Université Pierre et Marie Curie

École Doctorale Informatique, Télécommunications et Électronique

Compiling for a multithreaded dataflow architecture: algorithms, tools, and experience

par **Feng LI**

Thèse de doctorat d'Informatique

Dirigée par **Albert COHEN**

Présentée et soutenue publiquement le 20 mai, 2014

Devant un **jury** composé de:

,	Président
,	Rapporteur
,	Rapporteur
,	Examinateur
,	Examinateur
,	Examinateur

I would like to dedicate this thesis to my mother

Shuxia Li, for her love

Acknowledgements

I would like to thank my supervisor Albert Cohen, there's nothing more exciting than working with him. Albert, you are an extraordinary person, full of ideas and motivation. The discussions with you always enlightens me, helps me. I am so lucky to have you as my supervisor when I first come to research as a PhD student. I would also like to thank Dr. Antoniu Pop, I still enjoy the time we were working together, helped a lot during my thesis, as a good colleague, and a friend.

I am grateful to my thesis reviewers: Prof. Guang Gao and Dr. Fabrice Rastello. Thanks for your time carefully reading the thesis, and providing the useful feedbacks. For the revised version of the thesis, I would also like to thank Dr. Stéphane Zuckerman, he gives very useful suggestions and comments.

I would like to thank all the members in our PARKAS team. Prof. Marc Pouzet, Prof. Jean Vuillemin, and Dr. Francesco Zappa Nardelli invite lots of talks in our group, bring in new ideas and inspirations. Tobias Grosser, Jun Inoue, Riyadh Baghdadi, I still enjoy the time we discussing together, climbing together and flying to conferences. Boubacar Diouf, Jean-Yves Vet, Louis Mandel, Adrien Guatto, Timothy Bourke, Cédric Pasteur, Léonard Gérard, Guillaume Baudart, Robin Morisset, Nhat Minh Lê, you guys are wonderful people to work with!

Special thanks to my friends, Wenshuang Chen and Shunfeng Hu, always like your cooking! Also my friends Shengjia Wang, Ke Song, Yuan Liu, enjoy our beer time, the moments we discuss and implement a new idea.

Abstract

Across the wide range of multiprocessor architectures, all seem to share one common problem: they are hard to program. It is a general belief that parallelism is a software problem, and that perhaps we need more sophisticated compilation techniques to partition the application into concurrent threads. Many experts also make the point that the underlining architecture plays an equally important role: there needs to be a fundamental change in processor architecture before one may expect significant progress in the programmability of multiprocessors.

Our approach favors a convergence of these viewpoints. The convergence of dataflow and von Neumann architecture promises latency tolerance, the exploitation of a high degree of parallelism, and light thread switching cost. Multithreaded dataflow architectures require a high degree of parallelism to tolerate latency. On the other hand, it is error-prone for programmers to partition the program into large number of fine grain threads. To reconcile these facts, we aim to advance the state of the art in automatic thread partitioning, in combination with programming language support for coarse-grain, functionally deterministic concurrency.

This thesis presents a general thread partitioning algorithm for transforming sequential code into a parallel data-flow program targeting a multithreaded dataflow architecture. Our algorithm operates on the program dependence graph and on the static single assignment form, extracting task, pipeline, and data parallelism from arbitrary control flow, and coarsening its granularity using a generalized form of typed fusion. We design a new intermediate representation to ease code generation for an explicit token match dataflow execution model. We also implement a GCC-based prototype. We also evaluate coarse-grain dataflow extensions of OpenMP in the context of a large-scale 1024-core, simulated multithreaded dataflow architecture. These extension and simulated architecture allow the exploration of innovative memory models for dataflow computing. We

evaluate these tools and models on realistic applications.

Contents

Contents v

List of Figures ix

Nomenclature xii

1 Introduction 1

1.1 Hybrid Dataflow for Latency Tolerance 2

1.1.1 Convergence of dataflow and von Neumann 2

1.1.2 Latency Tolerance 3

1.1.3 TSTAR Multithreaded Dataflow Architecture 5

1.2 Task Granularity 7

1.3 Motivation 9

1.4 Dissertation Outline 10

2 Problem Statement 12

2.1 Explicit token matching shifts the challenges in hardware design to compilation 13

2.2 The complex data structure should be handled in an efficient way 14

2.3 Related Work 15

2.3.1 Compiling imperative programs to data-flow threads . . . 15

2.3.2 SSA as an intermediate representation for data-flow compilation 16

2.3.3 Decoupled software pipelining 16

2.3.4 EARTH thread partitioning 17

2.3.5 Formalization of the thread partitioning cost model 18

3 Thread Partitioning I: Advances in PS-DSWP 19

3.1 Introduction 19
3.1.1 Decoupled software pipelining 20
3.1.2 Loop distribution 21
3.2 Observations 21
3.2.1 Replacing loops and barriers with a task pipeline 21
3.2.2 Extending loop distribution to PS-DSWP 22
3.2.3 Motivating example 23
3.3 Partitioning Algorithm 30
3.3.1 Definitions 31
3.3.2 The algorithm 32
3.4 Code Generation 36
3.4.1 Decoupling dependences across tasks belonging to different treegions 36
3.4.2 SSA representation 37
3.5 Summary 40

4 TSTAR Dataflow Architecture 42

4.1 Dataflow Execution Model 42
4.1.1 Introduction 42
4.1.2 Past Data-Flow Architectures 43
4.2 TSTAR Dataflow Execution Model 47
4.2.1 TSTAR Multithreading Model 47
4.2.2 TSTAR Memory Model 48
4.2.3 TSTAR Synchronization 50
4.2.4 TSTAR Dataflow Instruction Set 52
4.3 TSTAR Architecture 55
4.3.1 Thread Scheduling Unit 56

5 Thread Partitioning II: Transform Imperative C Program to Dataflow Program 61

5.1 Revisit TSTAR Dataflow Execution Model 62

5.2 Partitioning Algorithms . . . 65
5.2.1 Loop Unswitching . . . 66
5.2.2 Build Program Dependence Graph under SSA . . . 67
5.2.3 Merging Strongly Connected Components . . . 69
5.2.4 Typed Fusion . . . 69
5.2.5 Data Flow Program Dependence Graph . . . 71
5.3 Modular Code Generation . . . 74
5.4 Implementation . . . 76
5.5 Experimental Validation . . . 78
5.6 Summary . . . 82

6 Handling Complex Data Structures 83
6.1 Streaming Conversion of Memory Dependences (SCMD) . . . 84
6.1.1 Motivating Example . . . 84
6.1.2 Single Producer Single Consumer . . . 86
6.1.3 Single Producer Multiple Consumers . . . 87
6.1.4 Multiple Producers Single Consumer . . . 91
6.1.5 Generated Code for Motivating Example . . . 92
6.1.6 Discussion . . . 96
6.2 Owner Writable Memory . . . 96
6.2.1 OWM Protocol . . . 97
6.2.2 OWM Extension to TSTAR . . . 99
6.2.3 Expressiveness . . . 101
6.2.4 Case Study: Matrix Multiplication . . . 102
6.2.5 Conclusion and perspective about OWM . . . 103
6.3 Summary . . . 105

7 Simulation on Many Nodes 106
7.1 Introduction . . . 107
7.2 Multiple Nodes Dataflow Simulation . . . 108
7.3 Resource Usage Optimization . . . 110
7.3.1 Memory Usage Optimization . . . 110
7.3.2 Throttling . . . 115

7.4 Experimental Validation 117
7.4.1 Experimental Settings 117
7.4.2 Experimental Results 118
7.4.2.1 Gauss Seidel 121
7.4.2.2 Viola Jones 124
7.4.2.3 Sparse LU 126
7.5 Summary 129

8 Conclusions and Future Work **131**
8.1 Contributions 131
8.2 Future Work 132

Personal Publications **136**

References **138**

List of Figures

1.1 TStar High level Architecture. 6
1.2 The Parallelism-Overhead Trade-off (from Sarkar) 8

2.1 General strategy for thread partitioning. 13

3.1 Barriers inserted after loop distribution. 21
3.2 Pipelining inserted between distributed loops. Initialize the stream (left), producer and consumer thread (right). 22
3.3 Uncounted nested loop before partitioning. 24
3.4 Uncounted nested loop in SSA form. 25
3.5 Loops after partitioning and annotated with OpenMP stream extension. 26
3.6 Pipelining and parallelization framework. 27
3.7 Definition and use edges in the presence of control flow. 28
3.8 Control dependence graph of Figure 3.3. Express the definition of treegion. 30
3.9 Split conditional statements to expose finer grained pipelining. . . 32
3.10 Algorithm for marking the irreducible edges. 33
3.11 Structured typed fusion algorithm. 35
3.12 Before partitioning (left), and After partitioning (right). Loop with control dependences. 36
3.13 Normal form of code (left) and using streams (right). 37
3.14 Normal form of code (left) and SSA form of the code (right). . . . 37
3.15 Apply our algorithm to generate the parallel code. Producer thread (left) and consumer thread (right). 39

3.16 Multiple producers with applied our algorithm, the generated code. 40

4.1 Firing rule example. 43
4.2 The basic organization of the static (a) and dynamic (b) model. . 44
4.3 Program dependence graph of an illustrative example. 50
4.4 Producer consumer relationship. 53
4.5 Producer consumer code. 54
4.6 TSTAR Highlevel Architecture. 57
4.7 Overview of the Thread Scheduling Unit (TSU). 58

5.1 Explicit Matching Operation. 62
5.2 Program Dependence Graph for explicit token matching compilation example. 64
5.3 Running example (left) and after unswitching (right). 66
5.4 SSA form before unswitching (left) and after (right). 67
5.5 SSA representation after loop unswitching. 68
5.6 SSA-PDG for the code example in Figure 5.5. 68
5.7 SSA PDG after merging the SCC. 69
5.8 (A) SSA-PDG after typed fusion. (B) Data Flow Program Dependence Graph. 71
5.9 Splitting data dependences: (A) the original SSA-PDG and (B) the generated DF-PDG. 73
5.10 SSA representation for a simple loop carried dependence 74
5.11 Corresponding SSA-PDG (left) and a partial DF-PDG (right) for code in 5.10. 74
5.12 Caller and callee, threaded version. 76
5.13 Implementation within GCC. 77
5.14 Merge Sort running on 4 cores. 80
5.15 Computing the 42th Fibonacci number on 4 cores (above) and 24 cores (below). 81

6.1 Non-linear access for array A within a loop. 84
6.2 Decouple computation and memory accesses with loop distribution. 85
6.3 Single producer single consumer 88

6.4 Single producer multiple consumers . . . 90
6.5 Multiple producers and single consumer . . . 93
6.6 Generated code for control program using SCMD. . . . 94
6.7 Generated code for control program using SCMD. . . . 95
6.8 Owner Writable Memory. . . . 100
6.9 Pipeline using OpenStream. . . . 101
6.10 OpenStream cache example. . . . 102
6.11 First phase: Matrix allocation and initialization. . . . 103
6.12 Second phase: Matrix multiplication. . . . 104
6.13 Third phase: Output the results. . . . 104

7.1 COTSon Architecture. . . . 107
7.2 Multiple nodes simulation on COTSon. . . . 108
7.3 Code example for simple frame memory allocator . . . 111
7.4 Simple frame memory allocator . . . 112
7.5 Code example for simple frame memory allocator . . . 113
7.6 Host Space TLS Frame Memory Allocator . . . 114
7.7 Excessive parallelism. . . . 116
7.8 Simulation Results . . . 119
7.9 Simulation Results . . . 120
7.10 Simulation Results . . . 121
7.11 Dependence of gauss seidel: A. same iteration dependences (left) B. cross iteration dependences (right) . . . 122
7.12 Gauss seidel implementation with OpenStream. . . . 123
7.13 Gauss seidel implementation with OWM support. . . . 124
7.14 Gauss seidel implementation with OWM support (final task). . . . 124
7.15 Viola Jones OpenStream kernel. . . . 125
7.16 Cascade data structure. . . . 126
7.17 Dynamic OWM memory allocation. . . . 127
7.18 Dependence patterns of sparse lu: A. $k = 0$ B. $k = 1$. . . . 128
7.19 Sparse lu OpenStream implementation for `bdiv` dependences. . . . 128
7.20 Sparse lu OpenStream implementation for `bdiv` dependences (with OWM support). . . . 129

8.1 High level overview of the compilation and simulation architecture. 134

Chapter 1

Introduction

The design complexity and power constraints of large monolithic processors forced the industry to develop Chip-Multiprocessor (CMP) architectures. Across the diverge range of multiprocessor architectures, all seems to share one common problem: they are hard to program. The application writer has to take detailed architecture specific measures to improve the performance, and the resulting code are hard to read, maintain or debug, and not even to mention performance portability.

It is a general belief that the difficulties of parallel programming is a software problem. Perhaps we need more sophisticated compilers, to partition the applications into tasks that can run in parallel and to coordinate or synchronize them to implement a given functional semantics. Or perhaps we need more abstract parallel programming languages, allowing the programmer to write applications or port legacy code in a productive way, and then rely on static and dynamic algorithms to exploit the target architecture

It is also argued that the design and programming interface of a multiprocessor architecture plays an equally important role. It may very well be that the conventional cache-coherent multiprocessors derived from the von Neumann model are not suitable for the scalable and productive exploitation of parallelism. Following this school of thought, there is a need for a fundamental change in processor architecture before we can expect significant progress in the use of multiprocessors.

This thesis presents a general thread partitioning algorithm for transform-

ing sequential code into a parallel data-flow program targeting a multithreaded dataflow architecture. Our algorithm operates on the program dependence graph and on the static single assignment form, extracting task, pipeline, and data parallelism from arbitrary control flow, and coarsening its granularity using a generalized form of typed fusion. We design a new intermediate representation to ease code generation for an explicit token match dataflow execution model. We also implement a GCC-based prototype. We also evaluate coarse-grain dataflow extensions of OpenMP in the context of a large-scale 1024-core, simulated multithreaded dataflow architecture. These extension and simulated architecture allow the exploration of innovative memory models for dataflow computing. We evaluate these tools and models on realistic applications.

1.1 Hybrid Dataflow for Latency Tolerance

The early study starts from dataflow architectures. By contrast with von Neumann architecture, the execution of an instruction in dataflow model is driven by the availability of data. In 1975, Dennis and Misunas proposed the static dataflow architecture Dennis & Misunas [1979], due to the limitation with iterative constructs and reentrancy Arvind & Culler [1986b], the dynamic, or tagged-token dataflow architecture was proposed by Watson and Gurd Watson & Gurd [1979], which exposes additional parallelism by allowing multiple instances of reentrant code, the main disadvantage of the dynamic model is the extra overhead in matching tags on tokens.

1.1.1 Convergence of dataflow and von Neumann

The limit of the static dataflow model and the large cost of dynamic dataflow execution of individual instructions led to the convergence of dataflow and von Neumann models. The hybrid architecture considered in this thesis moves from fine-grained parallelism towards coarse-grained execution model, which combines the power of the dataflow model for exploiting parallelism with the efficiency of the control-flow model. Two key features supporting this shift are low-cost sequential scheduling of instructions and the use of registers to temporarily hold

the results of instructions in a single thread.

Sequential instruction scheduling takes advantage of the fact that any dataflow program displays some level of sequential execution. For example, in the case where the output of one node goes to the next node, the dependences between two nodes could never executed in parallel. Therefore, there is little point to scheduling them to different processors. Sequential scheduling enables a coarser grain of parallelism compared to one instruction per task in pure dataflow model, allows to use of a simple control-flow sequencing within the grain. This comes from the fact that the data-driven sequencing is unnecessarily general and at a cost of overhead required to match tokens. Combining sequential scheduling with instruction-level context switching also gives another perspective of dataflow architectures: multithreading. In multithreaded architectures, a thread is defined as a sequence of statically ordered instructions, where once the first instruction is fired, the remaining will execute without interruption.

Multiple flavors of hybrid architectures exist: one approach essentially follows the von Neumann architecture with a few dataflow additions (**large-grain dataflow architecture**), another approach extends a dataflow architecture with some von Neumann additions (**multithreaded dataflow architecture**) Papadopoulos & Traub [1991]; Lee & Hurson [1994]; Iannucci [1988]; Nikhil [1989]; Nikhil *et al.* [1992]; Culler *et al.* [1991].

1.1.2 Latency Tolerance

The term *von Neumann bottleneck* was coined by John Backus in his 1977 ACM Turing Award lecture. According to Backus:

> Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von

Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it Backus [1978]

The shared bus between the program memory and data memory leads to the von Neumann bottleneck, the limited communication bandwidth between the CPU and memory limits the effective processing speed when the CPU is required to perform minimal processing on large amount of data - which is also referred as the memory wall. The memory wall is the growing disparity of speed between CPU and memory outside the CPU chip. Because of the memory wall, enhancing the CPU alone cannot guarantee improvements on system performance. From 1986 to 2000, CPU speed improved at an annual of 55 percent while memory speed only improved at 10 percent. More over, when memory is physically distributed, the latency of the network and network interface is added to that of accessing the local memory on the node. Given these trends, it was expected that memory latency would become an overwhelming bottleneck in computing performance.

As the rapid progress of the multiprocessor speed, two main issues has to be addressed: *latency* and *synchronization*. Latency appears in various forms in multiprocessor, from a few cycles at the instruction level such as pipeline bubbles caused by branch instructions, to tens of cycles cause by cache misses (memory latency), up to hundreds of cycles caused by inter-processor communication, or even worse, thousands of cycles caused by the IO. Synchronization is equally important to enforce the ordering of instruction executions according to their data dependencies, in order to ensure deterministic behavior.

There are basically two ways of fighting latency, reducing and tolerating. For instance, as a hardware approach, we might replace the disk with a new storage media, as fast as RAM to reduce the IO latency, or we could apply the software approach, removing some of the inter-process communications by apply compiler optimizations, or apply the branch prediction at instruction level to remove the pipeline bubbles. However, no matter how advanced the technology is, we could not eliminate all latencies. The latencies caused by communication and synchronization are inherent in parallel applications. Tolerating therefore maybe the only available option under this circumstances.

Prefetching and multithreading are two ways of tolerating latencies. Prefetching is a mechanism that loads data into the cache or local memory before it is

actually used, anticipating it will be used in the near future. Thus prefetching is very effective if the latency length can be predicted at compile time. By contrast, multithreading is more general and flexible in coping with unpredicted latencies.

In a multithreaded dataflow architecture (or multithreading with dataflow origin), a program is partitioned into multiple threads, each thread is the unit of execution and scheduled to run dynamically. Within a thread, instructions are issued to executed sequentially as on a von Neumann architecture. Among threads, they are scheduled to run when the firing rule is satisfied. A strict firing rule allows a thread to execute only when all its inputs are available. During the execution of a thread, if there is a long latency operation, the processor can switch to another ready thread and do other useful work. If there is enough parallelism in application and a multithreaded machine can do fast thread switching, latencies caused by long latency operations can be overlapped with the execution of useful instructions from other threads. Thus latencies are effectively covered on such a multithreading machine.

A key open question for multithreaded dataflow architecture is how best to partition the programs into threads and what degree of granularity is best.

1.1.3 TSTAR Multithreaded Dataflow Architecture

TSTAR is a multithreaded dataflow architecture we target on, where in the inter-thread level, threads are executed in a multithreading environment, scheduled according to the dataflow model to better exploit parallelism, while in the intra-thread level, each thread is compiled into sequential von Neumann processor.

TSTAR execution model is a feed-forward dataflow execution model with explicit token matching, where the producers write directly to their consumers. This model origins from several inputs of our partners in the TERAFLUX project[1] DDM execution model from Arandi & Evripidou Arandi & Evripidou [2011]; T* instruction set from Portero et al. Portero *et al.* [2011]. We will present the TSTAR execution model in detail in chapter 4.

Figure 1.1 shows the TSTAR top level architecture. `C2` is the single core which contains a processing element (x86-64 ISA with TStar extensions) along

[1] http://www.teraflux.eu

with its L1 cache. Each core also includes a partition of the L2 cache. In order to support the dataflow execution of threads, each core includes a hardware module that handles the scheduling of threads - the Local Thread Scheduling Unit (L-TSU). In addition to cores, the nodes also contain a hardware module to coordinate the scheduling of the data-flow threads among the cores - the Distributed Thread Scheduling Unit (D-TSU), as well as a hardware module that monitors the performance and faults of the cores - the Distributed Fault Detection Unit (D-FDU).Each core is identified with a unique id, the Core ID (CID), and each group of cores belongs to a node whose id is the Node ID (NID). Nodes are connected via an inter-node network, the Network on Chip (NoC). Cores within a node are connected via the NoC.

1.2 Task Granularity

An important issue in multiprocessor performance is the granularity of the program execution. The granularity of a parallel program can be defined as the average size of a sequential unit of computation in the program, with no interprocessor synchronization or communications. For a given machine with multiprocessors, there is a fundamental trade-off between the granularity (amount of parallelism) and the overhead of synchronization. It is desirable for a multiprocessor to have small granularity, so that it can exploit larger amount of parallelism. It is also desirable for a parallel program to have larger granularity, so that the synchronization and communication overhead could be relatively small compared to the actual workloads.

Sarkar Sarkar [1989] articulates this trade-off as the competing contributions of the ideal parallel execution time, the amount of time required to execute the program in the absence of the overhead, with the overhead factor, the extra work required to schedule and coordinate the tasks. The ideal parallel execution time is multiplied by the overhead factor to yield actual parallel execution time, the amount of time required to complete the problem for a given task granularity in the presence of scheduling overhead.

Figure 1.2 (from Sarkar) illustrates the general characteristic of the parallelism-overhead trade-off for a typical program running on a machine with ten proces-

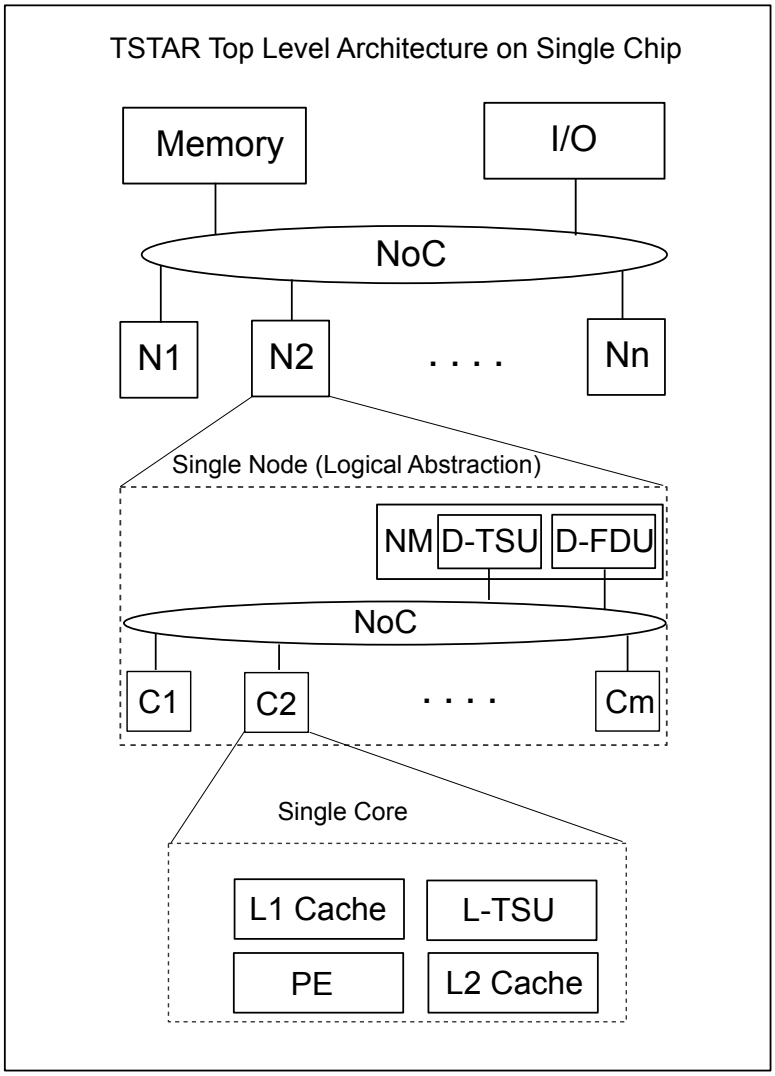

Figure 1.1: TStar High level Architecture.

sors. This plot is suggestive of what was experienced running various programs on contemporary multiprocessors, but it does not express the data from a specific machine or application. The normalized execution time is the ratio n/s where n is the number of processors and s is the actual speedup relative to a single processor. The normalized ideal parallel execution time increases from 1 to n as the task granularity increases from a single instruction, to 100,000 when the entire program executes as a single task. The task overhead factor is given by $(g + o)/g$ where g is the task size in the instructions and the o is the per-task

overhead, in this case 1000 instructions.

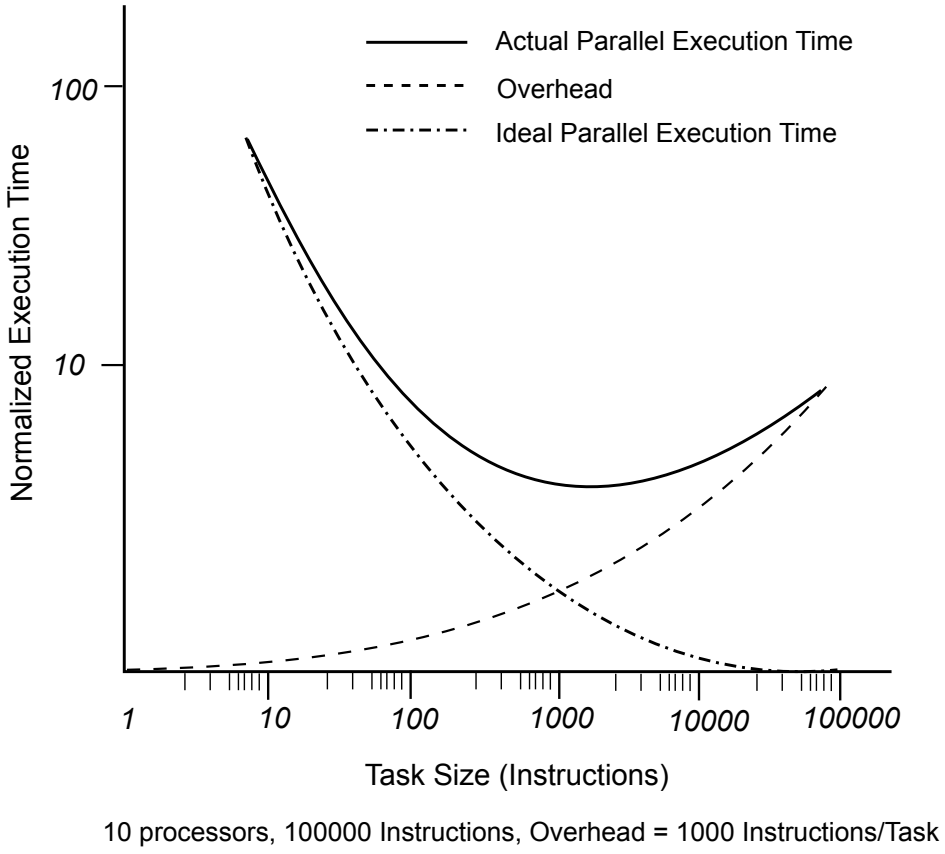

Figure 1.2: The Parallelism-Overhead Trade-off (from Sarkar)

It shows for a given multiprocessor, there is a minimum program granularity value below which the performance degrades significantly. Although the overhead shows in this figure is relatively long (*1000 instructions/task*), but we believe as the construct of the multiprocessor evolves, the overhead will decrease accordingly. There is always a trade-off between the number of parallelism and the task switching overhead. Within our multithreaded dataflow architecture, we assume a relatively low task overhead, by means of one to ten instructions per task. The hardware support largely reduce the synchronization and communication overhead. So we assume task overhead is not a first order issue, and the objective for thread partitioning is to expose the maximum amount of parallelism, and coarsen the granularity stays as an optimization pass in the procedure.

1.3 Motivation

There are three fundamental problems to be solved when compiling a program for parallel execution on a multiprocessor:

- Identifying parallelism in the program.
- Partitioning the program into sequential tasks.
- Scheduling the tasks on processors.

The problem of identifying or expressing parallelism belongs to the domain of programming language. Scheduling the tasks on processors is managed by a dedicated Thread Scheduling Unit (TSU) in the TSTAR multithreaded dataflow architecture. Partitioning the program into sequential tasks and targeting on TSTAR architecture is a main focus in this dissertation.

Multithreaded dataflow architectures require high-degree of TLP to tolerate latency. It is error-prone to ask programmers to partition the program to a large number of fine grain threads. On the other hand, porting the legacy code takes a large amount of work. Therefore, it is essential to have compiler support for multithreaded dataflow architectures so that they can be partitioned efficiently and widely accepted.

Automatic program parallelization techniques plus programmer efforts (e.g. annotations) can be applied to identify potential TLP. In our target architecture, threads are non-preemptive: once a thread starts its execution, it cannot be interrupted. In the non-preemptive model, thread start and end points are decided at compile time. It is the compiler's job to perform thread partitioning and optimize the partitioned code at compile time. Because of this, a compiler targeting multithreaded dataflow architecture faces real challenges when automatically partitioning threads:

- It must partition the program correctly with respect to the dependences constraints.
- It should perform optimizations on threads partitioned to make them execute efficiently.

- It should handle the complex data structures (like arrays) in an efficient way.

Thread partitioning is a challenging task in developing a compiler for multithreaded dataflow architectures. Measuring the performance metrics on the multithreaded dataflow architecture plays an equally important role. Before the real hardware processor is built, we need a flexible methodology to analyze the behavior of the proposed architecture. By performing simulations and analyzing the results with a full-system simulator, we can gain a thorough understanding how the proposed architecture behaves, how to improve it, and validate the results before it goes into the production cycle.

We will try to address all the issues mentioned above in this dissertation.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows.

In chapter 2 we present the problem statement when compiling for a multithreaded dataflow architecture.

As a preliminary research on thread partitioning, chapter 3 presents a thread partitioning algorithm. The algorithm extends loop transformations and Parallel Stage Decoupled Software Pipelining (PS-DSWP). Section 3.1 presents the related work in PS-DSWP and loop distribution. Section 3.2 presents our observations with extending loop distribution to PS-DSWP and a motivating example. Section 3.3 presents our thread partitioning algorithms based on SSA and treegion representation. After deciding the partition point between the original treegions, Section 3.4 explains the code generation challenges in presence of multiple producers and consumers.

Chapter 4 presents a general view of the TSTAR dataflow architecture. Section 4.1 gives an overview of the dataflow execution model. Section 4.2 further explains the TSTAR execution model from different aspects (multithreading model, memory model and synchronization), and section 4.3 gives an overview of the TSTAR architecture.

Chapter 5 presents a general algorithm for thread partitioning targets on

the TSTAR architecture, which transform sequential imperative programs into parallel data-flow programs. The algorithm operates on a program dependence graph in SSA form, extracting task, pipeline and data parallelism from arbitrary control flow, and coarsening its granularity using a generalized form of typed fusion. A prototype is implemented in GCC, and we give the evaluation results in the final section.

Chapter 6 complements chapter 5. In this chapter, we studies the method of handling complex data structure (such as arrays) in thread partitioning. Streaming Conversion of Memory Dependences (SCMD) connects independent writes and reads to the same memory location with stream dynamically. Owner Writable Memory model (OWM) is proposed to reduce the communication overheads when operating on complex data structures.

Chapter 7 presents the simulation infrastructure and benchmarks written with OWM memory model to validate the approach.

We conclude and draw some perspectives in chapter 8.

Chapter 2

Problem Statement

Figure 2.1 shows a general strategy for automatic thread partitioning. A serial program is translated to intermediate representations (Program Dependence Graph and Data Flow Program Dependence Graph). The IR is an input to the compile time analysis phase of the partitioning system. The output produced by compile-time analysis is a partition of the program graph into subgraphs which represent tasks. Finally, the code generation phase generate code for both x86 and TSTAR dataflow architectures.

Depending on the target, the compilation strategy diverges after thread partitioning. If the target is x86 architecture, the TSTAR backend in the compiler is disabled, the generated code will be linked to x86 dataflow runtime library. If target on TSTAR dataflow architecture, the backend will translate the dataflow builtin functions to dataflow ISA, so that it could be simulated on the full-system simulator.

Thread partitioning is a challenging task in developing a compiler for a multithreaded dataflow machine. Besides the points we discussed in previous chapter, we stress a few more points in this section.

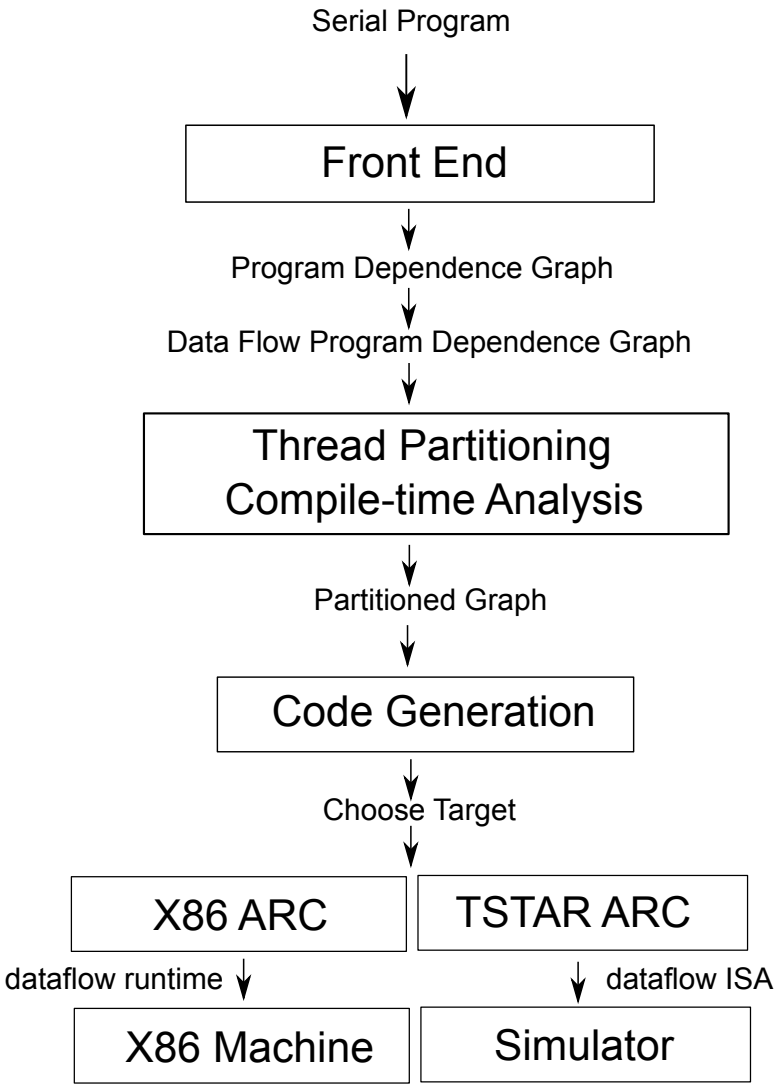

Figure 2.1: General strategy for thread partitioning.

2.1 Explicit token matching shifts the challenges in hardware design to compilation

In the non-preemptive thread model, each thread runs to completion once started. The data and control dependences need to be resolved and satisfied at partitioning stage. Unlike tagged-token dataflow machines, the tokens are stored separately in the *Frame Memory*, and use an explicit token matching strategy in TSTAR

execution model.

Tagged-token dataflow machines are implemented through a sophisticated matching hardware, which dynamically schedules operations with available operands Arvind & Culler [1986b] Memo & Culler [1983]. When a token arrives at a processor, the tag it carries is checked against the tags present in the token-store. If a matching token is found, it is extracted and the corresponding instruction is enabled for execution; otherwise, the incoming token is added to the store. This allows for a simple non-blocking processor pipeline that can overlap instructions from closely related or completely unrelated computations. However, the matching operation involves considerable complexity on the critical path of instruction scheduling Gajski *et al.* [1982].

The more subtle problem with the matching paradigm is that a failure to find a match implicitly allocates resources within the token store. Thus in mapping computations to processors places an unspecified commitment on the token storage hardware, if this resource becomes overcommited, the program may deadlock.

Explicit Token Matching shifts the complications from the hardware design of token-matching system to the compiler. The compiler has to match the producers and consumers explicitly. In TSTAR execution model, as producer data flow threads communicate by writing directly in the data flow frame of their consumers, it is necessary that, along all data dependence edges of the program dependence graph, the producer nodes know the data flow frame of the consumer nodes. It becomes more complicates when the producer and consumer are created by separate control programs, which means there needs to be a way to communicate such information. We address this issue in chapter 5.

2.2 The complex data structure should be handled in an efficient way

There are two basic approaches to represent data structures such as arrays: direct and indirect access schemes Gaudiot & Wei [1989]. The direct access scheme treats each structure element as individual data tokens. The dataflow functionality principle implies that all operations are side-effect free. The direct access

scheme is compatible with the dataflow principle. However, absence of side effect also means the token carries vectors, arrays or other complex data structures results in a new data structure, which will greatly increase the communication overhead in practice.

On the other hand, in an indirect access scheme, data structures are stored in special memory units and their elements are access through "read" and "write" operations. Literature has shown that storing the data structures and representing them by indirect access incurs less overhead that the direct access scheme in transmitting data, reconstructing the resultant array, and randomly accessing array elements.

However, the indirect access approach is not free of charge. When the array elements are stored in a separate storage, to preserve the functionality principle of dataflow, a write to a single element in the array might result in copying the entire array. If the array elements are stored in a virtual memory that is global addressable to all nodes, the consistent view of the virtual memory to related threads becomes a demanding task. We address this problem in Chapter 6.

2.3 Related Work

2.3.1 Compiling imperative programs to data-flow threads

The problem of compiling imperative programs for data-flow execution has been widely studied. Beck et al. Beck *et al.* [1989] propose a method for translating control flow to data flow, and show that data-flow graphs can serve as an executable intermediate representation in parallelizing compilers. Ottenstein et al. Ottenstein *et al.* [1990] study such a translation using the Program Dependence Web, an intermediate representation based on gated-SSA Tu & Padua [1995] that can directly be interpreted in either control-, data-, or demand-driven models of execution. Programs are transformed to the MIT dataflow program graph Arvind & Nikhil [1990], targeting the Monsoon architecture.

Najjar et al. evaluated multiple techniques for extracting thread-level data-flow Najjar *et al.* [1994]. These papers target a token-based, instruction-level data-flow model, analogous to the simulation of hardware circuits. In contrast,

our data-flow model does not require tokens or associative maps, shifting the effort of explicitly assign the consumer threads to their producers to the compiler. The comparison between our approach and the token-based solution is further discussed in chapter 5. In addition, thread-level data-flow requires additional efforts to coarsen the grain of concurrency, handling the dynamic creation of threads, and managing their activation records (data-flow frames).

2.3.2 SSA as an intermediate representation for data-flow compilation

The *static single assignment* form (SSA) is formally equivalent to a well-behaved subset of the *continuation-passing style* (CPS) Appel [1998]; Kelsey [1995] model, which is used in compilers for functional languages such as Scheme, ML and Haskell. The data-flow model has been tied closely to functional languages, since the edges in a data-flow graph can be seen both as encoding dependence information as well as continuation in a parallel model of execution. The SSA representation builds a bridge between imperative languages and the data-flow execution model. Our algorithm uses the properties of the SSA to streamline the conversion of general control flow into thread-level data-flow.

2.3.3 Decoupled software pipelining

Closely related to our work, and in particular to our analysis framework, is the *decoupled software pipelining* (DSWP) technique Ottoni *et al.* [2005]. It partitions loops into long-running threads that communicate via inter-core queues, following the execution model of Kahn process networks Kahn [1974]. DSWP builds a Program Dependence Graph (PDG) Ferrante *et al.* [1987], combining control and data dependences (scalar and memory). In contrast to DOALL and DOACROSS Cytron [1986] methods which partition the iteration space into threads, DSWP partitions the loop body into several stages connected with pipelining to achieve parallelism. It exposes parallelism in cases where DOACROSS is limited by loop-carried dependences on the critical path, it handles uncounted loops, complex control flow and irregular pointer-based memory accesses. Parallel-

Stage Decoupled Software Pipelining (PS-DSWP) Raman *et al.* [2008] is an extension to combine pipeline parallelism with some stages executed in a DOALL, data-parallel fashion. For example, when there are no dependences between loop iterations of a DSWP stage, the incoming data can be distributed over multiple data-parallel worker threads dedicated to this stage, while the outgoing data can be merged to proceed with downstream pipeline stages.

These techniques have a few caveats however. They offer limited support for decoupling along backward control and data dependences. They provide a complex yet somewhat conservative code generation method to decouple dependences between source and target statements governed by different control flow.

2.3.4 EARTH thread partitioning

Another thread partitioning method closely related to our work is used in EARTH architecture compilation technique Tang *et al.* [1997]. The thread partitioning method operates on the Program Dependence Flow Graph (PDFG). A PDFG graph could be built based on the Program Structure Tree (PST) representation. A PST tree of a program is a hierarchical representation of the control structure of the program (similar to the control dependence graph we used in the thesis). In the PST representation, all the nodes control dependent on the same node belongs to the same region. The PDFG is built upon PST by introducing data dependences inside each basic region. The cross region data dependences is promoted up to the same region level.

Our thread partitioning method presents in Chapter 5 differs in several aspects:

- In the EARTH partitioning method, the promotion of cross region dependences creates extra overhead upon each promotion — the outer regions needs to create continuations to wait for the data passing from the inner region. When the destination region get the data passing from the source, it still needs to pass the data down to the lower region where the actual dependence happens.

  Our method, instead of dependence promotion, we reserve the point to point cross region dependence, passing the consumer's thread information to its

producer by building and analyzing the DataFlow Program Dependence Graph (DF-PDG). A detailed algorithm is described in Chapter 5.2.5.

- Our method operates on the SSA-PDG form. There are two main reasons for relying on a SSA-based representation of the PDG: 1. SSA is formally equivalent to a well-behaved subset of the continuation-passing style model. By making the reaching definitions unique for each use, effectively converting the scalar flow into a functional program. 2. Reducing the complexity of analyzing def-use chains from $O(N^2)$ to $O(N)$, as the number of def-use edges can become very large, sometimes quadratic in the number of nodes.

2.3.5 Formalization of the thread partitioning cost model

Sarkar Sarkar [1989] has formally defined the macro-dataflow model and developed the cost model of the partitioning method targeting this dataflow architecture, shows that the problem of determining the optimal partition, with the smallest execution time is NP-complete in the strong sense. Tang Tang & Gao [1999] Tang *et al.* [1997] has formalized the EARTH dataflow execution model and developed the cost model for this architecture, shows the thread partitioning problem is NP complete.

The TSTAR dataflow architecture is similar to EARTH. The formalization is not our focus in this thesis since it has already been proven. Thus, in this thesis, we provide a simple heuristic algorithm for coarsening the granularity of the dataflow threads.

Chapter 3

Thread Partitioning I: Advances in PS-DSWP

As a preliminary research on thread partitioning, we presents a thread partitioning algorithm in this chapter. The algorithm extends loops distribution with pipelining to Parallel Stage Decouple Software Pipelining (PS-DSWP). By asserting a synchronous concurrency hypothesis, the data and control dependences can be decoupled naturally with only minor changes to existing algorithms that have been proposed for loop distribution and loop fusion.

Section 3.1 presents the related works in PS-DSWP and loop distribution. Section 3.2 presents our observations with extending loop distribution to PS-DSWP and a motivating example. Section 3.3 presents our thread partitioning algorithms based on SSA and treegion representation. After deciding the partition point between the original treegions, Section 3.4 explains the code generation challenges in presence of multiple producers and consumers.

3.1 Introduction

The most closely related work to this chapter is decoupled software pipelining and loop distribution. We recall the state-of-the-art in both and present the original finding at the source of this work: *by extending loop distribution with pipelining and asserting a synchronous concurrency hypothesis, arbitrary data and control*

dependences can be decoupled very naturally with only minor changes to existing algorithms that have been proposed for loop distribution Kennedy & McKinley [1990].

3.1.1 Decoupled software pipelining

Decoupled Software Pipelining (DSWP) Ottoni *et al.* [2005] is one approach to automatically extract threads from loops. It partitions loops into long-running threads that communicate via inter-core queues. DSWP builds a Program Dependence Graph (PDG) Ferrante *et al.* [1987], combining control and data dependences (scalar and memory). Then DSWP introduces a load-balancing heuristic to partition the graph according to the number of cores, making sure no recurrence spans across multiple partitions. In contrast to DOALL and DOACROSS Cytron [1986] methods which partition the iteration space into threads, DSWP partitions the loop body into several stages connected with pipelining to achieve parallelism. It exposes parallelism in cases where DOACROSS is limited by loop-carried dependences on the critical path. And generally speaking, DSWP partitioning algorithms handles uncounted loops, complex control flow and irregular pointer-based memory accesses.

Parallel-Stage Decoupled Software Pipelining Raman *et al.* [2008] (PS-DSWP) is an extension to combine pipeline parallelism with some stages executed in a DOALL, data-parallel fashion. For example, when there are no dependences between loop iterations of a DSWP stage, the incoming data can be distributed over multiple data-parallel worker threads dedicated to this stage, while the outgoing data can be merged to proceed with downstream pipeline stages.

These techniques have a few caveats however. They offer limited support for decoupling along backward control and data dependences. They provide a complex code generation method to decouple dependences among source and target statements governed by different control flow, but despite its complexity, this method remains somewhat conservative.

By building the PDG, DSWP also incurs a higher algorithmic complexity than typical SSA-based optimizations. Indeed, although traditional loop pipelining for ILP focuses on innermost loops of limited size, DSWP is aimed at processing large

control flow graphs after aggressive inter-procedural analysis optimization. In addition, the loops in DSWP are handled by the standard algorithm as ordinary control flow, missing potential benefits of treating them as a special case. To address these caveats, we turned our analysis to the state of the art in loop distribution.

3.1.2 Loop distribution

Loop distribution is a fundamental transformation in program restructuring systems designed to extract data parallelism for vector or SIMD architectures Kennedy & McKinley [1990].

In its simplest form, loop distribution consists of breaking up a single loop into two or more consecutive loops. When aligning loop distribution to the strongly connected components of the data-dependence graph, one or more of the resulting loops expose iterations that can be run in parallel, exposing data parallelism. Barriers are inserted after the parallel loops to enforce precedence constraints with the rest of the program. An example is presented in Figure 3.1.

```
for (i = 1; i < N; i++) {
S1  A[i] = B[i] + 1;
S2  C[i] = A[i-1] + 1;
}
```

```
for (i = 1; i < N; i++)
S1  A[i] = B[i] + 1;

<barriers inserted here>

for (i = 1; i < N; i++)
S2  C[i] = A[i-1] + 1
```

Figure 3.1: Barriers inserted after loop distribution.

3.2 Observations

It is quite intuitive that the typical synchronization barriers in between distributed data-parallel loops can be weakened, resulting into data-parallel pipelines. We aim to provide a comprehensive treatment of this transformation, generalizing PS-DSWP in the process.

3.2.1 Replacing loops and barriers with a task pipeline

In the previous example, we could remove the barriers between two distributed loops with pipelining so that the two loops could run in parallel.

```
/* Initialize the stream,
   inserting a delay. */
void INIT_STREAM() {
    produce(stream, A[0]);
}
```

```
/* Decoupled producer and
   consumer. */
for (i = 1; i < N; i++) {
S1  A[i] = B[i] + 1;
    produce(stream, A[i]);
}
for (i = 1; i < N; i++) {
    tmp = consume(stream);
S2  C[i] = tmp + 1;
}
```

Figure 3.2: Pipelining inserted between distributed loops. Initialize the stream (left), producer and consumer thread (right).

Figure3.2 shows that pipelined execution is possible: the `INIT_STREAM` function inserts one delay into a communication stream; the `produce`/`consume` primitives implement a FIFO, enforcing the precedence constraint of the data dependence on array `A` and communicating the value in case the hardware needs this information.

When distributing loops, scalar and array expansion (privatization) is generally required to eliminate memory-based dependences. The conversion to a task pipeline avoids this complication through the usage of communication streams. This transformation can be seen as an optimized version of scalar/array expansion in bounded memory and with improved locality Pop *et al.* [2009].

3.2.2 Extending loop distribution to PS-DSWP

The similarity between DSWP and distributed loops with data-parallel pipelines is striking. First, both of them partition the loop into multiple threads. Second, both of them avoid partitioning the loop iteration space: they partition the instructions of the loop body instead. But four arguments push in favor of refining DSWP in terms of loop distribution.

1. Loop distribution leverages the natural loop structure, where the granularity of thread partitioning can be easily controlled. Moreover, it is useful to have a loop control node to which to attach information about the iteration of the loop, including closed forms of induction variables; this node can also be used to represent the loop in additional transformations.

2. Using a combination of loop distribution and fusion, then replacing barriers with pipelining leads to an incremental path in compiler construction. This path leverages existing intermediate representations and loop nest optimizers, while DSWP relies on new algorithms and a program dependence graph.

3. Considering the handling of control dependences, a robust and general algorithm already exists for loop distribution. McKinley and Kennedy's technique handles arbitrary control flow Kennedy & McKinley [1990] and provides a comprehensive solution. The same methods could be applied for DSWP, transforming control dependences into data dependences, and storing boolean predicates into stream. After restructuring the code, updating the control dependence graph and data dependence graph, the code generation algorithm for PDGs Baxter & Bauer [1989]; Ferrante & Mace [1985]; Ferrante *et al.* [1988] can be used to generate parallel code. This solution would handle all cases where the current DSWP algorithm fails to clone a control condition.

4. Since loop distribution does not partition the iteration space, it can also be applied to uncounted loops. Unfortunately, the termination condition needs to be propagated to downstream loops. This problem disappears through the usage of a conventional communication stream when building task pipelines.

From this high-level analysis, it appears possible to extend loop distribution with pipelining to implement PS-DSWP and handle arbitrary control dependences. Yet the method still seems rather complex, especially the if-conversion of control dependences and the code generation step from the PDG. We go one step

further and propose a new algorithm adapted from loop distribution but avoiding these complexities.

3.2.3 Motivating example

Our method makes one more assumption to reduce complexity and limit risks of overhead. It amounts to enforcing the synchronous hypothesis on all communicating tasks in the partition Halbwachs *et al.* [1991]. A sufficient condition is to check if the source and target of any decoupled dependence is dependent on the same control node.

Consider the example in Figure 3.3. S1 and S7 implement the loop control condition and induction variable, respectively. S2, S3 and S6 are control dependent on S1. S3 is a conditional node, S4, S5 and L1 are control dependent on it. In the inner loop, L2 and L3 are control dependent on L1. When we apply DSWP to the outer loop, the control dependences originating from S1 must be if-converted by creating several streams (the number of streams depends on the number of partitions). When decoupling along the control dependence originating from S3, a copy of the conditional node must be created as well as another stream.

```
S1  while (p != NULL) {
S2    x = p->value;
S3    if(c1) {
S4      x = p->value/2;
S5      ip = p->inner_loop;
L1      while (ip) {
L2        do_something(ip);
L3        ip = ip->next;
        }
      }
S6    ... = x;
S7    p = p->next;
    }
```

Figure 3.3: Uncounted nested loop before partitioning.

Figure 3.4 shows the conversion to SSA form. Just like GCC, we use a loop-closed SSA form distinguishing between loop-Φ and cond-Φ nodes. The latter

```
S1  while (p1 = Φ^loop(p0,p2)) {
S2    x1 = p1->value;
S3    if(c1) {
S4      x2 = p1->value/2;
S5      ip1 = p1->inner_loop;
L1      while (ip2 = Φ^loop(ip1, ip3)) {
L2        do_something(ip2);
L3        ip3 = ip2->next;
        }
      }
      x3 = Φ^cond_c1(x1, x2);
S6    ... = x3;
S7    p2 = p1->next;
    }
```

Figure 3.4: Uncounted nested loop in SSA form.

take an additional condition argument, appearing as a subscript, to explicit the selection condition. The partitioning technique will build a stream to communicate this condition from its definition site to the cond-Φ node's task.

We build on the concept of *treegion*, a single-entry multiple-exit control-flow region induced by a sub-tree of the control dependence graph. In the following, we assume the control flow is structured, which guarantees that the control dependence graph forms a tree. Every sub-tree can be partitioned into concurrent tasks according to the control dependences originating from its root. Any data dependence connecting a pair of such tasks induces communication over a dedicated stream. We call `taskM_N` the `N`-th task at level `M` of the control flow tree.

In Figure 3.4, after building the control dependence tree, one may partition it into 3 tasks (`task1_1`, `task1_2` and `task1_3`) at the root level, and for `task1_2`, one may further partition this task into inner nested tasks `task2_1` and `task2_2`. One may then check for data parallelism in the inner loops; if they do not carry any dependence, one may isolate them in additional data-parallel tasks, such as `task3_1` in this example.

Figure 3.5 shows the task and stream-annotated code using an OpenMP syntax. Figure 3.6 shows the nested pipelining and data parallelization corresponding

```
//task0-0(main task)
S1  while (p1 = Φ^loop(p0, p2)) {
//persistent-task1-1
#pragma task firstprivate (p1) output(x1)
      {
S2      x1 = p1->value;
      }
//persistent-task1-2
#pragma task output(c1, x2)
      {
S3      if(c1) {
//persistent-task2-1
#pragma task firstprivate (p1) output(ip1) lastprivate(x2)
          {
S4          x2 = p1->value/2;
S5          ip1 = p1->inner_loop;
          }
//persistent-task2-2
#pragma task input(ip1)
          {
L1          while (ip2 = Φ^loop(ip1, ip3)) {
//parallel - task3-1
#pragma omp task firstprivate (ip2)
              {
L2              do_something(ip2);
              }
L3            ip3 = ip2->next;
            }
          }
        }
      }
//persistent-task1-3
#pragma task input(c1, x1, x2)
      {
        x3 = Φ^cond_c1(x1, x2);
S6      ... = x3;
      }
S7    p2 = p1->next;
    }
```

Figure 3.5: Loops after partitioning and annotated with OpenMP stream extension.

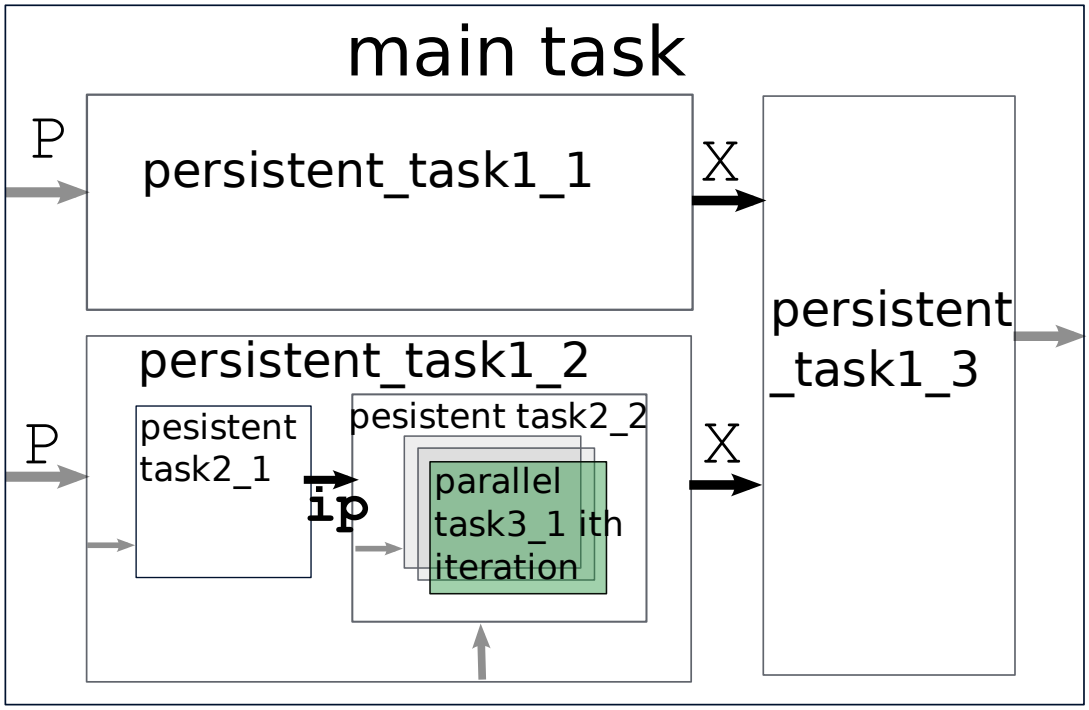

Figure 3.6: Pipelining and parallelization framework.

to the partitioned code. The main task will be executed first, and a pipeline will be created for the main task and its inner tasks three `task1_1`, `task1_2` and `task1_3`. Among these, the same variable `x` used to be defined in the control flow regions of both `task1_1` and `task1_2`, to be used in `task1_3`. This output dependence must be eliminated prior to partitioning into tasks, so that `task1_1` and `task1_2` could be decoupled, while `task1_3` may decide which value to use internally.

Nested tasks are introduced to provide fine grained parallelism. It is of course possible to adapt the partition and the number of nesting levels according to the load balancing and synchronization overhead. The generated code will be well structured, and simple top-down heuristics can be used.

In the execution model of OpenMP 3.0, a task instance is created whenever the execution flow of a thread encounters a task construct; no ordering of tasks can be assumed. Such an execution model is well suited for unbalanced loads, but the overhead of creating tasks is significantly more expensive than synchronizing persistent tasks. To improve performance, we use the persistent task model for

pipelining, in which a single instance will handle the full iteration space, consuming data on the input stream and producing on the output stream Pop & Cohen [2011a]. In Figure 3.6, all the tasks except `task3_1` use the persistent model to reduce the overhead of task creation; `task3_1` is an ordinary task following the execution model of OpenMP 3.0 (instances will be spawned every time the control flow encounters the task directive). All these tasks will be scheduled by the OpenMP runtime.

One problem with the partitioning algorithms is the fact that the def-use edges (scalar dependences) can become very large, sometimes quadratic with respect to the number of nodes Kennedy & Allen [2002]. Figure 3.7 (left) presents an example that illustrates this problem, Statements S1, S2 define the variable `x`. These definitions all reach the uses in the statements S3, S4 by passing through S5. Because each definition could reach every use, the number of definition-use edges is proportional to the square of the number of statements. These dependences constitute the majority of the edges in a PDG. SSA provide a solution to this problem. In SSA form, each assignment creates a different variable name and at point where control flow joins, a special operation is inserted to merge different incarnations of the same variable. The merge nodes are inserted just at the place where control flow joins. Figure 3.7 (right) is the original program under SSA form. A merge node (Φ) is inserted at S5, and killed the definition of S1 and S2. We could see here, in the SSA form, we could reduce the definition-use edges from quadratic to linear.

The systematic elimination of output dependences is also facilitated by the SSA form, with a Φ node in `task3_1`. Notice that the conditional expression from which this Φ node selects one or another input also needs to be communicated through a data stream.

When modifying loop distribution to rely on tasks and pipelining rather than barriers, it is not necessary to distribute the loop control node and one may run it all in the master task, which in turn will activate tasks for the inner partitions. The statements inside each partition form a treegion whose root is the statement that is dependent on the loop control node. With pipelining inserted, distributed loops could be connected with pipelining when there are data dependences.

One concern here is that loop distribution with task pipelines may not pro-

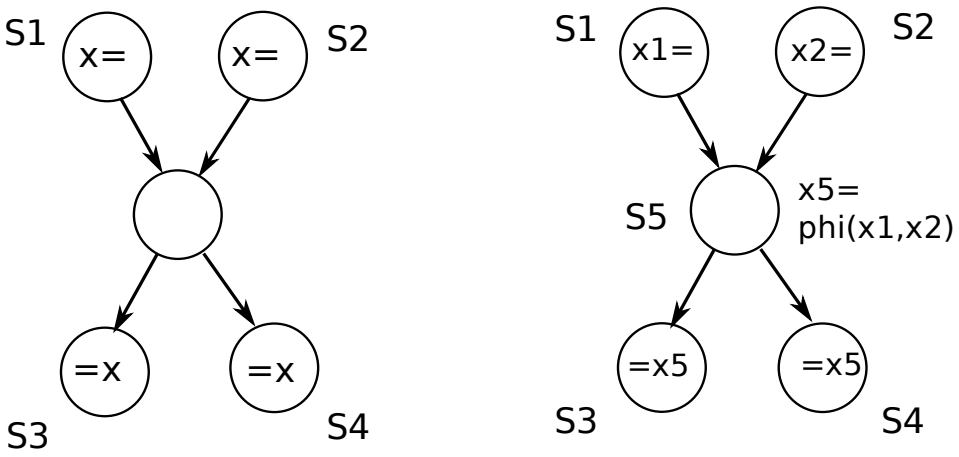

Figure 3.7: Definition and use edges in the presence of control flow.

vide expressiveness to extract pipeline parallelism. This is not a problem however, since we may apply the same method to every conditional statement rooted treegion, with some special care to the nested tasks, we could get fine grained parallelism without explicitly decoupling the control dependences. Considering again the example in Figure 3.3, its control dependence tree is given in Figure 3.8. The root treegion includes all the nodes in the control dependence graph, `treegion1_2` represents the treegion at conditional level 1 and its root is node 2, `treegion1_3` is at conditional level 1 and includes nodes (S3,S4,S5,L1,L2,L3). `treegion2_1` is in conditional level 2 and its root is node (L1), which is the loop control node of the inner loop.

So following our approach, we may start from the treegion at conditional level 0, which is the whole loop, an implicit task will be created as the master task. For the treegions at level 1, we could create them as sub-tasks running at the context of the main task. If there are data dependences between the treegions at the same level and without recurrence, we will connect them with communication streams. If there is a dependence from the master task to one inner task, the value from the enclosing context can be forwarded to the inner task like in a `firstprivate` clause of OpenMP. Dependences from an inner task to the master task are also supported, although `lastprivate` is not natively sup-

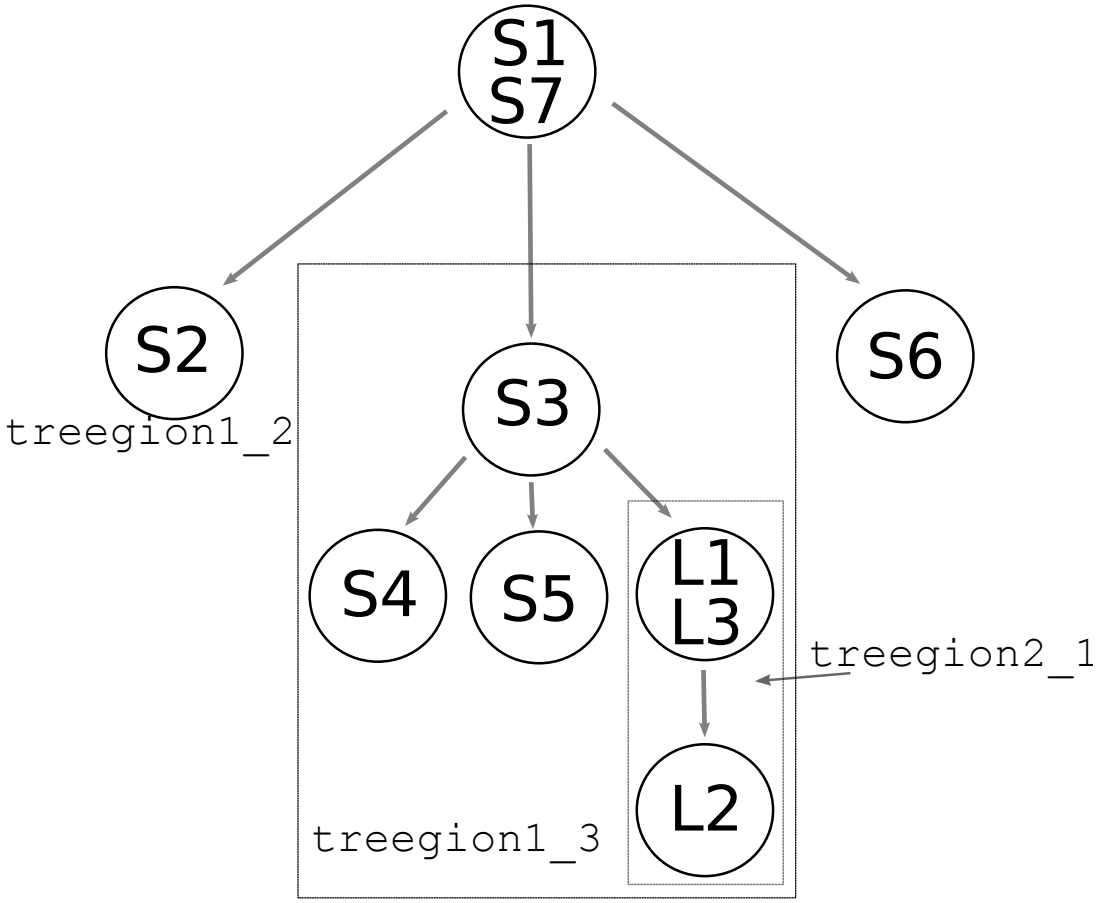

Figure 3.8: Control dependence graph of Figure 3.3. Express the definition of treegion.

ported for OpenMP3.0 tasks, it is a necessary component of our streaming task representation. `lastprivate(x)` is associated with a synchronization point at the end of the task and makes the value of `x` available to the enclosing context. The same algorithms could be recursively applied to the treegion at the next inner level. e.g. For `treegion1_3` at level 1, the sub treegion at level 2 is `treegion2_4`, `treegion2_5` and `treegion2_1`, we could create sub-tasks by merging `treegion2_4` and `treegion2_5` as one sub-task and `treegion2_1` (which is also the inner loop) as one sub-task, or just for part of them. To reveal data parallelism, we can reuse the typed fusion algorithm introduced by McKinley

and Kennedy Kennedy & Mckinley [1993]: it is possible to fuse communicating data-parallel nodes to increase the synchronization grain or improve the load balancing. In this example, the loop in node L2 does not carry any dependence, and we need to decouple it from its enclosing task to expose data-parallelism.

3.3 Partitioning Algorithm

In this section, we present our partitioning algorithm, based on the SSA and treegion representations. We define our model and the important constructs that will be used by our algorithm, then we present and describe our algorithm.

3.3.1 Definitions

In this work, we are only targeting natural structured loops Böhm & Jacopini [1966]. Such loops are single-entry single-exit CFG sub-graphs with one entry block and possibly several back edges leading to the header from inside of the loop. `break` and `continue` statements can be preprocessed to comply with this restriction, but we plan to lift it altogether in the future.

Treegion The canonical definition of a treegion is a non-linear, single-entry multiple-exit region of code containing basic blocks that constitute a sub-graph of the CFG. We alter this definition to bear on the Control-Dependence Graph (CDG) instead, so we will be looking at single-entry multiple-exit sub-graphs of the CDG.

Loop Control Node In the representation we employ later, we will use the loop control node to represent the loop. The loop control node include statements which will evaluate the loop control expression and determines the next iteration.

Although control dependences in loops can be handled by the standard algorithm by converting them to a control flow graph, there are advantages in treating them as a special case with coalescing them in a single node (loop control node): not only the backward dependence is removed by building the loop control node

so that the control dependence graph will form a tree, but also, this node can be used to represent the loop in all sort of transformations.

Conditional Level The control dependence graph of the structured code is a tree after building the loop control node. The root of the tree is the loop control node at the loop's outermost level. We define the conditional level for every node in the control dependence graph as the depth of the node in the tree. The root of the tree with depth 0 has conditional level 0.

We define the conditional level for the treegion is the conditional level of the root node of the treegion (subtree). We define `treegionN_M` to identify a treegion where `N` is the conditional level of the treegion and `M` is the root node number of the treegion.

3.3.2 The algorithm

The algorithm takes an SSA representation of a single function, and returns a concurrent representation annotated with tasks and communication streams.

Step 1: Transform Conditional Statements to Conditional Variables To achieve fine-grained pipelining, conditional statements are split to conditional variables. As showed in Figure 3.9. Full conversion to three-address SSA form is also possible (as it is performed in GCC or LLVM, for example).

```
if (condition(i))
//is transformed to
c1 = condition(i)
if (c1)
```

Figure 3.9: Split conditional statements to expose finer grained pipelining.

Step 2: Build the Program Dependence Graph under SSA By building the program dependence graph, the control dependence graph, data dependence graph (through memory) and scalar dependence graph (through registers) are built together.

The control dependence graph for the structured code is a tree, the root of the tree is the loop control node. The leaves of the tree are non-conditional statements and the other nodes inside the tree are the conditional statements or the loop control node of the inner loops. We start from building the control dependence graph, and evaluate the conditional level for each node in the graph. Every node inside the control dependence graph is an statement from the compiler's intermediate representation of the loop except for the loop control node. The loop control node will be built by searching the strongly connect component started from the loop header node (at each loop nest level) in the program dependence graph.

The data dependence graph could be built by the array dependence analysis Kennedy & Allen [2002] for the loop. We should analyze every pair of data dependences to mark the irreducible edges in a later step if there are recurrence.

Step 3: Marking the Irreducible Edges A partition can preserve all dependences if and only if there exists no dependence cycle spanning more than one output loop Allen & Kennedy [1987]; Kuck *et al.* [1981]. In our case, for the treegion at the same conditional level which shares a common immediate parent, if there are dependences that form a cycle, we mark the edges in between as irreducible. If we have statements in different conditional level or does not share a common immediate parent, we find the least common ancestor of both nodes, and walk backwards along the CDG, till reach the immediate nodes of the least common ancestor, and mark the edge in between as irreducible. The algorithm is presented in Figure 3.10.

Step 4: Structured Typed Fusion Before partitioning, to reveal data parallelism, we type every node in the dependences graph as `parallel` or `!parallel`. If there are loop-carried dependence inside this node, then it should be typed as `!parallel`, otherwise, typed as `parallel`.

The `parallel` type nodes are candidates for data parallelization. The goal is to merge this type of nodes to create the largest parallel loop, reducing synchronization overhead and (generally) improving data locality. Further partitioning can happen in the following step, starting from this maximally type-fused con-

```
// input: PDG Graph PDG(V,E) PDG--Program Dependence Graph
// input: CDG Graph CDG(V,E) CDG--Control Dependence Graph
// output: irreducible_edge_set Irreducible_edge_set
SCCS = find_SCCs(PDG)
For each SCC in SCCs:
  for each pair of node (Vx,Vy) in SCC:
    // CL represents for conditional level
    // in the Control dependence graph.

    // Find the least common ancestor of both nodes, and walk
    // backwards along the CDG, till reach the immediate nodes of the
    // least common ancestor, and mark the edges in between as
    // irreducible.
    Vca = get_least_common_ancestor (Vx, Vy)
    Vx = up_n_level(CDG, Vx.CL - Vca.CL - 1)
    Vy = up_n_level(CDG, Vy.CL - Vca.CL - 1)
    //mark edge (Vx,Vy) irreducible
    Irreducible_edge_set.insert(edge(Vx,Vy))
```

Figure 3.10: Algorithm for marking the irreducible edges.

figuration. Given a DAG with edges representing dependences and the vertexes representing statements in the loop body, we want to produce an equivalent program with minimal number of parallel loops. We want it to be as large as possible to balance the synchronization overhead. Even when we do not want that coarse grained parallel loops, we could also partition between iterations if possible.

In our case, we need a structured typed loop fusion algorithm. We revisit McKinley and Kennedy's *fast typed fusion* Kennedy & Mckinley [1993] into a recursive algorithm traversing the control dependence tree. Starting from the treegion at conditional level 0, which is the whole loop, we will check if there are loop carried dependences between iterations. If there are no loop carried dependence, we stop here by annotating the whole loop as parallel. If there are, we are going into each inner treegion, identifying those that have no loop carried dependences. If some of them carried no loop carried dependence, mark the nodes as parallel and try to merge them. There are some constraints when we fuse the nodes: (1) parallelization-inhibiting constraints; (2) ordering constraints. The parallelization-inhibiting fusion is that there are no loop-carried dependences before fusion, but will have the loop carried dependences after. So we should skip

this kind of fusion which will degrades data parallelism. The ordering constraints describe that two loops cannot be validly fused if there exists a path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.

The time complexity of the typed fusion algorithms is `O(E+V)` Kennedy & Mckinley [1993], and our structured extension has the same complexity.

```
void StructuredTypedFusion()
  Queue queue = new Queue()
  queue.push(treegions_at_level_0)
  while (not queue.empty()) {
    treegions = queue.pop()
    G = build_pdg_by_treegion(treegions)
    for each treegion in treegions:
      if loop_carried_no_dependence (treegion) {
        parallel_treegion.insert(treegion)
        update_typed_dependence_graph(G, treegion.num)
      }else{
        treegions_at_inner_level.insert(treegion)
      }
    queue.push (treegions_at_inner_level)
    B = Get_parallelization_inhabiting_edges
                  (parallel_treegion)
    t0 = 'parallel'
    TypedFusion (G, T, B, t0)
  }
procedure TypedFusion(G, T, B, t0)
  //G=(V,E) is the TYPED dependences graph,
  //including control,data,scalar dependences.
  //type(n) will return the type of a node.
  //B is the set of parallelization-inhabiting edges.
  //t0 is a specific type for which we will find a minimal fusion
end TypedFusion
```

Figure 3.11: Structured typed fusion algorithm.

Step 5: Structured Partitioning Algorithms Updating the CDG after typed fusion, start from the treegion which has conditional level 0 for our partitioning algorithms, and for all of its child treegions at conditional level 1, we should decide where to partition. The partition point could be any point between

each of these treegions at the same level except the irreducible edges that we have created in step 3. The algorithm may decide at every step if it is desirable to further partition any given task into several sub-tasks. Note: we did not provide any heuristic for making the partitioning decision (coarsen the granularity etc.). A detailed partitioning algorithm is presented in Chapter 5 where a simple heuristic algorithm for coarsen the granularity is applied.

Look at the example Figure 3.12:

```
for(i...)

  x =  work(i)

  if (c1)
    y = x + i;

    if (c2)
      z = y*y;

    q = z - y;
```

```
for (i...)
  BEGIN task1_1
    x =  work(i)
  END task1_1
  BEGIN task1_2
    if (c1)
      BEGIN task2_1
        y = x + i;
      END task2_1
      BEGIN task2_2
        if (c2)
          z = y*y;
      END task2_2
      BEGIN task2_3
        q = z - y;
      END task2_3
  END task1_2
```

Figure 3.12: Before partitioning (left), and After partitioning (right). Loop with control dependences.

The code in Figure 3.12 (left) is partitioned into 2 tasks, and one task (`task1_2`) is partitioned further into 3 sub-tasks.

3.4 Code Generation

After the partitioning algorithms, we have decided the partition point between the original treegions, with the support of the stream extension of OpenMP. We ought to generate the code by inserting the `input output` directives. With the support of nested tasks, relying on the downstream, extended OpenMP compilation algorithm (called OpenMP expansion). But some challenges remain,

especially in presence of multiple producers and consumers. We are using SSA form as an intermediate representation and generating the streaming code.

3.4.1 Decoupling dependences across tasks belonging to different treegions

Clearly if we decouple a dependence between tasks in the same treegion, the appropriate `input` and `output` clauses can be naturally inserted. But what about the communication between tasks at different level?

Considering the example in Figure 3.13, if we decide to partition the loop to 3 main tasks: task1_1 with S1, task1_2 with (S2,S3), and task1_3 with S4, task1_2 is further divided to task2_1 with S3. If we insert the produce and consume directly into the loop, unmatched production and consumption will result.

```
for (...) {
S1  x =  work(i)

S2  if (c1)
S3    y = x + i;
S4  ... = y;
}
```

```
for (i = 0; i < N; I++) {
S1  x =  work(i)
    produce(stream_x, x)    //task1_1 end

    x = consume(stream_x)
S2  if (c1)                 //task1_2 start
      x = consume(?)        //task2_1 start
S3    y = x + i;
      produce(?, y)         //task2_1 end
    produce(stream_y, y)    //task1_2 end
    y = consume(stream_y)
S4  ... = y;                //task1_3 end
}
```

Figure 3.13: Normal form of code (left) and using streams (right).

The answer comes from following the synchronous hypothesis and slightly modifying the construction of the SSA form in presence of concurrent streaming tasks.

3.4.2 SSA representation

We are using the Single Static Assignment (SSA) form as an intermediate representation for the source code. A program in SSA form if every variable used

in the program appears a single time in the left hand side of an assignment. We are using the SSA form to eliminate the output dependences in the code, and to disambiguate the flow of data across tasks over multiple producer configurations.

```
/* Normal form of the code. */
S1:  r1 = ...
S2:  if (condition)
S3:    r1 = ...
S4:  ... = r1
```

```
/* Code under SSA form. */
S1:  r1_1 = ...
S2:  if (condition)
S3:    r1_2 = ...
S4:  r1_3 = phi(r1_1, r1_2)
S5:  ... = r1_3
```

Figure 3.14: Normal form of code (left) and SSA form of the code (right).

Considering the example in Figure 3.14, if we partition the statements into (S1), (S2,S3), (S4), we need to implement precedence constraints for the output dependence between partition (S1) and (S2,S3), which decreases the degree of parallelism and induces synchronization overhead.

Eliminating the output dependences with the SSA form leads to the introduction of multiple streams in the partitioned code. In order to merge the information coming from different control flow branches, a Φ node is introduced in the SSA form. The Φ function is not normally implemented directly, after the optimizations are completed the SSA representation will be transformed back to ordinary one with additional copies inserted at incoming edges of (some) Φ functions. We need to handle the case where multiple producers in a given partition reach a single consumer in a different partition. When decoupling a dependence whose sink is a Φ node, the exact conditional control flow leading to the Φ node is not accessible for the out-of-SSA algorithm to generate ordinary code.

Task-closed Φ node In SSA loop optimization, there is a concept called loop-closed Φ node, which implements the additional property that no SSA name is used outside of loop where it is defined. When enforcing this property, Φ nodes must be inserted at the loop exit node to catch the variables that will be used outside of the loop. Here we give a similar definition for *task-closed* Φ node: if multiple SSA variables are defined in one partition and used in another, a phi node will be created at the end of the partition for this variable. This is the place where we join/split the stream. We need to make sure that different definitions of

the variable will be merged in this partition before it continues to a downstream one. This node will be removed when converting back from SSA.

Task-closed stream Our partitioning algorithms generate nested pipelining code to guarantee that all communications follow the synchronous hypothesis. For each boundary, if there are one or more definitions of a variable coming through from different partitions, we insert a consumer at this boundary to merge the incoming data, and immediately insert a producer to forward the merged data at the rate of the downstream control flow.

1. When partitioning from a boundary, if inside the treegion, there are multiple definitions of a scalar and it will be used in other treegions which has the same conditional level, we create a Φ node at the end of this partition to merge all the definitions, and also update the SSA variable in later partitions.

2. If there is a Φ node at the end of a partition, insert a stream named with the left-hand side variable of the Φ node.

3. At the place where this variable is used, which is also a Φ node, add a special stream-Φ node to consume.

4. To generate code for the stream-Φ, use the boolean condition associated with the conditional phi node it originates from.

Let us consider the SSA-form example in Figure 3.14 where we partition the code into (S1,S2,S3) and (S4,S5). A Φ node will be inserted at the end of the first partition, $r1_4 = phi(r1_1, r1_2)$, the Φ node in a later partition should be updated from $r1_3 = \Phi(r1_1, r1_2)$ to $r1_5 = \Phi(r1_4)$. In the second step, we find out that in partition (S1,S2,S3), there is a Φ node at the end, so we insert a stream to produce there. And in partition (S4,S5), after the Φ node there is a use of the variable, so we insert a stream consume. The generated code will look like Figure 3.15.

This example illustrates the generality of our method and shows how fine-grain pipelines can be built in presence of complex, multi-level control flow.

```
/* Producer. */
S1: r1_1 = ...
S2: if (condition)
S3:   r1_2 = ...
    r1_4 = phi(r1_1, r1_2)
    produce(stream_r1_4, r1_4)
```

```
/* Consumer. */
S4: r1_5 = phi(r1_4)
    r1_5 = consume(stream_r1_4, i)
S5: ... = r1_5
```

Figure 3.15: Apply our algorithm to generate the parallel code. Producer thread (left) and consumer thread (right).

If we decide to partition the statements into (S1), (S2,S3), (S4,S5), which is the case for multiple producers, the generated code will look like in Figure 3.16.

```
/* Producer 1. */
S1: r1_1 = ...
    r1_2 = phi(r1_1)
    produce(stream_r1_2, r1_2)

/* Producer 2. */
S2: if (condition)
      r1_3 = ...
    r1_4 = phi(r1_3)
    produce(stream_r1_4, r1_4)
```

```
/* Consumer. */
S4: r1_5 = phi(r1_2, r1_4)
    if (condition)
      r1_5 = consume(stream_r1_4, i)
    else
      r1_5 = consume(stream_r1_2, i)
S5: ... = r1_5
    i++
```

Figure 3.16: Multiple producers with applied our algorithm, the generated code.

For multiple consumers, the stream extension of OpenMP will broadcast to its consumers, which is appropriate for our case.

3.5 Summary

In this chapter, we propose a method to decouple independent tasks in serial programs, to extract scalable pipelining and data-parallelism. Our method leverages a recent proposition of a stream-processing extension of OpenMP, with a persistent task semantics to eliminate the overhead of scheduling task instances each time a pair of tasks need to communicate. Our method is inspired by the synchronous hypothesis: communicating concurrent tasks share the same control flow. This hypothesis simplifies the coordination of communicating tasks over nested levels of parallelism. Synchronous also facilitates the definition of gen-

eralized, structured typed fusion and partition algorithms preserving the loop structure information. These algorithms have been proven to be essential to the adaptation of the grain of parallelism to the target and to the effectiveness of compile-time load balancing.

These partitioning algorithms also handle DOALL parallelization inside a task pipeline. We are using a combination of SSA, control dependence tree and (non-scalar) dependence graph as an IR. With the support of SSA, our method eliminates the nested multiple producer and multiple consumer problems of PS-DSWP. SSA also provides additional applicability, elegance and complexity benefits.

persistent task model. The persistent task model benefits from eliminating the overhead of scheduling task instances, however, it reduces the potential parallelism in recurrent control structure.

synchronous hypothesis. The structured partitioning algorithm assumes a synchronous hypothesis, which simplifies the coordination of communicating tasks over nested levels of parallelism. However, this hypothesis results in too coarsen grained tasks sometimes (e.g. when marking the irreducible edges, if we have statements in different conditional level, we promote the inner to its ancestor until both of them are in the same treegion, and mark both root nodes as irreducible).

Conclusion. The partitioning algorithm presents in this chapter serves a preliminary study, the persistent task model and synchronous hypothesis restrict the exploration of parallelism. Enough parallelism is the key to cover latency. So in chapter 5, we present a more mature thread partitioning algorithm by removing the synchronous restricts which target on TSTAR dataflow architecture with a non-preemptive thread model, where we could exploit maximum parallelism. The TSTAR architecture and its detailed execution model is presented in chapter 4.

Chapter 4

TSTAR Dataflow Architecture

In this chapter, we present several abstraction layers of the TSTAR dataflow architecture, from the threading model to the dataflow ISA extensions. Section 4.1 gives an overview of the dataflow execution model and past dataflow architectures. Section 4.2 further explains the TSTAR execution model from different aspects (threading model, memory model ,synchronization details and ISA extension), and section 4.3 gives an overview of the TSTAR architecture.

4.1 Dataflow Execution Model

4.1.1 Introduction

In the data flow execution model, a program is represented by a directed graph Arvind & Culler [1986a]; Dennis *et al.* [1974]; Dennis & Misunas [1979]; Karp & Miller [1966]. The nodes of the graph are primitive instructions such as arithmetic or comparison operations. Directed arcs between the nodes represents the data dependencies between the instructions.

Execution of a test or operation is enabled by availability of the required value. The completion of one operation makes the resulting value or decision available to the elements of the program whose execution depends on them.

When the program begins, special activation nodes place data onto certain key input arcs, triggering the rest of the program. Whenever a specific set of input arcs of a node (called a firing set) has data on it, the node is said to be

fireable Arvind & Culler [1986a]; Veen [1986]; Davis & Keller [1982]. A fireable node is executed at some undefined time after it becomes fireable. The action of firing the actor results in removing the data token from each node in the firing set, performs its operation, and places new data token on its output arcs. It then cease the execution and waits to become fireable again.

The execution of a data flow program could be described by a sequence of snapshots. Each snapshot shows the data flow program with tokens and associated values placed on some arcs of the program. Execution of a data flow program advances from one snapshot to the next through the firing of some link or actor that is enabled in the earlier snapshot.

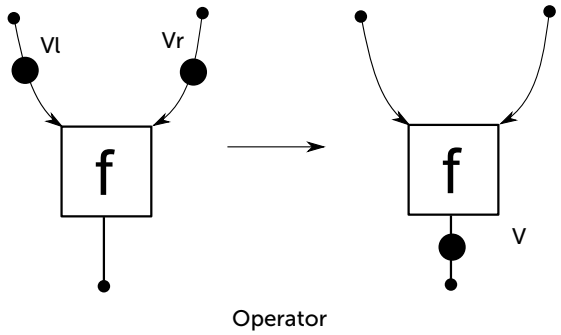

Figure 4.1: Firing rule example.

4.1.2 Past Data-Flow Architectures

The original motivation for dataflow study origins from the exploration of massive parallelism. One school of thoughts held that the conventional architecture is not suitable for the exploitation of parallelism, where the global program counter and global updatable memory becomes bottlenecks. The early study starts from dataflow architectures.

The static architecture. The static dataflow model was proposed by Dennis and Misunas Dennis & Misunas [1975]. In this model, the FIFO design of arcs is replaced by simple design where each arc could hold at most one data token. When there is a token on each input arc, and no token on any of the output arcs,

the node fires. Figure 4.2 (a) shows the general organization of the static data flow machine. The activity store contains instruction templates that represents the nodes in a dataflow graph. Each instruction template contains an operation code, slots for the operands, and destination addresses. To determine the availability of the operands, slots contain presence bits. The update unit is responsible for detecting instructions that are available to execute. When this condition is verified, the unit sends the address of the enabled instruction to the fetch unit via the instruction queue. The fetch unit fetches and sends a complete operation packet containing the corresponding opcode, data and destination list to one of the operation units and clears the presence bits. The operation unit performs the operation, forms result tokens, and sends them to the update unit. The update unit stores each result in the appropriate operand slot and checks the presence bits to determine whether the activity is enabled.

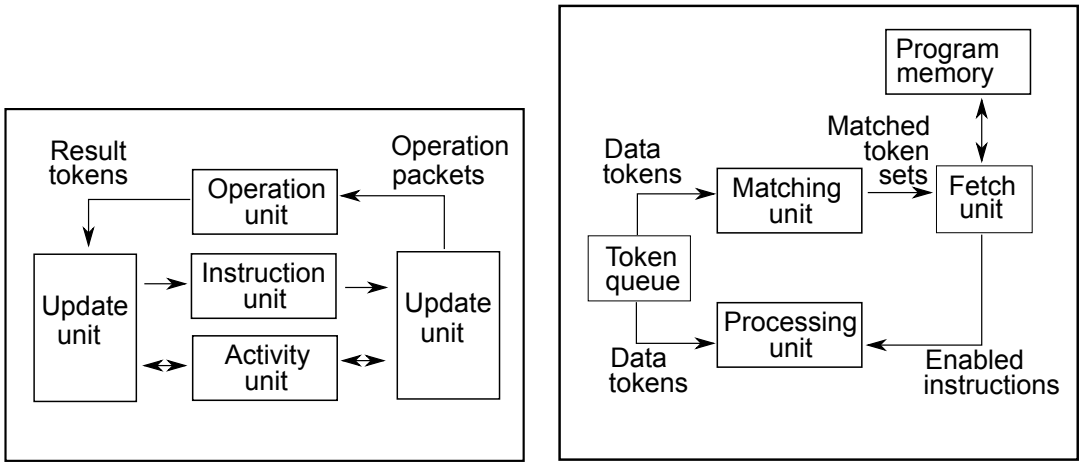

Figure 4.2: The basic organization of the static (a) and dynamic (b) model.

Advantage and Disadvantage. The main advantage of static dataflow model is its simplified mechanism for detecting the availability of required operands (fireable nodes). However, the static dataflow model has a serious drawback: the additional acknowledgement arcs increase data traffic by a factor of 1.5 to 2.0 in the system when dealing with iterative constructs and reentrancy Arvind & Culler [1986a]. The acknowledge scheme can transform a reentrant code into an

equivalent graph that allows pipelined execution of consecutive iterations, but the single token per arc limitation restricts that the execution of consecutive iterations can never fully overlap, even there is no loop carried dependences exist.

The dynamic architecture. The dynamic, or tagged-token architecture was proposed by Watson and Gurd Watson & Gurd [1979], Arvind and Culler Arvind & Culler [1983]. It exposes additional parallelism by allowing multiple instance of the reentrant code. The firing rule is: A node is enabled as soon as tokens with identical tags are present at each of its input arcs. Figure Figure 4.2 (b) show the general organization of the dynamic dataflow model. Tokens are received by the matching unit, which is a memory containing a pool of waiting tokens. The unit's basic operation brings together tokens with identical tags. If a match exists, the corresponding token is extracted from the matching unit, and the matched token set is passed to the fetch unit. If no match is found, the token is stored in the matching unit to await a matched token. In the fetch unit, the tags of the token pair uniquely identify an instruction to be fetched from the program memory.

Advantage and Disadvantage. The main advantage of dynamic dataflow model is its support for iterative constructs and reentrancy, higher parallelism is obtained by allowing multiple tokens on an arc. For example, a loop can be dynamically unfolded at runtime by creating multiple instances of the loop body and allowing concurrent execution of the iterations. It has been studied that this model offers maximum possible parallelism in any interpreter Arvind *et al.* [1977]. The main disadvantage of this model is the extra overhead in matching tags on tokens. Instead of simple presence or absence checking as in static dataflow model, tagged-token dataflow model execution requires a token to be checked against all other waiting tokens for a match. If a match is found then the matching token must be extracted from the waiting storage. The matching function and associated storage is most often realized with a large hash table and sophisticated insertion and extraction algorithms. The resulting hardware is highly complex and slow. A more subtle problem with token matching is the complexity of allocating memory cells. A failure to the matching mechanism implicitly allocates extra memory within the matching hardware. Usually the memory on such hardware is limited, once it is overused, it might leads to dead-

lock.

The hybrid architecture The limit of static dataflow model and the overhead in dynamic dataflow model led to the convergence of dataflow and von Neumann models. The hybrid architecture moves from fine-grained parallelism towards coarse-grained execution model, which combines the power of dataflow model for exploiting parallelism with the efficiency of the control-flow model.

The hybrid architecture diverge in two directions: one approaches are essentially von Neumann architectures with a few dataflow additions (large-grain dataflow architecture), another are essentially dataflow architectures with some von Neumann additions (multithreaded dataflow architecture) Iannucci [1988]; Nikhil [1989]; Nikhil *et al.* [1992] Culler *et al.* [1991].

In *multithreaded dataflow model*, the dataflow principle is adjusted so that a sequential instructions is issued consecutively by the matching unit without matching further token except for the first instruction of the thread. Data passed between instructions from the same thread is stored in register instead of writing them back to memory. In *large-grain dataflow model*, the dataflow graph is analyzed and divided to subgraphs, the subgraphs are then compiled into sequential von Neumann processes. These processes are executed in a multithreaded environment, scheduled according to the dataflow model to better exploit parallelism. Large dataflow machines typically decouple the matching stage from the execution stage by FIFO buffers. In which case off-the-shelf microprocessors can be used for the execution stage.

Large grain dataflow architectures. The large grain dataflow architecture is well studied and developed in the 1990s. *T Nikhil *et al.* [1992] is a direct descendant of dataflow architectures, especially of the Monsoon Papadopoulos & Culler [1990], and unifies them with von Neumann architecture. In the Associative Dataflow Architecture ADARC Strohschneider & Waldschmidt [1994], the processing units are connected via an associative communication network. The Pebble Roh & Najjar [1995] architecture is a large grain dataflow architecture with decoupling of the synchronization unit and the execution unit within the processing elements.

The EARTH Maquelin *et al.* [1996] Hendren *et al.* [1997] architecture is based on the Multithreaded Architecture and dates back to the Argument Fetch Dataflow Processor Dennis & Gao [1988]. An MTA node consists of an execution unit that may be an off-the-shelf RISC microprocessor and a synchronization unit to support dataflow thread synchronization. The synchronization unit determines which threads are ready to be executed. Execution unit and synchronization unit share the processor local memory which is cached. Accessing data in a remote processor unit communicate via FIFO queues: a ready queue containing ready thread identifiers link the synchronization unit with the execution unit, and an event queue holding local and remote synchronization signals connects the execution unit with the synchronization unit, but also receive signals from the network. A register use cache keeps track of which register set is assigned to which function activation. Our threading model is similar to EARTH threading model, both rely on non-preemptive threads.

4.2 TSTAR Dataflow Execution Model

4.2.1 TSTAR Multithreading Model

The basic execution model of TSTAR is a multithreading execution model that exploits application parallelism at different levels. It derives from the dataflow execution model, where the communication and synchronization latencies caused by inter-processor communication can overlap with useful information.

In TSTAR execution model, the instructions are not synchronized and scheduled individually, but are combined into larger units called DF-threads. A DF-thread is a sequence of instructions that execute sequentially and run to completion once started (non-preemptive).

DF-Thread DF-Thread is a non-preemptive dataflow thread, and have the following properties:

- They are activated only when all their input data are available. This data is available either in the DF-frame (Frame Memory) of the thread or in the OWM of the producer threads.

- They write their output to either the consumer threads allocated DF-frames in Frame Memory or to its own OWM (Owner Writable Memory).
- Each thread has a single entry point and a single exit point. Therefore there are
  - No jumps between DF-threads and
  - No jumps within a DF-thread.
- The compiler controls thread granularity for the most efficient execution.
- Data consistency is guaranteed, because it does not allow threads that read-/write the same address simultaneously.

The TSTAR execution model is based on U-Interpreter principles. The thread meta-data, such as producer-consumer relationships and ready count (the number of producers) of each thread are generated at compile time. The compiler also inserts the necessary code to perform the initialization which consists of loading meta-data of the threads into the Thread Scheduling Unit (TSU). The meta-data is stored in the Graph Memory (GM). Each instantiation of a thread is distinguished by its unique context that is maintained at execution time. The dynamic context is combined with static meta-data to uniquely identify each thread Ready Count (RC) entry and its input and output data. The DF-thread is ready to be scheduled to run when all its producers have finished execution and so their data are available. This guarantees that even though the results of a thread are written during the execution of the thread and to global memory space, they are only made "visible" to the consumers when the thread completes its execution. As a result, we can safely restart the thread's execution, in case of a failure.

4.2.2 TSTAR Memory Model

TSTAR memory model is a hybrid memory model. It consists the thread local storage, the owner writable memory and the frame memory. The thread local storage allows better exploit the locality in one single DF-thread. The owner

writable memory is served as a global addressable memory, to reduce the cost of the communication of complex data structures like arrays, vector etc. The frame memory is used to manage the synchronization of dataflow threads.

Thread Local Storage (TLS) Sequential scheduling avoid the overhead of the inefficient communication of tokens among nodes in past dataflow architectures. The TLS allows better exploit the locality in one single DF-thread. TLS is part of the thread's address space. Only the owner thread can read or write to this memory.

Whenever a DF-thread is ready to execute, it reads the input data from its associate Frame Memory (or Owner Writable Memory), and during the sequential execution of the DF-thread, the temporary variables could be stored in TLS, avoid extra overhead.

Owner Writable Memory (OWM) One general criticism leveled at dataflow computing is the management of data structures. The dataflow functionality principle implies that all operations are side-effect free. However, absence of side effect also means that if tokens are allowed to carry vectors, arrays, or other complex data structures, an operation on a data structure results in a new data structure. Which will greatly increase the communication overhead in practice. The problem of efficiently represent and manipulate complex data structures in dataflow execution model remains a challenge.

We proposed the OWM memory model to reduce the communication overheads when complex data structures passed over the threads. OWM is the global addressable memory, before a thread could write to a portion of memory, it has to claim ownership before hand. At any time point only the thread who has the ownership of the memory could write to it. When write ownership is successfully acquired, any read from another thread is not guaranteed to see consistent data. When write ownership is released, a consistent view of data must be visible to any other thread. Note the release operation could be performed explicitly by the thread or implicitly by the model. The latter is achieved when the OWM is used by a thread to write its results, which are made available to the consumer thread upon the completion of the execution of the thread. This memory can serve the

requirements of the single assignment semantics required for functional objects. However, the ability for other threads to subsequently reclaim write ownership adds to flexibility of usage.

Frame Memory (FM) FM is used to manage the synchronization of dataflow threads. The inputs to a consumer thread are written to its frame by one or more producer threads. Each thread has a synchronization count (SC) and it is decreased by each write. When the SC reached zero, the thread is ready to run.

4.2.3 TSTAR Synchronization

Figure 4.3 shows the illustrative examples. BB1, BB2, BB3 and BB4 are code regions corresponding to 4 different threads. Data dependences — represented by solid arrow lines — and control dependences — represented by dashed arrow lines — enforce a partial ordering on threaded execution.[1] Together, these data and control dependences form the Program Dependence Graph (PDG) of the function. BB2 may only be executed after its input value `z` and `x` have been produced by BB1.

To express producer-consumer and control dependences, we define an abstract data-flow interface suitable for parallelization passes in compilers as well as expert programmers developing low-level data-flow code.

The interface defines two main components: *data-flow threads*, or simply *threads* when clear from the context, together with their associated *data-flow frames*, or simply *frames*.

The frame of a data-flow thread stores its input values, and optionally some local variables or thread metadata. The frame's address also serves as an identifier for the thread itself, to synchronize producers with consumers. Communications between threads are single-sided: the producer thread knows the address of the data-flow frames of its dependent, consumer threads. A thread writes its output data directly into the data-flow frames of its consumers.

Each thread is associated with a Synchronization Counter (SC) to track the satisfaction of producer-consumer dependences: upon termination of a thread,

[1] Control dependence arcs are not to be mistaken for control flow arcs Wolfe [1995].

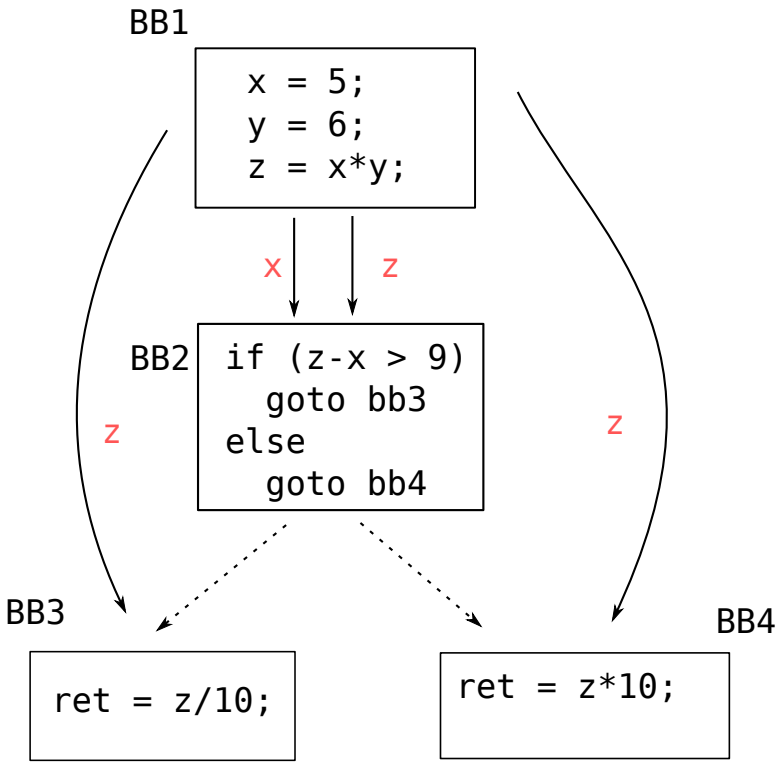

Figure 4.3: Program dependence graph of an illustrative example.

the SC of its dependent threads is decremented. A thread may execute as soon as its SC reaches 0, which may be determined immediately when the producer decrements the SC. The initial value of the SC is derived from the dependence graph: it is equal to the number of arguments of the thread, each one corresponding to an externally defined use.

In contrast, token-based approaches require checking the presence of the necessary tokens on incoming edges. This means that either (1) a scanner must periodically check the schedulability of data flow threads, or (2) data flow threads are suspendable. The former poses performance and scalability issues, while the latter requires execution under complex stack systems (e.g., cactus stacks) that may introduce artificial constraints on the schedule. The SC aggregate the information on the present and missing tokens for a thread's execution, allowing producer threads to decide when a given consumer is fireable.

In our example, thread BB2 has input `z` and `x`, it is not control dependent on

any node, therefore its initial SC is 2; thread BB3 has input `z` and it is control dependent on thread BB2, but its initial SC is 1 (BB3 will be created by BB2, thus satisfy the control dependence).

4.2.4 TSTAR Dataflow Instruction Set

We introduce the TSTAR dataflow instruction set to support our dataflow execution model. TSTAR dataflow instructions are inspired by T* ISA Giorgi *et al.* [2007]; Portero *et al.* [2011], and as an extension to x86 architecture. Paolo Faraboschi from HP Lab also contributes to the design of the interface.

They are implemented as compiler builtins, recognized as primitive operations of the compiler's intermediate representation. They can be implemented efficiently both in software or hardware. We will started describing the TSTAR ISA sets with a simple example, then in detail for each instruction.

Figure 4.4 shows a producer consumer pipeline relationship, producer produce value `x` and consumer consumes this value. Figure 4.5 shows the detailed implementation with TSTAR builtin functions.

A. The main thread

The main DF-thread takes care of creating the producer and consumer DF-threads and register dependences. In `S1,S2`, `tschedule` creates the DF-threads for producer and consumer, the synchronization counter could be either initialized due to the number of its producers (e.g. in `S2`, the consumer DF-thread's SC is initialized to 1), or used for controlling the schedule time of a certain thread (e.g. in `S1`, the producer DF-thread does not dependent on any other thread, but the SC is initialized to 1 at first, and decrease by 1 in `S5`. In `S3`, we use `tcache` to cache the DF-frame of the producer thread locally, so that we could directly read/write to it. And in `S4`, we register the consumer thread id to the producer thread, so that when the producer thread executed, it could get the frame address of the consumer, and write the results directly to it. `S5` decrease the SC of the producer, and the producer thread will be ready to execute (SC reached zero after decrementation).

B. The producer thread

The producer thread will do its computation (if have any), and pass the results to its consumers. In this example, `S7` loads the DF-frame of the current thread with `tload`. The `cfp` is locally cached and read only. In `S8`, it gets the consumer thread id that has been registered in the main thread and `S9` cache the consumer DF-frame locally. In `S11`, the result `x` is directly written to its consumer thread's DF-frame, and in `S12`, decrements (`tdecrease`) the SC of the consumer after writing, at which point the SC is decremented to zero, so the consumer thread will be ready to execute.

C. The consumer thread

The consumer thread load the DF-frame of the current thread in `S14`, and read the value written by the producer thread. Till now, the communication between producer and consumer has finished.

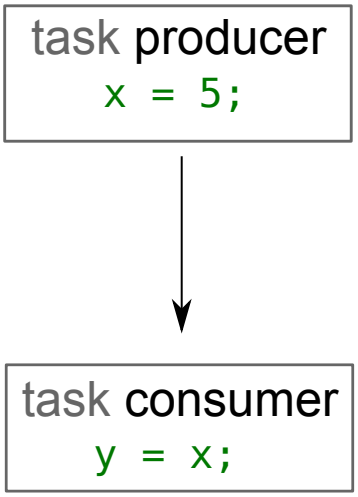

Figure 4.4: Producer consumer relationship.

Here is a detailed description of TSTAR instruction set interface.

```
dfthread_p tschedule (void (*func)(), int sc, int size);
```

Creates a new data-flow thread and allocates its associated frame. `func` is a pointer to the argument-less function to be executed by the data-flow

```
    A. Main task
    /* Main task, takes care of creating producer and
       consumer tasks, resolve dependences in between. */
    void main ()
    {
      /* Create producer task, the synchronization
         counter is set to 1. */
S1    producer_tid = tschedule (task_producer, 1, sz);

      /* Create consumer task, the synchronization
      counter is set to 1. */
S2    consumer_tid = tschedule (task_consumer, 1, sz);

S3    producer_fp = tcache (producer_tid);
S4    producer_fp->field_consumer_tid = consumer_tid;
S5    tdecrease (producer_tid);
S6    tdestroy ();
    }
```

```
    B. Producer task
    /* Producer task. Pass x to the consumer task. */
    void task_producer ()
    {
S7    cfp = tload ();
S8    consumer_tid = cfp->field_consumer_tid;
S9    consumer_fp = tcache (consumer_tid);
S10   x = 5;
S11   consumer_fp->x = x;
S12   tdecrease (consumer_tid);
S13   tdestroy ();
    }
```

```
    C. Consumer task
    /* Consumer task. Get x from the producer. */
    void task_consumer ()
    {
S14   cfp = tload ();
S15   x = cfp->x;
S16   y = x;
S17   tdestroy ();
    }
```

Figure 4.5: Producer consumer code.

thread, `sc` is the initial value of the thread's synchronization counter, and `size` is the size of the data-flow frame to be allocated. It returns a pointer to the allocated data-flow frame.

Once created, a thread cannot be canceled. Collection of thread resources is triggered by the completion of the thread's execution.

`void tdecrease(dfthread_p tid, n);`

Marks the target thread designated by frame pointer `fp` to be decremented by `n` upon termination of the current thread (lazy tdecrease). Once the current thread terminates, the synchronization counter of the target thread is decremented by `n`. When the synchronization counter of the target thread reach zero, it will be moved from the waiting queue to the ready queue.

In contrast with lazy tdecrease, there is also an eager tdecrease mode available in the implementation. Eager tdecrease will decrease the synchronization counter instantly at the moment tdecrease is called. There is always tread off between lazy tdecrease and eager tdecrease, lazy tdecrease will cache the potential tdecrease upon the thread termination, which potentially reduce the communication between threads, while on the other hand, the eager tdecrease's in time communication between threads makes the target thread becomes available as soon as possible, potentially increase the parallelism in the system.

`void destroy();`

Terminate the current thread and deallocates its frame. If it is running on lazy mode, the tdecrease will be merged and executed.

`void *load();`

Loads the DF frame of current thread into a locally allocated memory trunk (read only) that is directly accessible by the thread with standard loads.

`void *cache(void *tid);`

Allocates the frame of target thread id `tid` to a locally allocated memory chunk that is directly accessible by the thread with standard loads and stores.

4.3 TSTAR Architecture

TSTAR architecture is a multithreaded dataflow architecture, where in the inter-thread level, threads are executed in a multithreading environment, scheduled according to the dataflow model to better exploit parallelism, while in the intra-thread level, each thread is compiled into sequential von Neumann processor.

Figure 4.6 shows the TSTAR top level architecture. `C2` the single core which contains a processing element (x86-64 ISA with TSTAR extensions) along with its L1 cache. Each core also includes a partition of the L2 cache. In order to support the dataflow execution of threads, each core includes a hardware module that handles the scheduling of threads - the Local Thread Scheduling Unit (L-TSU). In addition to cores, the nodes also contain a hardware module to coordinate the scheduling of the data-flow threads among the cores - the Distributed Thread Scheduling Unit (D-TSU), as well as a hardware module that monitors the performance and faults of the cores - the Distributed Fault Detection Unit (D-FDU).Each core is identified with a unique id, the Core ID (CID), and each group of cores belongs to a node whose id is the Node ID (NID). Nodes are connected via an inter-node network, the Network on Chip (NoC). Cores within a node are connected via the NoC.

4.3.1 Thread Scheduling Unit

In TSTAR architecture, scheduling DF-threads is done by TSU. TSU is composed of local thread scheduling units (L-TSU) in each core and distributed thread scheduling units (D-TSU) in each node. Hence, management of resources, mainly continuations and DF-frames is done by TSU. The D-TSU is also informed by the FDU of any fault core, and so it includes execution recovery mechanisms. Figure 4.7 shows the overview of the TSU.

Each DF-threads is uniquely associated with a DF-frame. The DF-frame stores the data and meta data a DF-thread needed during execution. DF-threads are ready to run when all their input data is available in its associated DF-frame. The TSU track and update the Synchronization Counter (SC) which stored in the meta data section in each DF-frame. At the time a DF-thread is created, the SC will be initialized equal to the number of its producers. Each time when a

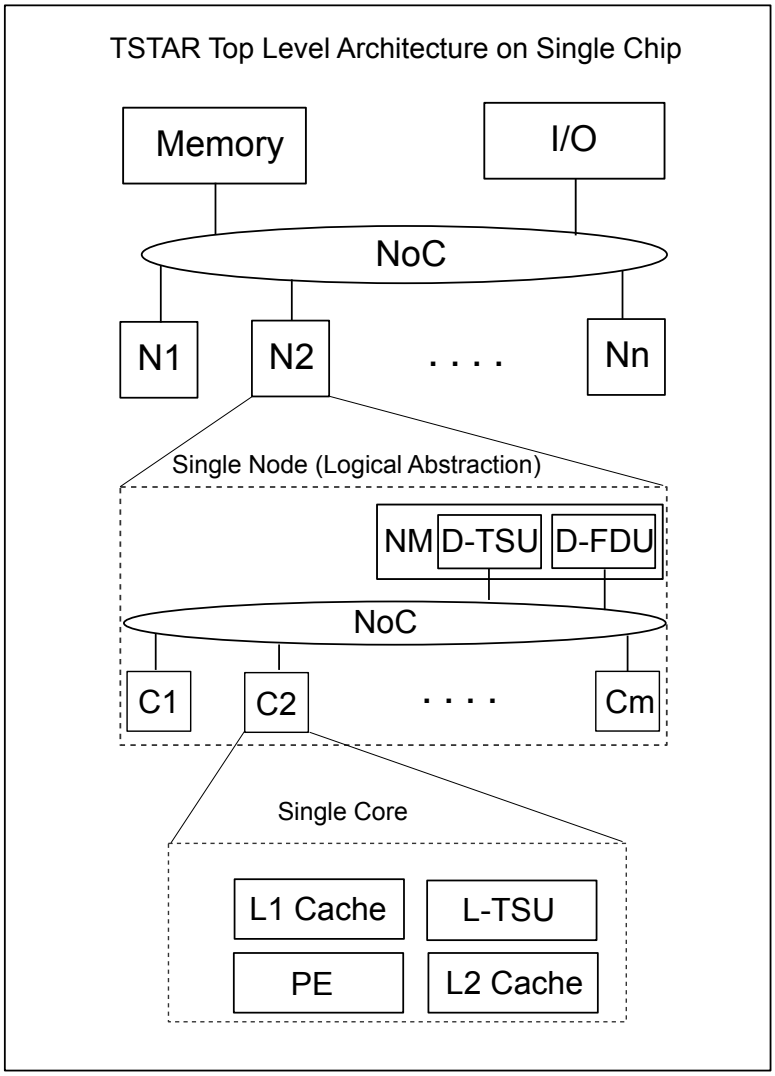

Figure 4.6: TSTAR Highlevel Architecture.

producer writes to its consumer, the consumer's SC will be decreased by 1.DF-threads are ready to run when SC decreased to 0 (all input data is ready).

Continuation is used to keep track of all the information of each thread. It consists of a tuple of three values:

1. *The Frame Pointer (FP)*: the address of a DF-frame (unique for every DF-thread).

2. *The Instruction Point (IP)*: the address of the DF-thread's code.

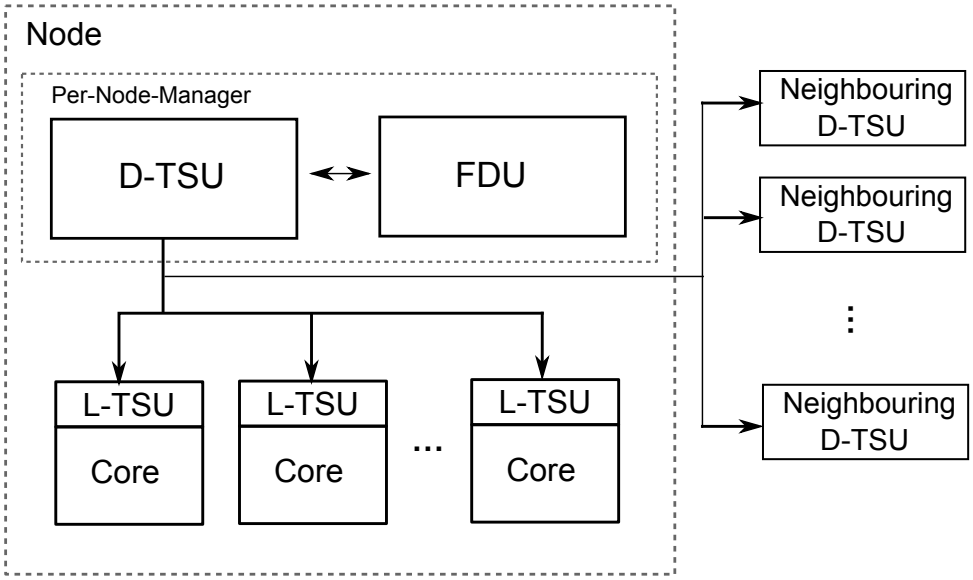

Figure 4.7: Overview of the Thread Scheduling Unit (TSU).

3. *The Synchronization Counter*: the number of input data required by the DF-thread before it can start execution.

Local Thread Scheduling Unit (L-TSU) The L-TSU handles the scheduling within the core in addition to the allocation and management of the DF-frames. The L-TSU uses three units to perform these operations:

- *Waiting List (WL)*: The waiting list stores the continuations (IP, SC, FP) of newly created DF-threads. There SCs are equal to the number of their producers.
- *Pre-LoadQueue (PQ)*: It stores the IP and FP of the DF-threads that are ready to execute, their associated SCs are decreased to 0. It is organized as a FIFO queue.
- *Free Frame List (FFL)*: It contains the FPs of the free DF-frames of the core. Whenever a DF-thread runs to completion, the associated DF-frame will be freed and added back to FFL.

The L-TSU keeps track of the dynamic continuations of threads in the Waiting List (WL). When a producer thread runs to completion and writes data to its

consumers, the SC value in the continuation will be decremented. When the SC reaches zero, it will be moved to the Pre-LoadQueue (PQ) and ready for execution.

- *Thread Creation.* The TCREATE instruction communicates to the L-TSU the IP and SC of the newly created DF-thread. This triggers the L-TSU to allocate a new DF-frame for the newly created DF-thread. The L-TSU locates the FP of the new DF-frame from the FFL. The L-TSU then stores in the WL the continuation (FP, IP, SC tuple) of the DF-thread. If there are no free DF-frames in FFL, then the TCREATE request is forwarded by the L-TSU to its node's D-TSU.

- *Thread Communication.* Producers and consumers communicate with each other through TWRITE. By using TWRITE, the producer thread could write its result(s) directly to its consumer(s). And once a TWRITE is issued, the SC of its consumer(s) will be decremented by one immediately (eager execution) or delayed to the end of the thread (lazy execution). When the SC decrease to zero, it will be placed to PQ. The TWRITE is snooped by the L-TSU so that L-TSU can decrement the SC associated with the DF-thread. If the TWRITE affects a DF-frame that is not stored locally, it will be routed by the NoC to the appropriate node where similar snooping actions will be performed.

- *Thread Completion.* When a DF-thread runs to completion, it invokes the TDESTORY instruction, which triggers the L-TSU to do a few things. It places the allocated DF-frame associated with the DF-thread on the FFL. It also transfers the continuation of the next ready DF-thread to the pipeline so that execution of that DF-thread begins and frees its associated PQ entry. Finally, L-TSU checks the WL for entries whose SCs have reached zero, and moves the FPs and IPs of those entries to the PQ and clears those entries in the WL.

Distributed Thread Scheduling Unit (D-TSU) Each node in the system contains a D-TSU that communicates with all the L-TSUs on the cores inside

that node. The D-TSU also communicates with the D-TSUs on the rest of the nodes in the system. The D-TSU communicates with the Fault Detection Unit (FDU) responsible for fault detection for the cores, the TSU in the node, as well as neighboring FDUs.

The D-TSU handles inter-core and inter-node thread scheduling. The D-TSU uses three units to perform those operations:

- *Free Frame Table (FFT).* The FFT stores the number of free DF-frames for each core in its node, which corresponds to that cores number of FFL filled entries. When a DF-frame is allocated/released in a core, the cores' corresponding entry in the FFT is decremented/incremented.
- *Frame Request Queue (FRQ).* The FRQ stores the frame allocation requests
- *Threads to Cores List(TCL).* The TCL keeps the information of all the DF-threads that exists in all the cores of its node. This information is used for fault recovery.

A D-TSU receives a TCREATE requests from other D-TSUs or from any of its nodes' L-TSUs. An L-TSU forwards a TCREATE request to a D-TSU when there are no free DF-frames locally in its core. A D-TSU forwards a TCREATE request to another nodes' D-TSU when there are no free DF- frames in its node. In both cases, when a TCREATE request arrives at a D-TSU it selects another core with free DF-frames to provide a free DF-frame. It first stores the TCREATE request in its FRQ, then checks its FFT to locate a core with a free DF-frame. It then forwards the IP and SC of the new DF-thread to a local L-TSU in its node that will provide the free DF-frame. When it receives the FP of the newly allocated DF-frame it forwards it to whoever made the TCREATE request (whether remotely from other D-TSUs or locally from one of its node's L-TSUs).

When a DF-thread invokes a TWRITE instruction with an FP that is remote (not local in the core where it is executing), the TWRITE is forwarded to the D-TSU of that DF-thread's node. The D-TSU then forwards it to the appropriate L-TSU if it is local in the node, otherwise it sends it over the NoC to the appropriate D-TSU where its core is located. Note that, the FP is unique for every DF-frame and holds information about the node and core at which the DF-frame is located.

Chapter 5

Thread Partitioning II: Transform Imperative C Program to Dataflow Program

We discuss a different way of looking at imperative programs in this chapter—as data-flow threads. These data-flow threads can be extracted from a conventional static single assignment (SSA) representation. Starting from the extraction of fine-grained parallelism, grain-coarsening transformations reduce synchronization overhead while avoiding significant loss of parallelism. As an important contribution, we lift all restriction on data dependence and control flow patterns, allowing for irreducible control flow and recursive calls. Our compilation algorithm also emphasizes modularity (separate compilation) and integration with existing development practices and binary interfaces.

This chapter is limited to the exploitation of scalar data dependences, expecting programmer intervention to expose producer-consumer relations from array and pointer code by means of explicit scalar dependences. Handling arrays and complex structures are further discussed in Chapter 6. Our approach is complementary to programming language efforts to express inter-task dependences as pragma annotations Planas *et al.* [2009]; Pop & Cohen [2011a]; Pop *et al.* [2009]; Stavrou *et al.* [2008]; it contributes to reducing the verbosity of such annotations.

5.1 Revisit TSTAR Dataflow Execution Model

We revisit the TSTAR dataflow execution model in this section, and explain in details on compilation challenges with the explicit token matching.

The Explicit Token Store To optimize the token matching problem in dynamic data flow architecture, Papadopoulos Papadopoulos & Culler [1990] proposed the explicit token store method. This method encodes the explicit rendezvous point for tokens to meet within a global address.

As shown in Figure 5.1, every location that corresponds to an activation of a two-input operator is augmented with a presence bit that is initially in the empty state. When the first token arrives, it notices the empty state, and writes its value, v, into the location. The presence bit is then set into the present state. The second token notices the present state and reads the location, and then sets the presence bit back to empty. The operation defined by the instruction s (from the tag) is performed on the incoming token's value and the value read, and a new token is generated with the resulting value, the same c, but a different s.

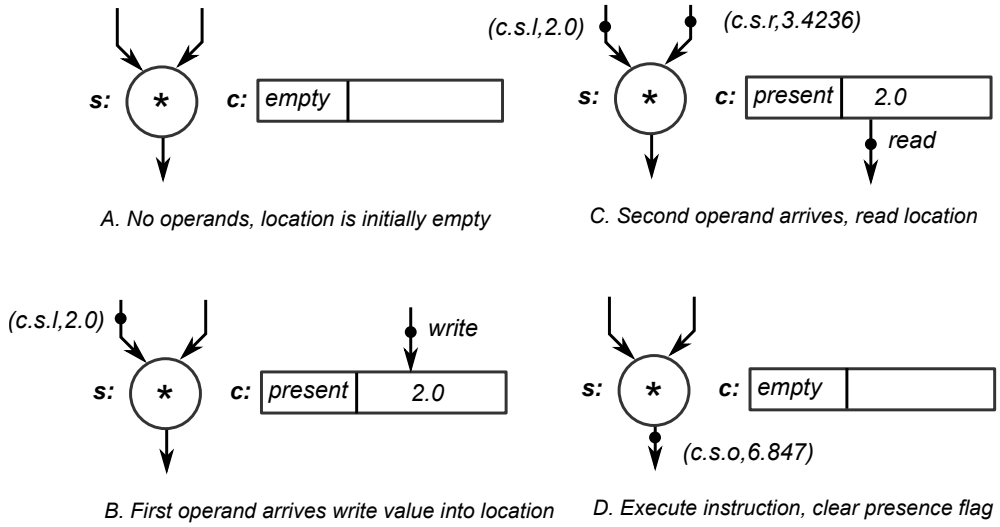

Figure 5.1: Explicit Matching Operation.

TSTAR Explicit Token Store We adapt the concept of explicit token store in TSTAR execution model to multiple inputs with introducing the synchronization

counter. The global addressable storage (Frame Memory) for dataflow frames is used to encodes the explicit rendezvous point. We will explain in details in the following paragraph.

TSTAR Dataflow Execution Model To express producer-consumer and control dependences, we define an abstract data-flow interface suitable for parallelization passes in compilers as well as expert programmers developing low-level data-flow code.

The interface defines two main components: *data-flow threads*, or simply *threads* when clear from the context, together with their associated *data-flow frames*, or simply *frames*.

The frame of a data-flow thread stores its input values, and optionally some local variables or thread metadata. The frame's address also serves as an identifier for the thread itself, to synchronize producers with consumers. Communications between threads are single-sided: the producer thread knows the address of the data-flow frames of its dependent, consumer threads. A thread writes its output data directly into the data-flow frames of its consumers.

Each thread is associated with a Synchronization Counter (SC) to track the satisfaction of producer-consumer dependences: upon termination of a thread, the SC of its dependent threads is decremented. A thread may execute as soon as its SC reaches 0, which may be determined immediately when the producer decrements the SC. The initial value of the SC is derived from the dependence graph: it is equal to the number of arguments of the thread, each one corresponding to an externally defined use.

In contrast, token-based approaches require checking the presence of the necessary tokens on incoming edges. This means that either (1) a scanner must periodically check the schedulability of data flow threads, or (2) data flow threads are suspendable. The former poses performance and scalability issues, while the latter requires execution under complex stack systems (e.g., cactus stacks) that may introduce artificial constraints on the schedule. The SC aggregate the information on the present and missing tokens for a thread's execution, allowing producer threads to decide when a given consumer is fireable.

The challenges in compilation *Explicit Token Matching* shifts the complications from the hardware design of token-matching system to the compiler. The compiler has to match the producers and consumers explicitly. In TSTAR execution model, as producer data flow threads communicate by writing directly in the data flow frame of their consumers, it is necessary that, along all data dependence edges of the program dependence graph, the producer nodes know the data flow frame of the consumer nodes. It becomes more complicates when the producer and consumer are created by separate control programs, which means there needs to be a way to communicate such information.

Figure 5.2 is the program dependence graph of a simple program. The dashed line represents control dependence and the solid line represents data dependence. If we partition each node as a task, we need a way to satisfy the control and data dependences. One simple way of satisfying the control dependences is to treat the control dependence edges as thread creation dependence (thread creation dependence is defined as, the target of the dependence will be created by the source of the dependence). In which way, `S1` and `C2` will be created by control node `C1`, and `S2` will be created by control node `C2`, the only dependence left is the data dependence between `S1` and `S2`.

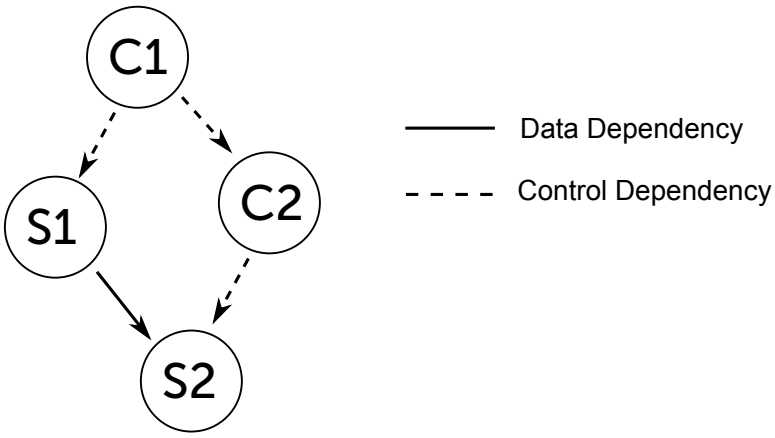

Figure 5.2: Program Dependence Graph for explicit token matching compilation example.

The problem needs to be solved is: How could we register `S2`(the consumer)'s DF-Frame information at the point `S1` is created? `S1` is created by `C1`, at the point when `S1` is created, `S1` does not have its consumer's information at all. It does not even know when `S2` will be created (`S2` is created by `C2`, the execution of `C2` depends on the dataflow execution model). We discuss the solution in details in later section with the help of our new IR (Data Flow Program Dependence Graph).

5.2 Partitioning Algorithms

The general approach for transforming sequential imperative programs into parallel data-flow programs extracts the finest grain of thread-level parallelism, splitting basic blocks at the statement level, which is then coarsened through typed fusion to reduce communication overhead. Strongly Connected Components (SCC) of the program dependence graph, where no parallelism can be exploited, are also coalesced.

Our algorithm operates on a low-level program representation in Static Single Assignment (SSA) Cytron *et al.* [1990, 1991] form, common in modern production compilers like GCC and LLVM. Our algorithm is implemented as a new parallelization pass of GCC's middle-end.

We only consider scalar data dependences and control dependences. As stated in the introduction, arrays and pointers are currently ignored from the dependence analysis. The correctness of program transformations requires programmers to expose dependences as scalar dependences in the source program.

Description of the algorithm We first build the Program Dependence Graph Ferrante *et al.* [1987] under SSA form (SSA-PDG) from the serial program, then coarsen the granularity by merging the SCCs in the graph and apply typed fusion. To align the flow of values and data-flow frames with the control dependences, we define the Data-Flow Program Dependence Graph (DF-PDG), translated from the SSA-PDG. The DF-PDG is then used to generate target data-flow code.

We illustrate our algorithm on the example in Figure 5.3 (left), where we assume that all functions are pure (no side effects, no state). In the loop body,

`bar(i)` is evaluated at each iteration, but only the last computed value will be used outside of the loop, along with the last value of `i`.

5.2.1 Loop Unswitching

The SSA form uses a unique name for each assignment to a variable. In this way, each use of a variable has a unique reaching definition. A merging Φ node is introduced in the SSA form at points where multiple control flow paths converge and a given variable is defined on more than one path. For the example on the left side of Figure 5.3, the Φ nodes for variables `i`, `a` and `b` will be placed before S9, in the loop header, as shown in Figure 5.4 (left).

Some of these Φ nodes in the header carry redundant data flow: the Φ node for variable `i` defines a value used inside the loop body, while the Φ nodes for `a` and `b` only define values used outside of the loop. To differentiate the type of Φ node, we apply loop unswitching so that the inductive Φ node for variable `i` will remain at the header while the Φ nodes capturing the last values of `a` and `b` will be placed at loop exit, before their respective uses, as in Figure 5.4 (right).

```
S1  a = 0;
S2  i = 0;
S3  b = 0;

S9  while (i < 100) {
S6    a = i;
S7    b = bar (i);
S8    i = next (i);
    }

S12 if (a > b)
S13   ret = a;
    else
S14   ret = b;
S16 return ret;
```

```
S1  a = 0;
S2  i = 0;
S3  b = 0;
S4  if (i < 100) {
      do {
S6      a = i;
S7      b = bar (i);
S8      i = next (i);
S9    } while (i < 100);
    }
S12 if (a > b)
S13   ret = a;
    else
S14   ret = b;
S16 return ret;
```

Figure 5.3: Running example (left) and after unswitching (right).

```
   # i1 = Φ (i0, i2);
   # a1 = Φ (a0, a2);
   # b1 = Φ (b0, b2);
S9 while (i1 < 100) {
S6   a2 = i1;
S7   b2 = bar (i1);
S8   i2 = next (i1);
   }
   /*use of a1, b1. */
```

```
   if (i1 < 100) {
     do {
       # i1 = Φ (i0, i2);
S6     a2 = i1;
S7     b2 = bar (i1);
S8     i2 = next (i1);
S9   } while (i1 < 100);
   }
   # a1 = Φ (a0, a2);
   # b1 = Φ (b0, b2);
   /* use of a1, b1. */
```

Figure 5.4: SSA form before unswitching (left) and after (right).

5.2.2 Build Program Dependence Graph under SSA

There are two main reasons for relying on a SSA-based representation of the PDG:

1. making the reaching definitions unique for each use, effectively converting the scalar flow into a functional program;

2. reducing the complexity from $O(N^2)$ to $O(N)$, as the number of def-use edges can become very large, sometimes quadratic in the number of nodes Kennedy & Allen [2002].

In SSA form, multiple reaching definitions for a use are factored through Φ nodes, which ensures that the number of def-use chains is bounded by the number of chains in the control flow graph.

Each node in the SSA-PDG represents a statement in SSA form and edges in the graph represent either control or data dependences. Control dependences can form several weakly connected graphs. We add control dependences from the entry node of the function to the root node of each weakly connected graph. The root node of each weakly connected graph is defined as the first node in the graph reached by the control flow.

Figure 5.5 shows the SSA representation, after loop unswitching, for the code on Figure 5.3 (right), and the corresponding SSA-PDG is presented on Figure 5.6, where dashed edges represent control dependences and solid edges represent data

dependences. We add control dependences from the function entry point `foo` to `S2, S4, S1, S3, S10, S11, S12, S15, S16`.

```
int foo () {
  S1  a0 = 0;
  S2  i0 = 0;
  S3  b0 = 0;
  S4  if (i0 <= 99)
        goto S5;
      else
        goto S10;

  S5  # i1 = Φ(i0, i2);
  S6  a1 = i1;
  S7  b1 = bar (i1);
  S8  i2 = next (i1);
  S9  if (i2 <= 99)
        goto S5;
      else
        goto S10;

  S10 # a2 = Φ(a0, a1)
  S11 # b2 = Φ(b0, b1)
  S12 if (a2 > b2)
        goto S13;
      else
        goto S14;

  S13 ret0 = a2;
      goto S15;

  S14 ret1 = b2;

  S15 # ret2 = Φ(ret0, ret1)
  S16 return ret;
}
```

Figure 5.5: SSA representation after loop unswitching.

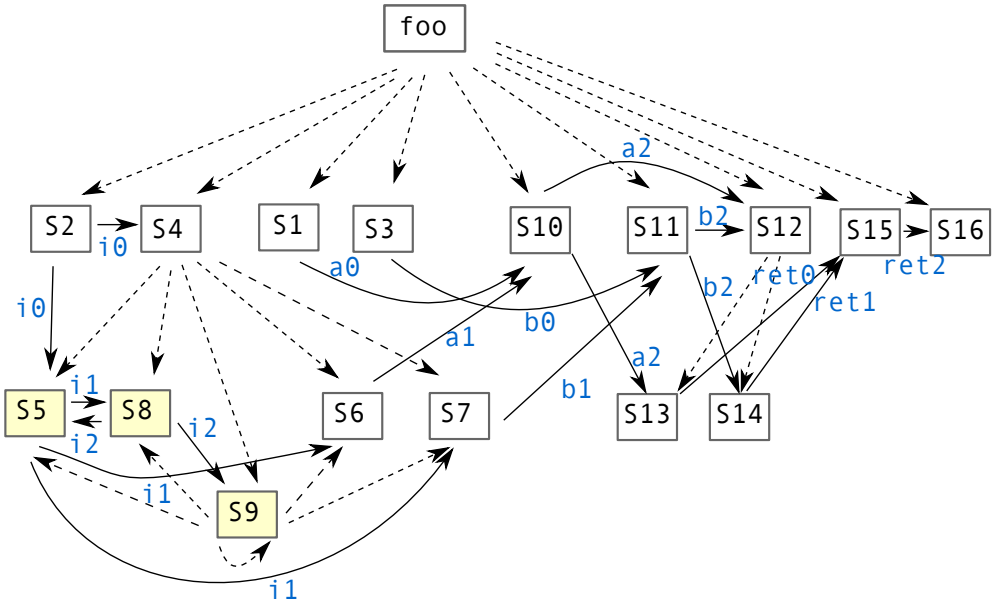

Figure 5.6: SSA-PDG for the code example in Figure 5.5.

5.2.3 Merging Strongly Connected Components

In the SSA-PDG, SCCs present no parallelization opportunities. Their execution is sequential, so generating data flow threads would mostly incur parallelization overhead.[1] We coalesce SCCs, as shown on Figure 5.7 where the new node `SCC1` replaces nodes `S5, S8` and `S9` from Figure 5.6. This SCC corresponds to the induction on variable `i` in the loop.

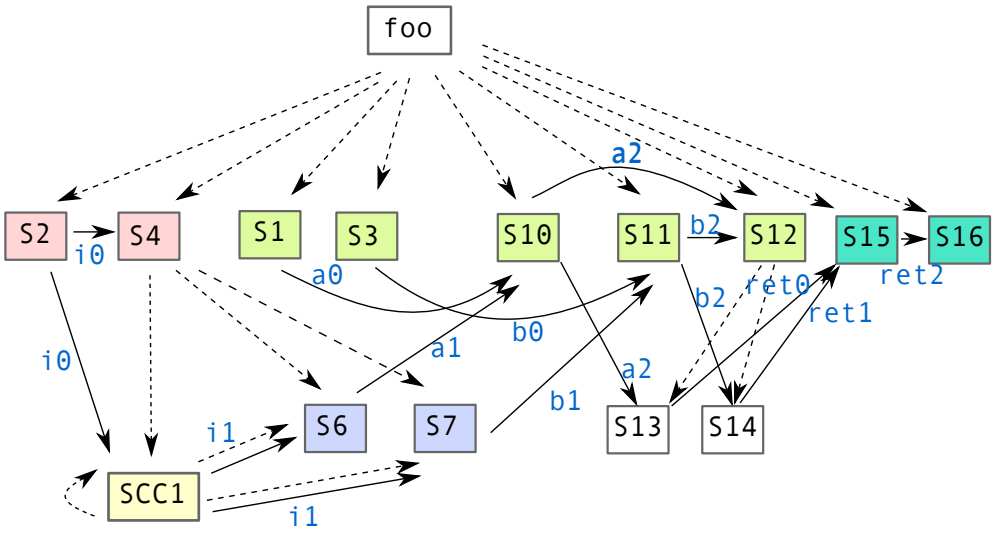

Figure 5.7: SSA PDG after merging the SCC.

5.2.4 Typed Fusion

Before partitioning, we coarsen the granularity of each data-flow thread using a generalized form of the typed fusion algorithm of McKinley and Kennedy Kennedy & Mckinley [1993]. In SSA-PDG, we assign a *type* to each node according to its control dependences: all nodes sharing identical control dependences are assigned the same type. Nodes of the same type are candidates for typed fusion. On Figure 5.7, the nodes `S6` and `S7` have the same type, as both are control dependent on `S4`, and can potentially be fused. Similarly, nodes `S2, S4, S1, S3, S10, S11, S12, S15` and `S16` could be fused, but this would lead to

[1] Some latency-hiding benefits exist, but we concentrate on parallelism extraction in this paper.

adverse side effects, in particular reducing parallelism by creating artificial SCCs. In our example, such a fused node would lead to SCC with nodes `S13` and `S14` because of the dependence chains `S10-S13-S15` and `S11-S14-S15`.

For this reason, we limit the fusion algorithm to avoid introducing new SCCs or increasing the size of existing SCCs, which are on the critical path of the program's execution.

Our approach for typed fusion follows a simple greedy algorithm, which starts from a random node in each typed set and adds new nodes of the same type to the fusion set as long as: (1) the new additions do not lead to creating a new SCC in the SSA-PDG after fusion; and (2) the fusion set does not contain SCCs itself. As this algorithm is applied after fusing the existing SCCs, the latter condition simply means that nodes that have self-dependences are not considered candidates even if their type matches.

Figures 5.7 and 5.8 (A) present one possible outcome of typed fusion applied to the running example. There are five different types in the graph on Figure 5.7: `(foo)`, `(S2, S4, S1, S3, S10, S11, S12, S15, S16)`, `(SCC1, S6, S7)`, `(S13)` and `(S14)`. Note that `S13` and `S14` have distinct types because their control dependences correspond to different truth values for `S12`.

For the typed set `(SCC1, S6, S7)`, `SCC1` cannot be fused with any other node as it would increase the size of an existing SCC (restriction (2)). This only leaves `S6` and `S7`, which can be fused and yield the fused node `F2` on Figure 5.8 (A). For the typed set `(S2, S4, S1, S3, S10, S11, S12, S15, S16)`, there are many possible outcomes, depending on the traversal order of typed sets. In this case and as we discussed above, restriction (1) does not allow, for example, nodes `S4` and `S10` to be in the same fusion set because of the dependence chain `S4-S6-S10`. The algorithm will always lead to at least three fusion sets for this type due to the long dependence chain `S4-S6-S10-S13-S15`.

This technique coarsens the granularity of data flow threads, without inserting redundant computations and without increasing the number of instructions belonging to SCCs (which usually happens when relying on basic blocks to partition the computation).

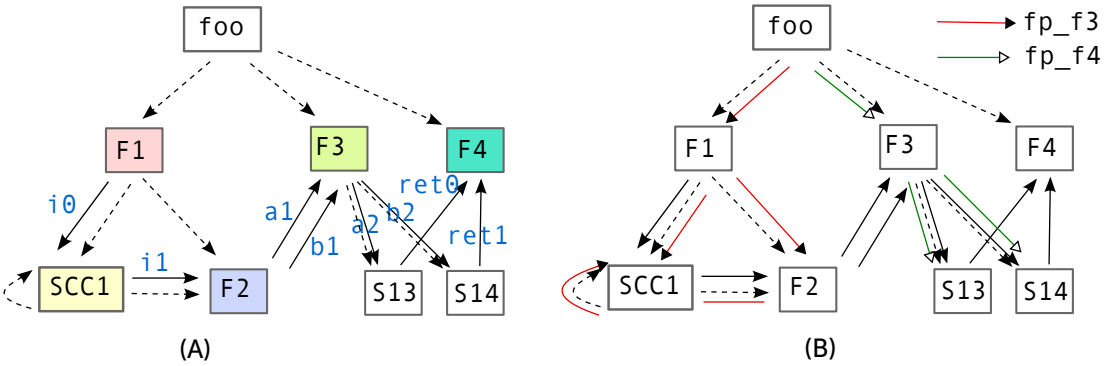

Figure 5.8: (A) SSA-PDG after typed fusion. (B) Data Flow Program Dependence Graph.

5.2.5 Data Flow Program Dependence Graph

As data flow threads communicate by writing directly in the data flow frame of their consumers, it is necessary that, along all data dependence edges of the SSA-PDG, the producer nodes know the data flow frame of the consumer nodes. We transform the SSA-PDG graph to reflect the communication required to this effect.

Thread creation point Thread creation occurs, conditionally, along each control dependence edge of the SSA-PDG. The thread creation points for a given node are its predecessor nodes in the control dependence graph. On Figure 5.8 (B), where dashed lines represent control dependences, the nodes `foo, F1, F3` and `SCC1` are thread creation points as they have outgoing control dependence edges. Control dependences can be conditional, like in the case of `F3` which creates either `S13` or `S14` depending on the conditional statement `S12`, or unconditional in the case of the function entry point `foo`.

At a thread creation point, the data-flow frame for the newly created thread is known and it needs to be passed to all threads producing data for this new thread.

Passing data-flow frame information The DF-PDG for our running example is shown on Figure 5.8 (B). We add data dependences to the SSA-PDG for

passing the data-flow frame information of consumer threads to producer threads. The edges with white triangular arrow communicate the data-flow frame of `F4` (consumer) to the data-flow threads `S13, S14` (producers). The values of `ret0` and `ret1`, produced in `S13` and `S14` are consumed by `F4`. As the data is stored directly in the frame of the consumer, the producer must get a pointer to the frame of `F4`. The thread creation point for `F4` is the function entry point, `foo`, and it needs to forward this frame pointer to all producers, which involves thread `F3` in our example.

We rely on the following algorithm to pass the data flow frame pointers of consumers to producers. For each data dependence from a node `A` to a node `B` in the SSA-PDG, we visit the predecessors of each node along control dependences until we reach a common predecessor `P`.

- If `P` is an immediate predecessor of the consumer node `B`, then `P` is the thread creation point for `B` and therefore knows the data-flow frame of `B`. We add data dependences for the frame pointer of `B` along all control dependence paths in the graph to `A`.

- If `P` is not an immediate predecessor of `B`, we need to split the data dependence as the frame of the consumer cannot be known due to diverging control flow paths, as illustrated on Figure 5.9. We remove the original data dependence from `A` to `B` and we add data dependences, for the same variable, from `A` to all successors `D` of `P` in the SSA-PDG such that there is a control dependence path from `D` to `B`. We further add data dependences for the data flow pointer of `D` from `P` to `A` and also for the data itself, from `D` to `B`.

This second case is illustrated on Figure 5.9, where data dependences from `S1` to `S2` and `S3` need to be split. The common predecessor is the function entry node `foo`, which is not an immediate predecessor of either `S2` or `S3`. The successor of `foo` that is a predecessor of `S2` and `S3` is `C1`, which is used to forward the data produced by `S1`. The frame pointer of `C1` is sent to `S1` from its thread creation point.

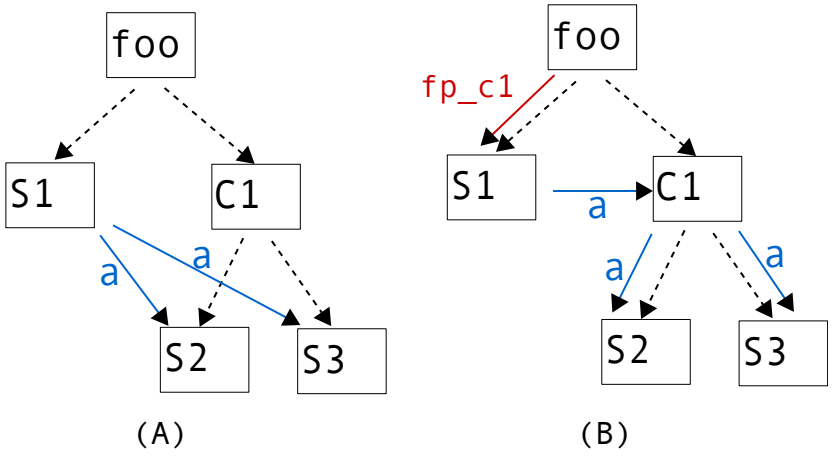

Figure 5.9: Splitting data dependences: (A) the original SSA-PDG and (B) the generated DF-PDG.

Strongly connected components in the DF-PDG The algorithm for building the DF-PDG presented above introduces additional data dependences that can lead to new SCCs in the graph. These new constraints need to be enforced, which serializes the execution. For this reason, we perform one additional pass of fusion of SCCs once the DF-PDG is constructed.

The example on Figure 5.10 and corresponding SSA-PDG in Figure 5.11 illustrate this issue. There are two data dependences: `S1` to `S2` and the loop-carried dependence `S3` to `S1`. For the latter, the DF-PDG construction algorithm explores every control dependence paths linking `S3` and `S1` from a common predecessor, namely `C0`→`S3` and `C0`→`C1`→`S1`. As `S1` is only reached through `C1`, the data dependence is split and the data forwarded to `S1` through `C1`, as shown by the extra data dependence edges on Figure 5.11 (right). Similarly, the data dependence from `S1` to `S2` needs to be split, resulting in the gray path. There are four different combinations for each data dependence and we only show one on Figure 5.11, yet it already results in a SCC involving nodes `S1, S3` and `C1`.

```
C0    if (c) goto S1;
      else goto D1;
S1    a2 = Φ (a0, a1);
S2    ... = a2;
S3    a1 = ...;
C1    if (c) goto S1
      else goto D1
D1    ...
```

Figure 5.10: SSA representation for a simple loop carried dependence

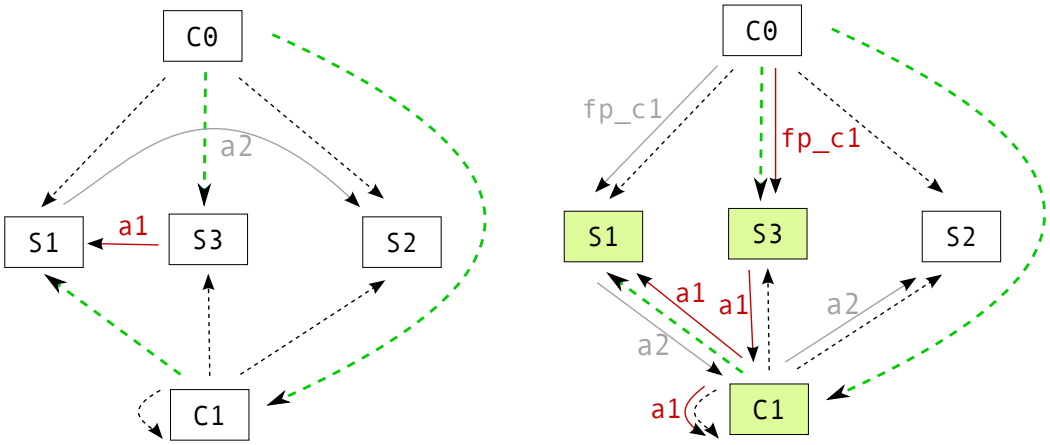

Figure 5.11: Corresponding SSA-PDG (left) and a partial DF-PDG (right) for code in 5.10.

5.3 Modular Code Generation

We put a strong emphasis on generality and flexible integration of data-flow compilation tools in a state-of-the-art development process. Modularity has not been considered a first-class objective in previous thread-level data-flow algorithms. We show that modular code generation is possible, provided that each processor core (or instruction fetch unit) is associated with a private user-level stack. The stack only needs to be accessed by this particular core. Data-flow threads themselves do not need any internal stack; they are non-suspendable and run sequentially w.r.t. other data-flow threads scheduled on the same core. Any data-flow thread scheduled on a given core is free to use the core's private stack. This stack streamlines the implementation of classical (blocking) function calls.

It may also be used to spill registers within a thread, although data-flow frames may accommodate free space for this purpose.

For modular compilation purposes, externally visible functions in a compilation unit should be cloned, to preserve the original control flow interface, while the clone is compiled into data-flow threads. The original function can be used when calling the function from outside of a parallel data-flow region or to avoid saturating the system with threads. The clone must be exported among the module's symbols to seamlessly compose threaded code over separately compiled modules.

Within a parallel region, all functions are called asynchronously. Internal functions, within the scope of the compilation unit, are directly compiled into data-flow threads. External functions, linked from separately compiled modules, and builtin functions from the compiler are wrapped into a data-flow thread in which they are called synchronously.

Every threaded clone of a function is split into three stages: the entry thread, multiple compute threads, and the return thread.

- The entry thread implements the entry block of the control flow graph, creating all the threads for the blocks that are unconditionally executed upon entering the function.

- The compute threads are systematically created by the immediate predecessors in the control dependence graph, as described in the previous section. Thread creation is conditional on the predicate at the source of the control dependence. Input arguments and pointers to the frames of the dependent threads are handled according to the DF-PDG.

- The return thread propagates the return value to the continuation threads at the call site of the function.

Figure 5.12 illustrates the calling convention. Calling a data-flow function creates the entry thread (`callee.entry`) of the callee and the caller's continuation thread (`caller.bb.2`); the latter will wait for the value of the callee's return thread (`callee.return`).

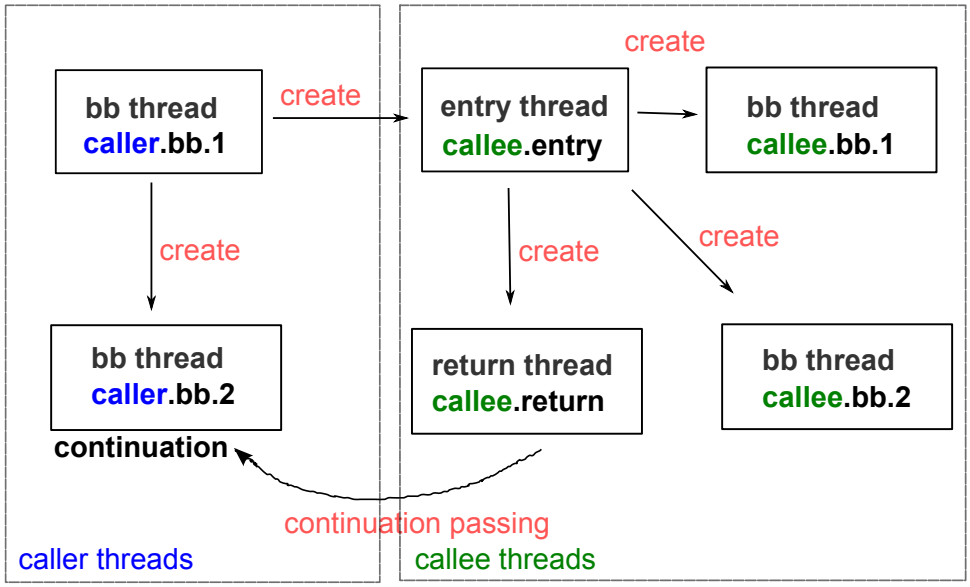

Figure 5.12: Caller and callee, threaded version.

5.4 Implementation

We target data-flow execution on a shared-memory multiprocessor with hardware coherence.

Our prototype has been implemented within GCC (GNU Compiler Collection) 4.7.0. GCC has been widely used for compilation/architecture research like automatic vectorization Nuzman *et al.* [2006] Nuzman & Henderson [2006], thread level speculation Liu *et al.* [2006] Renau *et al.* [2005], induction variable recognition Trifunovic *et al.* [2010] etc.

Compilation Figure 5.13 illustrates the GCC compilation framework. The compilation framework could be splitted into 3 parts: the front end, the middle end, and the back end. The middle end is independent of language or architecture, and most optimizations happens here. Language dependent code in the front end will be lowered to GENERIC. GENERIC is an IR that provides a language-independent way of representing an entire function in trees, and GENERIC will be lowered to GIMPLE (gimplification) in the middle end. GIMPLE is a three-address IR derived from GENERIC by breaking down GENERIC expressions into

tuples of no more than 3 operands. Most optimizations happens with GIMPLE representation.

The partitioning algorithm is implemented as an optimization pass in GCC (THREAD PARTITION). We first build the Program Dependence Graph under SSA form (SSA-PDG) from the serial program, then coarsen the granularity by merging the SCCs in the graph and apply typed fusion. To align the flow of values and data-flow frames with the control dependences, we define the Data-Flow Program Dependence Graph (DF-PDG), translated from the SSA-PDG. The DF-PDG is then used to generate target data-flow code.

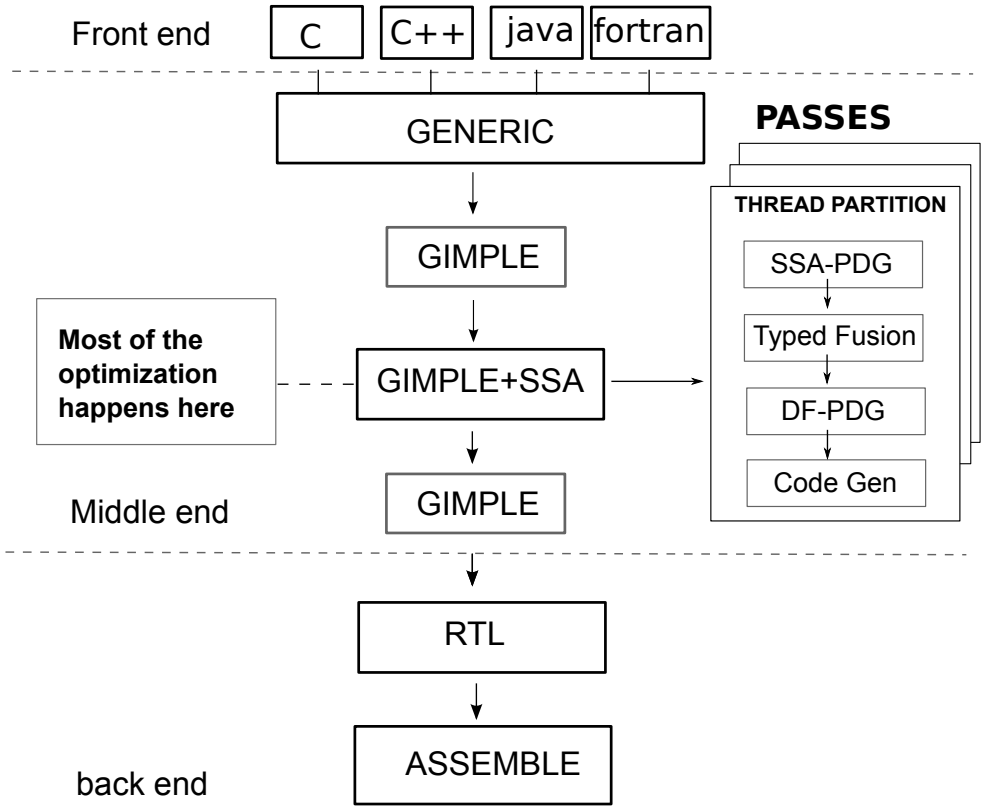

Figure 5.13: Implementation within GCC.

Runtime It replaces the `libgomp` OpenMP runtime of GCC. Parallel data-flow code runs within an `omp parallel single` region, extending the scheduler for OpenMP tasks. The runtime system is called `dfrt` and is implemented in C++.

Worker threads are created by `libgomp`. Upon entering an `omp parallel` region, they start scheduling tasks from the (shared) ready queue. The scheduler takes data-flow threads whose SC reached 0 and moves them to the ready queue.

We currently assume there are enough ready threads to occupy the processor cores and hide latency. This hypothesis eliminates the need for a waiting queue collecting threads whose SC has not reach 0, since we do not need to start prefetching data or code for these threads. Revisiting this hypothesis may be necessary when studying applications with limited parallelism degree where scheduling and memory latencies are harder to hide.

Data-flow frames are allocated from a dedicated memory pool. This pool internally uses slab allocation to accelerate the allocation and deallocation of frames of predefined sizes. The frame structure itself is laid out as follows:

- a thread template pointer referring to invariant meta-data shared by all thread instances of this template, including the function (code) pointer and the size of the frame;
- the thread's synchronization counter (SC);
- pointers to frames of data- and control-dependent threads;
- the thread's arguments.

The last two items correspond to the frame structure exposed in the abstract data-flow interface and generated by the compiler.

5.5 Experimental Validation

We validate our approach on two universal examples of tree recursion, Fibonacci and merge sort. The objectives are:

1. checking the method on diverse data and control flow, including loops over arrays, divide and conquer recursion, and data-dependent conditions;
2. Fibonacci exhibits the finest-grain threads possible, which gives a precise reference on the break even point and scalability for thread-level data flow compared to fine-grain data flow;

3. merge sort is more realistic and allows to illustrate typed fusion for grain coarsening.

We target an Intel Core i7-2720QM 4-core laptop (Sandy Bridge chip) and an AMD Opteron 6164 HE 24-core blade server (two Magny Cours chips). Both benchmarks are recursive, sequential C programs, and automatically parallelized.[1]

To assess the effect of thread granularity, we set a threshold for parallel recursive calls. Below this threshold, the serial version is executed. This programmer-controlled granularity complements the effects of the automatic typed fusion algorithm. Modular compilation allows the serial version to be called seamlessly as an external function.

As an illustration, in the Fibonacci implementation below, `fib.threaded` will be transformed to data-flow threads, and `fib.serial` will not since it is declared as an external function.

```
extern int fib.serial (int);

int fib.threaded (int n) {
  if (n < THRESH)
    return fib.serial (n);
  else
    return fib.threaded (n-1) + fib.threaded (n-2);
}
```

Figure 5.14 reports the performance of merge sort on $200,000$ random integers between 0 and $10,000$. The compiler automatically partitions the function into data-flow threads, then converts the data and control dependences into the proper frame operations. The algorithms not only parallelize the recursive division of the array, but also the merge operation. The latter is a good candidate for typed fusion: the array comparisons and assignments are dominated by the same loop header and can be fused into a coarser-grain thread.

[1]The array dependences in merge sort are covered by scalar dependences on the indexes, and can safely be ignored.

The grain threshold ranges from 2^0 to 2^{18} (262, 144, effectively sequentializing the execution). The figures show the speedup as a function of the granularity of the parallel threads. As the grain threshold increases, speedup gained from parallel execution on multiple cores exceeds the overhead of thread creation. The generated code breaks even at the threshold of 2^4. This low break-even threshold is a benefit of the applicability of typed fusion on the merge operation. As a divide and conquer algorithm, the problem size is reduced in each division, the array eventually fitting into the cache; it reaches a maximal speedup of 2.82.

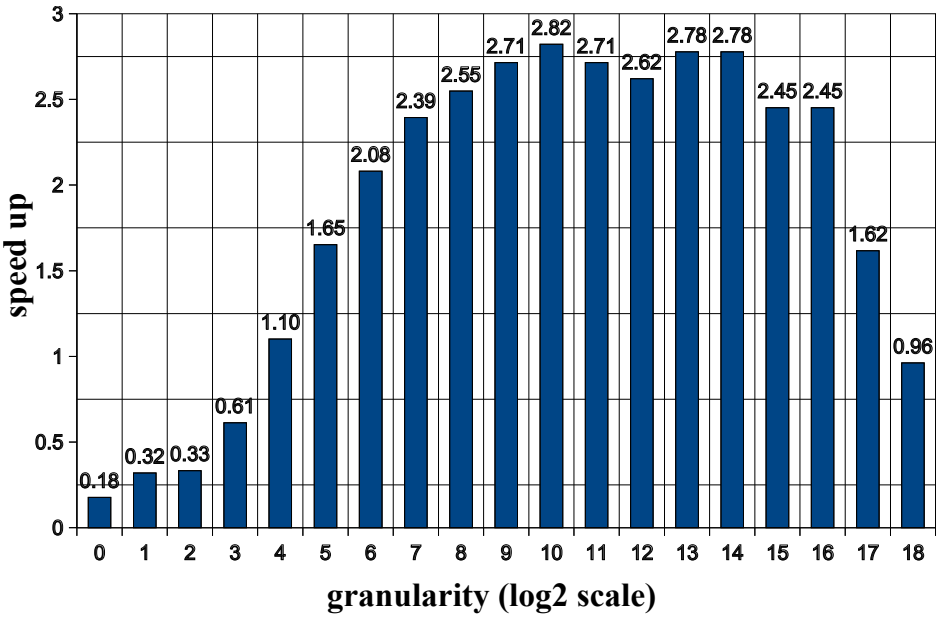

Figure 5.14: Merge Sort running on 4 cores.

Figure 5.15 shows the performance results for `fib(42)`. Fibonacci is an extreme case where typed fusion is ineffective, since no pairs of instructions share the same control dependence. The generated code breaks even when setting the threshold at `fib(15)`, where 2^{27} threads are created. But as granularity increases, the overhead of thread synchronization decreases. Our results on 24 cores reach a speedup of 11.86, which validates our algorithm's ability to exploit parallelism effectively.

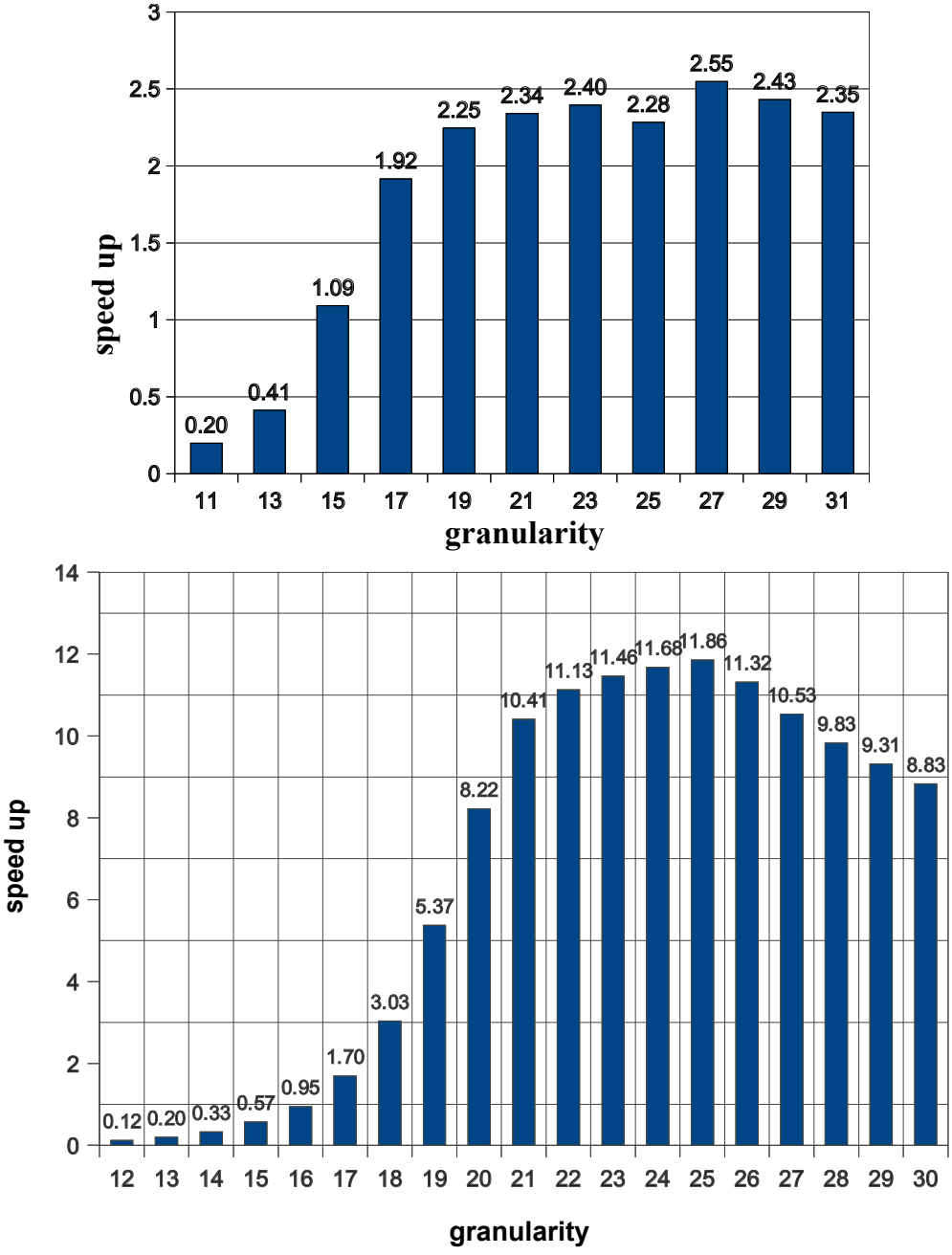

Figure 5.15: Computing the 42th Fibonacci number on 4 cores (above) and 24 cores (below).

5.6 Summary

We presented an automatic parallelization algorithm to compile arbitrary imperative control flow to a multithreaded data-flow model in this chapter. The algorithm operates on an SSA form PDG, extracting task, pipeline and data parallelism, then applying a generalized form of typed fusion to coarsen the synchronization grain, and finally expressing the communications in a suitable way for tokenless threaded data-flow execution. Our prototype is implemented in a production compiler; it currently supports scalar dependences only.

In next chapter, we complement this algorithm with two methods of handling complex data structures.

Chapter 6

Handling Complex Data Structures

One general criticism leveled at dataflow computing is the management of data structures. The functional nature of pure dataflow programs implies that all operations are side-effect free. The absence of side effect means that if tokens are allowed to carry vectors, arrays, or other complex data structures, an operation on a data structure results in a new data structure. Which will greatly increase the communication overhead in practice. The problem of efficiently representing and manipulating complex data structures in a dataflow execution model remains a challenge.

In this chapter, we propose two methods of handling complex data structures. Section 6.1 presents Streaming Conversion of Memory Dependences (SCMD), a hybrid compilation-time and runtime algorithm, which decouples the memory accesses with computations automatically and connects independent accesses to the same memory locations with streams in finest granularity. Section 6.2 present the Owner Writable Memory (OWM) model to reduce the communication overhead when complex structures passed over threads.

6.1 Streaming Conversion of Memory Dependences (SCMD)

Streaming Conversion of Memory Dependences (SCMD) is a hybrid compilation-time and runtime algorithm that automatically parallelize programs with arrays at the finest granularity, it connects independent `writes` and `reads` access to the same memory location with streams dynamically, in which way, the computation could be decoupled with even dynamic memory access patterns. In conventional dataflow, arrays are treated as a single memory region, the dependences are coarsened accordingly, and the parallelism opportunity is largely reduced.

6.1.1 Motivating Example

Figure 6.1 shows a general data access pattern for array A. `S1` writes to memory location $A[m(i)]$, `S2` reads from memory location $A[g(i)]$ and `S3` writes to memory location $A[f(i)]$. `m(int)`, `g(int)`, `f(int)`, `compute_1(int)` and `compute_2(int)` are side effect free functions.

```
  for (i = 0; i < N; i++)
    {
S1    A[m(i)] = compute_1(i);
S2    a = A[g(i)];
S3    A[f(i)] = compute_2(i);
    }
```

Figure 6.1: Non-linear access for array A within a loop.

The dynamic array accesses and unknown dependences make it impossible to be parallelized with compile time analysis. One might think a solution to decouple the computation and memory accesses with loop distribution and then parallelize the distributed loops as presented in Figure 6.2.

But there are a few caveats here:

- The loop distribution method, as we discussed in Chapter 3, inserts a barrier between distributed loops. In this example, the barriers are inserted between `L1` and `L2`, `L2` and `L3`. `L3` could only be executed after `parallel for` execution of `L2` finishes.

```
L1  parallel for (i = 0; i < N; i++)
    {
      Compute1[i] = compute_1 (i);
    }

L2  parallel for (i = 0; i < N; i++)
    {
      Compute2[i] = compute_2 (i);
    }

L3  for (i = 0; i < N; i++)
    {
      A[m(i)] = Compute1[i];
      a = A[g(i)];
      A[f(i)] = Compute2[i];
    }
```

Figure 6.2: Decouple computation and memory accesses with loop distribution.

- `L3` could not be executed in parallel. But at run time, there are still parallelization opportunities. Consider the situation where `g(i)`, `f(i)` and `m(i)` are all evaluated to $(0, 1, 2, \ldots, N)$ at runtime. By using SCMD, the tasks could be matched automatically and executed in parallel.

- Loop distribution works when the loop is well structured. When complex control dependences get involved, the computation and memory accesses can be hardly analyzed at compilation time.

SCMD decouples the memory accesses with computations automatically and connects the accesses to different memory locations with streams in finest granularity, which could exploit maximum parallelism in the program at execution time. We will explain SCMD in details.

Consider the access order for a single memory location $\{A[m]|m \in [0, N-1]\}$. Let $\vec{S}_m$ denotes the access order for memory location $A[m]$, and for $\{\{x \in \vec{S}_m\}, x \in \{W, R\}\}$. W denotes *write access* and R denotes *read access*. The *access order* represents the *read* and *write* accesses to a single memory location during the execution of the program. In our motivating example, $\vec{S}_m$ depends on how `m(i)`, `g(i)` and `j(i)` evaluate in the code. e.g., if $N = 1$, `m(i)` , `g(i)` and `f(i)` all evaluate to 0, we have the access order $\vec{S}_0 = \{W, R, W\}$; if $N = 2$, and `m(i)`

evaluates to $\{0, 1\}$, `g(i)` evaluates to $\{1, 1\}$ and `f(i)` evaluates to $\{1, 0\}$, then the access order for memory location $A[0] = \{W, W\}$, and $A[1] = \{R, W, W, R\}$. Note in the access order for each memory location, the first element in the sequence must be W (producer always comes before consumer). The access order for memory location $A[1]$ we showed here is just a partial access order, the $A[1]$ must be initialized before this loop (e.g. when the array is initialized the first time).

In SCMD, in order to get the finest granularity and maximum parallelism, each *write* to a memory location with its computation are forked as a producer, and each *read* to a memory location with its computation are forked as a consumer. In which way, the *reads* and *writes* to a each memory location will be connected with a dataflow stream.

In the TSTAR dataflow execution model, the producer writes to its consumer(s) directly, which indicates, the producer has to have the knowledge of its consumer(s) before it is being executed. But in a program with dynamic array access patterns, at the point the producer is created, its consumers are unknown until the next access to the same memory location is executed.

The question is, in a totally dynamic program, how do we connect writes and reads to the same memory location with dataflow streams?

We will consider the access order separately in 3 possibly sequences, and discuss each in the following sections:

- Single producer and single consumer where $\vec{S}_m = \{W, R\}$.
- Single producer and multiple consumers where $\vec{S}_m = \{W, R, R\}$.
- Multiple producers and single consumer where $\vec{S}_m = \{W, W, R\}$.

6.1.2 Single Producer Single Consumer

Assume we have the access order $\vec{S}_i = \{W, R\}$ for memory location $\{A[i] | i \in [0, N-1]\}$. When the value assigned to $A[i]$ is produced (W in its access order $\vec{S}_i$), we do not have the knowledge in which iteration of the for loop, it will be consumed (R in its access order $\vec{S}_i$). In order to solve this problem, a proxy thread will be created and wait for the result produced by the producer, and also, wait for its consumer's information (DF-Frame). When the consumer is created, The

pointer to its DF-Frame will be written to this proxy thread. When the proxy thread have both the result and the pointer to its consumer's DF-Frame, it will be executed and write this result to its consumer.

Figure 6.3 illustrates this solution. The dotted line with white triangle represents the execution order in the control program: the control program for consumer 1 (`CPC1`) could only be executed after control program for producer A (`CPPA`) finishes its execution. The dashed lines represent thread creation relationship from its source to its destination, in this case, `CPPA` will create the producer thread (`ProducerA`) and the its proxy thread (`proxy_p`), store the frame pointer of `proxy_p` (`fp_proxy_p`) to shared buffer `CTBL` indexed by memory location A(i). The proxy thread has SC initialized to 2, and wait for the result from producer thread and the DF-Frame address of the consumer. `CPC1` will create the consumer thread, and write the frame pointer of the consumer (`fp_consumer`) to `fp_proxy` by looking up the shared memory indexed by memory location `A(i)`. When the synchronization counter of `proxy_p` decrease to 0, it will write the result to the consumer thread pointed by `fp_consumer`, and then the consumer will be ready for execution.

6.1.3 Single Producer Multiple Consumers

Assume we have the access order $\vec{S_i} = \{W, R, R\}$ for memory location $\{A[i]|i \in [0, N-1]\}$. In this case, we have multiple consumers that needs to consume the same value produced by the same producer. One might have the solution that count for the number of its consumers N_c, and have a proxy thread with SC equals to $N_c + 1$. Wait for the information of all its consumers, and when all the consumers are created, broadcast the result. This method will block the execution of all its consumers till the last consumer is created, and works only if the number of consumers are known when proxy thread is created. In a dynamic program like Figure 6.1, we do not have the knowledge how many consumers one producer have until the entire loop is executed.

So instead of one single proxy thread for the producer, we also create one proxy thread for each consumer. This consumer proxy thread will wait for the result passed from previous proxy thread, the DF-Frame information from the

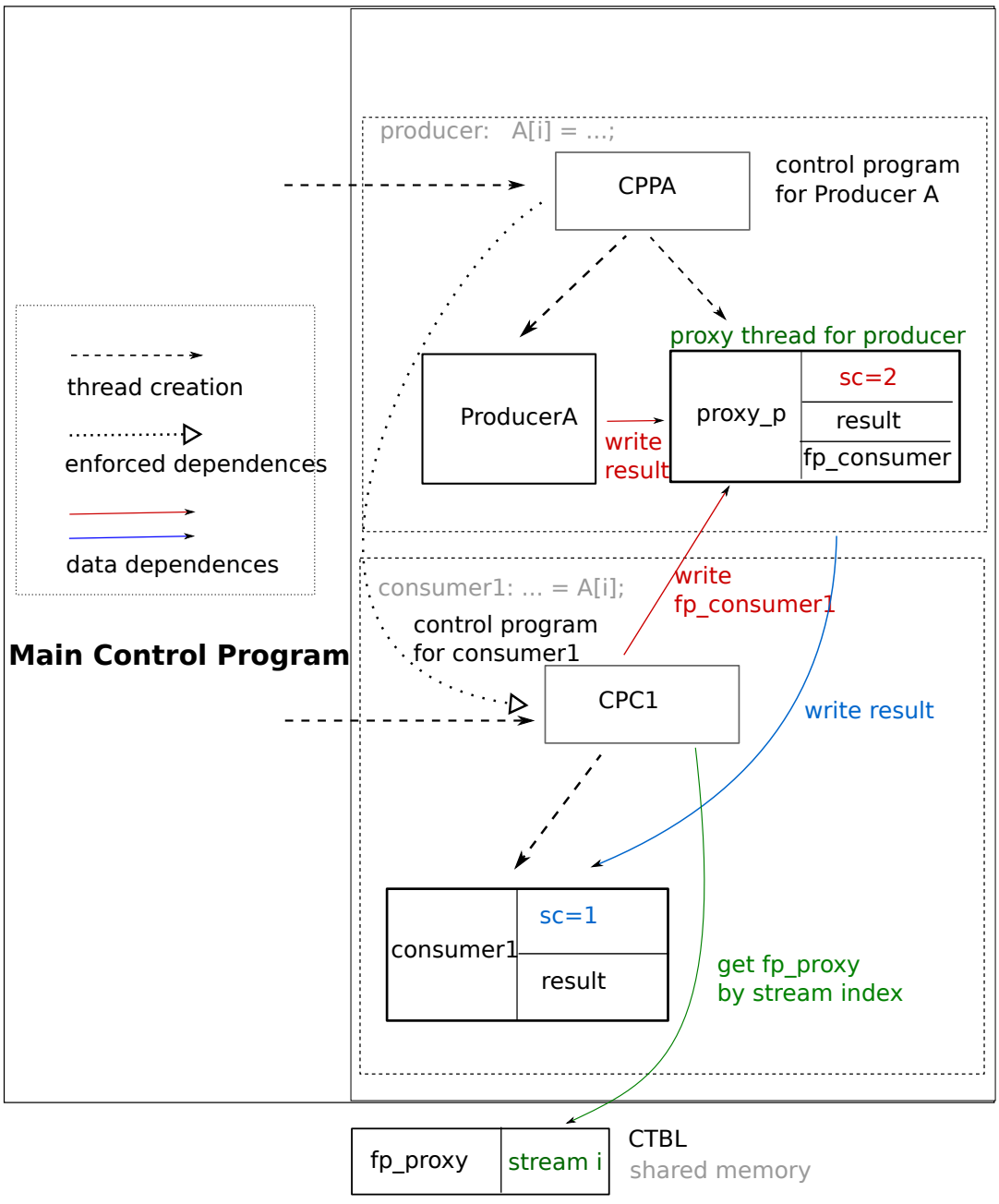

Figure 6.3: Single producer single consumer

next consumer, and next consumer's proxy thread. Once a proxy thread got those information, it will write the result to the consumer and this consumer's proxy thread.

Figure 6.4 illustrates this solution. The dotted line with white triangle represents enforced dependence in the control program: the control program for consumer 1 could only be executed after control program for producer A finishes its execution. The dashed lines represent thread creation relationship from its source to its destination, and the red and blue lines represent the data dependences.

- **CTBL.** CTBL is the check up table indexes for each memory location. CTBL resides in the shared memory, it is used by *the control program* to resolve the dependences between producers and consumers.

- **The control program.** The control program takes care of thread creation and matching producers with its consumers. For example, the control program for Producer A (`CPA`) creates the producer thread (`ProducerA`) and its proxy thread (`proxy_p`). And set `CTBL` for memory location `A[i]` to this proxy thread's DF-Frame address (`fp_proxy_p`); the control program for consumer1 (`CPC1`) creates the consumer thread (`consumer1`) and its proxy thread (`proxy_c1`), and writes the DF-Frame information of this proxy thread, and the consumer thread to the previously created producer's proxy thread (`proxy_p`). The previous proxy thread's DF-Frame address could be retrieved from `CTBL`. The same applies for `consumer2`.

- **The producer thread.** The producer thread does the computation, and stores the result to its proxy thread once it is finished (`ProducerA` in this example). Note the producer and its proxy thread are created together by `CPA`, the proxy thread's DF-Frame pointer is written to the producer's DF-Frame when both are created, so that the producer thread could restore its proxy thread at execution.

- **The proxy thread.** The proxy thread takes three inputs: one field for the result, one field for its next consumer, and one field for the next proxy thread of this consumer. Once this proxy thread gets all the inputs, it is

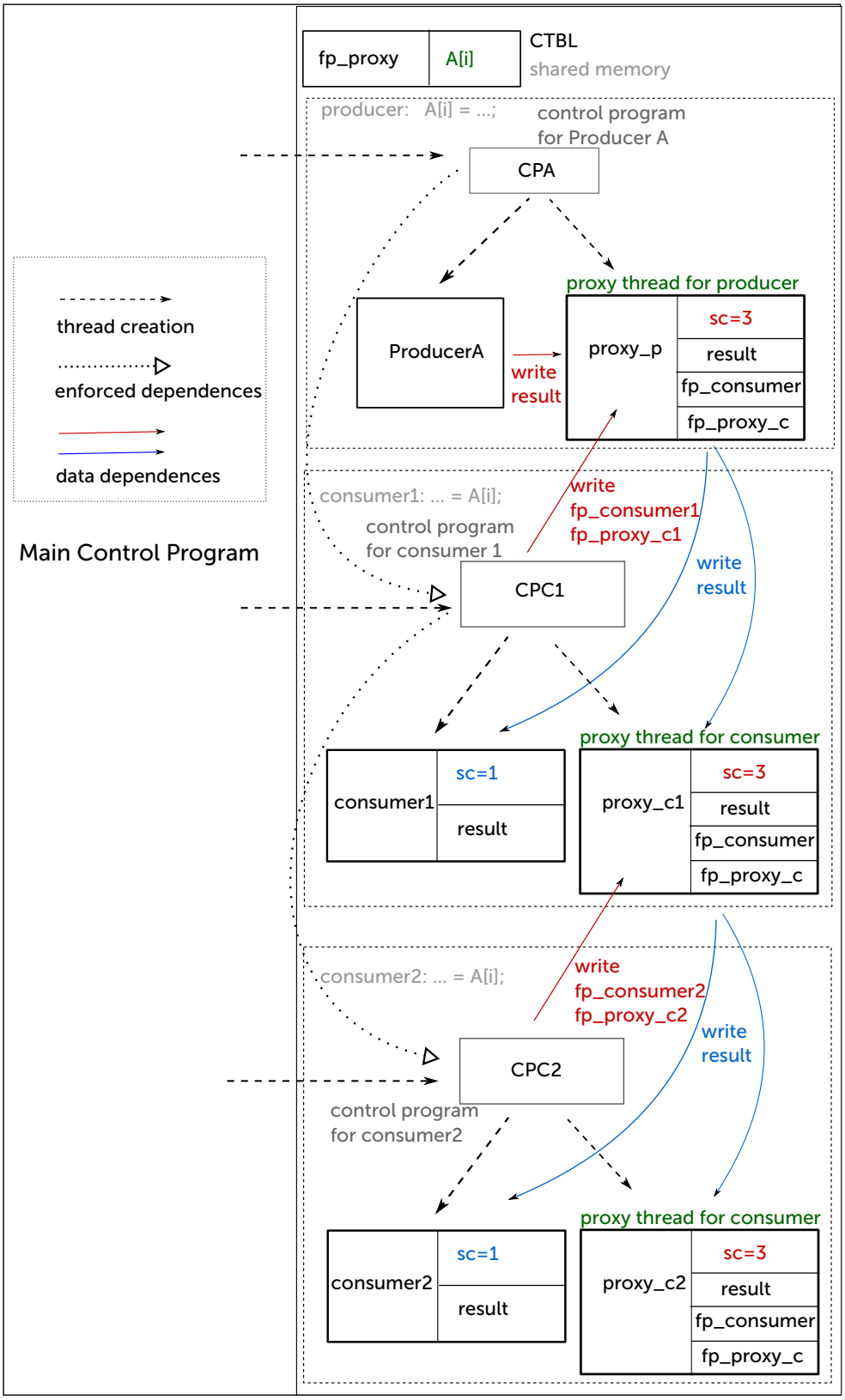

Figure 6.4: Single producer multiple consumers

ready for execution. Once being executed, it will write the result to the consumer thread and the consumer proxy thread. For example, producer A's proxy thread (`proxy_p`) have SC initialized to 3, and its dataflow frame waits for 3 inputs: `result`, its consumer thread (`fp_consumer`) and its consumer's proxy thread (`fp_proxy_c`). Both `fp_consumer` and `fp_proxy_c` are written to `proxy_p` by the control program for consumer1 (`CPC1`) at the point they are created.

- **The consumer thread.** The consumer thread takes one input for the result produced by its producer. The result is written to the consumer thread by its producer's proxy thread (`consumer1` gets result from `proxy_p`) or by previous consumer's proxy thread (`consumer2` gets its result from `proxy_c1`).

6.1.4 Multiple Producers Single Consumer

Assume we have the access order $\vec{S_i} = \{W, W, R\}$ for memory location $\{A[i]|i \in [0, N-1]\}$. In this case, we have multiple producers and one single consumer to the same memory location. In the case with access order $\vec{S_i} = \{W, W, R\}$, the first produced value should be abandoned and the second should be consumed. Since the access order is not statically decidable, at the time the first producer is created, we do not have the knowledge if another producer to the same memory location will be created right before the consumer is created.

If we follow the method as in single producer single consumer discussed before, it will still work. `CTBL` maintains a mapping from the memory location to the most recent producer's proxy thread. At the time the consumer to the same memory location is created, it looks up in `CTBL` and gets the most recent producer's proxy thread. It then writes back the consumer's dataflow thread information. But note in this way, the first producer's proxy thread will become a zombie thread, and will never be executed.

So we adapt the algorithm, once a producer is created, it lookups in the table `CTBL`, if the lookup for this its memory location is not empty, it means either a previously created producer's proxy thread (multiple consumers situation) or a consumer's proxy thread (multiple consumers situation) is already created. In

either way, both the proxy thread should be deallocated since a new `write` to the same memory location comes. Write a NULL value instead of newly created consumer thread's DF-Frame pointer, in which case, at the time the proxy thread is being executed, it could deallocate itself by checking if this value is NULL.

Figure 6.5 illustrates this solution. 2 producers for the same memory location $A[i]$ are created before the consumer. The first producer (`producer1`) is created by the control program for producer 1 (`CPP1`). `CPP1` set `CTBL` to the DF-Frame pointer of proxy thread for producer 1 (`fp_proxy_p1`). The second producer (`producer2`) is created by the control program for producer 2 (`CPP2`). `CPP2` will first lookup in the table `CTBL` for this memory location $A[i]$, and get previously created producer 1's proxy thread `fp_proxy_p1`. It will write a NULL value to this proxy thread and decrease the `SC` by 2. When `proxy_p1` is executed and get this NULL value, it will run to deallocate itself and release the resources.

6.1.5 Generated Code for Motivating Example

Figure 6.6 and Figure 6.7 presents the generated code by applying SCMD algorithm for our motivating example in Figure 6.1.

Figure 6.6 shows the dataflow tasks generated. For each producer, a proxy thread will be created, waits for the computation decoupled from the producer task, and also waits for its consumer's DF-Frame information. The producer task computes the result, and writes the result back to its proxy thread once computation is finished. e.g. `func_s1` is the producer thread, it gets the current DF-Frame information (`get_cfp()`), and loads the inputs it needs (`i` for computation and `producer_proxy` for its proxy thread's DF-Frame). Once the computation finishes, it writes the result to proxy thread's DF-Frame, and decrease the SC by 1 (`tdecrease (producer_proxy)`). The proxy thread (`func_proxy_s1`) waits not only for the result computed from the producer thread, but also waits for its consumer's information. When it gets its consumer's DF-Frame, its consumer proxy's DF-Frame and the computed result, it will be ready for execution. In case of multiple producers' case ($\vec{S_i} = \{W, W, R\}$), the NULL value is written as the consumer's DF-Frame. When the producer get a NULL value (`if (fp_consumer == NULL)`), it deallocates itself by calling `tdestroy`.

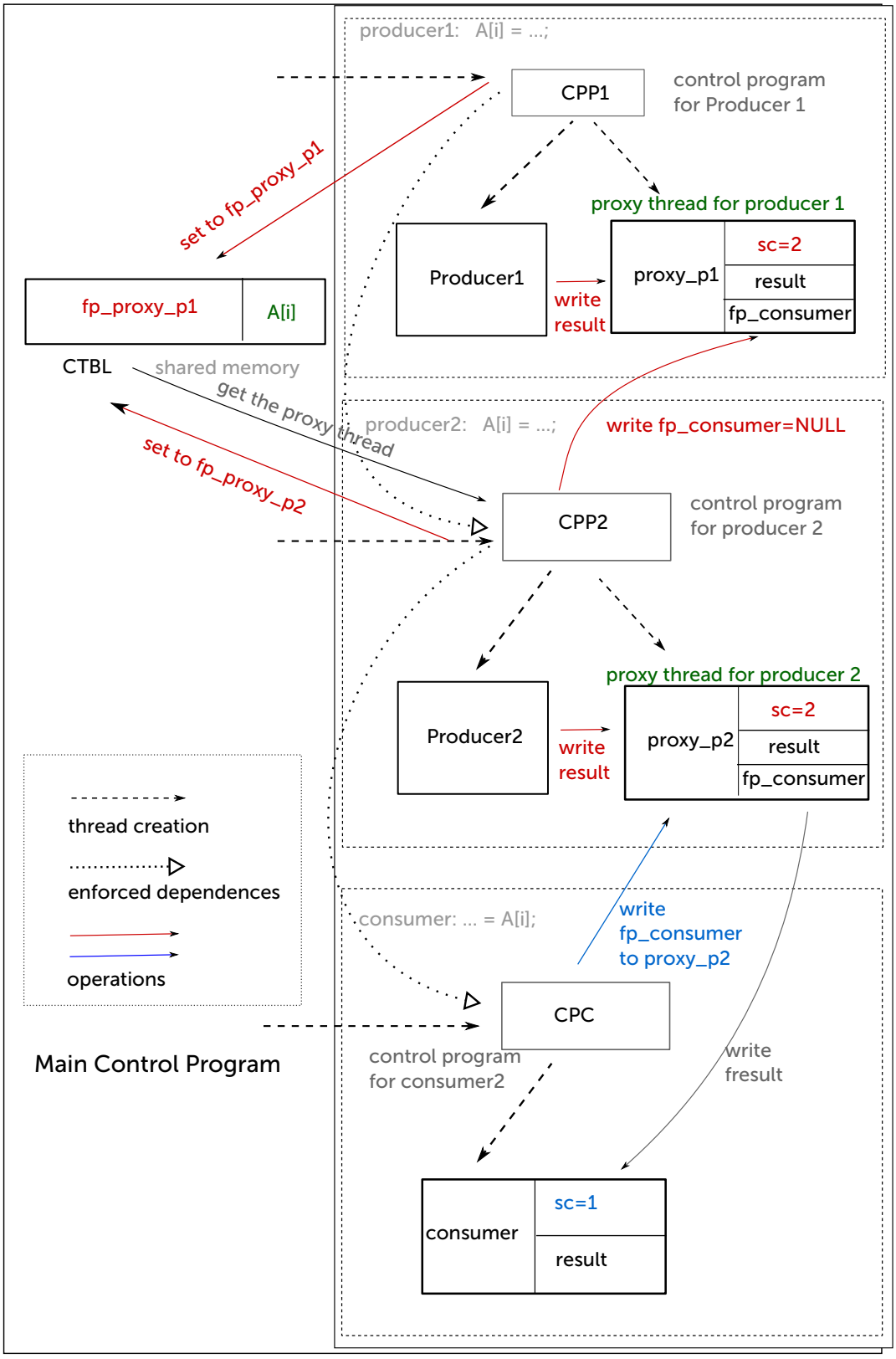

Figure 6.5: Multiple producers and single consumer

```
/* Producer thread (S1). */
void func_s1 ()
{
  current_fp = get_cfp ();
  i = current_fp->i;
  result = compute_1 (i);

  producer_proxy = current_fp->proxy;
  tdecrease (producer_proxy);
  tdestroy ();
}
/* Producer's proxy thread (S1). */
void func_proxy_s1 ()
{
  current_fp = get_cfp ();
  result = current_fp->result;

  fp_consumer = current_fp->consumer;
  /* Multiple producers case. Destroy
     this zombie proxy thread. */
  if (fp_consumer == NULL)
    tdestroy ();

  fp_consumer->result = result;
  tdecrease (fp_consumer);

  fp_consumer_proxy =
         current_fp->consumer_proxy;
  fp_consumer_proxy->result = result;
  tdecrease (fp_consumer_proxy);
  tdestroy ();
}
```

```
/* Consumer thread (S2). */
void func_s2 ()
{
  current_fp = get_cfp ();
  result = current_fp->result;
  tdestroy ();
}

/* Consumer's proxy thread (S2). */
void func_proxy_s2 ()
{
  current_fp = get_cfp ();
  result = current_fp->result;

  fp_consumer = current_fp->consumer;

  if (fp_consumer == NULL)
    tdestroy ();

  fp_consumer->result = result;
  tdecrease (fp_consumer);

  fp_consumer_proxy =
         current_fp->consumer_proxy;
  fp_consumer_proxy->result = result;
  tdecrease (fp_consumer_proxy);
  tdestroy ();
}
```

Figure 6.6: Generated code for control program using SCMD.

For each consumer, a proxy thread will also be created in multiple consumers' case ($\vec{S_i} = \{W, R, R\}$). At the time consumer task is executed, it will get the current DF-Frame, and load the result for further processing. The consumer's proxy thread (`func_proxy_s2`) takes care of passing the result computed by the producer to the next consumer in case of multiple consumers. And will be deallocated automatically when a NULL value is received.

Figure 6.7 shows the generated control program. The control program takes care of resolving all the memory dependences. We will take the control program for producer `S1` as an example. We specify each statement with `Ln`, represents

the *nth* line in the code in the following discussion.

```
1  void foo.control (){
2    init_CTBL ();
3    for (i = 0; i < N; i++){
4      // Control Program for producer S1
5      idx_m = m(i);
6      /* Create producer thread. */
7      fp_producer_s1 = tschedule (func_s1, 1, sz_p_s1);
8      /* Create producer's proxy thread. */
9      fp_producer_proxy_s1 = tschedule (func_proxy_s1, 1, sz_pp_s1);
10     /* Register the proxy thread of the producer. x */
11     fp_producer_s1->proxy = fp_producer_proxy_s1;
12     fp_producer_s1->i = i;
13     /* Schedule the producer to execute */
14     tdecrease (fp_producer_s1);
15
16     tmp = get_CTBL (idx_m);
17     fp_current_proxy = tmp.val;
18     status = tmp.status;
19     /* handle multiple producers case. Write NULL (dummy) value to
20        destroy the proxy thread. */
21     if (status != EMPTY){
22       fp_current_proxy->val = NULL;
23       tdecrease (fp_current_proxy.val);
24     }
25     set_CTBL (idx_m, fp_producer_proxy_s1);
26
27     // Control Program for consumer S2.
28     idx_g = g(i);
29     /* Create the consumer thread. */
30     fp_consumer_s2 = tschedule (func_s2, 1, sz_p_s2);
31     /* Create the consumer's proxy thread. */
32     fp_consumer_proxy_s2 = tschedule (func_proxy_s2, 1, sz_pp_s2);
33     fp_current_proxy = get_CTBL(idx_g).val;
34     /* Register both consumer and consumer's proxy in case of
35        multiple consumers. */
36     fp_current_proxy->consumer = fp_consumer_s2;
37     fp_current_proxy->consumer_proxy = fp_consumer_proxy_s2;
38     set_CTBL (idx_g, fp_consumer_proxy_s2);
39     // Control Program for producer S3. Similar to S1, omited.
40   }
41 }
```

Figure 6.7: Generated code for control program using SCMD.

`L7` creates the producer thread, and `L9` creates the producer's proxy thread. `L11` register the proxy thread's DF-Frame to the producer by writing to the pro-

ducer thread's DF-Frame. `L12` writes `i` to the producer's DF-Frame. `tdecrease (fp_producer_s1)` in `L14` decreases the SC of producer thread, schedules the producer to execute.

`get_CTBL` gets the proxy thread's information stored in $CTBL$ (`L16`). `L21` will check multiple producer's case. If it is not empty, it means either a previous producer or a consumer's proxy thread is written to `CTBL` for this memory location. To deallocate this unused producer proxy thread and release the resources it allocates, writes a NULL value so that it could deallocate itself once it is being executed. `set_CTBL` sets the $CTBL$ with memory location $A[idx_m]$ to the producer's proxy thread just created.

6.1.6 Discussion

SCMD decouples the memory access with computation automatically and connects the accesses to different memory locations with streams in a finest granularity, which could exploit maximum parallelism in the program at execution time — at the cost of thread creation. For each memory location, at least one thread will be created. As we discussed in chapter 1, we assume the cost of task creation is relatively cheap in TSTAR architecture, and the extracted parallelism and decoupled computation still benefits.

6.2 Owner Writable Memory

The Owner Writable Memory model (OWM) is used to reduce the communication overheads when complex data structures passed over threads. The name and idea origins from Prof. Ian Watson from Unversity of Manchester, our work in this chapter mainly includes the design from compilation point of view.

OWM is the global addressable memory, before a thread could write to a portion of memory, it has to claim ownership before hand. At any time point only the thread who has the ownership of the memory could write to it. When write ownership is successfully acquired, any read from another thread is not guaranteed to see consistent data. When write ownership is released, a consistent view of data must be visible to any other thread. Note the release operation could

be performed explicitly by the thread or implicitly by the model. The latter is achieved when the OWM is used by a thread to write its results, which are made available to the consumer thread upon the completion of the execution of the thread. This memory can serve the requirements of the single assignment semantics required for functional objects. However, the ability for other threads to subsequently reclaim write ownership adds to flexibility of usage.

6.2.1 OWM Protocol

For a complete description of the OWM memory model, we also include the OWM protocol here. The OWM protocol was firstly formalized by François Gindraud Gindrand *et al.* [2013], who is doing his master thesis at our lab. We give a short introduction here, and continue with our work from compilation side for OWM design in later sections.

The OWM protocol is inspired from a distributed, directory-based MSI cache coherence protocol. The global OWM memory address is mapped locally to each node on the NoC. Before a task can access to an OWM subregion, it has to claim ownership before hand. The owner will always keep track of the nodes that holds a valid copy of the subregion. One important property of resolving the ownership of a OWM subregion is handled as follows:

- The global addressable OWM memory is distributed to each nodes, we could tell the node it is allocated at the first place by the address, we call this node as its *first owner*.
- When the ownership changes, the first owner always keeps the information of the current owner. When claim ownership or data requests has been received, it forwards the requests to its owner and renew the ownership information.

One problem with the MSI is the atomicity of bus events. On the NoC, we assume all the messages will eventually arrive without packet loss or duplication, in any order. So we must ensure that a task accesses a region in `W` mode will invalidate all the copies of that region on other nodes before the tasks depends on being executed.

We assure this property by adding a memory semantic `tpublish`. When all the modifications are done within the OWM subregion, the owner task has to execute tpublish on the region explicitly to ensure all the other nodes depend on the new data will be invalidated.

Protocol description Each node on the NoC operates on two message queues, a send queue and a receive queue. Nodes communicates via messages. The message sending is abstracted as removing one message from the send queue of the source node, and add it atomically to the receive queue of the destination node. The protocol could be divided into three message types: (DataRequest, DataAnswer), (OwnerRequest, OwnerAnswer), (InvalidateRequest, InvalidateAck). We will discuss each in details.

- **Data request.** *DataRequest* and *DataAnswer* messages are equivalent to *BusRd* event in MSI. The request will be sent to the first owner of this region, and forwarded to the current owner. When the owner node receives this request, it replies with a *DataAnswer* message contains the fresh data, and add the request node to the list of valid nodes. When the request node receives the *DataAnswer*, it update the local copy of the OWM region, sets the valid flag as true, and resets the requested flag.

- **Ownership request.** *OwnerRequest* and *OwnerAnswer* are similar to the *BusM* event in MSI. In snooping MSI the bus is guaranteed that only one busM event could occur. In OWM memory model, the enforced dependences are added between tasks so that no ownership change could occur if there is another node claimed the ownership and did not publish the data yet. The request message will be sent to the first owner of this region, and will be forwarded to the current owner. The first owner will update the ownership information by checking the *OwnerRequest* message. When the destination node receive this message, it sets the valid flag to be true, and send *OwnerAnswer* which packs the data and ownership response meta data information to the new owner. When the request node receives this message, it will update the region it requests by the data received. The *valid set* information is also sent in the meta data by the previous owner,

the request node will update this information, and add the previous owner to this set.

- **Invalidation request.** Invalidation complements the ownership transfer process. We explicitly send invalidation request to other nodes that have a local copy upon modification. The *InvalidateRequest* is sent to all the nodes in the valid set. The *valid set* will be copied to *waiting invalidation acknowledge set (WIAS)* before it is reset. When the node receive a *InvalidateRequest*, it set the `valid` flag to false, and send back the *InvalidateAck* message to acknowledge the sender. When the sender receives *InvalidateAck*, it removes the source node from *WIAS*.

6.2.2 OWM Extension to TSTAR

OWM is one single memory region, but it could be further divided into smaller subregions for finer granularity. We introduce `owm_tsubscribe` and `owm_tpublish` as an extension to the TSTAR ISA for supporting OWM. One could subscribe (by calling `owm_tsubscribe`) part of OWM region to a thread, which means, before this thread is executed, the ownership of the subregion should be acquired, and ready for access. One thread could publish the modifications to the OWM region it acquired by calling `owm_tpublish`. Before the modifications are published, any read from another thread is not guaranteed to see consistent data.

OWM is a weak memory model, it is the programmer's responsibility to take care of data consistency and dependences. In Figure 6.8, `OWM subregion A` is subscribed both to `DF-thread A` and `DF-thread B`. At the time `DF-thread B` is executed, the data it sees depends on whether `owm_tpublish` in `DF-thread A` is executed. If it is executed, then it sees the modified version, otherwise, it sees the old data. If you want `DF-thread B` always sees the updated data, the programmer should add an enforced dependence from `DF-thread A` to `DF-thread B`.

Here is the detailed description for the OWM ISA:

```
void owm_tsubscribe(void *tid, int off, int offowm, int size, int mode);
```

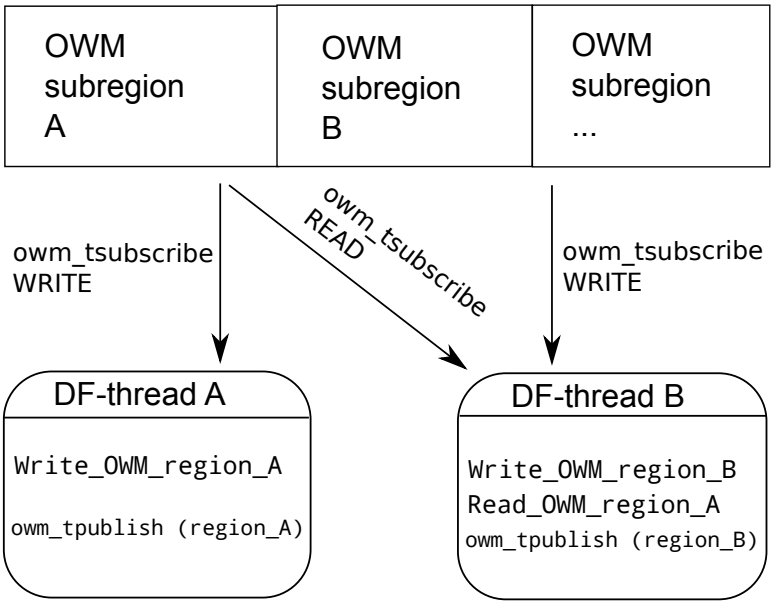

Figure 6.8: Owner Writable Memory.

Subscribe the OWM subregion described by (`offowm`, `size`, `mode`) to be cached before executing dataflow thread with thread id `tid`: `offowm` is the initial offset to the global OWM region, `size` is the size of the OWM subregion to be subscribed, `mode` describes the access mode to the region, it could be `read-only`, `write-only` or `read-write`. The pointer to the local cached OWM region is stored in DF-frame described by (`tid`, `off`), where `tid` is the thread id, and `off` is the offset in the thread's DF-frame.

```
void owm_tpublish(void *regptr, int size)
```

Publish the modification to the OWM region described by (`regptr`, `size`). If size is 0, it writes the region starting at `regptr` using the size that was registered during the `owm_subscribe` operation. This way, different threads can be subscribed to different segments of the same region using different sizes.

6.2.3 Expressiveness

OWM is integrated into OpenStream compiler as a language extension. OpenStream is a highly expressive stream-computing extension to OpenMP3.0 designed by Antoniu Pop Pop & Cohen [2011b]. One could use OpenStream to decompose programs into tasks and explicits the flow of data among them, thus exposing data, task, and pipeline parallelism.

The code in figure 6.9 shows a simple example of expressing pipeline parallelism with OpenStream. `x` is defined as a stream. The first loop produces value to stream `x` and the second loop consumes from this stream. For more information about OpenStream, interested readers could refer to Pop & Cohen [2013] and Pop & Cohen [2011b].

```
int x __attribute__((stream));

for (i = 0; i < N; ++i) {
  #pragma omp task output (x)
  x = ... ;
}
for (i = 0; i < N; ++i)
  #pragma omp task input (x)
  ... = x;
}
```

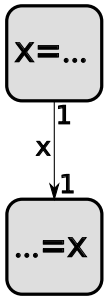

Figure 6.9: Pipeline using OpenStream.

OWM extension to OpenStream OWM extension extends OpenStream with OWM memory model, thus reduces the communication overhead when complex data structures involved. The extension is a simple `cache` pragma:

```
#pragma omp task cache (ACCESS_MODE:  MEM[OFF:SIZE])
```

The `cache` pragma subscribes the task with OWM subregion described by `MEM[off:size]` with read (`R`) ,write (`W`) or read-write (`RW`) access mode. The current pragma supports only one dimension array, but it could be easily extended to multiple dimension arrays.

The simple usage of the pragma is described in Figure 6.10. `tstar_owm_alloc` allocates the OWM memory with size `N*N*sizeof(DATA)`. `Task 1` writes to this OWM memory region and `task 2` reads from this OWM region. Note two tasks are synchronized by stream `sync`, `task 2` will only be executed when `task 1` finishes.

```
int sync __attribute__ ((stream));

DATA *A = tstar_owm_alloc (N * N * sizeof (DATA));
/* task 1. */
#pragma omp task cache (W: A[:N*N]) output (sync)
{
  for (i = 0; i < N; i++)
    A[i][i] = i;
}
/* task 2. */
#pragma omp task cache (R: A[:N*N]) input (sync)
{
  for (i = 0; i < N; i++)
    ... = A[i][i];
}
```

Figure 6.10: OpenStream cache example.

6.2.4 Case Study: Matrix Multiplication

Matrix multiplication is a good example to illustrate the expressiveness of OWM extension in user cases. We illustrate this example in three phases: in the first phase, one task allocates and initializes all the matrices in the OWM memory; in the second phase, the matrix is partitioned to several blocks, each task will cache the OWM subregion it needs and compute the results, then store the results to the output matrix; and a final task will wait till the end of all the previously created tasks, print and verify the results. We will explain in details following the path of the three phases.

Matrix allocation and initialization. Figure 6.11 shows the code for matrix allocation and initialization. The input matrices A,B and output matrix C are

allocated by calling `tstar_owm_allocate`, `fill_matrix` initialize all the matrices. The `cache` pragma subscribes matrices `A`, `B`, `C` in `write` mode. At the time `fill_matrix` is executed, all the OWM subregion it subscribes will be ready for writing. The modification will be published at the end of the task. Stream `init` is used to synchronize between phase one and phase two, so that the computation could only be started when the initialization finishes.

```
int init __attribute__((stream));

DATA *A = tstar_owm_alloc (N * N * sizeof (DATA));
DATA *B = tstar_owm_alloc (N * N * sizeof (DATA));
DATA *C = tstar_owm_alloc (N * N * sizeof (DATA));

#pragma omp task cache (W: A[:N*N], B[:N*N], C[:N*N]) output (init)
fill_matrix (A, B, C, N);
```

Figure 6.11: First phase: Matrix allocation and initialization.

Matrix multiplication. The main computations are done in this phase. Figure 6.12 shows the code for matrix multiplication. The matrix is divided into blocks, each thread caches `BLOCKSZ` rows of matrix `A`, and the entire matrix `B` in `read` mode, and `BLOCKSZ` rows of matrix `C` in `write` mode. Once the thread is executed, it computes $A_{BLOCKSZ*N} * B_{N*N} = C_{BLOCKSZ*N}$ at the end of each thread, the modification to matrix `C` is published and thus available for reading by other threads. Each task creates in this phase writes a single value to stream `finish`. Stream `finish` acts as a waiting barrier in the last task, which will wait for the termination of all threads created in this phase.

Output the results. Figure 6.13 shows the final thread, which waits for the termination of all the threads created in phase two. Once all the computations are done, it will output the results and do the verification if necessary. Stream `finish` acts as a barrier, waits for `N/BLOCKSZ` inputs from stream `finish`. Each thread created in phase two writes to stream `finish` once finished.

```
for (j = nb = 0; j < N; j += BLOCKSZ, ++nb) {
    int aoff = N * j; int boff = 0; int coff = N * j;

    #pragma omp task cache (R: A[aoff:N*BLOCKSZ], B[boff:N*N]) \
    cache (W: C[coff:N*BLOCKSZ]) output (finish)
    {
      for (int jj = j; jj < j + BLOCKSZ; ++jj) {
        for (int i = 0; i < N; i++) {
          DATA t = 0;
          for (int k = 0; k < N; k++) {
            t += A[jj * N + k] * B[k * N + i];
          }
          C[i + jj * N] = t;
        }
      }
    }
  }
```

Figure 6.12: Second phase: Matrix multiplication.

```
#pragma omp task cache (R: A[:N*N], B[:N*N], C[:N*N]) \
input (finish >> final_view[N/BLOCKSZ])
{
  dump_result_and_verify (A, B, C, N);
}
```

Figure 6.13: Third phase: Output the results.

6.2.5 Conclusion and perspective about OWM

The validation of OWM memory model is presented in Chapter 7. We have studied four benchmarks with OWM support (matrix multiplication, sparse LU, gauss seidel and viola jones), those benchmarks are valided with a system level simulator.

In this chapter, we stress a few points here:

- **The OWM memory model is a loosely coupled memory model.** Compared to word-based cache coherence, the protocol is largely simplified with the assumption that users have to synchronize all the tasks that access to the same OWM subregion to preserve the ownership atomicity. There is usually a trade off between programmability and flexibility, we shift the

complexity of the hardware design to the user, but at the same time, provide a compilation tool chain to simplify this procedure.

- **OWM extension to OpenStream provides an easy to use compilation support.** As we presented in matrix multiplication case study, the OWM extension could be easily integrated into dataflow programs, the user could use OpenStream to synchronize between tasks. In chapter 7, we present another user case where dynamic memory allocation could also be easily replaced with OWM extension with less effort. We have also integrated our backend support to OpenStream compiler, the lowered builtin functions will be translated directly to TSTAR ISA, and linked with part of the OpenStream library (runtime related with streaming operations), and part of the runtime support in our simulator. The user could just write OpenStream programs, and leave the rest for the compilation support, automatically target for our multithreaded dataflow architecture.

- **The OWM extension could be easily extended to support multiple dimension arrays.** One of the limitation of current OWM support is the lack of support on multiple dimension arrays. In the implementation of benchmarks where two dimensional arrays are used, we usually have to remap the memory regions as a single dimension array, which might have extra cost. But the OWM extension could be easily extended. An abstract polyhedral representation could be used in this case to represent an OWM region in multiple dimension arrays situation.

6.3 Summary

To complement the thread partitioning algorithm proposed in chapter 5, we presented two ways of handling complex data structures in this chapter: SCMD is used to decouple computations and memory accesses in finest granularity, which exploits as much parallelism as possible. The OWM memory model is used to reduce overhead when complex data structures are exchanged across threads. For expressiveness and as part of our compilation tool chain, we extend OpenStream

with OWM extension, the programmer could write dataflow programs with OWM support easily.

Chapter 7

Simulation on Many Nodes

Thread partitioning is a challenging task in developing a compiler for multithreaded dataflow architecture. Measuring the performance metrics on the multithreaded dataflow architecture plays an equally important role. Before the real hardware processor is built, we need a flexible methodology to analyze the behavior of the proposed architecture. By performing simulations and analyzing the results with a full-system simulator, we can gain a thorough understanding how the proposed architecture behaves, how to improve it, and validate the results before it goes into the production cycle.

Our focus in this chapter is not about the precise timing model in simulation, but the capability of simulating interesting benchmarks on thousands of cores (multiple nodes) targeting TSTAR architecture. We have rewritten and adapted a few interesting benchmarks targeting for TSTAR architecture, and simulate with manycore configuration, show its potential scalability provided a precise timing model presents.

Another focus in this chapter is the resource requirements in many nodes simulation. Multiple nodes simulation of parallel programs requires more resources than single node simulation and sequential execution. Unless precautions are taken, programs with tremendous parallelism or large number of nodes will saturate, and even deadlock, in a host machine with reasonable size.

As one of our contribution to the simulator, we analyze the resource requirements in host and guest machine, and propose our solutions which could large reduce the memory usage both in host machine and guest machine. The solutions

are implemented and integrated in COTSon simulator.

This chapter is organized as follows. Section 7.1 gives a short introduction on the COTSon simulation framework. Section 7.2 presents the many nodes simulation scenario. Section 7.3 presents the memory usage optimization and thread throttling for resource management. Section 7.4 presents our benchmarks and the results of experimental validation.

7.1 Introduction

Figure 7.1 sketches the structure of COTSon architecture. COTSon uses AMD's SimNow simulator for the functional simulation of each node in the cluster. For the timing simulation, COTSon attached a specific timing model for each component (i.e., cores, caches, memory, disks and network interfaces) of the target architecture.

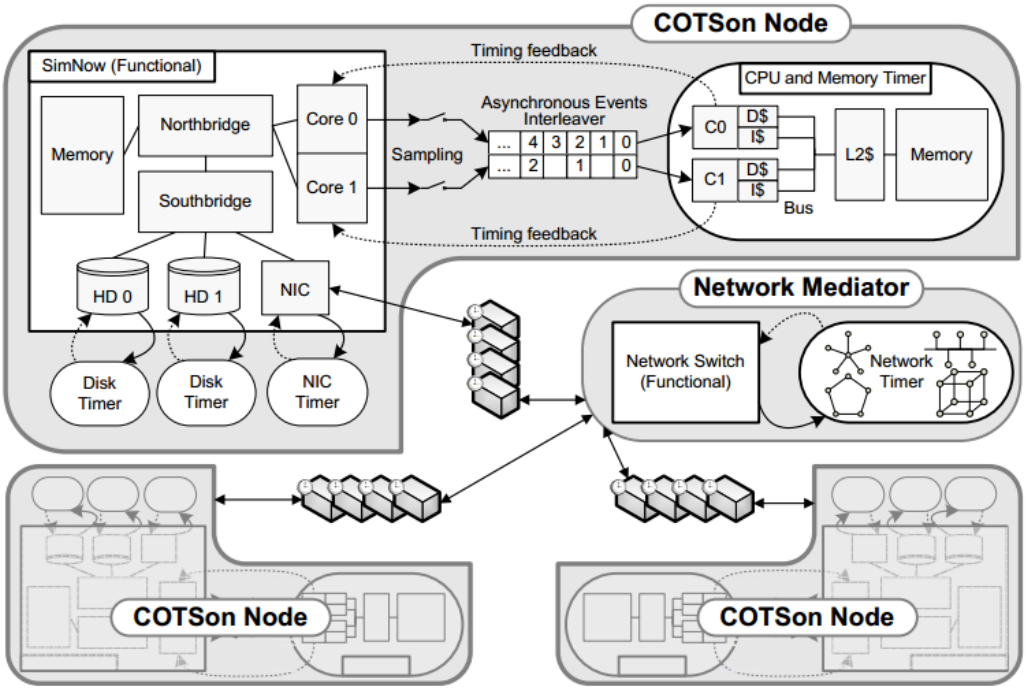

Figure 7.1: COTSon Architecture.

The support for a many-nodes simulation is given by interconnecting a large

number of COTSon nodes. Each node is composed of a certain number of cores with their hardware components. Nodes are connected with each other through their network interfaces.

When a particular application needs to communicate using the network, it executes some code that eventually reaches the NIC. This procedures a NIC event that reaches the NIC synchronous timing model. The response time from the timing model is then used by the functional simulator to schedule the emission of the packet into the external world. The packet is sent to an external entity which is called the network mediator.

7.2 Multiple Nodes Dataflow Simulation

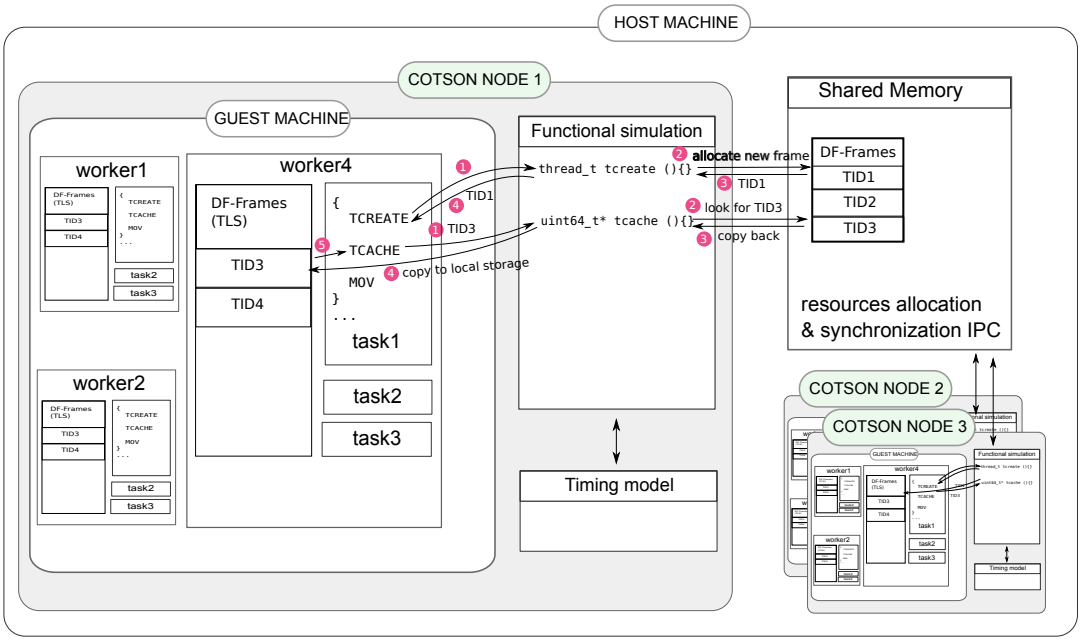

Figure 7.2: Multiple nodes simulation on COTSon.

Figure 7.2 shows the multiple nodes simulation structure on COTSon. It constitutes several components: the *host machine*, the *guest machine*, and the *COTSon nodes*.

The *host machine* is where the COTSon instances are running on. COTSon

supports multiple nodes simulation by allowing multiple instances of COTSon, the communication and synchronization of the instances go through the mediator.

The *guest machine* is the machine (both hardware and operating system level) that is simulated by COTSon instance. We create one worker for each CPU within the guest machine. Each worker will poll the centralized task queue for ready tasks.

At the execution of each task, the TSTAR instructions will be trapped by COTSon for functional simulation. In Figure 7.2, `task1` in `worker 4` (`COTSon node 1`), `TCREATE` and `TCACHE` will be trapped by COTSon, and call the registered functions `tcreate` and `tload` on COTSon node where guest machine simulated on respectively. We will describe the TSTAR instructions accordingly in the COTSon infrastructure.

TCREATE. `TCREATE` will be trapped by COTSon to the functional simulation, and then the registered function `tcreate` will be called (Figure 7.2, step 1 and 2). It will try to allocate a new DF-frame for the new DF-thread in the shared memory. If allocation is successful, the new identifier for the DF-frame (`TID1` in this case) will be returned as the result of the execution of the assemble `TCREATE`.

DF-frames in shared memory is shared by all COTSon processes, and protected with locks.

TCACHE. `TCACHE` is used to cache the remote frame locally. It will be trapped by the functional simulation, and then the registered function `tcache` will be called. The DF-frame id is passed along with `TCACHE`. In step 2, it will look up for `TID3` in the shared `DF-Frames`, if it is found, the entire DF-frame will be copied from host to guest, more precisely, the DF-frame will be copied from the shared memory to the local heap for this worker thread. And the local copy's pointer will be returned to `TCACHE` finally (step 5). Then in this task, we could directly modify/read this DF-frame. At the time `tdestroy` is called, the modifications will be synchronized and could be seen by other tasks/nodes.

TLOAD. `TLOAD` is a shortcut for `TCACHE (current_thread)`. It will be trapped by the functional simulation, and then the registered function `tload` will be called.

The current thread id is stored within thread local storage and used to get current DF-frame in the shared `DF-Frames`, if it is found, the DF-Frame will be copied from host to guest, and the local copy's pointer will be returned to `TLOAD`. Another difference between `TLOAD` and `TCACHE` is that the frame loaded by `TLOAD` is read-only. The data stored in the DF-frame is needed by the computations in the current thread.

TDECREASE. `TDECREASE` makes the target thread designated by thread id to be decremented by `n` either at the time it is called (eager tdecrease) or upon termination of the current thread (lazy tdecrease, at the time `TDESTORY` is called). It will be trapped by the functional simulation, and the registered function `tdecrease` will be called. In eager tdecrease, the target DF-frame id and `n` is passed along with `TDECREASE`. It will look up for the target DF-frame, once it is found, decrease the `SC` by `n`. Check the value `SC` after decrement, if it reaches to zero, move it to the ready queue. In lazy tdecrease, the `TDECREASE` instruction will be cached.

TDESTROY. `TDESTROY` will be trapped by the functional simulation, and call the registered function `tdestroy`. `tdestroy` will terminates the current thread and deallocates its DF-frame in Shared DF-Frames. If running on lazy mode, it will aggregate and execute the cached instructions (e.g. several `TDECREASE` to the same thread will be aggregated to a single `TDECREASE`) before deallocation.

7.3 Resource Usage Optimization

We consider two approaches to reduce resource usage at runtime. The first one optimizes memory usage irrespectively of the number of concurrently active threads. The second one aims at throttling task creation in highly parallel applications.

7.3.1 Memory Usage Optimization

The TLS corresponds to each worker in the *GUEST MACHINE* is a chunk of preallocated heap memory. At the time `TCACHE` or `TLOAD` is called, the allocator

will allocate certain memory from the TLS heap memory before the DF-frame is copied, and the heap memory will be released at the end of the thread.

Consider the code example in Figure 7.3:

```
    /* A task example */
    void task ()
S0  {
S1    current_fp = tload ();
S2    i1 = current_fp->i1;
S3    consumer_tid = current_fp->consumer_tid;
S4    consumer_fp = tcache (consumer_tid);
S5    i2 = consumer_fp->i2;
S6    consumer_fp->result = i1+i2;
S7    tdecrease (consumer_fp);
S8    tdestroy ();
    }
```

Figure 7.3: Code example for simple frame memory allocator

Figure 7.4 shows how the simple allocator works for this code example. Each worker is initialized with its TLS heap memory for caching the DF-Frames locally. In this example, `cp` points to the start of the heap (`0x80000000`). At the point (tload) (`S1`) is called, the current DF-Frame will be cached locally (copied from shared memory to TLS heap memory), `cp` will advance by the size of this DF-Frame. In the meantime, the mapping between allocated TLS DF-Frame and thread id will be inserted, in which way, another call to `tcache`/`tload` in the same thread will return cached DF-Frame directly. At the point `tcache` is called (`S4`), the DF-Frame will be cached locally with `consumer_fp`, and `cp` will advance by the size of consumer DF-Frame. The mapping between TLS and shared DF-Frame will be inserted. At the point `tdestroy` is called, the local TLS heap memory will be freed by pointing `cp` to the beginning of the heap (`0x80000000`).

The simple allocator is simple yet efficient, it operates on a task basis, at the end of each task, the TLS memory will be freed. It satisfies our needs at most of the time. However, for the control tasks, it occurs that more `tcache` involved.

The control task takes care of task creation, initialization and dependence resolution. In the dependence resolution, the consumer's thread id needs to be written to its producer after both are created. e.g. In Figure 7.5, the `control_task`

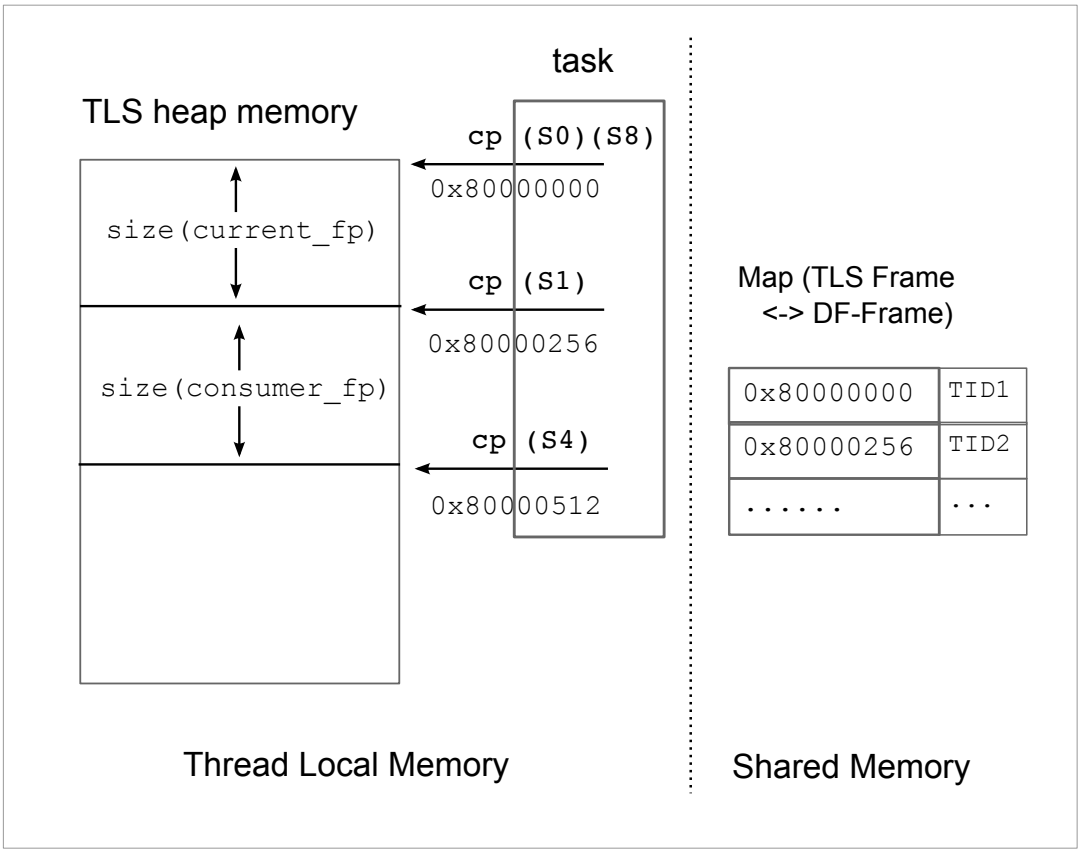

Figure 7.4: Simple frame memory allocator

creates producer and consumer tasks, it register the consumer thread id in the producer thread's DF-Frame. In order to write to a thread, it has to cache this thread locally and then operates on it. The `tcache` will be called `NUM` of times for different thread id. When the `NUM` increase, the preallocated heap will raise out of memory error.

```
   /* A task example */
   void control_task ()
S0 {
     for (i = 0; i < NUM; i++) {
S1       producer_tid = tschedule (task_p, 1, sz);
S2       consumer_tid = tschedule (task_c, 1, sz);
S3       fp = tcache (producer_tid);
S4       fp->consumer = consumer_tid;
S5       tdecrease (producer_tid);
     }
S6   tdestroy ();
   }
```

Figure 7.5: Code example for simple frame memory allocator

In order to solve the memory consumption problem in the guest machine, we will need to consider the question:

- **Where the allocator should be implemented for TLS heap memory?**

  One obvious option is to implement the allocator in the guest space, since the TLS heap memory reside in guest memory. We could add a builtin function `tls_alloc` before `tcache`/`tload` is called. And pass the returned TLS frame pointer to `tcache` as a new argument for caching it locally. And at the point where the cached frame is not used any more, insert a new builtin function `tls_free`, free the allocated memory. But this method push too much pressure either on the compiler or the programmer. The compiler or the programmer needs to decide where the `tls_alloc` or `tls_free` should be inserted.

  Another option is to manipulate the TLS heap memory from host space. `tcache` or `tload` represent the exact allocation point , then it will be

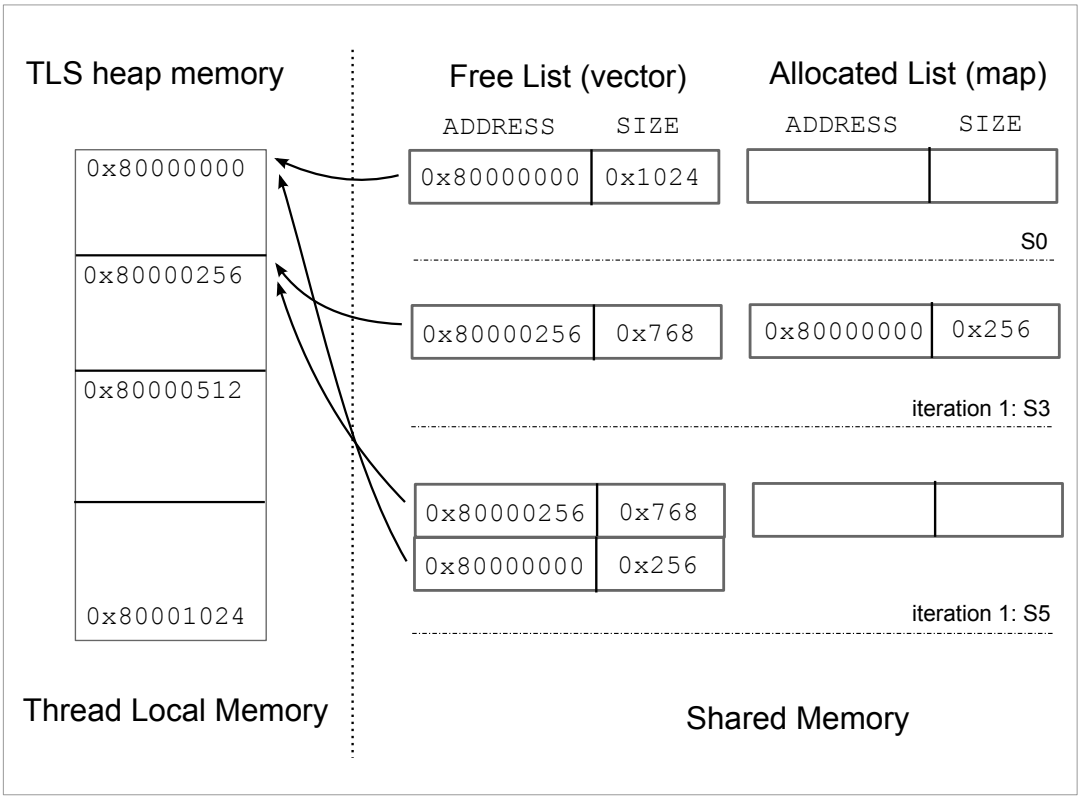

Figure 7.6: Host Space TLS Frame Memory Allocator

trapped by the functional simulation. The allocator then could allocate the memory needed and return the guest space address. But still, the `tls_free` needs to be inserted when the cached frame is no longer being used. `tlf_free` could be inserted by the compiler at the point where the cached frame will not be used anymore to free the memory, at the cost of one more instruction. Or being merged directly into `tdecrease`.

Figure 7.6 shows how the code in Figure 7.5 works with new allocator. The new allocator is implemented in host space, manipulate TLS heap memory. The free list stores a vector of currently available TLS heap memories. The `ADDRESS` and `SIZE` marks the start and the end of the free region. The allocated list stores the allocated TLS heap memories. The `ADDRESS` and `SIZE` marks the start and the end of the allocated region.

At the start of the worker, Free list will be initialized with the entire TLS heap

memory, and allocated list will be empty (`S0`). At the first iteration, `tcache (S3)` will be trapped by functional simulation, and the new allocator will be used to allocate the new TLS DF-Frame. The allocator will look in the free list, try to find the exact matched free memory location. If exact match does not exist, it will try to find the closest match (In this case, it is `0x1024`), and split the memory, move the allocated memory into the allocated list. At the point `tdecrease (S5)` is called, the allocator will free the cached TLS DF-Frame and push it back to the free list.

In the next iteration, since the cached size will always be the same, the exact match will apply with the `(ADDRESS, SIZE)` tuple `(0x80000000, 0x256)` in the free list. And it will be freed and pushed back when `tdecrease (S5)` is called. In this special case, the memory allocator will always work with any `NUM` iterations.

7.3.2 Throttling

Parallel execution of programs requires more resources and more complex resource management than sequential execution. Unless precautions are taken, programs with tremendous parallelism will saturate, and even deadlock, a machine of reasonable size. This resource problem arises in any system which allows dynamic generation of concurrent tasks, especially in dataflow architectures where the parallelism is well exploited. Ideally, enough parallelism should be exposed to fully utilize the machine on which the program is executing, while minimizing the resource requirements of the program.

A particularly nasty case occurs in certain loops, consider the example in Figure 7.7. The loop `L0` in the `control_task` executes very rapidly and all the tasks created within this loop are started at about the same time, the memory usage is proportional to N, the number of iterations. Disaster! Control the parallelism with throttling is necessary in this case.

Culler Culler & Arvind [1988] proposed the K-bounded loop technique as a dynamic software solution. The compiler analyzes the code and determines the maximum store usage for a loop cycle. At runtime, the hardware decides how many loop cycles are allowed to execute in parallel from the activity level of the machine and the static information about maximum store usage per cycle.

```
   void control_task ()
L0 {
     for (i = 0; i < N; i++) {
S1       producer_tid = tschedule (task_p, 1, sz);
S2       tdecrease (producer_tid);
     }
S3   tdestroy ();
   }
```

Figure 7.7: Excessive parallelism.

The problem with K-bounded loops is that it is not a general solution (e.g., it cannot handle recursion), and there is the overhead of the extra instructions inserted to control the loops.

Our solution is a hardware solution, trying to solve the problem of resource limitation on the frame memory, but it should also apply to other type of resources. Frame memory is used each time a new thread is created (where `tschedule` is executed), and is freed at the end of the thread when `tdestroy` is executed. We mark the thread that is going to create other tasks, where the resource usage will always increase once being executed, as `greedy`, and mark the thread that does not create tasks, as `generous`. Once a `generous` thread is executed, it assures to release the resources in the end and increase the total available resources in the entire system.

Our heuristic is simple, each time `tschedule` is executed, it will check the availability of the resources for the allocation of the thread, if available, it allocates the resources it needs and returns the allocated DF-Frame address. If not, it will try to schedule a ready task marked as `generous` and execute it right away. Once the resources occupied by this `generous` thread is released, it will check again the availability, continue scheduling `generous` threads until enough resources being released. If there are no threads marked as `generous` in the ready queue, executes `greedy` threads will only increase the demands in the resources. In this case, the program will raise exception for insufficient resources and exit.

7.4 Experimental Validation

We conducted multiple-nodes simulation on 6 benchmarks: Fibonacci, Gauss Seidel, Matrix Multiplication, Sparse LU and Viola Jones.

The benchmarks have been implemented in two different flavors. One flavor is to write programs with the low level TSTAR instruction set directly (Fibonacci and Matrix Multiplication); the other flavor uses OpenStream and compiler support to express dataflow parallelism, and has been used for the more complex benchmarks (Gauss Seidel, Sparse LU and Viola Jones). The multi-node implementation for the latter benchmarks use the OpenStream extension for OWM. The runtime support library for OpenStream (to match dependences over streams) is integrated into the COTSon runtime.

7.4.1 Experimental Settings

The experimental evaluation involved the work and support of several collaborators. Including Dr. Paolo Faraboschi from HP Labs who led the design and implementation of the TSTAR extension of COTSon, and the group of Prof. Roberto Giorgi at the University of Siena for the hardware platform and support running the simulations.

Software stack We use the `DF-proxies` branch of OpenStream compiler, where we integrated our TSTAR backend implementation and OWM support. The simulated architecture uses SimNow version `4.6.2`, and the most recent version of COTSon with support for TSTAR architecture (the TSUF branch). We also have our resource usage optimization implementation integrated in TSTAR.

Hardware stack The host machine for simulation is a DL-Proliant DL585 G7 based on AMD Opteron^{TM} 6200 Series, with 64 CPUs and 1TB RAM. The guest machine is configured using SimNow with different number of CPUs and RAM size configuration. The guest machine is running Ubuntu `9.10`.

Applications We have chosen five benchmark for evaluation: Fibonacci, Gauss Seidel, Matrix Multiplication, Sparse LU and Viola Jones. Except for Fibonacci,

all the other benchmarks are integrated with OWM support, which is used to reduce the communication overhead and enable multiple nodes support.

- **Fibonacci** is a simple benchmark which recursively compute the nth Fibonacci number. In the dataflow version, it spawns tasks recursively.
- **Gauss Seidel** is an iterative method which is being widely used to solve a linear system of equations in numerical linear algebra.
- **Matrix Multiplication** is a well known algorithms that get the product of two matrices.
- **Sparse LU** factors a matrix as the product of a lower triangular matrix and an upper triangular matrix.
- **Viola Jones** is an object detection framework proposed by Paul Viola and Michael Jones to provide competitive object detection rates in real-time.

7.4.2 Experimental Results

Figure 7.8 shows the simulation results with functional simulation. Our focus in this chapter is not about the precise timing model in simulation, but the capability of simulating interesting benchmarks on multiple nodes configuration targeting TSTAR architecture with OWM support. All benchmarks are written with OpenStream with OWM extension.

The simulation includes multiple node configurations. Each shared-memory node runs 4 cores, so in the case of the 128 cores configuration we have 32 nodes in action. Each node is a COTSon instance, simulating a guest machine with our selected operating system. The timing of all nodes are synchronized via COTSon's network mediator. And in term of the application level synchronization, one node will be selected as master node, where the other nodes are left as salve nodes. The master node initializes all the shared resources between nodes (e.g., the shared task queue) and waits for the the slave nodes to finish their initialization. Each worker will poll from the centralized task queue and schedule the task when the task is ready.

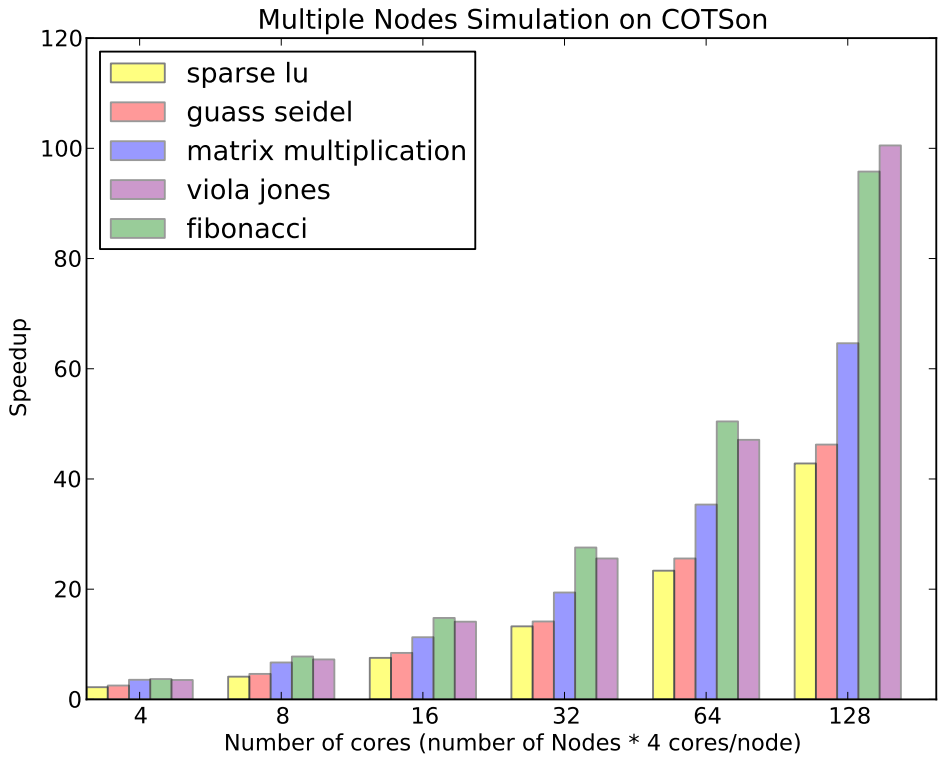

Figure 7.8: Simulation Results

The functional simulation reflects the parallelism in each benchmark (Fibonacci >Viola Jones >Matrix Multiplication >Gauss Seidel >Sparse LU). The non linear speed up shows the potential overhead in the compilation and simulation environment. The overhead mainly comes from the control program. The control program takes care of creating all the tasks and resolves dependences. OpenStream manages stream data structures through shared memory, so we have to constrain all the control programs to be scheduled on the same node.

Figure 7.9 shows the simulation results of viola jones with up to 1024 cores (tuned for the number of cores). We tuned the results by adapting the *scale* parameter in this benchmark, so that in each configuration, we have enough tasks for each worker. From the simulation results, we could see that it is fully functional in manycores simulation, and the results show quite nice parallelism opportunities.

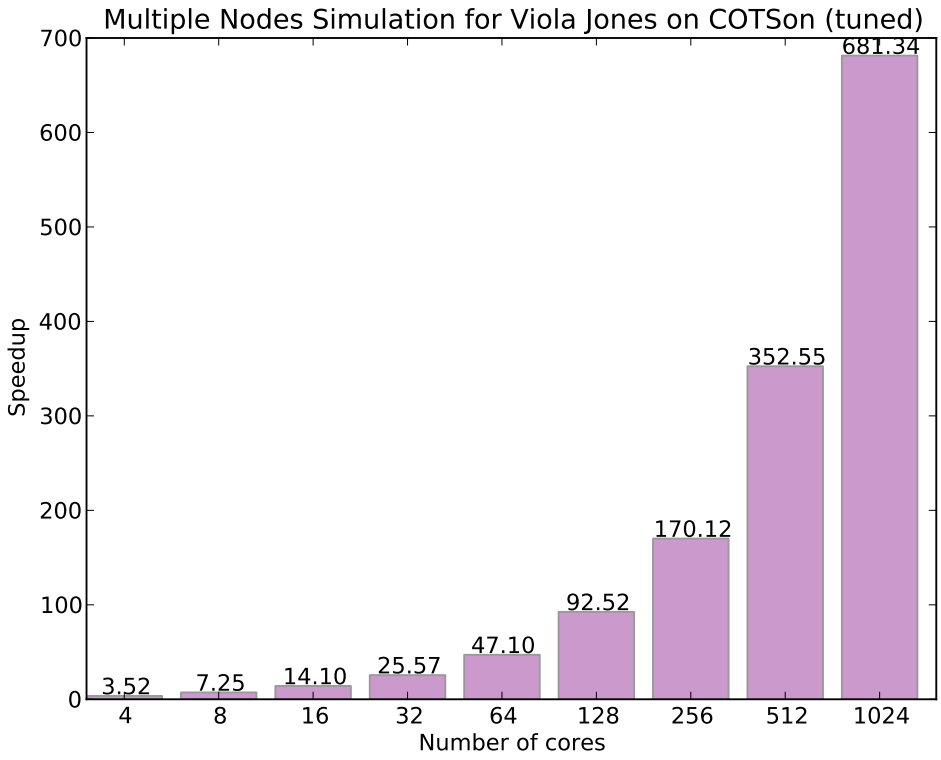

Figure 7.9: Simulation Results

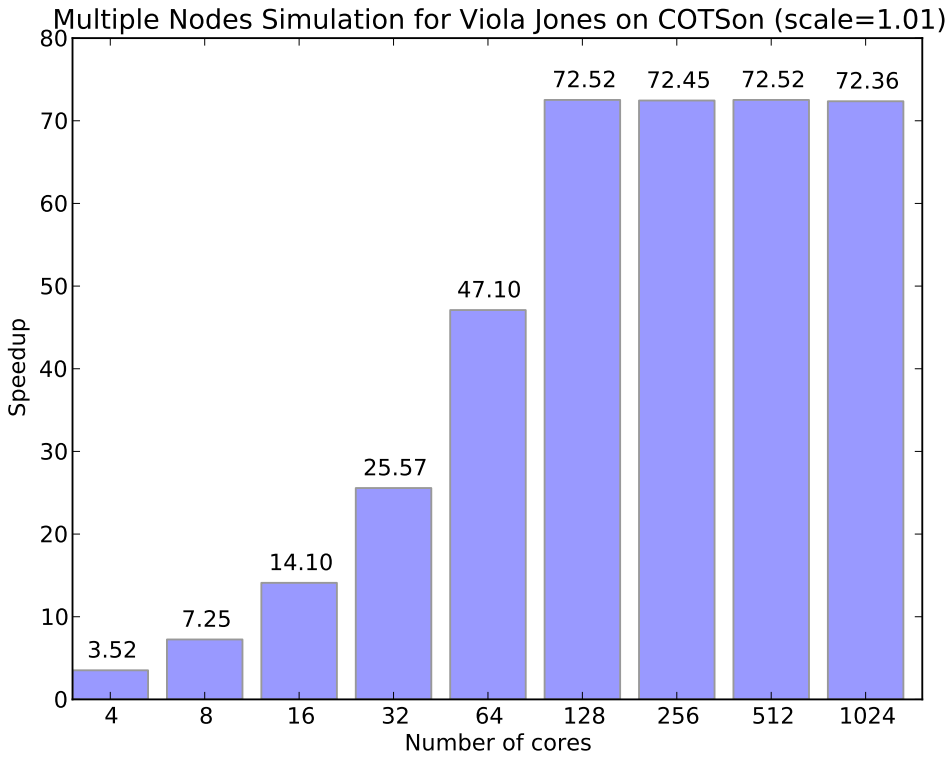

Figure 7.10: Simulation Results

Figure 7.10 shows the simulation results of viola jones with up to 1024 cores with *scale* parameter fixed ($scale = 1.01$). The problem size of the benchmark limits the parallelism in the simulation — from 128 cores, the speedup stop increasing, the total amount of tasks does not satisfy the large number of workers.

We provide a detailed analysis of each benchmark in the following subsections. (Matrix Multiplication is skipped since it was presented in chapter 6).

7.4.2.1 Gauss Seidel

Figure 7.11 shows the dependence details of Gauss Seidel. In this figure, (A) shows the dependences in the same iteration and (B) shows the cross iteration dependences. We will go through both in detail.

In Figure 7.11, every square represents a block in seidel computation. The red line represents the dependence in the same iteration, and the dashed black

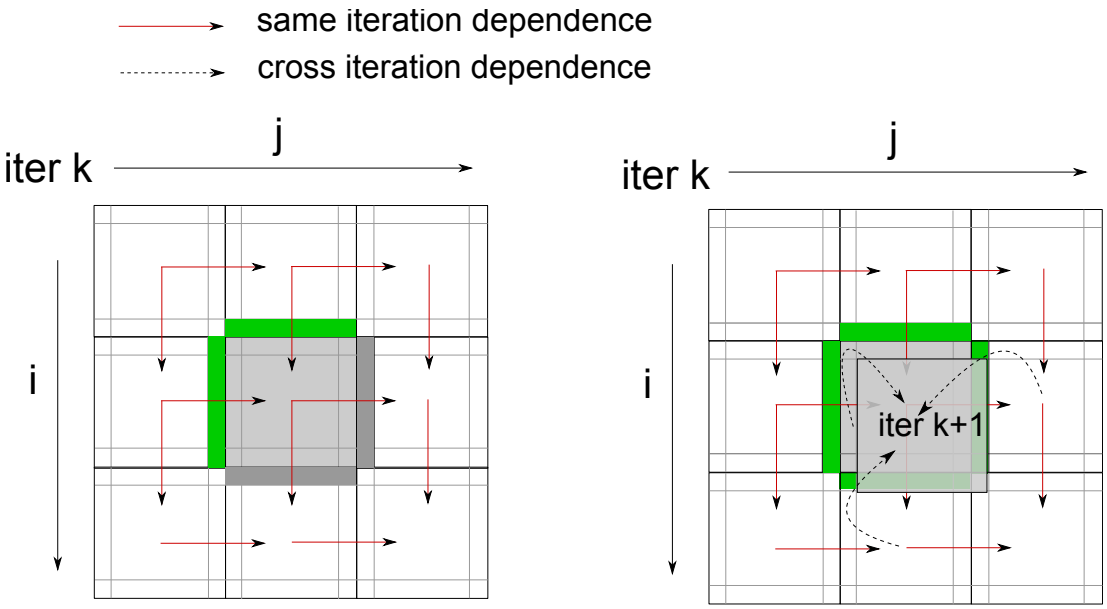

Figure 7.11: Dependence of gauss seidel: A. same iteration dependences (left) B. cross iteration dependences (right)

line represents cross iteration dependences. We take the block in the center as an example (the light grey square in left figure), the computation of this block depends on its left neighbor block and its top neighbor block. But we should note the computation of the center block does not depends on all the values of the top and the left block, just the part marked in the green rectangle. So we could actually just pass the relevant data in dataflow streams to remove the unnecessary cost of passing the entire block. The dependence from the center block to its right and bottom neighbor block should also be preserved, it needs the old value of those two blocks before they are computed in this iteration (marked as a grey rectangle).

Figure 7.12 shows our OpenStream implementation of the dependences of this center block. In this code, `sbottom_ref`, `stop_ref`, `sright_ref`, `sleft_ref`, `scenter_ref` are array of streams for the computation. e.g. in the `input` clauses, `stop[green_rect_t]` represents the stream for the computed value from its top neighbor (green rectangle) and `sbottom_ref[sbottom_ref]` represents the stream for the computed value from its bottom neighbor (grey rectangle). The input streams and output streams should be correctly indexed so that the computation

gets the correct value: the top and left rectangle should come from the computation in this iteration, the right and the bottom rectangle should come from the computation of the previous iteration. In the output clauses, when the computation has been done, the value of this center block in this iteration has been updated, the borders of this centered block will be sent to the output streams. `gauss_seidel_df` computes the center block with all its inputs and sends the results to output streams.

```
for each block
#pragma omp task \
  input(sbottom_ref[grey_rect_b] >> top_in[block_size], \
        stop_ref  [green_rect_t] >> bottom_in[block_size], \
        sright_ref [grey_rect_r] >> left_in[block_size], \
        sleft_ref [green_rect_l] >> right_in[block_size], \
        scenter_ref[lgrey] >> center_in[block_size*block_size]) \
  output(stop_ref[next_t] << top_out[block_size], \
        sbottom_ref[next_b] << bottom_out[block_size], \
        sleft_ref [next_l] << left_out[block_size], \
        sright_ref [next_r] << right_out[block_size], \
        scenter_ref[next_c] << center_out[block_size*block_size])
    {
      gauss_seidel_df(
                  blocks_x, blocks_y, block_size, numiters,
                  it, id_x, id_y,
                  left_in, top_in, bottom_in, right_in, center_in,
                  left_out, top_out, bottom_out, right_out, center_out);
    }
```

Figure 7.12: Gauss seidel implementation with OpenStream.

The dependence pattern of Gauss Seidel does not incur communication overhead, we just pass the relevant data in streams. But still, we could use OWM memory model to further reduce the communication overhead by implementing the matrix in OWM. The initialization tasks and the terminal tasks could directly write to this matrix instead of communicating via streams. Figure 7.13 shows the terminal task that directly cache the relevant block, and write the result to this matrix. In this figure, `matrix` is stored in OWM memory, and this task cache the relevant block `matrix[matrix_offset:block_elements]`, `matrix_offset` is the offset of this block and `block_elements` is the size of this block. It is cached

with *write* mode, and `gauss_seidel_df_finish` will directly write the final result of this block to `matrix`. Note the `final_stream` acts as a barrier stream, waits for all the computations are done and the final task could read results from this matrix as showed in Figure 7.14.

```
for each block
#pragma omp task
    input(sleft_ref[numiters*blocks+id] >> left_in[block_size], \
          stop_ref[numiters*blocks+id] >> top_in[block_size], \
          scenter_ref[numiters*blocks+id] >> center_in[block_elements]) \
    cache (W: matrix[matrix_offset:block_elements]) output (final_stream)
    {
      gauss_seidel_df_finish(&matrix[matrix_offset],
                                      id_x, id_y, N, block_size,
           left_in, top_in, NULL, NULL, center_in);
    }
```

Figure 7.13: Gauss seidel implementation with OWM support.

In Figure 7.14, the `final_stream` waits for all terminal tasks to finish before this task is executed. Currently, the OWM interface supports only one dimensional array, so in this seidel implementation, we have another extra step which transform back to two dimensional array. This step will be eliminated with multiple dimensional array support in the OWM interface.

```
#pragma omp task
   input (final_stream >> final_view[blocks_x * blocks_y])
   cache (R: matrix[:N*N])
   { /*use the results matrix. */}
```

Figure 7.14: Gauss seidel implementation with OWM support (final task).

7.4.2.2 Viola Jones

Viola jones is an object detection framework which provides competitive object detection rates in real-time proposed by Paul Viola and Michael Jones Viola & Jones [2001]. The first version we used is implemented by Daniel Gracia Perez from Thales Research and Technology. We have integrated OWM support in this implementation so that it could run on multi-node instances.

Figure 7.15 shows the OpenStream version of the computation intensive part in Viola Jones (we omitted some parts in this display example for simplicity), `runStageOnTile` does the main computation work. The parallelism in the code is straight forward, in both phases, we could either parallelize the code in the outermost loop (`scaleIndex`), or for finer granularity, the inner loops (`irow` or `icol`). The dependence between phase one and phase two could be synchronized via array of streams `sycn`. And in second phase, the computed information `centerX`, `centerY` and other information could be passed to the final task via dataflow streams.

```
for (scaleIndex=0; scaleIndex < scaleIndexMax, scaleIndex++){
  /* Phase one: Detection. */
#pragma omp task output (sync[scaleIndex])
  for (irow = 0; irow < nTileRows; irow += rowStep) {
    for (icol = 0; icol < nTileCols; icol += colStep) {
      for (iStage = 0; iStage < nStages; iStage++) {
        if (goodPoints[irow * width + icol]) {
          // Update goodPoints according to detection threshold
          if (runStageOnTile(imgInt_f, imgSqInt_f, irow, icol,
          tileHeight, tileWidth, height, width,
          &(cascade->stageClassifier[iStage]), scaleFactor,
          scale_correction_factor)) {
            goodPoints[irow * width + icol] = 0;
          }}}}}
  /* Phase two: Determine the position of the object detection. */
#pragma omp task input (sync[scaleIndex]) \
  output (centerX_stream << m_centerX[NB_MAX_DETECTION]) \
  output (centerY_stream << m_centerY[NB_MAX_DETECTION] \
  ...
  for (irow2 = 0; irow2 < gpRows; irow2++) {
    for (icol2 = 0; icol2 < gpCols; icol2++) {
      if (goodPoints[irow2][icol2]) {
        // Calculation of the Center of the detection
        centerX = (tileWidth - 1) * 0.5 + (icol2 * colStep);
        centerY = (tileHeight - 1) * 0.5 + (irow2 * rowStep);
      }
    }}
}
```

Figure 7.15: Viola Jones OpenStream kernel.

Viola Jones accesses large amounts of shared data, most of which are complex

data structures and *read only*. How convenient it is to integrate OWM into this benchmark is an important and interesting question. Figure 7.16 shows the data structure of `cascade`—one of many complex data structures in Viola Jones. The structure of `CvHaarClassifierCascade` is through several level of indirect references. The `cascade` is dynamically allocated from heap memory and initialized once, used read only throughout the entire program.

```
/* cascade or tree of stage classifiers */
typedef struct CvHaarClassifierCascade {
    int  count; /* number of stages */
    int orig_window_sizeR;
    int orig_window_sizeC;
    /* array of stage classifiers */
    CvHaarStageClassifier* stageClassifier;
} CvHaarClassifierCascade;
```

Figure 7.16: Cascade data structure.

In order to simplify the usage of OWM memory in this scenario, we provide customized replacements for `malloc` and `free`. One could link `malloc` and `free` in the program to the OWM version without modifications in the allocation and deallocation of those data structures.

Figure 7.17 shows this solution. We cached the OWM memory pool (`owm_mem_pool`) used for allocation in the first task. `loadImage` and `readClassifCascade` are the initializer for the image and cascade. We link `malloc` and `free` with our OWM version without any modifications in the code. The returned allocated pointer to OWM global memory address will be stored and passed between tasks through `firstprivate`. The `firstprivate` is implicit by default in OpenStream, we emphasize the point by making it explicitly. As we could see in the final code, to use OWM memory as a global addressable read only memory is very easy even in the case of dynamic allocation.

7.4.2.3 Sparse LU

The dependence pattern for Sparse LU is more restrictive in terms of parallelism exploitation, as shown in Figure 7.18. Each block in the graph represents a `read+write` block. The lines with arrows represent dependences between blocks,

```
  CvHaarClassifierCascade* cascade = NULL;
  CvHaarClassifierCascade** cascade_p = &cascade;
  Image *pgm = NULL;
  Image **pgm_p = &pgm;

#pragma omp task output (sync_stream) \
  cache (W: owm_mem_pool[:MAX_POOL_SIZE])
  {
    *pgm_p = loadImage((char *)imgName);
    *cascade_p = readClassifCascade(haarFileName);
  }

#pragma omp task input (sync_stream) \
  firstprivate (cascade_p) firstprivate (pgm_p) \
  cache (R: owm_mem_pool[:MAX_POLL_SIZE])
  {
    CvHaarClassifierCascade* cascade = *cascade_p;
    Image *pgm = *pgm_p;
    /* Usage of cascade and pgm. */
  }
```

Figure 7.17: Dynamic OWM memory allocation.

both the source and destination block will be read by the corresponding function, and when the computation is done, the results will be written to the destination block. E.g., for the red dotted line, the destination block is computed by $dest = bdiv(src, dest)$, and for the green solid line, the destination block is computed by $dest = bmod(src1, src2, dest)$.

Figure 7.18 A shows the dependences when the outer loop iteration equals 0 ($k = 0$), and inner loops executes in serial. B shows the dependences when the outer loop iteration equals to 1 ($k = 1$). Due to the dependence patterns, the next iteration of the outer loop should not start until it is previous iteration finishes. But the computation in each iteration could be executed following dataflow dependences. Part of the implementation for the dependences represented by red line in this figure is presented in Figure 7.19

One characteristic in Sparse LU is its communication pattern. Entire blocks will be passed via dataflow streams multiple times due to its dependence pattern. e.g. In Figure 7.18 A, `block(0,0)` will be passed to `block(0,1)`, `block(0,2)`,

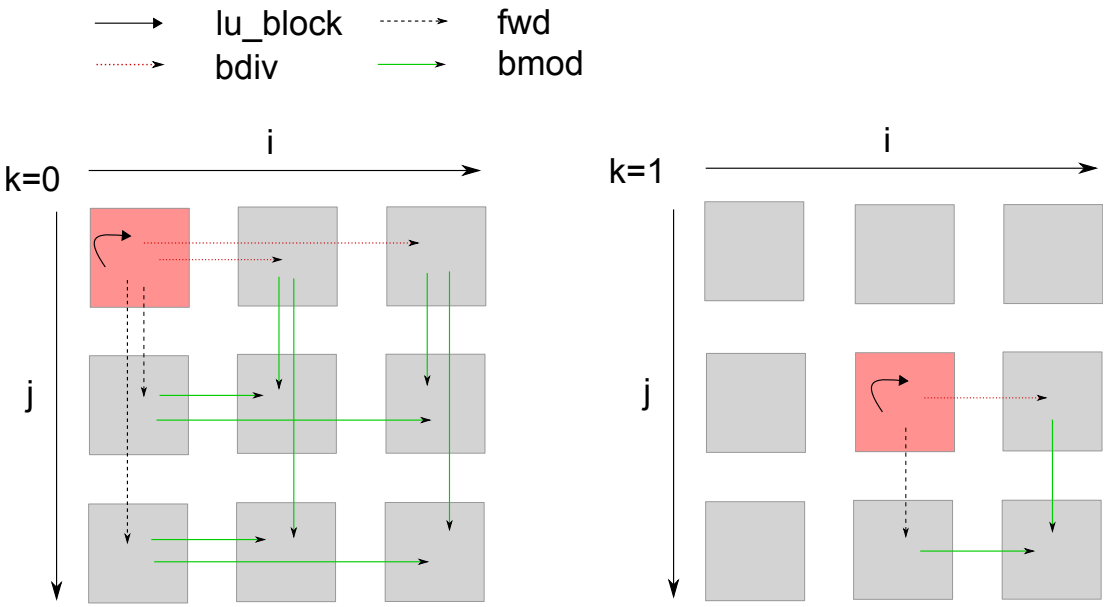

Figure 7.18: Dependence patterns of sparse lu: A. $k = 0$ B. $k = 1$.

```
for (k = 0; k < num_blocks; ++k){
  for (i = k + 1; i < num_blocks; ++i) {
    if ((*fill_flags)[i][k] != 0)
    {
      #pragma omp task \
      input  (streams[i*num_blocks + k] >> iview[bds]) \
      output (streams[i*num_blocks + k] << oview[bds]) \
      peek   (streams[k*num_blocks + k] >> iview2[bds])
      {
        bdiv (block_size, iview2, iview);
        memcpy (oview, iview, bds * sizeof (double));
      }
}}}
```

Figure 7.19: Sparse lu OpenStream implementation for `bdiv` dependences.

`block(1,0)` and `block(2,0)`. Those overheads could be reduced with OWM, where the synchronization patterns keeps the same, but we keep the matrix in OWM, so that when the dependence satisfies, the task could directly read and write to the relevant OWM region. The OWM version corresponds to Figure 7.19 is presented in Figure 7.20.

```
for (k = 0; k < num_blocks; ++k){
  for (i = k + 1; i < num_blocks; ++i) {
    if ((*fill_flags)[i][k] != 0)
    {
      #pragma omp task \
      input  (streams[i*num_blocks + k] >> iview) \
      output (streams[i*num_blocks + k] << oview) \
      peek   (streams[k*num_blocks + k] >> iview2) \
      cache (RW: matrix[(i*num_blocks+k)*bds:bds]) \
      cache (R: matrix[(k*num_blocks+k])*bds:bds)
      {
        bdiv (block_size, &matrix[(i*num_blocks+k)*bds],
        &matrix[(k*num_blocks+k)*bds]);
      }
}}}
```

Figure 7.20: Sparse lu OpenStream implementation for `bdiv` dependences (with OWM support).

7.5 Summary

This chapter covered the following:

- *Simulator* **Resource usage optimization largely reduce the memory constraints on host machine.** With the optimization applied on COTSon, the dataflow programs could free and reuse the frame memory allocated on guest machine constantly (leads to loose the memory constraints on host machine), so that the memory consumption with multiple nodes simulation becomes affordable.

- *Simulator* **Throttling enables larger size application simulatable.** Simulation scales to many cores needs larger input size for the benchmarks, the current implementation in OpenStream generate one control task that creates all the tasks in advance. The task creation speed exceeds the task execution, where the resource demands keeps stacking up. Our throttling methods could release the resources being used by scheduling the `generous` task when the resource shortage happens in allocating a new task.

- *Compiler* **OWM extension for OpenStream make it easier to write**

dataflow programs with OWM support. The OWM extension integrated into OpenStream is straightforward and easy to use for the programmer. It could be automatically lowered down into builtin functions in GCC middle end, and compiled to OWM ISA in the compiler backend, and then simulated on COTSon with TSTAR support.

- *Compiler+Simulator* **A great effort has been made in simulating OpenStream programs on COTSon.** OpenStream has made efforts with its efficient runtime system, one part of which is used for managing streams, resolving dependences; and another part is used for scheduling, managing threads creation etc. We have to replace the scheduling and threads creation part with TSTAR backend, generating TSTAR ISA, and still linking the stream management in the objective file. On COTSon, the runtime system on guest machine, that hooking as a pre main function, for workers creation, has also to be linked within the final objective file.

- *Benchmarks* **A few interesting benchmarks are simulated with OWM support.** We have written a few interesting benchmarks with OWM support, to express the parallelism and to minimize the communication overhead. Those benchmarks are compiled down to TSTAR ISA and being simulated on COTSon under multiple nodes environment (up to 32 nodes).

Note *Our focus in this chapter is not about the precise timing model in simulation, but the capability of simulating interesting benchmarks on thousands of cores targeting TSTAR architecture. But still, it will be interesting to see the results with a precise timing model provided. This thesis is related to TERAFLUX Solinas* et al. *[2013] Giorgi* et al. *[2014]* [1] *project, and the timing model is taken care of by our other partner, at the time this thesis is being written, the timing model still suffers in multiple nodes simulation (multiple nodes sync mechanism conflicting with other lock mechanism in the simulator, which results in potential deadlock issues). We will provide a simulation results with precise timing model given its available.*

[1] more information about this project could refer to http://www.teraflux.eu

Chapter 8

Conclusions and Future Work

To conclude this thesis, we review our main contributions, and then present ongoing and future work opportunities for our thread partitioning work.

8.1 Contributions

The contributions in this dissertation includes:

- A compilation algorithm extending loop transformations and Parallel Stage Decoupled Software Pipelining (PS-DSWP). By asserting a control-flow hypothesis inspired from synchronous concurrency, the data and control dependences can be decoupled naturally with only minor changes to existing algorithms that have been proposed for loop distribution and typed fusion.

- The design of the TSTAR instruction set to support our dataflow execution model, and its implementation within GCC as part of an end-to-end compilation tool chain.

- A complete algorithm that extracts dataflow parallelism from imperative programs. Our algorithm operates on a program dependence graph in static single assignment form. It exhibits task, pipeline, and data parallelism from arbitrary control flow, and allows to coarsen its granularity using a generalized form of typed fusion. A new intermediate representation — the Data Flow Program Dependence Graph (DF-PDG) — has been designed

to ease code generation for explicit token match, or feed-forward dataflow execution models. A prototype is implemented as an optimization pass in GCC.

- The design of a programming interface for Owner Writable Memory model (OWM), to support complex data structures and thread partitioning with in-place computations. The OWM extension gas been proposed and evaluated as an extension of the OpenStream research language and the hardware interface has been implemented in GCC backend.

- A hybrid compilation-time and runtime algorithm for the Streaming Conversion of Memory Dependences (SCMD), to automatically parallelize programs with complex data structures at the finest granularity. SCMD decouples the memory accesses from computations and connects the accesses to different memory locations through streams and a dynamic dependence resolution method.

- Enhancements to a simulation platform for large-scale dataflow architectures based on the COTSon framework. These include memory usage optimization for multi-node configurations, and contributions to the implementation of TSTAR in COTSon.

- Conversion of realistic benchmarks for multithreaded dataflow execution, and evaluation of these on the large-scale simulation environment with OWM support.

8.2 Future Work

Figure 8.1 shows our general attempt to address the problem described in the introduction section: "How to program a large-scale, multithreaded dataflow architecture in an efficient way?"

The attempt has been conducted in two directions in our lab: a language design to express the data, task and pipeline parallelism in a program, and an

automatic thread partitioning method to exploit finer grain thread level parallelism.

Following the first direction, Antoniu Pop designed the OpenStream research language. It is a highly expressive stream-computing extension to OpenMP3.0. Combining OpenStream with the second direction detailed in this thesis, one may decompose programs into tasks and explicit the flow of data among them, while relying on automatic parallelization techniques in the compiler and at runtime to expose finer grain data, task and pipeline parallelism.

The two directions and the associated methods are implemented in GCC as two separate passes. Our goal is to eventually combine the two paths. One could use OpenStream to identify the coarse grain parallelism in the program, and the automatic thread partitioning method will further exploit finer grain parallelism inside each task.

The final view is pictured in Figure 8.1. The OpenStream Parallel Program written by the user could be passed through the compiler front end, then in the compiler middle end, two passes will be applied : the OpenStream Compilation Pass will transform the program into coarse grain dataflow tasks communicated with dataflow streams. Each of those tasks will be further passed to Automatic Thread Partitioning Pass, which will automatically analyze each task, generate finer grain dataflow tasks. Then in the code generation pass, the compiler backend will generate code depends on user's choices, either on x86 architecture with TSTAR runtime support, or directly on TSTAR architecture, which could be simulated on COTSon with TSTAR support.

But there are still challenges in combining the two passes:

- **Intermediate Representation.** OpenStream pass uses control flow graph (CFG) as its IR, and the tasks and their dependences will be outlined by the annotation boundary. Automatic Thread Partitioning pass uses Program Dependence Graph (DF-PDG) and DataFlow Program Dependence Graph (DF-PDG) as its IR. There needs to be a consistent way of representing tasks and their dependences in both passes.

- **Communication.** The way coarse grain tasks are partitioned by OpenStream, as well as the dependences/communications channels between the

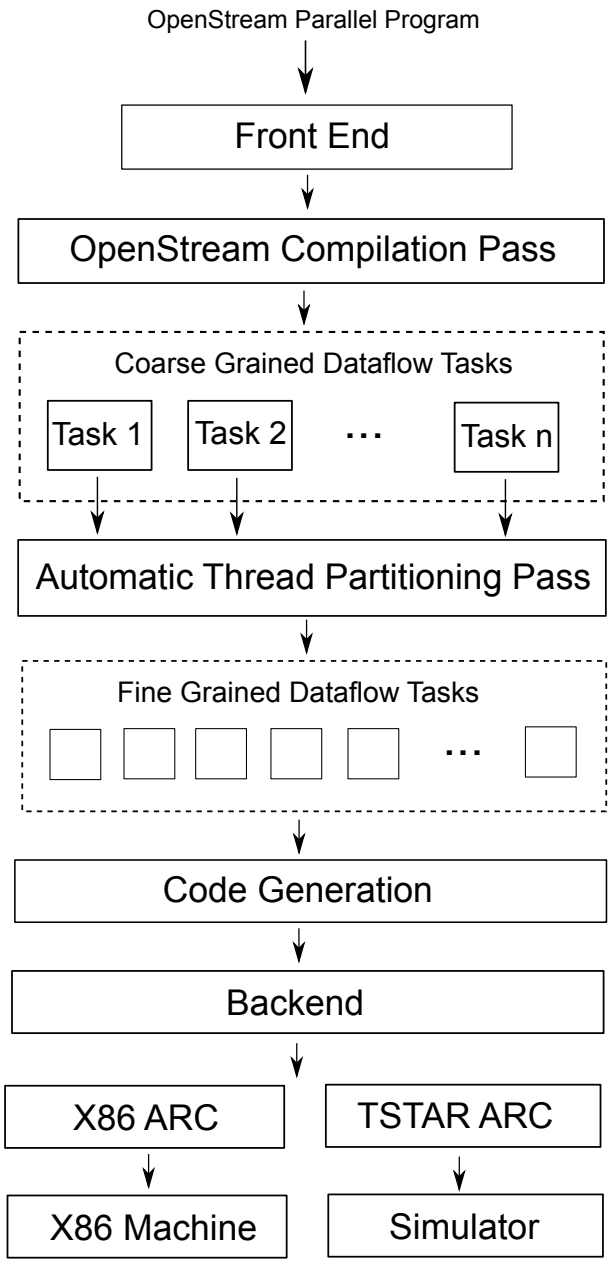

Figure 8.1: High level overview of the compilation and simulation architecture.

coarse grain tasks, should be preserved after Automatic Thread Partitioning pass. We may need an outlet input task and an outlet output task, and probably their continuations in each coarse grain task, to consume inputs from the outlet outputs from other coarse grain tasks, and to produce

outputs to outlet inputs in other coarse grain tasks.

- **Implementation.** OpenStream pass is an early pass in GCC middle end, the program is represented with GIMPLE representation. Automatic Thread Partitioning is implemented in an late pass in GCC middle end, where SSA representation is available in the GIMPLE-SSA form. How to communicate/encodes the dependences information generated at the first pass to the second pass without losing information remains a challenge.

Another interesting future work related with the timing model, as part of TERAFLUX project, the compilation framework generate code for a simulated multithreaded data-flow architecture. In order to evaluate the results more precisely, an accurate timing model needs to be integrated into COTSon. This work will be done in collaboration with partners of the project to complete the validation of the approach.

Personal Publications

1. Li, F., Pop, A. & Cohen, A. (2012b). Automatic extraction of coarse-grained data-flow threads from imperative programs. Micro, IEEE, 32, 1931.

2. Li, F., Arnoux, B. & Cohen, A. (2012a). A Compiler and Runtime System Perspective to Scalable Data-Flow Computing. In 5th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), Paris, France.

3. Li, F., Antoniu, P. & Cohen, A. (2011). Advances in Parallel-Stage Decoupled Software Pipelining. In Workshop on Intermediate Representations (WIR), Chamonix, France.

4. Solinas, M., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G.R., Garbade, A., Girbal, S., Goodman, D., Koliai, S., Li, F., Lujn, M., Morin, L., Mendelson, A.,Navarro, N., Pop, A., Trancoso, P., Ungerer, T., Valero, M.,Weis, S., Watson, I., Zuckermann, S. & Giorgi, R. (2013). The teraflux project: Exploiting the dataflow paradigm in next generation teradevices. In Euromicro DSD, Santander, Spain.

5. Trifunovic, K., Cohen, A., Razya, L. & Li, F. (2011). Elimination of memory-based dependences for loop-nest optimization and parallelization. In GCC Research Opportunities Workshop (GROW11), Chamonix, France.

6. Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jaga- sia, H., Ladelsky, R., Pop, S., Sjo din, J. & Upadrasta, R. (2010). GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In GCC Research Opportunities Workshop (GROW10), Pisa, Italie.

7. Giorgi, R., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G.R., Gayatri, R., Garbade, A., Girbal, S., Goodman, D., Khan, B., Koliai, S., L, N.M., Li, F., Lujn, M., Morin, L., Mendelson, A., Navarro, N., Patejko, T., Pop, A.,Trancoso, P., Ungerer, T., Weis, S., Watson, I., Zuckermann, S. &Valero, M. (2014). Teraflux: Harnessing dataflow in next generation teradevices. MICPRO Special Issue on European Projects in Digital Systems Design.

References

Allen, R. & Kennedy, K. (1987). Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, **9**, 491–542.

Appel, A.W. (1998). SSA is functional programming. *SIGPLAN Not.*, **33**, 17–20.

Arandi, S. & Evripidou, P. (2011). Ddm-vmc: The data-driven multithreading virtual machine for the cell processor. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, 25–34, ACM, New York, NY, USA.

Arvind & Culler, D.E. (1986a). Annual review of computer science vol. 1, 1986. chap. Dataflow Architectures, 225–253, Annual Reviews Inc., Palo Alto, CA, USA.

Arvind & Culler, D.E. (1986b). Dataflow architectures. *Annual Review of Computer Science*, **1**, 225–253.

Arvind, A. & Culler, D. (1983). The tagged token dataflow architecture (preliminary version). Tech. rep., Tech. Rep. Laboratory for Computer Science, MIT, Cambridge, MA.

Arvind, A., Gostelow, K.P. & Plouffe, W. (1977). Indeterminacy, monitors, and dataflow. *ACM SIGOPS Operating Systems Review*, **11**, 159–169.

Arvind, K. & Nikhil, R.S. (1990). Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, **39**, 300–318.

Backus, J. (1978). Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, **21**, 613–641.

Baxter, W. & Bauer, H.R., III (1989). The program dependence graph and vectorization. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, POPL '89, 1–11, ACM, New York, NY, USA.

Beck, M., Johnson, R. & Pingali, K. (1989). From control flow to dataflow. Tech. rep., Cornell University.

Böhm, C. & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, **9**, 366–371.

Culler, D.E. & Arvind (1988). Resource requirements of dataflow programs. *SIGARCH Comput. Archit. News*, **16**, 141–150.

Culler, D.E., Sah, A., Schauser, K.E., von Eicken, T. & Wawrzynek, J. (1991). Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. *SIGARCH Comput. Archit. News*, **19**, 164–175.

Cytron, R. (1986). Doacross: Beyond vectorization for multiprocessors. In *Intl. Conf. on Parallel Processing (ICPP)*, Saint Charles, IL.

Cytron, R., Ferrante, J. & Sarkar, V. (1990). Experiences using control dependence in PTRAN. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, 186–212, Pitman Publishing, London, UK, UK.

Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. & Zadeck, F.K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, **13**, 451–490.

Davis, A. & Keller, R. (1982). Data flow program graphs. *Computer*, **15**, 26–41.

Dennis, J. & Misunas, D. (1979). Data processing apparatus for highly parallel execution of stored programs. US Patent 4,153,932.

Dennis, J., Fosseen, J. & Linderman, J. (1974). Data flow schemas. In A. Ershov & V.A. Nepomniaschy, eds., *International Symposium on Theoretical Programming*, vol. 5 of *Lecture Notes in Computer Science*, 187–216, Springer Berlin Heidelberg.

Dennis, J.B. & Gao, G.R. (1988). An efficient pipelined dataflow processor architecture. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, 368–373, IEEE Computer Society Press, Los Alamitos, CA, USA.

Dennis, J.B. & Misunas, D.P. (1975). A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, ISCA '75, 126–132, ACM, New York, NY, USA.

Ferrante, J. & Mace, M. (1985). On linearizing parallel code. In *Proc. of the 12th ACM SIGACT-SIGPLAN Symp. on Principles of programming languages*, POPL '85, 179–190, ACM, New York, NY, USA.

Ferrante, J., Ottenstein, K.J. & Warren, J.D. (1987). The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, **9**, 319–349.

Ferrante, J., Mace, M. & Simons, B. (1988). Generating sequential code from parallel code. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, 582–592, ACM, New York, NY, USA.

Gajski, D., Padua, D., Kuck, D. & Kuhn, R. (1982). Second opinion on data flow machines and languages. *Journal Name: Computer*.

Gaudiot, J.L. & Wei, Y.H. (1989). Token relabeling in a tagged token dataflow architecture. *Computers, IEEE Transactions on*, **38**, 1225–1239.

Gindrand, F., Cohen, A. & Nardelli, F.Z. (2013). Definition, code generation, and formal verification of a software controlled cache coherence protocol. In *Master thesis*.

Giorgi, R., Popovic, Z. & Puzovic, N. (2007). DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems. In *SBAC-PAD*, 263–270.

Giorgi, R., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G.R., Garbade, A., Gayatri, R., Girbal, S., Goodman, D., Khan, B., Kolia, S., Landwehr, J., L, N.M., Li, F., Lujn, M., Mendelson, A., Morin, L., Navarro, N., Patejko, T., Pop, A., Trancoso, P., Ungerer, T., Watson, I., Weis, S., Zuckerman, S. & Valero, M. (2014). Teraflux: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, –.

Halbwachs, N., Caspi, P., Raymond, P. & Pilaud, D. (1991). The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, **79**, 1305–1320.

Hendren, L., Tang, X., Zhu, Y., Ghobrial, S., Gao, G., Xue, X., Cai, H. & Ouellet, P. (1997). Compiling c for the earth multithreaded architecture. *International Journal of Parallel Programming*, **25**, 305–338.

Iannucci, R.A. (1988). Toward a dataflow/von neumann hybrid architecture. *SIGARCH Comput. Archit. News*, **16**, 131–140.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In J.L. Rosenfeld, ed., *Information processing*, 471–475, North Holland, Amsterdam, Stockholm, Sweden.

Karp, R. & Miller, R. (1966). Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, **14**, 1390–1411.

Kelsey, R.A. (1995). A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, 13–22, ACM, New York, NY, USA.

Kennedy, K. & Allen, J.R. (2002). *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Kennedy, K. & McKinley, K.S. (1990). Loop distribution with arbitrary control flow. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, 407–416, IEEE Computer Society Press, Los Alamitos, CA, USA.

Kennedy, K. & Mckinley, K.S. (1993). Typed fusion with applications to parallel and sequential code generation. Tech. rep., Rice University.

Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B. & Wolfe, M. (1981). Dependence graphs and compiler optimizations. In *ACM SIGPLAN-SIGACT symp. on Princ. of Prog. Lang.*, POPL '81, 207–218, ACM, New York, NY, USA.

Lee, B. & Hurson, A. (1994). Dataflow architectures and multithreading. *Computer*, **27**, 27–39.

Li, F., Antoniu, P. & Cohen, A. (2011). Advances in Parallel-Stage Decoupled Software Pipelining. In *Workshop on Intermediate Representations (WIR)*, Chamonix, France.

Li, F., Arnoux, B. & Cohen, A. (2012a). A Compiler and Runtime System Perspective to Scalable Data-Flow Computing. In *5th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, Paris, France.

Li, F., Pop, A. & Cohen, A. (2012b). Automatic extraction of coarse-grained data-flow threads from imperative programs. *Micro, IEEE*, **32**, 19–31.

Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J. & Torrellas, J. (2006). Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, 158–167, ACM, New York, NY, USA.

Maquelin, O., Gao, G.R., Hum, H.H.J., Theobald, K.B. & Tian, X.M. (1996). Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, 179–188, ACM, New York, NY, USA.

Memo, C.S.C. & Culler, D.E. (1983). Tagged token dataflow architecture.

Najjar, W., Roh, L. & Wim Bhm, A. (1994). An evaluation of medium-grain dataflow code. *Intl. J. of Parallel Programming*, **22**, 209–242, 10.1007/BF02577733.

Nikhil, R.S. (1989). Can dataflow subsume von neumann computing? *SIGARCH Comput. Archit. News*, **17**, 262–272.

Nikhil, R.S., Papadopoulos, G.M. & Arvind (1992). T: A multithreaded massively parallel architecture. *SIGARCH Comput. Archit. News*, **20**, 156–167.

Nuzman, D. & Henderson, R. (2006). Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, 281–294, IEEE Computer Society, Washington, DC, USA.

Nuzman, D., Rosen, I. & Zaks, A. (2006). Auto-vectorization of interleaved data for simd. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, 132–143, ACM, New York, NY, USA.

Ottenstein, K.J., Ballance, R.A. & MacCabe, A.B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*, PLDI '90, 257–271, ACM, New York, NY, USA.

Ottoni, G., Rangan, R., Stoler, A. & August, D.I. (2005). Automatic Thread Extraction with Decoupled Software Pipelining. In *IEEE/ACM Intl. Symp. on Microarchitecture*, vol. 0, 105–118, IEEE Computer Society, Los Alamitos, CA, USA.

Papadopoulos, G.M. & Culler, D.E. (1990). Monsoon: An explicit token-store architecture. *SIGARCH Comput. Archit. News*, **18**, 82–91.

Papadopoulos, G.M. & Traub, K.R. (1991). Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, 342–351, ACM, New York, NY, USA.

Planas, J., Badia, R.M., Ayguadé, E. & Labarta, J. (2009). Hierarchical Task-Based Programming With StarSs. *Intl. J. on High Performance Computing Architecture*, **23**, 284–299.

Pop, A. & Cohen, A. (2011a). A Stream-Comptuting Extension to OpenMP. In *Proc. of the 4th Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*.

Pop, A. & Cohen, A. (2011b). A Stream-Computing Extension to OpenMP. In *Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*.

Pop, A. & Cohen, A. (2013). Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, **9**, 53:1–53:25.

Pop, A., Pop, S. & Sjödin, J. (2009). Automatic Streamization in GCC. In *GCC Developer's Summit*, Montreal, Quebec.

Portero, A., Yu, Z. & Giorgi, R. (2011). T-Star (T*): An x86-64 ISA extension to support thread execution on many cores. In *HiPEAC ACACES-2011*, 277–280, Fiuggi, Italy.

Raman, E., Ottoni, G., Raman, A., Bridges, M.J. & August, D.I. (2008). Parallel-stage decoupled software pipelining. In *Proc. of the 6th annual IEEE/ACM Intl. Symp. on Code Generation and Optimization*, CGO '08, 114–123, ACM, New York, NY, USA.

Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K. & Torrellas, J. (2005). Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, 179–188, ACM, New York, NY, USA.

Roh, L. & Najjar, W.A. (1995). Design of storage hierarchy in multithreaded architectures. In *Proceedings of the 28th annual international symposium on Microarchitecture*, 271–278, IEEE Computer Society Press.

Sarkar, V. (1989). *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA.

Solinas, M., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G.R., Garbade, A., Girbal, S., Goodman, D., Koliai, S., Li, F., Lujn, M., Morin, L., Mendelson, A., Navarro, N., Pop, A., Trancoso, P., Ungerer, T., Valero, M., Weis, S., Watson, I., Zuckermann, S. & Giorgi, R. (2013). The teraflux project: Exploiting the dataflow paradigm in next generation teradevices. In *Euromicro DSD*, Santander, Spain.

Stavrou, K., Nikolaides, M., Pavlou, D., Arandi, S., Evripidou, P. & Trancoso, P. (2008). TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *Intl. Conf. on Parallel Processing (ICPP'08)*, 25–34, Portland, Oregon.

Strohschneider, J. & Waldschmidt, K. (1994). Adarc: A fine grain dataflow architecture with associative communication network. In *EUROMICRO 94. System Architecture and Integration. Proceedings of the 20th EUROMICRO Conference.*, 445–450, IEEE.

Tang, X. & Gao, G.R. (1999). Automatically partitioning threads for multithreaded architectures. *Journal of Parallel and Distributed Computing*, **58**, 159 – 189.

Tang, X., Wang, J., Theobald, K.B. & Gao, G.R. (1997). Thread partitioning and scheduling based on cost model. In *Proceedings of the Ninth An-*

nual ACM Symposium on Parallel Algorithms and Architectures, SPAA '97, 272–281, ACM, New York, NY, USA.

Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J. & Upadrasta, R. (2010). GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italie.

Trifunovic, K., Cohen, A., Razya, L. & Li, F. (2011). Elimination of memory-based dependences for loop-nest optimization and parallelization. In *GCC Research Opportunities Workshop (GROW'11)*, Chamonix, France.

Tu, P. & Padua, D. (1995). Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of the 9th Intl. Conf. on Supercomputing*, ICS '95, 414–423, ACM, New York, NY, USA.

Veen, A.H. (1986). Dataflow machine architecture. *ACM Comput. Surv.*, **18**, 365–396.

Viola, P. & Jones, M. (2001). Robust real-time object detection. In *International Journal of Computer Vision*.

Watson, I. & Gurd, J. (1979). A prototype data flow computer with token labelling. *Managing Requirements Knowledge, International Workshop on*, **0**, 623.

Wolfe, M.J. (1995). *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.