256-entry physical RF; 8-wide issue
Execution Units	4 Int ALUs; 1 Int MUL/DIV; 4 Floating ALUs; 1 Floating MUL/DIV; 2 LDST units
Memory Unit	128-entry load queue; 128-entry store queue
Fabric	8-entry buffers; same execution units as OOO per strip; 16 strips; 3 pass regs per FU; 16 Live-in FIFOs, 16 Live-out FIFOs
Config. Cache	16-entry, direct mapped, 16-byte blocks, 3-bits saturation counter, threshold value 4

Table 5.2: Evaluation system parameters

Benchmarks DYNASPAM is evaluated using eleven programs from the Rodinia benchmark suite [11]. An overview of these benchmarks are shown in Table 5.3.1. We evaluate the full DYNASPAM system using the OpenMP version of all benchmarks, with OpenMP

pragmas disabled to enable a sequential version. All the benchmarks are compiled with -O3 flag.

5.3.2 Results

Area The datapath and FIFO buffers are designed separately. Table 5.4 shows that the size of each data path block, containing pass registers and multiplexers, is almost as large as an integer ALU, and that the area of FIFOs are much smaller. The overall fabric size is $2.9mm^2$ with 8 stripes (A 2-core AMD Bulldozer is $30.0mm^2$ at this technology node). The area of the configuration cache is obtained from CACTI, and the number is $0.003mm^2$.

Trace Coverage For different benchmarks, the coverage of dynamically accelerated instructions by the fabric vary, depending on the loop regularity and hardware resources. Figure 5.5 shows the percentage of dynamic instructions executed on the host OoO pipeline (Normal), the percentage of instructions that have been mapped to the fabric but not of-flooded yet (Mapping), and the percentage that are accelerated and run on the fabric (Accelerating). We evaluate with pre-set trace lengths ranging from 16 to 40 instructions. From the figure, we observe that a small fraction of instructions are executed during the mapping phase for all programs. Generally, traces with longer lengths have higher coverage. However, if a trace contains only a few instructions from a block, it will force more instructions to run on the host pipeline. As an example, imagine a single block with 33 instructions that executes in a loop. At a trace length of 32, 32/33 instructions execute on the fabric and 1/33 instructions executes on the host OoO pipeline. At 40 instructions, the trace enters a new block and 40/66 instructions execute on the fabric and 26/66 instructions execute on the fabric, thereby reducing coverage. NW with 24 instructions, and SRAD with 40 instructions are examples of this effect at work. We use a trace length of 32 instructions for all the following experiments.

Benchmark Name	Domain	Kernel	Description
Back Propagation (BP)	Pattern Recognition	bpnn_train_kernel	Machine learning algorithm to train the node weights of a neural network
Breadth-First Search (BFS)	Graph Algorithms	BFSGraph	Breadth-first search on a graph
B+ Tree (BT)	Search	kernel_cpu	Search in a B+ tree
Hotspot (HS)	Physics Simulation	compute_tran_temp	Estimate processor temperature based on power simulation
Kmeans (KM)	Data Mining	kmeans_clustering	Clustering algorithm for data-mining
LU Decomposition (LD)	Linear Algebra	lud_base	Matrix decomposition
K-Nearest Neighbors (KNN)	Data Mining	main	Finding the k-nearest neighbors from an unstructured data set
Needleman-Wunsch (NW)	Bioinformatics	runTest	Nonlinear global optimization method for DNA sequence alignments
PathFinder (PF)	Grid Traversal	run	Shortest path finder on a 2-D grid using dynamic programming
Particle Filter (PTF)	Medical Imaging	particleFilter	Statistical estimator of targeted object location given noisy measurements
SRAD (SRAD)	Image Processing	main	Diffusion method for ultrasonic and radar imaging applications

Table 5.3: Programs tested from the Rodinia Benchmark Suite.

Module names	Area(μm^2)	Module names	Area(μm^2)
sparc_exu_alu	4660	fpu_add	34370
sparc_mul_top	47752	fpu_mul	62488
sparc_exu_div	11227	fpu_div	13769
data_channel	4717	fifo	848

Table 5.4: Area Comparison for different components

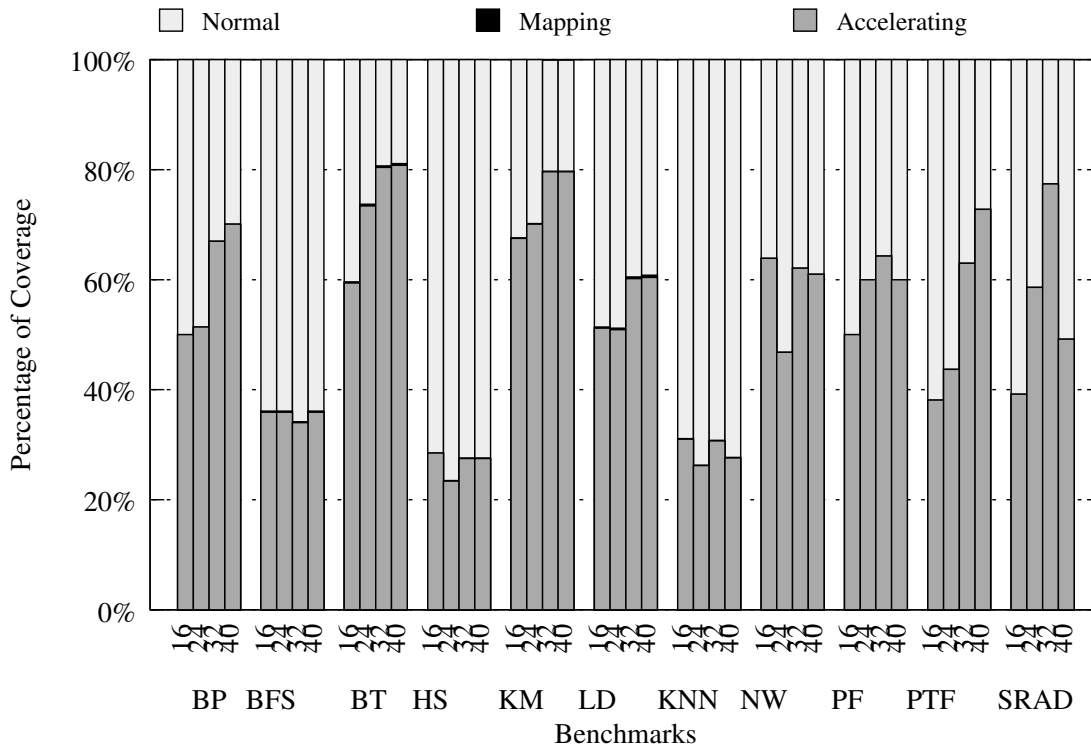


Figure 5.5: Trace Coverage. “Normal” is the percentage of dynamic instructions executed on the host OoO pipeline, “Mapping” is the percentage of dynamic instructions which are detected as hot traces, but not offloaded to the spatial fabric yet, and “Accelerating” is the percentage of dynamic instructions which are offloaded to the fabric successfully.

Configuration Lifetime Table 5.5 shows the number of traces that are detected and mapped successfully (mapped trace), and the number traces that are actually offloaded (offloaded trace). Some of the traces are mapped but never offloaded due to their low frequency of execution. The last three columns of Table 5.5 show the average configuration lifetime, which starts when the fabric is configured by one trace and ends when the fabric is reconfigured by another trace. From Table 5.5, we find that the average configuration

Benchmark Name	Mapped Traces	Offloaded Traces	Avg. Config. Lifetime (Invocation)		
			1 fabric	2 fabrics	4 fabrics
BP	2	2	6505.5	13013.0	13013.0
BFS	24	10	6.4	8.5	63.9
BT	4	3	197.4	246.8	987.0
HS	11	2	1065.0	2130.0	2130.0
KM	1	1	2750.0	2750.0	2750.0
LD	9	5	81.8	334.4	7690
KNN	4	3	2750.0	2750.0	2750.0
NW	1	1	13276.0	13276.0	13276.0
PF	2	1	6514.0	6514.0	6514.0
PTF	2	2	46.2	9240.0	9240.0
SRAD	1	1	33574.0	33574.0	33574.0

Table 5.5: Detected Traces and Average Configuration Lifetime. The “Mapped Traces” column shows how many hot traces are detected and translated to configurations by the hardware, and “Offloaded Traces” column shows among them how many traces are really offloaded to the fabric. The “Avg. Config. Lifetime” shows how many invocations the offloaded traces need to be evicted from the fabrics for the new offloaded traces. The larger the better.

lifetime is above 40 trace invocations with 1 on-chip fabric for all programs except BFS, which has only 6.4 invocations per configuration. Investigating BFS reveals that there are many unbiased control branches within the loop. Multiple fabrics can be used to reduce reconfiguration times and increase efficiency. We modeled architectures with 2 and 4 fabrics and use a least-recently-used (LRU) policy to manage reconfiguration. The experiment results show that with 4 fabrics, BFS’s average configuration life time is 64 invocations, and reaches 2045 invocations with 8 fabrics (not shown in the table).

Performance We compared the performance of DYNASPAM with three different configurations to the baseline OoO pipeline, as shown in Figure 5.6. DYNASPAM with “mapping” only maps the detected traces but does not offload them to the fabric. DYNASPAM with “mapping + acceleration w/ speculation” both maps and offloads the traces to the fabric

while using memory speculation. DYNASPAM with “mapping + acceleration w/o speculation” maps and offloads the detected traces while conservatively preserving all load-store and store-store orderings.

Recall that mapping overhead comes from two sources: 1) time draining the pipeline backend when the trace mapping starts; and 2) the cost of pausing instruction issue for long latency functional units during mapping. The simulation results show that the overhead of mapping is small, and causes less than 3% slowdown for all the benchmarks.

Without memory speculation, DYNASPAM produces a $1.23\times$ geomean performance and causes slowdown in two programs, NW and SRAD, which have a large fraction of dynamic memory instructions. With memory speculation enabled, DYNASPAM produces a $1.42\times$ geomean performance improvement without ever causing program slowdown. This shows the importance of effectively using memory speculation.

Work on DYNASPAM has primarily focused on the feasibility and applicability of dynamic mapping, and has not focused on optimizing area usage. In future work, research will be done to adjust the number of functional units according to instruction type distributions of the benchmarks.

Energy We measured the energy consumption in the simulation of both the baseline processor and SPAS. Figure 5.7 demonstrates the energy consumption of different hardware components to show the energy increase/decrease in each component. The overall energy consumption is reduced by 2.5%-36.86%, with geomean 23.9%. For each benchmark, it is clear that the energy consumption from Fetch, Rename, Instruction Scheduling (InstSchedule), and the bypass networks (Datapath) are reduced. On the other hand, power consumption for the memory system is increased, since SPAS cannot reduce memory activity. The energy consumption of the fabric includes both the functional units and datapath, which is

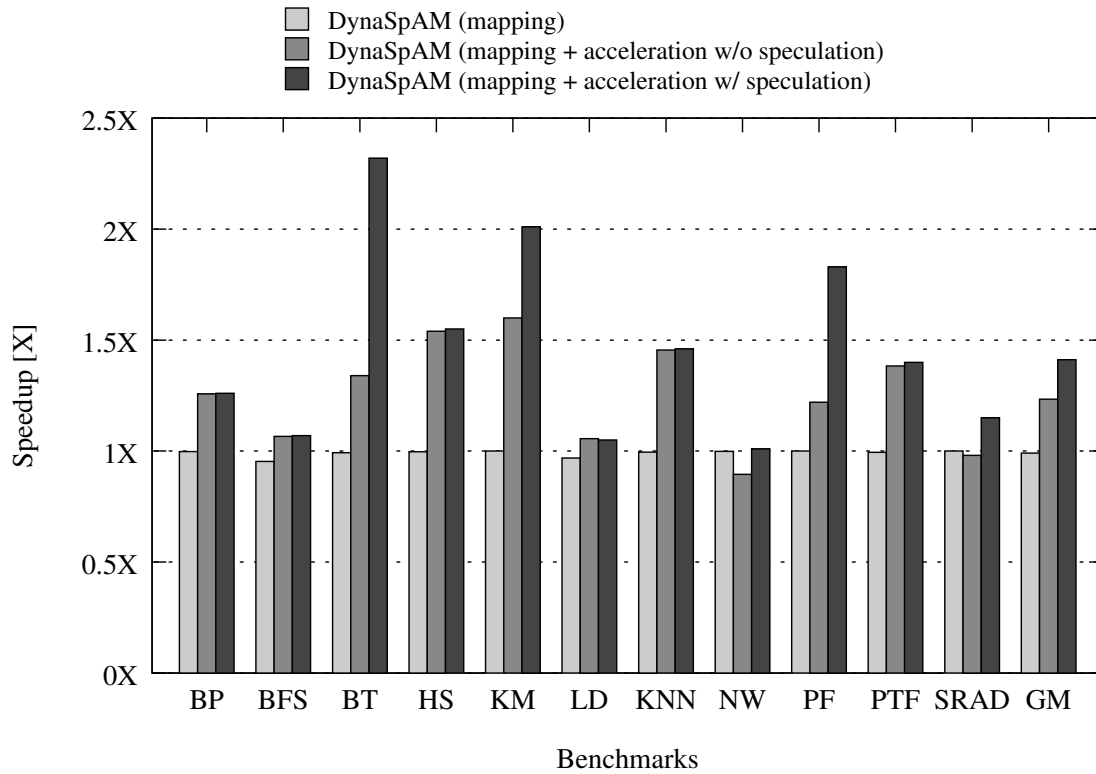


Figure 5.6: Performance Comparison with Respect to Host OoO Pipeline.

greater than the energy consumed by Execution on the OoO pipeline but smaller than the sum of Execution, Datapath, and InstSchedule.

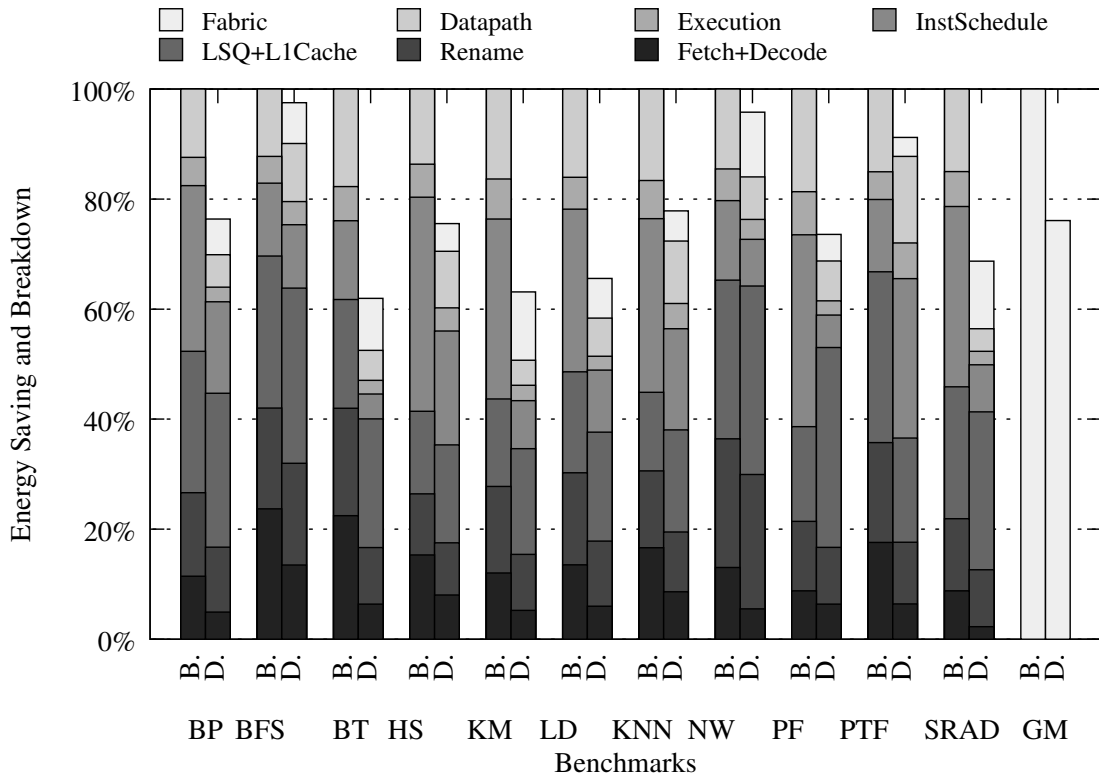


Figure 5.7: Energy Comparison with Respect to Host OoO Pipeline.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

Reconfigurable spatial architectures can be more efficient than OoO processors. However, the state of the art spatical architecture design relies heavily on profiling and compiler techniques to explore instruction and loop level parallelism. To make it work for outer loops with irregular memory access patterns and control flows, this dissertation presents Coarse-Grained Pipelined Accelerators (CGPA), an HLS framework that synthesizes efficient specialized accelerator modules for individual loops by utilizing coarse-grained pipeline parallelism technique. The combination of coarse-grained pipelining and exploitation of parallelism within each pipelined stage allows CGPA to design efficient accelerators for C/C++ programs with irregular memory accesses and complex control flows. Compared to the unparallelized version, CGPA shows speedups of 3.0x–3.8x for 5 kernels from programs in different domains.

This dissertation also exploits a dynamic mapping method for spatial architecture, to overcome the disadvantage of static methods, such as the inability to adapt to different

workloads and the lack of compatibility. It identifies the major challenge of dynamic instruction mapping for spatial architectures is getting large enough instruction scheduling window. To solve this issue, This dissertation presents DYNASPAM, a framework to dynamically detect, map and accelerate long hot instruction sequences from an OoO pipeline on a spatial fabric. Specifically, DYNASPAM leverages existing features from the OoO pipeline and actively guides instruction issue to generate efficient mappings for the fabric. This new method is both low-cost and efficient. Experimental results, a geomean 1.42X speedup with 23.9% energy consumption reduction for 11 benchmarks from Rodinia Benchmark Suite, demonstrate the potential gains from symbiotic combination of an OoO pipeline and spatial dataflow architecture.

6.2 Future Directions

For both the static and dynamic instruction mapping techniques presented in this dissertation, there are several improvements that can be exploited in the future.

First of all, the CGPA compiler only focuses on outer loops and relies on the adaptive backend to optimize inner loops which are suitable for targeting by traditional methods. In the future, additional optimizations using the existing research results such as prefetching and other loop-level parallelism techniques can be applied to the inner loops, especially on each stage separately. Additionally, the bandwidth of the memory system can be increased by synchronizing the memory accesses from parallel stage workers [43].

Secondly, the spatial fabric designed in SPAS and has same functional unit settings per row, and it is an interesting direction to optimize the types of functional unit and the connections between them for better area overhead and performance improvement. For example, the ratio between ALUs and LDST units can be optimized according to the ratio of memory operations in the programs. Moreover, multiple spatial fabrics with different

functional units settings can be integrated to the same chip, so workloads with different resource requirements can be offloaded to their optimal targets.

Last but not least, it is interesting to see DYNASPAM can be applied to other spatial fabrics, such as DySER [34] and SGMF [85], which are using static mapping methods to translate program code to fabric configurations. The topology of these fabrics are different from that of DYNASPAM and requires adding new scheduling constraints to the *resource-aware* scheduler of DYNASPAM.

Bibliography

- [1] Altera corp. <http://www.altera.com>, 2018. [Online; accessed 01-March-2018].
- [2] Autoesl/vivado. <https://www.xilinx.com/products/design-tools/vivado/integration.html>, 2018. [Online; accessed 01-March-2018].
- [3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
- [4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.
- [5] A. Bracy, P. Prahlaad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *In Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, pages 18–29, 2004.
- [6] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–26, 2004.

- [7] M. Butler and Y. N. Patt. An investigation of the performance of various dynamic scheduling techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO)*, pages 1–9, 1992.
- [8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 33–36, 2011.
- [9] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 29–38, 1995.
- [10] J. E. Carrillo and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 141–150, 2001.
- [11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.
- [12] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 270–281, 2000.
- [13] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 142–153. IEEE Computer Society, 1998.
- [14] 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

- Proceedings of the 32Nd Annual International Symposium on Computer Architecture (ISCA)*, pages 272–283, 2005.
- [15] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–400, 2008.
- [16] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Proceedings of the 49rd Annual Design Automation Conference (DAC)*, pages 1229–1234, 2012.
- [17] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, pages 433–438, 2006.
- [18] P. Diniz and J. Park. Automatic synthesis of data storage and control structures for fpga-based computing engines. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 91–100, 2000.
- [19] B. Draper, W. Nar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins. Compiling and optimizing image processing algorithms for fpgas. In *Proceedings of Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 222–231, 2000.
- [20] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping applications to the rapid configurable architecture. In *Proceedings of The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr 1997.
- [21] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN*

- '94 *Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, June 1994.
- [22] H. Esmailzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.
- [23] C. Fallin, C. Wilkerson, and O. Mutlu. The heterogeneous block architecture. Technical report, SAFARI Group, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2014.
- [24] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.
- [25] E. Fiksman, Y. Birk, and O. Mencer. Asc-based acceleration in an fpga with a processor core using software-only skills. In *Proceedings of 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 271–272, 2006.
- [26] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, pages 162–171, Nov 1994.
- [27] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings 31th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 173–181, December 1998.
- [28] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the streams-c c-to-fpga compiler: An applications perspective. In *Proceedings of the 2001 ACM/SIGDA Ninth*

- International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 134–140, 2001.
- [29] R. Ghiya and L. J. Hendren. Is it a Tree, DAG, or Cyclic Graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 1996.
- [30] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 121–133, January 1998.
- [31] M. B. Gokhale and J. M. Stone. Napa c: compiling for a hybrid risc/fpga architecture. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 126–135, 1998.
- [32] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High LevelLanguage. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–56, 2000.
- [33] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: A co/processor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, pages 28–39, 1999.
- [34] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 503–514, 2011.

- [35] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 12–23, 2011.
- [36] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*, pages 37–47, 2010.
- [37] J. R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, 1997.
- [38] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu. Elastic cgras. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 171–180, 2013.
- [39] Z. Huang and S. Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 735–745, 2001.
- [40] Z. Huang and S. Malik. Exploiting operation level parallelism through dynamically reconfigurable datapaths. In *Proceeding of 39th Design Automation Conference*, pages 337–342, 2002.
- [41] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery.

- The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.
- [42] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 93–103, 2003.
- [43] T. B. Jablin. *Automatic Parallelization for GPUs*. PhD thesis, Princeton, NJ, USA, 2013.
- [44] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2017.
- [45] N. P. Johnson, T. Oh, A. Zaks, and D. I. August. Fast condensation of the program dependence graph. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 39–50, 2013.
- [46] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang,

- E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.
- [47] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 381–394, 2007.
- [48] O. Kocberber, B. Grot, Picorel J., Falsafi B., Lim K., and Ranganathan P. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.
- [49] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.
- [50] H. H. Lee, Y. Wu, and G. Tyson. Quantifying instruction-level parallelism limits on an epic architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 21–27, April 2000.
- [51] S. Li, J.-H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.
- [52] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *Proceedings of the*

- 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 36–47, New York, NY, USA, 2013. ACM.
- [53] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August. Dynaspam: Dynamic spatial architecture mapping using out of order instruction schedules. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*, pages 541–553, New York, NY, USA, 2015. ACM.
- [54] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August. Cgpa: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, pages 78:1–78:6, New York, NY, USA, 2014. ACM.
- [55] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.
- [56] D. S. McFarlin, C. Tucker, and C. Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 241–252, 2013.
- [57] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the Conference on Field Programmable Logic*, volume 2778, pages 61–70, 2003.
- [58] E. Mirsky and A. DeHon. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, Apr 1996.
- [59] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. Tartan: Evaluating spatial computation for whole program execution. In

Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 163–174, 2006.

- [60] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.
- [61] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. Technical report, HP Laboratories, 2009.
- [62] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 13–25, 1997.
- [63] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 495–506, 2013.
- [64] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.
- [65] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, pages 206–218, 1997.
- [66] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 24–35, New York, NY, USA, 2013. ACM.

- [67] M. Quax, J. Huisken, and J. van Meerbergen. A scalable implementation of a reconfigurable wcdma rake receiver. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*, pages 30230–30240, 2004.
- [68] E. Raman. *Parallelization Techniques with Improved Dependence Handling*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, June 2009.
- [69] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.
- [70] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO)*, pages 63–74, December 1994.
- [71] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, 1994.
- [72] B. Robotmili, D. Li, H. Esmailzadeh, S. Govindan, A. Smith, A. Putnam, D. Burger, and S. W. Keckler. How to implement effective prediction and forwarding for fusible dynamic multicore architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 460–471, 2013.
- [73] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO)*, pages 24–34, December 1996.

- [74] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003.
- [75] M. B. Seth and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 216–227, March 2003.
- [76] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.
- [77] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 291–303, 2003.
- [78] Synopsys Design Compiler. <https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html>, 2018. [Online; accessed 01-March-2018].
- [79] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpfen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 2–12, 2004.
- [80] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale. Trident: an fpga compiler framework for floating-point algorithms. In *International Conference on Field Programmable Logic and Applications*, 2005.

- [81] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 49–59, 2007.
- [82] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, pages 205–218, 2010.
- [83] J. Villarreal, A. Park, W. Nar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–134, 2010.
- [84] C. Villavieja, J. A. Joao, R. Miftakhutdinov, and Y. N. Patt. Yoga: A hybrid dynamic vliw/ooo processor. Technical report, High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas 78212-0240, USA, 2014.
- [85] D. Voitsechov and Y. Etsion. Single-graph multiple flows: Energy efficient design alternative for gpgpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, pages 205–216, 2014.
- [86] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.
- [87] M. Weinhardt and W. Luk. Pipeline vectorization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(2):234–248, 2001.

- [88] R. D. Wittig and P. Chow. Onechip: an fpga processor with reconfigurable logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Apr 1996.
- [89] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO)*, pages 274–286, December 1996.
- [90] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
- [91] J. Xu, N. Subramanian, A. Alessio, and S. Hauck. Impulse c vs. vhdl for accelerating tomographic reconstruction. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 171–174, 2010.
- [92] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 225–235, 2000.
- [93] Q. Zhu, K. Vaidyanathan, O. Shacham, M. Horowitz, L. Pileggi, and F. Franchetti. Design automation framework for application-specific logic-in-memory blocks. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 125–132, July 2012.