Global Instruction Scheduling for Multi-Threaded Architectures

Guilherme de Lima Ottoni

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Advisor: David I. August

September 2008

Abstract

Recently, the microprocessor industry has moved toward multi-core or chip multiprocessor (CMP) designs as a means of utilizing the increasing transistor counts in the face of physical and micro-architectural limitations. Despite this move, CMPs do not directly improve the performance of single-threaded codes, a characteristic of most applications. In effect, the move to CMPs has shifted even more the task of improving performance from the hardware to the software.

Since developing parallel applications has long been recognized as significantly harder than developing sequential ones, it is very desirable to have automatic tools to extract thread-level parallelism (TLP) from sequential applications. Unfortunately, automatic parallelization has only been successful in the restricted domains of scientific and data-parallel applications, which usually have regular array-based memory accesses and little control flow. In order to support parallelization of general-purpose applications, computer architects have proposed CMPs with light-weight, fine-grained (scalar) communication mechanisms. Despite such support, most existing multi-threading compilation techniques have generally demonstrated little effectiveness in extracting parallelism from non-scientific applications. The main reason for this is that such techniques are mostly restricted to extracting parallelism within local, straight-line regions of code. As observed in several limit studies, local regions of code contain limited parallelism, and control dependence analysis and multiple control units are necessary to extract most of the parallelism opportunities.

This thesis describes a general compiler framework for Global Multi-Threaded (GMT) instruction scheduling, i.e. to simultaneously schedule instructions from a global region of code to extract TLP for multi-threaded architectures. Our compiler framework is based on a Program Dependence Graph (PDG) representation, efficient graph partitioning algorithms, and novel multi-threaded code generation algorithms. Central to this framework are our multi-threaded code generation algorithms, which produce efficient code for arbitrary partitions of the PDG into threads. Based on this framework, three thread-partitioning

strategies for GMT instructions scheduling are proposed. The first one, called GREMIO, extends list-scheduling to target multi-threaded architectures and to operate on global code regions. The second technique, called Decoupled Software Pipelining (DSWP), extracts pipelined TLP from arbitrary loop nests. We also propose Parallel-Stage DSWP, an extension of DSWP that allows multiple threads to concurrently execute the same stage of the pipeline. These techniques are implemented in the VELOCITY compiler and evaluated on an accurate CMP simulator built on top of validated Itanium 2 core models. The experiments show that our techniques balance applicability and scalability differently, with each technique resulting in the best speedup in different scenarios. Overall, the results demonstrate the effectiveness of the proposed compilation techniques, with significant speedups on a number of real benchmark applications written in C.

Acknowledgments

First, I thank my advisor, David, for making this work possible in various ways. I thank David for the liberty of picking my research topic, and for believing in me throughout this journey. David's leadership inside the Liberty Research Group also enabled the good research infrastructure that made this work possible. His critical feedback certainly helped both shape and improve this work. I also appreciate his help to improve my English and oral-presentation skills. I also thank David for the financial support for my studies, which allowed me to focus on my research. Finally, I thank David for motivating me to work hard and to continuously try to improve the quality of this work.

This work would also be hardly possible without the support from my fellow Liberators. The VELOCITY compiler was written by a group of students, including myself, Spyridon Triantafyllis, Matthew Bridges, Neil Vachharajani, Easwaran Raman, and others. I am very grateful to these colleagues for helping build such a nice infrastructure to do compiler research. The highly accurate multiprocessor simulator used in this thesis' experiments was built by Ram Rangan, based on a validated uniprocessor simulator written by David Penry. The work in this thesis was partly motivated by an earlier work by Ram Rangan, Neil Vachharajani, and David August on manually parallelizing loops that traverse recursive data structures. The IMPACT compiler from the University of Illinois at Urbana-Champaign provided not only the front-end for our compiler, but also a credible infrastructure in which I originally implemented several of the ideas in this thesis. I also thank the various Liberators for listening to my research ideas, reading my paper drafts, and providing me with invaluable feedback. I am also grateful to the Liberators with whom I had the opportunity to exchange ideas and coauthor papers. I am particularly indebted to Thomas Jablin who, as the only English native speaker in my office for a while, I constantly bugged close to paper deadlines. Thomas also generously proofread this dissertation. Finally, I am indebted with the Liberators responsible for maintaining our computer systems, including Neil Vachharajani, Matthew Bridges, Thomas Jablin, and Arun Raman. Without their help and the Condor job-management system, the 100,000-hour simulation results presented in this thesis would not be possible.

I thank the members of my thesis committee, David August, Vivek Sarkar, Sharad Malik, Margaret Martonosi, and David Walker, for taking the time to participate in my committee. Their feedback have definitely improved the quality of this work. Particularly, I thank the readers in my committee, Vivek Sarkar and Sharad Malik. Vivek Sarkar provided me with very qualified and detailed comments on various technical aspects of this work. His large experience in this field also helped me better understand the relations and contributions of my work compared to previous research in this area. I thank Sharad Malik for reading this dissertation in the short period that was left to him.

I also thank the entire staff of Princeton, and of the Department of Computer Science in particular. Their professionalism really makes this such a great place to study and to do research. I especially thank Melissa Lawson who, as the highly effective Graduate Coordinator, makes all the bureaucracy so simple and allows us to completely focus on our studies.

My studies at Princeton have been financially supported by grants from Intel Corporation, National Science Foundation, and Gigascale Systems Research Center, graduate fellowships from the Intel Foundation and Princeton's School of Engineering and Applied Science (SEAS), and special awards from the Department of Computer Science and SEAS. I also thank Intel for the opportunities to do two Summer internships, which provided me not only great industrial research experience, but also funds to survive the grad-student life.

I am also indebted to the Brazilian people, who support the good public (free) educational system in Brazil. Without the background acquired at Juvenal Miller, CTI-FURG, FURG, and UNICAMP, I would not have reached this far.

I am very fortunate to have been born an raised in an extremely happy and fun family. The distance from my parents, Elias and Lúcia, and my siblings, Gustavo, Tatiana, and Frederico, definitely made my move to the United States harder. Fortunately, the Internet and Skype enabled our long weekend conversations that virtually reduced the distance between us and made me, for a couple of hours a week, feel as if I was back in Brazil.

Keeping sane while going through a Ph.D. program is probably impossible without the company of friends to help one relax and forget the work for a few hours a week. For that, I thank many friends, especially Desirée, Luis Fernando, Diogo, Letícia, Michel, Adriana, Diego, Marcio, Juliana, Renato, and others. Through some bottles of wine or a good *churrasco* (Brazilian barbecue), and speaking *um bom português (ou "gauchês")*, they helped me get the necessary breaks from my research. I also thank some of these friends and the rest of the guys who appeared for our joyful weekly *peladas* (soccer games) at Rutgers.

Finally, and most importantly, I thank my wife, Desirée. Her love, care, understanding, incentive, and company were invaluable to give me strength throughout not only the Ph.D. journey, but most of my life so far. I deeply thank her understanding of one of my lifetime dreams, which was to study in one of the best schools in the world. Her love and understanding made my dream possible, despite all the changes, uncertainties, challenges, and difficulties that this life's choice brought to both of us. I also thank her for the strength to face adverse real-life situations, for our two little angels, and for our hopefully coming baby. *Mori, eu te amo muito!!!*

"The difficulties were created to be overcome, and we must never give up until we are absolutely certain that we are incapable."

Sebastião P. Ottoni

For my precious wife Desirée

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since their invention, computers have been used for an increasing number of applications. Today, these uses range from embedded applications (e.g. car and aircraft control, voice processing in cell phones) to desktop applications (e.g. Internet browsing, gaming) to supercomputer applications (e.g. weather forecast, seismic simulation). This wide adoption of computers is a result of both their constant cost reduction and their continuous increase in performance. This work focuses on improving the performance of applications on modern computers.

For the vast majority of the applications, performance improvements have mainly been a result of two factors. First, and most importantly, processors have become faster due to improvements in the process technology. The clock rate of processors has improved from 5 KHz in the 1940s (ENIAC [111]) to 3 GHz in 2000s (Pentium 4 [45]). The second factor for increase in performance is the constant improvement in the processors' micro-architecture (e.g. caches and branch prediction) and accompanying compiler technology. Together, the effect of micro-architectural and compilation improvements is estimated to account for 40% increase in performance per year [10]. Besides these two main factors, in limited application domains that require extraordinary computing power and contain abundant parallelism (e.g. physical simulations), supercomputers consisting of multiple processors have been used to provide extra performance.

This long-lasting scenario for improving performance is now changing. Recently, the microprocessor industry has reached hard physical limits, including heat dissipation and power consumption, that are preventing faster clock rates. Furthermore, more aggressive micro-architectural techniques adversely affect these physical issues, both by requiring more complex, power-hungry hardware components and by adding logic to the critical path to execute instructions. Moreover, micro-architectural techniques generally provide diminishing returns. Therefore, neither clock rate nor micro-architectural techniques are contributing to significantly improve performance anymore. As a result, the performance of most applications is not increasing at the same pace it used to for several decades. Figure 1.1 illustrates this effect on three generations of the SPEC CPU Integer benchmark suite [99]. Each point in this graph represents a different machine. For each month in the x-axis, the best performance among the machines reported during this month is plotted. The different benchmark suites were normalized using common processors reported. The two different trend lines are linear regressions. The steepest line was computed using the processors released before 2004, and it represents the past performance trend. The other line was computed for processors released starting in 2004, and it represents the current performance trend.

The current reduction in the performance trend is a result of the aforementioned limitations in both clock frequency and micro-architecture. Since these were the two main sources of increase in performance, improving performance now lies on parallelism, which was the third and far less important source of performance for decades.

The good news is that parallelism is becoming cheaper. With Moore's Law's continuous increase in transistor count per chip [63], multi-core processors have become the norm. Multi-threaded and/or multi-core processors are found today even in desktops, laptops, and cell phones. These parallel machines are available in most systems, even if the users/applications do not fully utilize them. Furthermore, compared to traditional symmetric multi-processor (SMP) systems, the tightly integrated nature of single-chip multi-

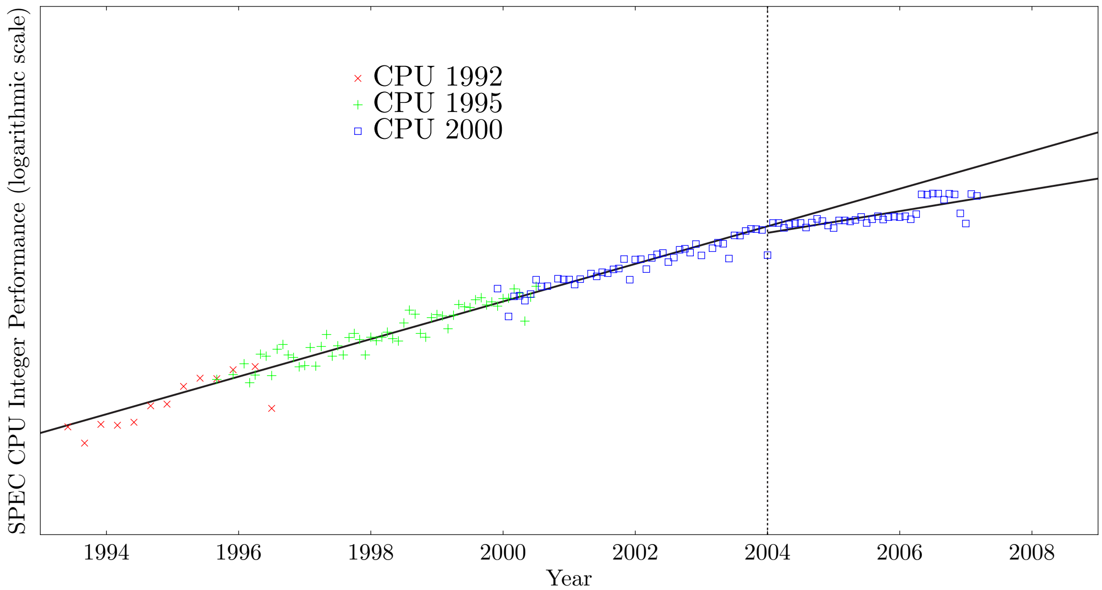


Figure 1.1: Performance trend of SPEC Integer benchmark suite on various machines released over time. Different versions of the benchmark suite were normalized using common machines. Source data from SPEC [99].

processors brings the potential to exploit parallelism at finer granularities.

Despite the recent widespread availability of parallel machines, most existing applications do not benefit from this computation power. This fact is due to two main reasons: the lack of applications written in parallel languages, and the restricted applicability of automatic parallelization techniques.

Parallel applications can be obtained by having the programmer write applications in parallel programming languages. Nevertheless, most applications are still written in sequential languages for the ease of programming. In parallel programming paradigms, the programmer needs to reason about concurrency, communication costs, data locality and to worry about new problems caused by a large number of possible interleavings of multiple executing threads. Common concurrency bugs include data races, deadlocks, and livelocks. The great interest in tools to help detect these errors [20, 25, 27, 61, 94] is a testament to their commonality, difficulty, and cost. Given the higher costs of development and maintenance of parallel programs, from a software engineering perspective, it is definitely preferable to utilize sequential programming models.

Even if written in sequential paradigms, applications can potentially be translated into parallel codes. This can be achieved with the assistance of automatic tools, especially compilers. Unfortunately, however, automatic parallelization has only been successful in restricted application domains, mostly where data parallelism is abundant. In particular, automatic parallelization has been effective for scientific and media applications. For more general application domains, these techniques have not been able to extract useful parallelism. Particularly, the presence of irregular memory accesses (as in linked data structures) and intensive and arbitrary control flow render inapplicable most of the compiler parallelization techniques proposed over the last several decades.

In the next two sections, we briefly introduce some of the previous work on extracting TLP at both coarse and fine granularities. This overview helps understand the limitations of these techniques, thus putting in context and motivating our work. We then introduce our approach for TLP extraction and summarize the contributions of this thesis.

1.1 Coarse-Grained Thread-Level Parallelism

The great majority of the automatic parallelization techniques have focused on extracting coarse-grained parallelism at the loop level. These techniques typically exploit parallelism among different iterations of a loop whose iterations are all independent. Such loops are known as *DOALL loops* [2, 115]. The challenges in exploiting DOALL parallelism come from two complementary alternatives: (a) *analyzing* a loop nest to prove that its iterations are independent; and (b) *transforming* a non-DOALL loop nest into an equivalent DOALL one. Many techniques have been proposed to address both alternatives, including the GCD and Omega tests, polyhedral methods, loop skewing, and loop distribution [2, 115]. Compiler frameworks based on these techniques can generally find reasonable amounts of parallelism in regular scientific applications. In this application domain, programs typically contain counted loops, very little control flow, and regular array-based memory accesses.

However, outside the scientific computation domain, these techniques are rarely applicable. The reasons for this are mainly the large amount of irregular pointer-based memory accesses, uncounted loops, and control flow. These characteristics typically create inter-iteration (or loop-carried) dependences that are very hard, if not impossible, to eliminate.

In the presence of loop-carried dependences, one notable loop-parallelization technique has been proposed, called DOACROSS [22, 78]. With DOACROSS, loop iterations are executed in multiple processors in a round-robin fashion. To respect loop-carried dependences, synchronization primitives are inserted in the code, such that instructions in one iteration wait for their dependent instructions in the previous iterations to complete. Despite some success, the benefits of DOACROSS are generally limited by two factors. First, the number and position of loop-carried dependences lead to synchronizations in the code that limit the amount of parallelism. Second, by dividing the loop iterations among the processors in a round-robin fashion, DOACROSS inserts the synchronizations in the critical path to execute the loop. In other words, the cost of synchronization is paid between every pair of consecutive iterations, essentially multiplying the synchronization cost by the number of iterations of the loop. Together, these two issues generally negate the benefits of DOACROSS. However, DOACROSS has found applicability when combined with speculation, in techniques generally known as *thread-level speculation* (TLS) [9, 47, 98, 101, 106, 120]. Unfortunately, even with the support of speculation, the amount of parallelism obtained by these techniques has been limited, hardly justifying the complex hardware support necessary to keep the overhead of these techniques tolerable.

1.2 Fine-Grained Thread-Level Parallelism

Given the difficulty of finding coarse-grained parallelism in most applications, a few researchers have investigated the possibility of extracting fine-grained thread-level parallelism. The potential of exploiting parallelism at finer granularities is enabled by multi-core processors. Compared to traditional symmetric multi-processors (SMPs), there are two key differences that enable fine-grained parallelism in multi-core processors. First, there is the possibility of communicating directly between the cores, without having to go off-chip. This can provide a much higher communication bandwidth. Second, there is the possibility to integrate specialized hardware mechanisms into the chip, so as to lower the inter-core communication overheads. In this section, we give an overview of such communication mechanisms, as well as the proposed compilation techniques to exploit them.

A notable example of on-chip inter-core communication mechanism is the *scalar operand network* in the RAW processor [104, 110]. This mechanism consists of a programmable mesh inter-connect among the cores. Associated with each core, there is a routing co-processor. To the software, the scalar operand network can essentially be abstracted as a set of hardware queues for scalar inter-core communication. Each core communicates with the other ones through its router. The accesses to the communication queues are encoded as reads and writes to specialized registers. For this reason, RAW's hardware queues are called register-mapped queues.

The key component of RAW's C compiler (RAWCC) is its instruction scheduler [59, 60]. For such spacial architecture, it is necessary to schedule instructions in both time and space (cores). Furthermore, the scheduler must also generate the necessary communications to satisfy inter-core dependences. The space-time scheduling approach used in the RAWCC compiler works as follows. For each basic block in the program, the *basic block orchestrator* partitions the instructions among the cores, using a space-aware variation of list scheduling [35]. Later, communications are generated to implement inter-core dependences. The basic block orchestrator processes each basic block independently: the assignment of instructions to cores in one basic block does not affect the assignment for other basic blocks. The coordination of the control flow among the cores is implemented using *asynchronous branching*. In this approach, the core containing the branch that terminates the basic block broadcasts the direction of the branch to all other cores through the hardware queues. Upon receiving the branch direction, the other cores mimic the same branching behavior. With this technique, all cores essentially follow a control-flow graph that is isomorphic to the original one, and each core executes a subset of the original program's instructions. As such, this approach basically utilizes multiple cores/threads to exploit *instruction-level parallelism* inside each basic block. For this reason, we call this a *local multi-threaded (LMT) instruction scheduling* technique. The results reported for this technique show that it hardly matches the parallelism obtained by an aggressive out-of-order processor [59].

Inter-core hardware queues have also been employed in *decoupled access-execute* processors [97]. In these processors, one core is dedicated to performing memory accesses, while another core executes the remaining instructions. The idea behind these processors is to try to decouple the execution of the non-memory-related instructions from the memory accesses, hopefully hiding the memory latencies. To implement the dependences between these two sets of instructions, two classes of scalar hardware queues are used. One class of queues is used to communicate scalar operands that either feed or result from memory instructions. The other is a *branching queue*, which is used to send the branch results from the execute core to the access core, so that the latter can follow the same execution path as the former. This compilation technique is classified as *LMT instruction scheduling*, since it essentially exploits parallelism inside basic blocks, with both threads executing all basic blocks in the program [88]. Experimental evaluations of decoupled access-execute architectures showed very little benefits, sometimes resulting in slowdowns because of the communication overhead and cyclic dependences between the threads [88].

Another hardware mechanism to implement inter-core scalar communication is the *synchronization array* [85]. In this mechanism, the processor's instruction set architecture (ISA) is extended with two new instructions, `produce` and `consume`, which respectively send and receive a scalar value through a given hardware queue. In [85], this hardware mechanism was used to support the parallelization of loops that traverse recursive/linked data structures. These loops were manually partitioned into two threads, one executing the data structure traversal, and the other executing the loop body. Similar to the decoupled-access-execute work, the idea was to hide the memory access latencies, which can be quite expensive in traversals of linked data structures due to the lack of locality. Unlike decoupled access-execute, however, reasonable speedups for selected pointer-intensive benchmarks were obtained in [85]. The key to their success was that, by creating a unidirectional flow of dependences between the threads, truly decoupled execution was achieved. This thread partitioning scheme, which produces *pipelined multi-threading*, was coined *Decoupled Software Pipelining* (DSWP). In this thesis, we propose general code partitioning and code generation algorithms that are able to extract an arbitrary number of threads from general loop regions (not only recursive data structure traversals) that exploit pipelined multi-threading. In fact, the techniques proposed in this thesis are even more general, not being limited to pipelined multi-threading.

Several alternatives to the synchronization array mechanisms were investigated with the goal of reducing the hardware costs for pipelined multi-threading [84]. As an interesting design point, we observed that the communication queues can be implemented through shared memory, for a small performance penalty. This not only reduces the hardware costs, but it also facilitates virtualization. To obtain good performance through shared memory, however, two hardware changes are still necessary. First, blocking `produce` and `consume` instructions are usually needed to avoid the costs of implementing synchronization in software. Second, it is necessary to change the cache coherence protocol to perform *forwarding of cache lines* used to implement the queues. By having a cache line forwarded from the producer core when it becomes full, stalls to access the queues in the consumer core are eliminated.

1.3 Parallelism: Limits and Limitations

The previous sections described the main efforts on automatically exploiting parallelism. Despite the lack of success of automatic parallelization techniques, several limit studies have shown that even applications known as hard-to-parallelize have large amounts of parallelism [56, 108]. However, these limit studies are generally performed by execution-trace analysis, and they give very little insight on how to extract such parallelism.

So, the interesting question that arises is: what prevents such parallelism from being achieved by the hardware and/or the compiler? This is prevented by several limitations both in the hardware and in the compilers. We first describe the limitations of the hardware. Even with aggressive out-of-order execution, branch prediction, and large instruction windows, the processors still have fundamental limitations. In particular, processors only see the stream of dynamic instructions that actually execute. As a result, they are unable to detect merge points in a program's control-flow graph and to understand true control dependences. This results in hardware techniques being overly conservative in case of branch mispredictions, squashing even instructions that are not dependent on the branch. This fundamentally prevents a processor from extracting multiple threads of execution from a single program thread.

There are also several limitations that prevent compilers from automatically extracting parallelism, as we pointed out in [77]. Most importantly, parallelism is not very regular, rarely matching the DOALL and DOACROSS patterns that are implemented in many parallelizing compilers, such as the Intel Compiler [46], IBM's XL compiler [43], SUIF [38], and Parascope [18]. Compilers focusing on fine-grained TLP, such as RAWCC [59, 60], have tried to address this issue by looking at more irregular parallelism. However, these techniques have their own limitations. By fully replicating the control flow in all threads, they only exploit *local* parallelism. As such, local multi-threading instruction scheduling techniques essentially have the same limitations as the hardware: the inability to exploit parallelism across independent control regions to truly benefit from multiple threads of execution. As pointed out by Lam et al. [56], it is key to overcome these two limitations in order to extract reasonable amounts of parallelism. This brings another obstacle identified in [77] that traditionally inhibits parallelizing compilers: the scope of parallelization. To find most parallelization opportunities, it is generally necessary to look for parallelism across large regions of code, possibly inside an entire procedure or even across the whole program. Another parallelization obstacle is analyzing memory dependences. Statically determining precise memory aliasing is, in fact, a fundamental limitation [57, 80]. One way to get around this limitation is to rely on *speculation*. With speculation, certain dependences can be ignored, as long as run-time mechanisms to detect and recover from mis-speculations are available.

More recently, another fundamental limitation of automatic parallelization was identified in [10]. This work demonstrated that significantly more parallelism can be unlocked by changing a program's output by another, equivalently good output. However, benefiting from these opportunities clearly requires the programmer's intervention in order to convey the cases where this is valid. To enable these opportunities, simple source-code annotations were proposed in [10]. These annotations were shown to work very well in combination with the techniques proposed in this thesis.

1.4 Our Approach

The goal of this work is to design techniques to overcome the main aforementioned limitations of previous automatic parallelization techniques, so as to uncover some of the potential suggested by various limit studies. The general approach proposed in this work exploits fine-grained thread-level parallelism. To achieve so, this work generalizes instruction scheduling techniques to target multi-threaded architectures. Since most applications contain lots of control flow, a key goal in this work was to design techniques that look for parallelism in large code regions, beyond extended basic blocks[1] and inner loops. These

[1] An extended basic block (EBB) is a sequence of instructions with a single entry (the first instruction) and potentially multiple exits.

techniques typically operate at outer loops or entire procedures, thus being classified as GMT instruction scheduling [72].

To represent the program region being scheduled, we use a variation of the *Program Dependence Graph* (PDG) [29]. The PDG is an elegant representation that has some important properties. First, it encodes all the program dependences that need to be preserved to keep the program's semantics [40, 91]. Second, the PDG can be built for low-level code, even at the assembly level. This property is important for the extraction of fine-grained parallelism, which is better performed in a compiler's back-end.

The approach for GMT instruction scheduling proposed in this thesis is very general. One of its key properties is the separation of concerns into *thread partitioning* and *code generation*, as illustrated in Figure 1.2. These two components are interfaced through a *partitioned PDG* representation. This design allows a thread partitioning algorithm to operate on an abstract, dependence-centric representation, the PDG, in order to find parallelism. By partitioning the PDG nodes among different blocks (threads), a partitioner exposes thread-level parallelism. Such approach enables different thread partitioning algorithms to be easily integrated into the same framework. From the partitioned PDG, multi-threaded code is generated. The multi-threaded code consists of the original instructions in the code, plus communication and synchronization instructions inserted to enforce all inter-thread dependences.

The framework proposed in this thesis preserves the order among all instructions assigned to the same thread. This design simplifies not only the interface between thread partitioning and code generation, but also the code generation algorithms described in this work. We rely on traditional single-threaded instruction scheduling to obtain a better ordering of the instructions in each thread, by taking into account the details of the target cores. However, this creates a phase-ordering problem because the extraction of parallelism is tackled independently in two different phases of the compilation. This problem is aggravated because single-threaded instruction schedulers are typically unaware of

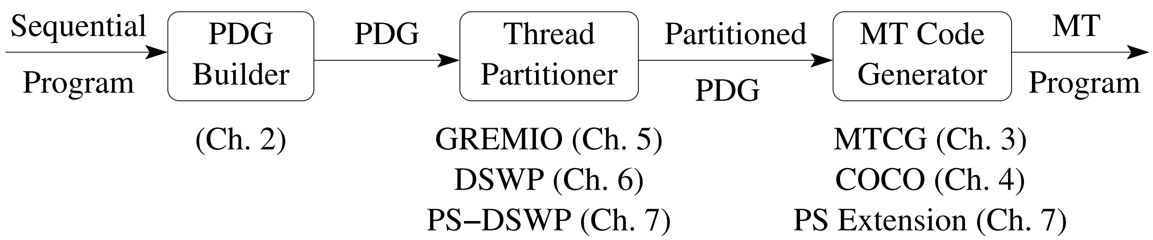


Figure 1.2: Framework for global multi-threaded instruction scheduling.

inter-thread communication and thread-level parallelism. However, this problem can be mitigated by making the single-threaded instruction scheduler aware of inter-thread communication primitives, by giving special scheduling priorities to these primitives.

The techniques proposed in this thesis are *non-speculative*, and they require no special hardware support. Nevertheless, given the granularity of the parallelism typically found in applications, simple hardware mechanisms to reduce the overheads of inter-thread communication and synchronization can be very beneficial. Examples of such hardware mechanisms include synchronization arrays [84, 85] and scalar operand networks [104]. As mentioned in Section 1.2, these mechanisms have been previously proposed to support other fine-grained TLP approaches.

1.5 Contributions

In summary, the main contributions of this thesis are:

1. The design and implementation of a general framework to perform global instruction scheduling for multi-threaded architectures [72, 74, 76]. This framework separates concerns into thread partitioning and code generation, thus providing great flexibility that enables various global scheduling algorithms to be easily implemented. This framework has been implemented both in the IMPACT [96] and in the VELOCITY [105] compilers, and it has been used to support a few research efforts on GMT instruction scheduling [10, 72, 73, 74, 76, 77, 81, 82, 83, 109].

2. Algorithms to generate efficient multi-threaded code for arbitrary code partitions [73, 74]. The proposed *multi-threaded code generation* (MTCG) algorithm allows an arbitrary partition of PDG nodes to threads, and it produces code that does not replicate the entire control-flow graph for all produced threads. This is key to truly exploit parallelism across different control regions. In order to implement inter-thread dependences, the MTCG algorithm inserts communication and synchronization primitives. While the basic MTCG algorithm [74] is somewhat naïve in how it inserts communication primitives in the code, we also propose more advanced algorithms to obtain a better placement of these primitives. This communication optimization framework, called COCO [73], is based on novel thread-aware data-flow analyses and graph min-cut algorithms, and it simultaneously optimizes all types of inter-thread dependences (control, register, and memory).

3. Algorithms to partition instructions into threads in order to expose thread-level parallelism. Based on the proposed compiler framework for GMT instruction scheduling, this thesis describes partitioning algorithms that extend single-threaded instruction scheduling techniques to generate TLP. More specifically, these thread-partitioning techniques are:

   (a) A list-scheduling-based partitioner, called GREMIO [72].

   (b) A Decoupled Software Pipelining (DSWP) thread partitioner [74, 76, 77].

   (c) An extension of DSWP that enables parallel stages in the pipeline, called Parallel-Stage Decoupled Software Pipelining (PS-DSWP) [77, 81]. (This technique is the result of a collaboration with Easwaran Raman.)

4. Correctness proofs of the proposed algorithms. Given the design of our framework, the correctness of the generated code only depends on the code generation algorithms (MTCG and COCO).

5. Implementations and experimental evaluations of all the proposed techniques, which were implemented in the VELOCITY compiler [105].

1.6 Overview

This dissertation is organized as illustrated in Figure 1.2, and it is divided in two parts. In Part I, which consists of Chapters 2 through 4, we describe the program representation and multi-threaded code generation algorithms. More specifically, Chapter 2 describes the variation of the program dependence graph we use. In Chapter 3, we describe the basic multi-threaded code generation (MTCG) algorithm, and we present the COCO communication optimizations in Chapter 4. In Part II, consisting of Chapters 5 through 7, we present the thread-partitioning techniques. Chapter 5 describes the list-scheduling-based partitioner (GREMIO). The Decoupled Software Pipelining (DSWP) partitioner is described in Chapter 6, while the Parallel-Stage DSWP extension is presented in Chapter 7. Chapter 8 presents the conclusions of this work and provides directions for future research on global multi-threaded instruction scheduling and code optimizations in general.

Part I

Program Representation and Multi-Threaded Code Generation

This part describes the program representation and general multi-threaded code generation algorithms. In Chapter 2, we describe the central program representation that we use in this work. This intermediate representation is a variation of the program dependence graph (PDG), and it can be built for arbitrary code, even at the machine level. The PDG representation serves as the interface between the thread partitioning techniques described in Part II and the code generation algorithms described in this part.

In Chapter 3, we describe our general multi-threaded code generation (MTCG) algorithm. This algorithm produces efficient and correct multi-threaded code for an arbitrary partition of the PDG nodes into threads. A key property of MTCG is that it does not replicate the control flow entirely in all threads. This property is key to enable truly thread-level parallelism. To the best of our knowledge, the MTCG algorithm is the first method to satisfy this property and to support arbitrary control flow and thread partitions. Furthermore, our MTCG algorithm is elegant and simple to implement.

The basic MTCG algorithm described in Chapter 3 uses a simple strategy to determine the program points where inter-thread communications are inserted. In Chapter 4, we describe a general framework, called COCO, to improve the placement of inter-thread communications. COCO improves the performance of the resulting code both by reducing the number of inter-thread communications and by eliminating synchronization points that prevent parallelism.

Chapter 2

Dependence Graphs

Dependence graphs are a traditional form of representing dependences among statements or instructions in a program. In this chapter, we introduce various forms of dependence graphs proposed in the literature and describe the variation used in this work.

2.1 Data Dependence Graphs

Data dependence graphs are a basic representation of dependences among instructions or statements of a program. In data dependence graphs, each vertex represents an instruction (or statement) in the program, and directed arcs specify data dependences between instructions. At a low-level representation, data dependence arcs can take two forms: either (virtual) register[1] data dependences or memory data dependences. Furthermore, data dependences can be of three kinds, depending on whether the involved instructions read from or write to the data location: *flow dependence*, which goes from a write to a read; *anti-dependence*, which goes from a read to a write; and *output dependence*, which goes from a write to another write [54]. Register data dependences can be efficiently and precisely computed through data-flow analysis. For memory data dependences, compilers typically

[1]In this dissertation, we assume a low-level IR with virtual registers, which may correspond to either temporaries emitted by the compiler or to program variables that can be promoted to physical registers [19]. We generally omit the word *virtual* in the text.

rely on the results of static memory analyses to determine loads and stores that may access the same memory locations. Although computationally much more complicated, practical existing pointer analysis can typically disambiguate a large number of non-conflicting memory accesses even for type-unsafe languages like C (e.g. [15, 36, 70]). Inter-procedural pointer analyses typically compute a summary of the memory accesses per function. With such information, complete memory data dependences can be computed not only between loads and stores, but also involving function calls.

Many memory analysis techniques have also been proposed for regular, array-based memory references [2, 115]. These analyses require a high-level intermediate representation, with explicit array accesses. With these techniques, it is possible to refine dependence arcs involving array accesses in loop nests with dependence distance vectors. Although these techniques are very important for array-based, scientific applications (typically written in FORTRAN), they are not so applicable to more irregular applications (typically written in C). For this reason, we do not use such memory analyses in most of this work.

Data dependence graphs are generally employed in scheduling and parallelization techniques allowing only very primitive or no forms of control flow. Such uses include instruction scheduling of basic blocks or inner loops with a straight-line body [65], and the vectorization and parallelization of simple inner loops [2].

2.2 Control Dependence Graphs

For techniques that operate beyond straight-line code (basic block) boundaries, the notion of *control dependences* becomes relevant. An instruction (or statement) X *controls* an instruction Y if, depending on the direction taken at X, Y must execute along one path and may not execute along another path. Similar to data dependences, control dependences among the statements or instructions of a program can be represented as arcs in a directed graph.

Different notions of control dependence exist. For structured programs, the syntax-based notion of control dependences is applicable [54]. At this level, a statement is control dependent on its closest enclosing control construct. Although very natural and easy to compute, this definition of control dependence is not applicable to unstructured programs (containing `goto`s, `break`s, and `continue`s) or even to structured programs after going through traditional compiler optimizations [1, 4, 17, 65].

Ferrante et al. [29] generalized the notion of control dependence to arbitrary control-flow graphs.[2] Their graph-based definition, which is based on the post-dominance relation [65], enables control dependences to be computed for low-level program representations. For this reason and some of its key properties, we use Ferrante et al.'s definition of control dependence in this work.

Definition 1 (Control Dependence (Ferrante et al. [29]))**.** *Let G be a CFG, and X and Y two nodes in G. Y is control dependent on X iff:*

1. *there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y; and*
2. *X is not strictly post-dominated by Y.*

Cytron et al. [23] proposed an efficient algorithm to compute control dependences according to this definition.

To deal with the possibility of calling a function that may not return (e.g. because it may call `exit` in C), some extra control dependences are necessary. To properly compute control dependences in this case, auxiliary control-flow arcs are drawn from the instructions calling such possibly non-returning functions to the program's *END* node.[3] With these arcs inserted, control dependences are computed normally using standard algorithms [23, 29].

[2] In fact, Ferrante et al. [29] defninition is restricted to control-flow graphs where all nodes can reach the *END* node.

[3] Auxiliary control-flow arcs can also be used to model other non-explicit control transfers, such as exceptions.

In the resulting control dependences, some instructions will now be dependent on these function-call instructions. We name these dependences *call-exit control dependences*.

2.3 Program Dependence Graphs

Dependence graphs including both data and control dependences are generally called *Program Dependence Graphs* (PDGs) [29, 54]. Compilers widely use PDGs as an intermediate representation due to several important properties. In particular, using syntax-based control dependences, Horwitz et al. [40] proved the *Equivalence Theorem*. According to this theorem, two programs with the same PDG are equivalent. Later, Sarkar [91] proved a similar result for PDGs using control dependences according to Definition 1.

Traditionally, PDGs contain two kinds of nodes [29]: regular nodes, representing statements/instructions, and region nodes, representing a control-equivalent region of the program. In this work, however, we opted to have only regular nodes. Control dependence arcs are drawn directly from branch instructions to the dependent instructions. As will be clear in the following chapters, this allows a more homogeneous treatment of the PDG, since all its node represent actual instructions in the program.

As an example, consider the C code in Figure 2.1(a), with corresponding low-level representation in Figure 2.1(b). Figures 2.1(c)-(e) illustrate the corresponding CFG, post-dominance tree, and PDG. In Figure 2.1(e), solid arcs represent data dependences, which are all register dependences in this example, and dotted arcs represent control dependences. Register dependence arcs are labeled with the register involved in the dependence, while control arcs are labeled with the branch direction that causes the dependent instruction to execute.

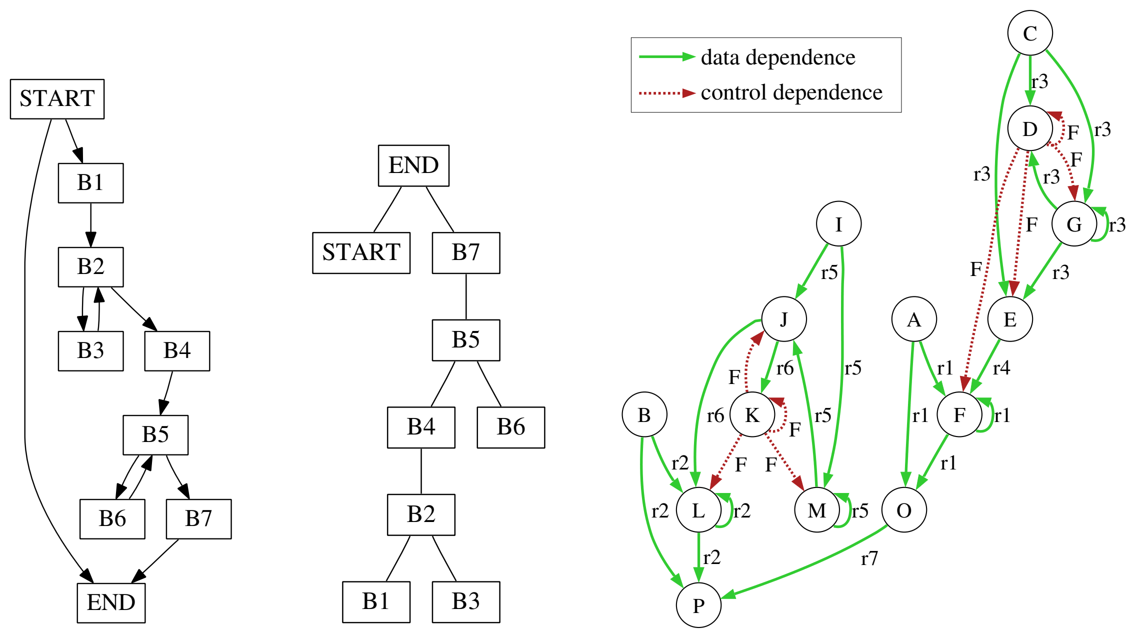
In the PDG used in most of this work, only true (flow) register dependences are included. Since there is no ambiguity in register accesses, renaming can generally be used to eliminate output and anti-dependences. As described in Chapter 3, our algorithms take into account these dependences only when they are necessary.

```
s1 = 0;
s2 = 0;
for(p=head; p != NULL;
    p = p->next){
  s1 += p->value;
}
for(i=0; a[i] != 0; i++){
  s2 += a[i];
}
printf("%d\n", s1*s1/s2);
```

(a) Example code in C

```
(A) B1: move   r1 = 0         ;; s1 in r1
(B)     move   r2 = 0         ;; s2 in r2
(C)     load   r3 = [head]    ;; p in r3
(D) B2: branch r3 == 0, B4
(E) B3: load   r4 = [r3]      ;; load p->value
(F)     add    r1 = r1, r4
(G)     load   r3 = [r3+4]    ;; load p->next
(H)     jump   B2
(I) B4: move   r5 = @a        ;; &a[i] in r5
(J) B5: load   r6 = [r5]      ;; load a[i]
(K)     branch r6 == 0, B7
(L) B6: add    r2 = r2, r6
(M)     add    r5 = r5, 4
(N)     jump   B5
(O) B7: mult   r7 = r1, r1
(P)     div    r8 = r7, r2
```

(b) Low-level IR



(c) CFG (d) post-dom. tree (e) PDG

Figure 2.1: Example code in: (a) C, (b) low-level IR, (c) CFG, (d) post-dominance tree, and (e) PDG.

Chapter 3

Multi-Threaded Code Generation

As discussed in Section 1.4, one of the key enablers of GMT instruction scheduling is the ability to generate efficient code for a given code partition. In this chapter, we describe our general multi-threaded code generation algorithm, called MTCG [74].

There are several requirements for a code generator to support GMT instruction scheduling. First, in order to handle global code regions, it is desirable to use a PDG representation. As discussed in Chapter 2, respecting the dependences in a PDG guarantees the preservation of the program's semantics. Another requirement of a general multi-threaded code generator is the ability to handle *arbitrary code partitions*. By code partition, we mean an assignment of instructions to threads.[1] Handling arbitrary code partitions is important to support different thread partitioning algorithms, and it is central to provide the separation of concerns into partitioning and code generation mentioned in Section 1.4. A third requirement of a multi-threaded code generation algorithm is to be able to produce efficient code. To achieve this, it is particularly important to avoid replicating the control flow as much as possible, which is the main drawback of LMT scheduling techniques (Section 1.2). The MTCG algorithm proposed in this work was designed to meet all the above requirements. In Section 3.1, we describe the MTCG algorithm. Section 3.2 presents the proof of correctness of the algorithm.

[1] A code partition does not specify the order among the instructions in each thread.

3.1 MTCG Algorithm

This subsection describes the MTCG algorithm [74], whose pseudo-code is illustrated as Algorithm 1. This algorithm takes as input a program dependence graph (*PDG*) for the code region being scheduled and a given partition ($\mathcal{P}$) of its nodes (instructions) into threads. Furthermore, for simplicity, we assume here that a control-flow graph (*CFG*) representation of the original code region is also given.[2] As output, this algorithm produces, for each of the resulting threads, a new CFG containing its corresponding instructions and the necessary communication instructions.

The thread model assumed here is that the program executes sequentially until it reaches a region that has been scheduled on multiple threads. Upon reaching such parallelized region, auxiliary threads are spawned, and each of them will execute one of the CFGs produced by MTCG. In this model, auxiliary threads do not spawn more threads. The main thread (which may also execute one of the new CFGs) then waits for completion of all the auxiliary threads. Once all auxiliary threads terminate, sequential execution resumes. This thread model is known as *fork-join*. In practice, the cost of spawning and terminating threads can actually be reduced by creating the threads only once, at the start of the program. The threads then execute a loop that continuously waits on the address of the code to be executed and invokes the corresponding code, until they are signaled to terminate. In this model, the work is statically assigned to threads. This is in contrast to dynamic scheduling [90].

Before going into the details in Algorithm 1, let us introduce the notation used. T_i denotes a block (thread) in $\mathcal{P}$, and CFG_i denotes its corresponding control-flow graph. For a given instruction I, $bb(I)$ is the basic block containing I in the (original) *CFG*, and $point_j(I)$ is the point in CFG_j corresponding to the location of I in *CFG*.

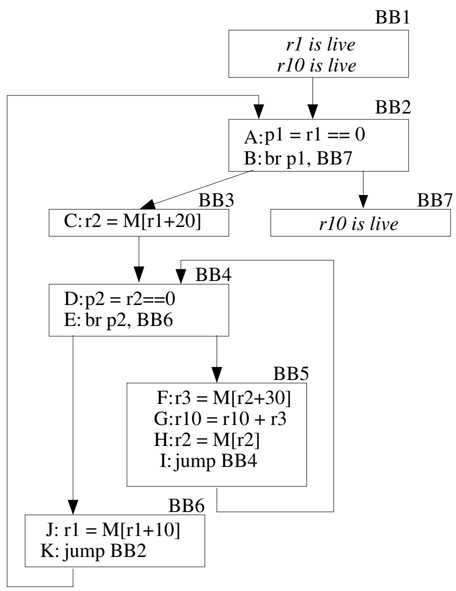
[2]There are well-known algorithms to construct a CFG from a PDG derived from a sequential program [28, 100].

Algorithm 1 MTCG

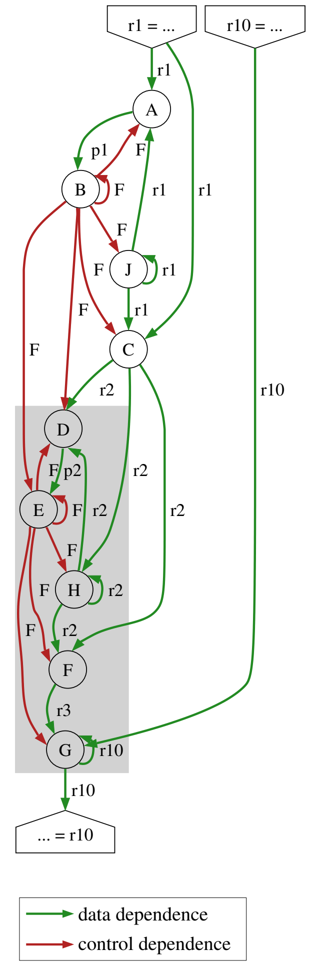
```
Require: CFG, PDG, 𝒫 = {T_1, ..., T_n}
for each T_i ∈ 𝒫 do                                                    // 1. create CFG_i's basic blocks
    V_{CFG_i} ← create_relevant_basic_blocks(CFG, PDG, T_i)
end for
for each instruction I ∈ V_{PDG} do                                    // 2. put instruction in its thread's CFG
    let i | I ∈ T_i
    add_after(CFG_i, point_i(I), I)
end for
COMM ← ∅
for each arc (I → J) ∈ PDG [incl. transitive control dependences] do   // 3. insert communication
    let T_i, T_j be such that I ∈ T_i and J ∈ T_j
    if (T_i = T_j) ∨ ((I, T_j) ∈ COMM) then
        continue
    end if
    COMM ← COMM ∪ {(I, T_j)}
    q ← get_free_queue()
    if dependence_type(I → J) = Register then                          // register dependences
        r_k ← dependent_register(I → J)
        add_after(CFG_i, I, "produce [q] = r_k")
        add_after(CFG_j, I, "consume r_k = [q]")
    else if dependence_type(I → J) = Memory then                       // memory dependences
        add_after(CFG_i, I, "produce.rel [q]")
        add_after(CFG_j, I, "consume.acq [q]")
    else                                                               // control dependences
        r_k ← register_operand(I)
        add_before(CFG_i, I, "produce [q] = r_k")
        add_before(CFG_j, I, "consume r_k = [q]")
        add_after(CFG_j, I, duplicate(I))
    end if
end for
for each T_i ∈ 𝒫 do
    for each live-in register r in T_i do                              // initial communication
        q ← get_free_queue()
        add_last(START, "produce [q] = r")
        add_last(START_i, "consume r = [q]")
    end for
    for each live-out register r defined in T_i reaching END do        // final communication
        q ← get_free_queue()
        add_last(END_i, "produce [q] = r")
        add_last(END, "consume r = [q]")
    end for
    q ← get_free_queue()
    add_last(END_i, "produce.rel [q]")
    add_last(END, "consume.acq [q]")
end for
for each T_i ∈ 𝒫 do                                                    // 4. create CFG_i's arcs
    for each branch instruction I ∈ T_i do
        redirect_target(I, closest_relevant_postdom_i(target(I)))
    end for
    for each B ∈ V_{CFG_i} do
        CRS ← closest_relevant_postdom_i(succ(orig_bb(B)))
        add_last(B, "jump CRS")
    end for
end for
```

```
while (list != NULL) {
  for (node = list->head; node != NULL;
       node = node->next) {
    total += node->cost;
  }
  list = list->next;
}
```

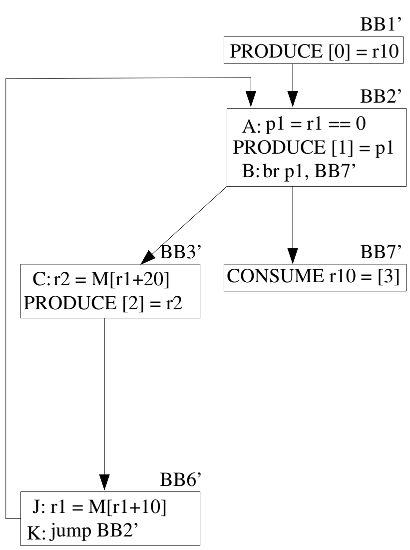
(a) Example code in C



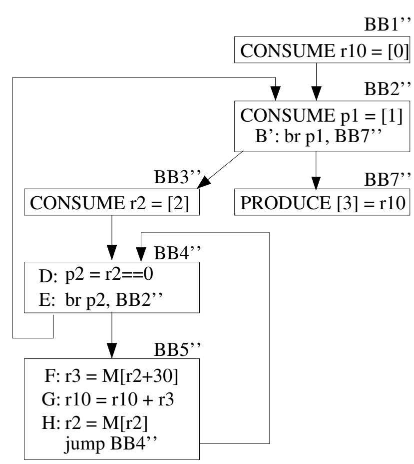
(b) Original *CFG*



(c) Partitioned PDG



(d) CFG_1 (main thread)



(e) CFG_2 (auxiliary thread)

Figure 3.1: Example of the MTCG algorithm. (a) C source code; (b) original low-level code; (c) partitioned PDG; and (d)-(e) resulting multi-threaded code.

In essence, the MTCG algorithm has four main steps. Each of the following subsections describes one of these steps. As we describe these steps, we illustrate them on the example in Figure 3.1. Figure 3.1(a) contains the source code in C, and Figure 3.1(b) contains the corresponding code in a low-level IR. We illustrate how MTCG generates code for the partitioned PDG in Figure 3.1(c), where the nodes in the shaded rectangle are assigned to one thread, and the remaining instructions to another. In the PDG, data dependence arcs are annotated with the corresponding register holding the value, while control dependence arcs are labeled with the corresponding branch condition. In this example, there are no memory dependences. Special nodes are included in the top (bottom) of the graph to represent live-in (live-out) registers. The resulting code for the two threads is illustrated in Figures 3.1(d) and (e). In this example, the main thread also executes the code for one of the generated threads (CFG_1).

3.1.1 Creating New CFGs' Basic Blocks

MTCG creates a new CFG (CFG_i) for each thread T_i. Each CFG_i contains a basic block for each *relevant basic block* to T_i in the original *CFG*. The notion of relevant basic block is defined below. In its last step, described in Section 3.1.4, MTCG adds the arcs connecting the basic blocks in each CFG_i. These arcs are inserted in a way that guarantees the equivalence between the condition of execution of each new basic block and its corresponding block in the original *CFG* (see proof in Section 3.2).

Definition 2 (Relevant Basic Block)**.** *A basic block B is* relevant *to a thread T_i if B contains either:*

1. *an instruction scheduled to T_i; or*
2. *an instruction on which any of T_i's instructions depends (i.e. a source of a dependence with an instruction in T_i as the target); or*
3. *a branch instruction that controls a* relevant *basic block to T_i.*

The reason for including basic blocks containing instructions in T_i is obvious, as they will hold these instructions in the generated code. The reason for adding the basic blocks containing instructions on which T_i's instructions depend is related to where communication instructions are inserted by MTCG, as described shortly. The third rule in Definition 2 is recursive, and it is necessary to implement the correct condition of execution of the basic blocks. This rule is related to how inter-thread control dependences are implemented by MTCG, as described Section 3.1.3.

Consider the example in Figure 3.1. For thread $T_1 = \{A, B, C, J\}$, besides the start block BB1 and the end block BB7, the only relevant basic blocks are BB2, BB3, and BB6. All these blocks are relevant to T_1 by rule 1 in Definition 2, since there is no incoming dependence into this thread. For thread $T_2 = \{D, E, F, G, H\}$, besides the start and end blocks, several other basic blocks are relevant for different reasons. BB4 and BB5 contain instructions in T_2, and therefore are relevant by rule 1. Because of the dependences from T_1 to T_2, with source instructions B and C (Figure 3.1(c)), the basic blocks that contain these instructions (BB2 and BB3) are relevant to T_2 by rule 2. Again, no basic block is relevant due to rule 3 in this case.

3.1.2 Moving Instructions to New CFGs

The second step of MTCG (lines 4 to 7) is to insert the instructions in T_i into their corresponding basic blocks in *CFG*$_i$. MTCG inserts the instructions in the same relative order as in the original code, so that intra-thread dependences are naturally respected.

Figures 3.1(d) and (e) illustrate the instructions inserted in their threads, in the basic blocks corresponding to the instructions' original basic blocks.

3.1.3 Inserting Inter-Thread Communication

The third step of the algorithm (lines 8 to 44) is to insert communication primitives to implement each inter-thread dependence. We represent communication primitives by `produce` and `consume` instructions [84]. The `produce` and `consume` instructions are inserted in pairs, so that, statically, each `produce` feeds one `consume`, and each `consume` is fed by one `produce`. For each inter-thread dependence, a separate communication queue is used. Although a single queue is sufficient, multiple queues are used to maximize the freedom of subsequent single-threaded scheduling. Furthermore, a queue-allocation technique can later reduce the number of queues necessary, analogously to register allocation.

Depending on where the communication primitives are inserted with respect to the parallel region, they can be of three types: *intra-region* (i.e. inside the parallelized region), *initial*, or *final* communications. We first describe the insertion of the most interesting ones, the intra-region communications, followed by the initial and final communications.

Intra-Region Communications

In order to preserve the conditions under which each dependence occurs, MTCG adopts the following strategy.

Property 1 (Basic Communication Placement)**.** *Each intra-region dependence is communicated at the program point of its source instruction.*

This strategy is simple to implement, and it also simplifies the proof of correctness of the algorithm (Section 3.2). However, this strategy can result in a suboptimal number of inter-thread communications. Better algorithms to place communication primitives are described in Chapter 4.

In Algorithm 1, $add\_before(\textbf{\textit{CFG}}_i, I, instr)$ inserts $instr$ in $\textbf{\textit{CFG}}_i$ at the point right before instruction I's position in the original ***CFG***, and add_after works analogously. The $add\_last(B, I)$ function inserts instruction I as the last instruction in basic block B.

The actual communication instructions inserted depend on the type of the dependence, as illustrated in Algorithm 1:

- Register Dependences: These are implemented by communicating the register in question.

- Memory Dependences: For these, purely synchronization primitives are inserted to enforce that their relative order of execution is preserved. This is dependent on the target processor's memory consistency model. In this work, we assume Itanium 2's memory consistency model. In Algorithm 1, synchronization primitives are represented by `produce.rel` and `consume.acq` instructions, which implement the release and acquire semantics in the memory system.[3]

- Control Dependences: In the source thread, before the branch is executed, its register operands are sent. In the target thread, `consume` instructions are inserted to get the corresponding register values, and then an equivalent branch instruction is inserted to mimic the same control behavior. For simplicity, Algorithm 1 assumes that branches have a single register operand.

  The call-exit control dependences discussed in Section 2.2 require a different code generation strategy, which is not illustrated in Algorithm 1. For these control dependences, a `consume` instruction from a special queue is inserted in the target thread at the point corresponding to the call-site. We name the value consumed from this special queue the *terminate* value. After the `consume` of this *terminate* value, a branch is inserted to check this value and to jump to terminating code if the value is *true*. In the source thread, a `produce` *false* into this special queue is inserted after the function call. This signals the target thread to continue execution in case the function returns normally. Additionally, in the source thread, a `produce` *true* is inserted in a handler installed to executed upon program termination. This causes the target thread to terminate as well.

In the example from Figure 3.1, there are two inter-thread, intra-region dependences. First, instruction C in T_1 writes into register $r2$ that is used by instructions D, F, and H

[3]In other words, `produce.rel` is guaranteed to execute only after all previous memory instructions commit. Analogously, `consume.acq` is guaranteed to commit before all successive memory instructions are executed.

in T_2. This register data dependence is communicated through queue 2. The `produce` and `consume` instructions are inserted right after the points corresponding to instruction C in BB3. The other inter-thread dependence is a control dependence from branch B in T_1, which controls instructions in T_2. This dependence is satisfied through queue 1. In T_1, the register operand ($p1$) is sent immediately before instruction B. In T_2, the value of $p1$ is consumed and a copy of branch B (B') is executed.

As mentioned in line 9 of Algorithm 1, the MTCG algorithm also needs to communicate *transitive control dependences* in the PDG in order to implement the relevant blocks for a thread. This is related to rule 3 in Definition 2. The reason for this is control dependences' implementation, which consumes the branch operands and duplicates the branch in the target thread. In order to enforce the correct conditions of execution of these instructions in the target thread (i.e. to create the relevant basic blocks to contain these instructions), it is necessary to communicate the branches controlling the blocks where these instructions are inserted. This reasoning follows recursively, thus resulting in the necessity of communicating the transitive control dependences.

Initial Communications

Communication primitives may need to be inserted for register values that are live at the entry of the parallelized region, similarly to OpenMP's `firstpriviate` [71]. More specifically, if a register definition from outside the parallelized region reaches a use in an instruction assigned to thread T_i, then the value of this register has to be communicated from the main thread to T_i.[4] These communication primitives are inserted at the *START* basic block in *CFG* and at its corresponding *START*$_i$ basic block in *CFG*$_i$. This step is illustrated in lines 31-35 in Algorithm 1.

In the example from Figure 3.1, the main thread executes the code for T_1, so no initial communication is necessary for T_1. For T_2, instruction G uses the value of $r10$ defined

[4]The exception is in case T_i is run on the main thread.

outside the parallelized region, and therefore this register needs to be communicated. This dependence is communicated through queue 0, and the `produce` and `consume` instructions are inserted in BB1, the *START* basic block.

Since auxiliary threads are spawned at the entry of the parallelized region[5], no initial communication for memory and control dependences are necessary. In other words, auxiliary threads are only spawned when the execution reaches the parallelized region, and any memory instruction before this region will have already executed.

Final Communications

At the end of the parallelized region, the auxiliary threads may have to communicate back to the main thread, which is similar to OpenMP's `lastprivate` [71]. Final communications may be necessary not only for register values, but also for memory. We discuss these two cases below.

For every register r defined by an instruction assigned to thread T_i, r needs to be sent to the main thread if two conditions hold. First, r must be live at the end of the parallelized region (END). Second, a definition of r in T_i must reach END. Lines 36-40 in Algorithm 1 handle this case.

In Figure 3.1, it is necessary to communicate the value of register $r10$ from T_2 to the main thread at the end of the parallelized region. This communication uses queue 3, and the `produce` and `consume` instructions are inserted in the END blocks (corresponding to BB7).

A problem arises in case multiple threads have definitions of r that reach the exit of the parallelized region. Depending on the flow of control, a different thread will have the correct/latest value of r, which needs to be communicated back to the main thread. To address this problem, two solutions are possible. One alternative is to associate a time-stamp for r in each thread, so that it is possible to tell which thread wrote r last. In this scheme, at the

[5]At least conceptually; see discussion earlier in Section 3.1.

end of the parallelized region, all possible writers of r communicate both their value of r and the corresponding time-stamp to the main thread. The main thread can then compare the time-stamps to determine which value of r is the latest. The time-stamps can be computed based on a combination of topological order and loop iterations [52]. An alternate solution is to enforce that a single thread always has the latest value of r. This can be enforced in two ways. First, the possible partitions of the PDG can be restricted to require that all instructions writing into r must be assigned to the same thread. Unfortunately, this restricts the valid partitions, breaking both the generality of the code generation algorithm and the separation of concerns between thread partitioning and code generation. A better alternative is to include in the PDG output dependences among instructions defining the same live-out register r. Effectively, these dependences will be communicated inside the parallelized region, similarly to true register dependences. These register output dependences need not be inserted among every pair of instructions defining a register r. Instead, one of these instructions (say, I) can be chosen, so that the thread containing I will hold the latest value of r. Register output dependences can then be inserted from every other instruction writing into r to instruction I. This way, a final communication of r is necessary only from the thread containing I to the main thread.

At the exit of the parallelized region, synchronizations may also be necessary to enforce memory dependences (lines 41-43 in Algorithm 1). Specifically, for every auxiliary thread that contains a memory instruction, it is necessary to insert a synchronization from it to the main thread upon the exit of the parallelized region. This synchronization prevents the main thread from executing conflicting memory instructions that may exist in the code following the parallelized region. Analyses can be performed to determine if such synchronization is unnecessary, or to move it as late as possible in the code.

3.1.4 Creating New CFGs' Arcs

The last step of the MTCG algorithm (lines 45-53) is to create the correct control flow in the new CFGs. To achieve this, it is necessary both to adjust the branch targets and to insert jumps in these CFGs. Because not all the basic blocks in the original CFG have a corresponding block in each new CFG, finding the correct branch/jump targets is non-trivial. The key property we want to guarantee is that each basic block in a new CFG must have the same condition of execution of its corresponding basic block in the original CFG. In other words, the control dependences among the basic blocks in a new CFG must mimic the control dependences among their corresponding basic blocks in the original CFG.

Since control dependences are defined based on the post-dominance relation among basic blocks (Definition 1), it suffices to preserve the post-dominance relation among basic blocks. Therefore, for a new CFG, CFG_i, the branch/jump targets are set to the *closest post-dominator basic block B* of the original target/successor in the original CFG such that *B* is *relevant* to CFG_i. Notice that such post-dominator basic block always exists as every vertex is post-dominated by END, which is relevant to every new CFG.

Demonstrating that the control dependences among the new basic blocks match the control dependences among their corresponding blocks in the original CFG is central in the correctness proof of the MTCG algorithm. Section 3.2 presents the formal proofs.

For simplicity, the MTCG algorithm inserts unconditional jumps at the end of every new basic block. Many of these jumps, however, can be later optimized by standard code layout optimizations [65].

Figures 3.1(d)-(e) illustrate the resulting code, with the control-flow arcs inserted by MTCG. Determining most of the control-flow arcs is trivial in this example. An interesting case occurs in CFG_1 for the outgoing arc of BB3'. In the original *CFG* (Figure 3.1(b)), control flows from BB3 to BB4. However, BB4 is not relevant to T_1. Therefore, the output arc from BB3' goes to the closest post-dominator of BB4 relevant to T_2, which is BB6'. Another interesting case happens for branch E in T_2. In the original code, the branch target of E is BB6. However, BB6 is not relevant to *CFG*$_2$. Therefore, branch E is redirected to BB6's closest post-dominator relevant to T_2, which is BB2".

The code in Figures 3.1(d)-(e) has been subject to code layout optimizations. For example, no jump is necessary at the end of BB3' if we place BB6' immediately after BB3' in the generated code.

3.2 Correctness of the MTCG Algorithm

In this section, we prove the correctness of the MTCG algorithm. This is achieved by demonstrating that the code produced by MTCG preserves the semantics of the original code, for any given thread partitioning. The key step of this proof is showing that the MTCG algorithm preserves all the data and control dependences of the original program. In order to prove this result, we first demonstrate several useful lemmas.

In the following, we denote by $Y\ pdom\ X$ and $Y\ pdom_i\ X$ the relations that Y post-dominates X in *CFG* (the original CFG) and *CFG*$_i$, respectively. Additionally, we denote $Y\ \cancel{pdom}\ X$ the fact that Y does not post-dominate X. For simplicity of notation, we denote the corresponding basic blocks in all CFGs by the same name. The referred CFG is made clear by the context.

The lemma below shows that every path p_i in *CFG*$_i$ is a subsequence of a path p in *CFG*.

Lemma 1. *Let $B_0, B_k \in V_{CFG_i}$. Then there is a path $p_i = (B_0, \ldots, B_k)$ in CFG$_i$ iff there is a path $p = (B_0, \ldots, B_k)$ in CFG such that p_i is the subsequence of p restricted to V_{CFG_i}.*

Proof. By induction on $|p_i|$, the length of p_i.

Base: $|p_i| = 0$. Then $p_i = p = (B_0)$.

Inductive step: By induction, our assumption holds for the sub-path $(B_0, \ldots, B_{k-1})$, so we need to show that the property also holds when we add B_k to the path.

$\Rightarrow$ Consider the arc $(B_{k-1} \rightarrow B_k) \in E_{CFG_i}$. By construction of *CFG*$_i$'s arcs (lines 45-53 in Algorithm 1), there is a successor S of B_{k-1} in *CFG* such that B_k *pdom* S. Thus there is a path $B_{k-1} \rightarrow S \rightsquigarrow B_k$ in *CFG*.

$\Leftarrow$ Conversely, let B_k be the first vertex in V_{CFG_i} to appear in a path in *CFG* starting at a successor S of B_{k-1}. In addition, let p' be such that $p = (B_0, \ldots, B_{k-1}, \overbrace{S, \ldots, B_k}^{p'}, \ldots)$. We need to show that $(B_{k-1} \rightarrow B_k) \in E_{CFG_i}$, and thus $p_i = (B_0, \ldots, B_{k-1}, B_k)$ is a path in *CFG*$_i$. We argue that B_k is the closest post-dominator of S in *CFG* that is relevant to *CFG*$_i$. First, notice that B_k *pdom* S. Otherwise, there would exist another vertex $T \in V_{CFG_i}$ before B_k in p' such that B_k would be control dependent on T. But then T would be relevant to *CFG*$_i$, which contradicts the choice of B_k as the first vertex in p' to be relevant to *CFG*$_i$. Now assume that there is another post-dominator $U \in V_{CFG_i}$ of S in p'. This also contradicts the fact that B_k is the first vertex in p' to be relevant to *CFG*$_i$. Therefore, B_k is the closest post-dominator of S in *CFG* that is relevant to *CFG*$_i$, and thus the arc $(B_{k-1} \rightarrow B_k)$ is added to *CFG*$_i$ by the MTCG algorithm. □

The following lemma shows that the post-dominance relation in *CFG*$_i$ is the same as in *CFG* restricted to the basic blocks relevant to T_i.

Lemma 2. *Given two basic blocks X and Y in CFG$_i$, Y pdom$_i$ X iff Y pdom X.*

Proof. We first show that $Y\ pdom_i\ X \implies Y\ pdom\ X$. For contradiction, assume that Y $pdom_i$ X and Y $\cancel{pdom}$ X. Then there is a path $p : X \rightsquigarrow END$ in *CFG* that avoids Y. By Lemma 1, there is a corresponding path $p_i : X \rightsquigarrow END$ in *CFG*$_i$ which does not contain Y, a contradiction.

We now show that $Y\ pdom\ X \implies Y\ pdom_i\ X$. For contradiction, assume Y *pdom* X and Y $\cancel{pdom_i}$ X. Then there must exist a path $p_i : X \rightsquigarrow END$ in *CFG*$_i$ that avoids Y. By Lemma 1, there is a corresponding path $p : X \rightsquigarrow END$ in *CFG* that avoids Y, a contradiction. □

The following lemma follows directly from the definition of relevant basic blocks (Definition 2).

Lemma 3. *If CFG_i contains block Y, it also contains every block X on which Y is control dependent.*

Proof. This follows immediately from rule 3 in Definition 2. □

Theorem 1. *Let $Y \in V_{CFG_i}$, Y is control dependent on X in CFG_i iff Y is control dependent on X in CFG.*

Proof. First, by Lemma 3, V_{CFG_i} contains every X on which Y is control dependent in *CFG*. We analyze each condition in the control dependence definition (Definition 1) individually. Lemma 2 says that the post-dominance relation between X and Y is the same in CFG_i and *CFG*. Therefore, condition 2 of Definition 1 holds for CFG_i iff it holds for *CFG*. In addition, Lemma 1 and Lemma 2 together guarantee that condition 1 of Definition 1 is true for CFG_i iff it is true for *CFG*, completing the proof. □

Lemma 4. *The code generated by the MTCG algorithm preserves all data dependences.*

Proof. The control dependence between the basic blocks is preserved by Theorem 1. Given that the instructions are inserted in the basic blocks corresponding to their original basic blocks, their condition of execution is preserved. Intra-thread dependences are naturally satisfied by the MTCG algorithm. In addition, inter-thread register dependences are satisfied by communicating the value of the related register, and memory dependences are respected by sending a synchronization token. In both cases, the communications are inserted at points corresponding to the location of source instruction, therefore preserving the condition under which the dependence occurs. □

Theorem 2. *The code generated by the MTCG algorithm preserves all dependences in the PDG.*

Proof. Theorem 1 shows that the basic blocks in the MT code preserve their control dependences. Lemma 4 shows that all data dependences are preserved. Finally, inter-thread control dependences are satisfied by communicating their register arguments and duplicating the branch instruction at the CFG containing the target of the control dependence. This ensures that branch instructions are inserted in the code to reconstruct the appropriate control flow. □

From Theorem 2, the correctness of the MTCG algorithm then follows immediately.

Theorem 3 (Correctness of MTCG Algorithm)**.** *The MTCG algorithm preserves the semantics of the original code.*

Proof. From Theorem 2, all PDG dependences are preserved. Therefore, using the Equivalence Theorem [91], the semantics of the original program is preserved. □

Theorem 3 establishes the correctness of the MTCG algorithm, for any thread partition. As discussed before, this is the key enabler for our general GMT instruction scheduling framework.

3.3 Significance

Program Dependence Graphs (PDGs) [29, 54] are a very elegant representation of programs. PDGs represent all the dependences in a program in a unified way, allowing a more homogeneous treatment of data and control dependences. Due to these characteristics, PDGs have been widely used for program analyses, program slicing, loop parallelizations, and global instruction scheduling. Although the idea of PDGs was born for high-level, syntactical program representations [54], the work of Ferrante et al. [29] extended PDGs for low-level representations based on control-flow graphs (CFGs). This extension has enabled the use of PDGs for low-level, back-end code optimizations, including global instruction scheduling [8, 112, 113].

High-level parallelizing compilers use PDGs for loop distribution [49] and for partitioning programs to exploit DOALL parallelism [90]. Ferrante et al. [28] showed how to generate sequential code back from a PDG. However, to the best of our knowledge, our MTCG algorithm is the first general technique to produce multi-threaded code from a low-level PDG, with arbitrary control flow and thread partitions, and without replicating the whole CFG in each thread.

The work that comes closest to our goal of generating multi-threaded code for arbitrary low-level PDGs was proposed by Newburn et al. [66, 68]. Although operating at a low-level representation, this work reconstructs the program's control hierarchy. To do so, they extend the notion of region nodes in Ferrante et al. [29]'s PDG, with region nodes for different control structures (e.g. conditionals, DOALL loops, non-DOALL loops). Due to the possibility of arbitrary control flow in low-level, highly optimized code, Newburn's PDG defines region nodes for multi-predecessor control regions, irreducible loops, among other specialized region nodes. The code generation from this extended PDG is performed hierarchically, treating each type of region node especially. There are several drawbacks of this approach compared to our homogeneous treatment of control dependences in the MTCG algorithm. First, Newburn's code generation is much more complicated because of the various special cases that need to be handled. This also makes it harder to demonstrate and verify the correctness of their code generation algorithm. Furthermore, their code generation operates hierarchically on the control structure of the PDG, which prevents the exploitation of some types of parallelism. For example, their code generation allows either DOALL parallelism, or intra-iteration parallelism only, in which case the threads synchronize at the end of every loop iteration [67, 68]. Thus, their algorithm cannot produce code that exploits pipeline parallelism, such as code produced by the DSWP technique described in Chapters 6 and 7. Therefore, although operating on a low-level representation, Newburn's code generation has essentially the same limitations of structural approaches that use a syntax-based PDG. On the other hand, the MTCG algorithm proposed in this chapter is a simple, general algorithm for generating multi-threaded code for PDGs with arbitrary control flow and thread partitions. The simplicity of our algorithm lies in its homogeneous treatment of dependences, especially by handling all control dependences in a similar way.

Although elegant, the MTCG algorithm has two drawbacks. The first is that its simple communication-placement strategy may be suboptimal. Chapter 4 studies better strategies to place communication in the generated code. The second drawback is that, as described in this chapter, the MTCG algorithm does not allow the exploitation of iteration-level parallelism. Chapter 7 describes how to extend the MTCG algorithm to enable the exploitation of this type of parallelism as well.

Chapter 4

Inter-Thread Communication Optimizations

Although very general, the MTCG algorithm (presented in Chapter 3) may generate excessive inter-thread communications. In this chapter, we present more elaborate algorithms to better place communication primitives.

This chapter describes the COmpiler Communication Optimization (COCO) framework [73]. COCO optimizes the placement of communication required to implement the three kinds of inter-thread dependences: register, control, and memory dependences. Since communication instructions are overhead inserted in the code, reducing the number of these instructions can improve performance. Furthermore, communication instructions represent synchronization points in the program. Therefore, improving their placement can also improve the parallelism among the generated threads.

Property 1 in Chapter 3 described a simple strategy for placing communication instructions, which is to place them at the program point corresponding to the source instruction of the dependence. The simplicity of this strategy lies in that each dependence is communicated whenever its source instruction executes. For example, for a register dependence, at the program point right after the register's definition (the source of the dependence), it is guaranteed that the value communicated is the latest value assigned to that register. This may not be true, for instance, if the value of the register is instead communicated immediately before it is used, since definitions in different threads may reach a single use.

Nevertheless, the simplicity of placing communication according to Property 1 comes at a cost: it may result in excessive communication. To illustrate that, consider the example in Figure 4.1. Figure 4.1(a) shows the original code in a CFG, and Figure 4.1(b) contains the corresponding PDG. Assume a partition into 2 threads: $T_1 = \{A, B, C, D, E, G\}$ and $T_2 = \{F\}$. Figures 4.1(c)-(d) show the code generated by MTCG for each thread. Thread 1 (T_1) has instructions in all basic blocks, and thus they are all relevant to it. For thread 2, B3 is relevant because it holds instruction F assigned to this thread, and B1 and B2 are relevant because they contain instructions on which F depends. As can be seen in the PDG, there are 3 inter-thread dependences in this case: two register dependences $(A \rightarrow F)$ and $(E \rightarrow F)$ involving $r1$, and a transitive control dependence $(D \rightarrow F)$. This transitive control dependence exists because D controls E, which is the source of an inter-thread register dependence. In Figures 4.1(c)-(d), there is a pair of `produce` and `consume` instructions for each inter-thread dependence, inserted according to Algorithm 1.

In the example in Figure 4.1, the set of communication instructions inserted by MTCG is not optimal. It would be more efficient to simply communicate $r1$ at the beginning of blocks B3' and B3", for two reasons. First, this would avoid communicating $r1$ twice on the path (B1, B2, B3). Second, this would make it unnecessary to have branch D'' in thread 2, thus saving the communication of $r2$ as well. In the remaining of this chapter, we describe the algorithms used by COCO to efficiently perform these register-communication and control-flow optimizations in general, as well as to optimize the memory synchronizations inserted by the basic MTCG algorithm.

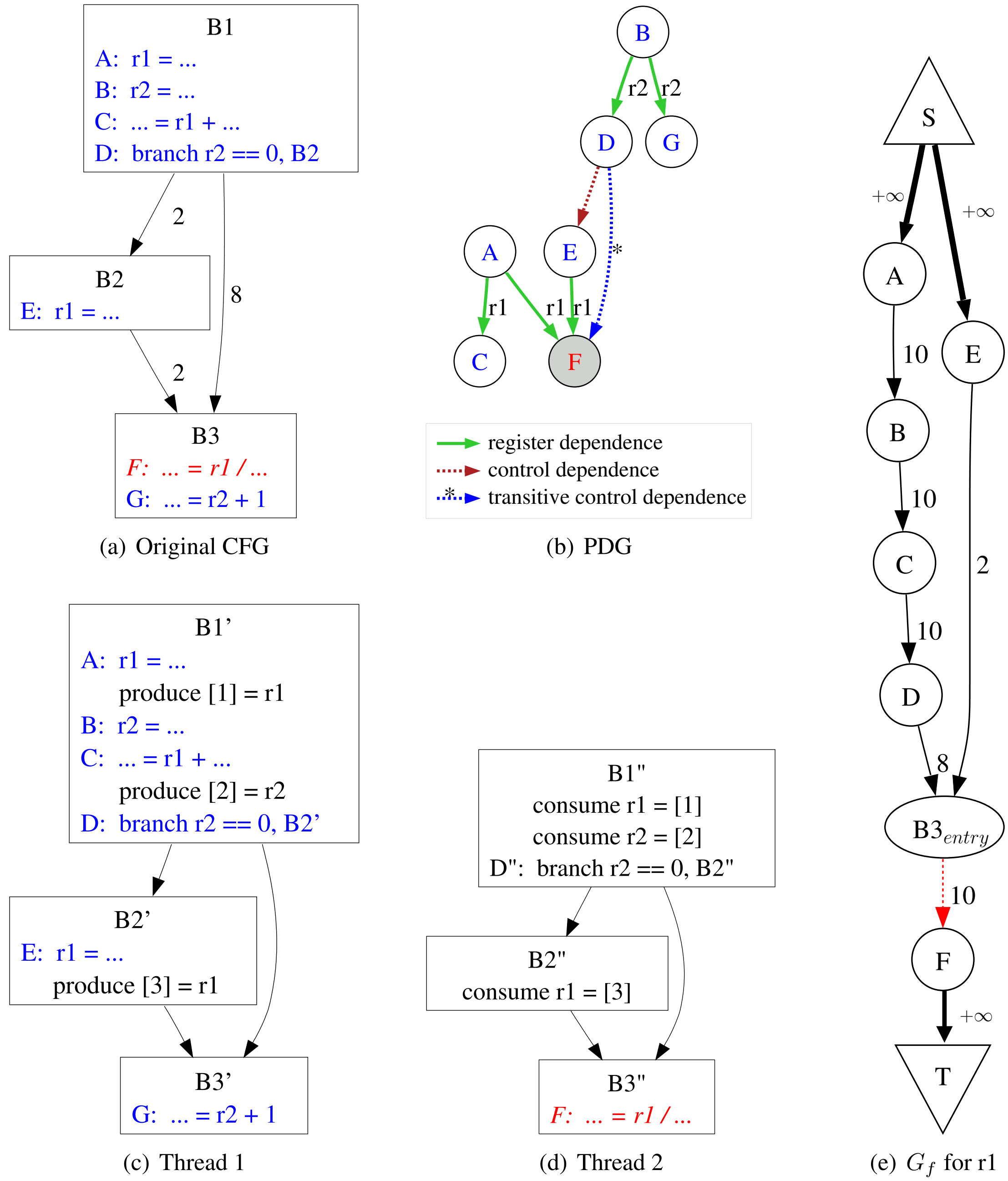


Figure 4.1: Simple example of the MTCG algorithm.

4.1 Problem Formulation

As motivated by the example in Figure 4.1, the goal of COCO is to reduce the number of dynamic communication and synchronization instructions executed in the generated MT code. Unfortunately, as with many program analysis and optimization problems, this problem is undecidable in general, since it is undecidable to statically determine which paths will be dynamically executed. To illustrate this difficulty, consider the example in Figure 4.2. In this example, both definitions of $r1$ are assigned to one thread, while the use of $r1$ is assigned to another thread (the thread boundaries are shown by dashed lines in Figure 4.2). Thus the value of $r1$ needs to be communicated from one thread to another. For simplicity, assume that blocks B1, B3, and B5 are relevant to both threads. Depending on the dynamic execution frequency of arcs A, B, and C, different places to communicate $r1$ result in the minimum amount of communication. For example, with freq(A) = 1, freq(B) = 5, and freq(C) = 10, the optimal solution is to communicate $r1$ at B1 and B3. This results in 6 dynamic communications. Now consider freq(A) = 1, freq(B) = 10, freq(C) = 5. In this case, the optimal solution is to communicate $r1$ at B5, which results in 5 dynamic communications. As this example illustrates, there is no static placement of communication that guarantees the minimum number of dynamic communications for every execution.

Given the necessity of knowing profile weights to optimally place communications, COCO uses a profile-based approach, in which an estimate on the execution count of each CFG edge is available. These estimates can be obtained through standard profiling techniques (e.g. [7]) or through static analyses, which have been demonstrated to be also very accurate [116]. Given the profile weights, the problem can be formulated as to minimize the communication assuming each CFG edge's execution frequency as indicated by its weight.

Before formulating the communication optimization problems and describing the algorithms, we introduce several definitions and properties that are necessary. First, in Definition 3 below, we define the notion of *relevant branches* to a thread. This definition parallels the notion of relevant basic blocks described earlier in Definition 2 in Section 3.1.1. However, Definition 3 is more general in that it enables the placement of communication at points other than the point of a dependence's source instruction.

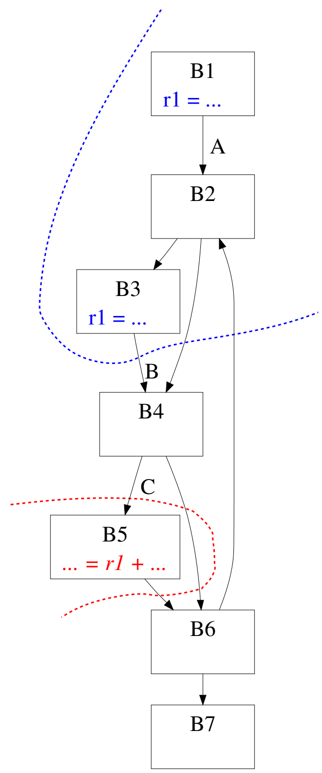


Figure 4.2: Example CFG illustrating the impossibility of statically placing communications optimally.

Definition 3 (Relevant Branches)**.** *A branch instruction B is* relevant *to thread T if either:*

1. *B is assigned to T in the partition; or*

2. *B controls the insertion point of an incoming dependence into an instruction assigned to T; or*

3. *B controls another branch B' relevant to T.*

The intuition is that relevant branches are those that thread T will contain, either because they were assigned to this thread or because they are needed to implement the correct condition under which an incoming dependence into T must happen.

Based on the notion of relevant branches for a given thread, we define its *relevant points* in the program.

Definition 4 (Relevant Points)**.** *A program point p in the original CFG is* relevant *to thread T iff all branches on which p is control dependent are relevant branches to T.*

In other words, the relevant points for thread T are those that depend only on T's relevant branches. This means that the condition of execution of these points can be implemented without adding new branches to T. In Definition 4, we require *all* branches controlling p to be relevant to T in order to enforce the correct condition of execution of p through all CFG paths.

The communication instructions generated by MTCG obey an important property to enable the TLP intended by the thread partitioner:

Property 2. *All inter-thread communications in the generated MT code correspond to dependence arcs in the PDG, including transitive control dependence arcs.*

This property guarantees that only dependences represented in the PDG will be communicated. This is important because, as illustrated in Figure 1.2, the partitioner is based on the PDG. If Property 2 was not respected, then MTCG could hurt the parallelism intended by the partitioner. For example, the DSWP partitioner (described in Chapter 6) assigns instructions to threads so as to form a pipeline of threads, with unidirectional dependences among them. However, if Property 2 is not respected, a dependence cycle among the threads can be created.

In the MTCG algorithm, since communication is always inserted at the point corresponding to the source of the dependence, *all* inter-thread transitive control dependences need to be implemented [74]. In other words, each of these dependences will require its branch operands to be communicated and the branch to be duplicated in the target thread. With COCO, however, a better placement of data communications can reduce the transitive control dependences that need to be implemented. For this reason, besides Property 2, COCO also respects the following property to limit the transitive control dependences that need to be implemented:

Property 3. *The communication instructions to satisfy a dependence from thread T_s to thread T_t must be inserted at relevant points to T_s.*

Essentially, this property prevents branches from becoming relevant to thread T_s merely for implementing a dependence emanating from T_s. In other words, no new branches must be added to T_s in order to implement a dependence from it to T_t. However, additional branches may be made relevant to the target thread T_t in order to implement one of its input dependences.

Besides Properties 2 and 3, the placement of register communications must also respect the *Safety* property for correctness:

Property 4 (Safety)**.** *A register dependence from T_s to T_t involving (virtual) register r must be communicated at* safe *points, where T_s has the* latest *value of r.*

This property is necessary for correctness because communicating r at an unsafe point would, in some control paths, overwrite r in T_t with a stale value. In other words, a dependence from a definition of r not in T_s to a use of r in T_t would not be respected, thus changing the program's semantics. Notice that MTCG's placement of communication is safe, since the registers are communicated immediately after they are defined.

The set of registers that are safe to communicate from thread T_s to any other thread at each program point can be precisely computed using the data-flow equations (4.1) and (4.2) below. In these equations, DEF_{T_s} and USE_{T_s} denote the set of registers defined and used by instruction n if it is assigned to T_s, and DEF means the set of registers defined by n regardless of which thread contains n.

$$\begin{aligned} \text{SAFE}_{\mathit{out}}(n) &= \text{DEF}_{T_s}(n) \cup \text{USE}_{T_s}(n) \cup \\ &\quad (\text{SAFE}_{\mathit{in}}(n) - \text{DEF}(n)) \quad (4.1) \\ \text{SAFE}_{\mathit{in}}(n) &= \bigcap_{p \in \text{Pred}(n)} \text{SAFE}_{\mathit{out}}(p) \quad (4.2) \end{aligned}$$

The intuition behind these equations is that T_s is guaranteed to have the latest value of a register r right after T_s either defines or uses r. Furthermore, the value of r in T_s becomes stale after another thread defines r. Finally, the data-flow analysis to compute safety is a forward analysis, and the SAFE_{in} sets are initially empty.

This safety data-flow analysis is said to be *thread-aware* because, although operating on a single CFG, it takes the partition of instructions among the threads into account. An equivalent analysis could operate on multiple CFGs (one per thread) simultaneously, given the correspondence between the basic blocks in all CFGs.

Having defined Properties 2, 3, and 4, the communication-placement optimization problem can be defined as follows. Given a thread partition and the original CFG with profile weights, find insertion points in the CFG to place the necessary communications such that: (a) Properties 2, 3, and 4 are satisfied; and (b) the total profile weight of all communication insertion points is minimized.

Given this problem formulation, the next subsection focuses on optimizing the communication between a pair of threads. Based on this pairwise optimization, the general algorithm to handle any number of threads is described in Section 4.3.

4.2 Optimizing a Pair of Threads

We first describe how COCO optimizes register communications in Section 4.2.1. Then Section 4.2.2 shows how to extend COCO to also minimize control flow. Finally, Section 4.2.3 demonstrates how COCO optimizes memory synchronizations as well.

4.2.1 Optimizing Register Communication

We now formulate the problem of optimizing register communications from a source thread T_s to a target thread T_t. Since the communication of each register requires its own set of instructions, it is possible to optimize the communication of each register independently. So let r denote the register whose communication is to be optimized.

The register communication optimization problem can be precisely modeled as a *min-cut problem in directed graphs*, by constructing a graph $G_f = (V_f, A_f)$ derived from the CFG as described shortly. The intuition behind the construction of G_f is that a cut in this graph will correspond to communicating r at the program points corresponding to the arcs in this cut.

The set of vertices in V_f contains the original code instructions where r is *live with respect to* T_t. That means the live range of r considering only the uses of r in the instructions assigned to T_t. This can be computed using a thread-aware data-flow analysis very similar to the standard liveness analysis. In addition, V_f also contains one vertex corresponding to the entry of each basic block where r is live with respect to T_t. The need for these vertices will become clear shortly. Finally, there are two special nodes: a source node S, and a target (or sink) node T. The arcs in A_f are of two kinds. *Normal arcs* represent possible flows of control in the program, corresponding to the arcs in the CFG constructed at the granularity of instructions. These arcs have a *cost* equal to the profile weight of their corresponding CFG arcs. In addition, there are also *special arcs* from S to every definition of r in T_s, and from every use of r in T_t to T. The costs of the special arcs are set to infinity to prevent them from participating in a minimum cut. This is necessary because special arcs do not correspond to program points, and thus cannot have communication instructions placed on them.

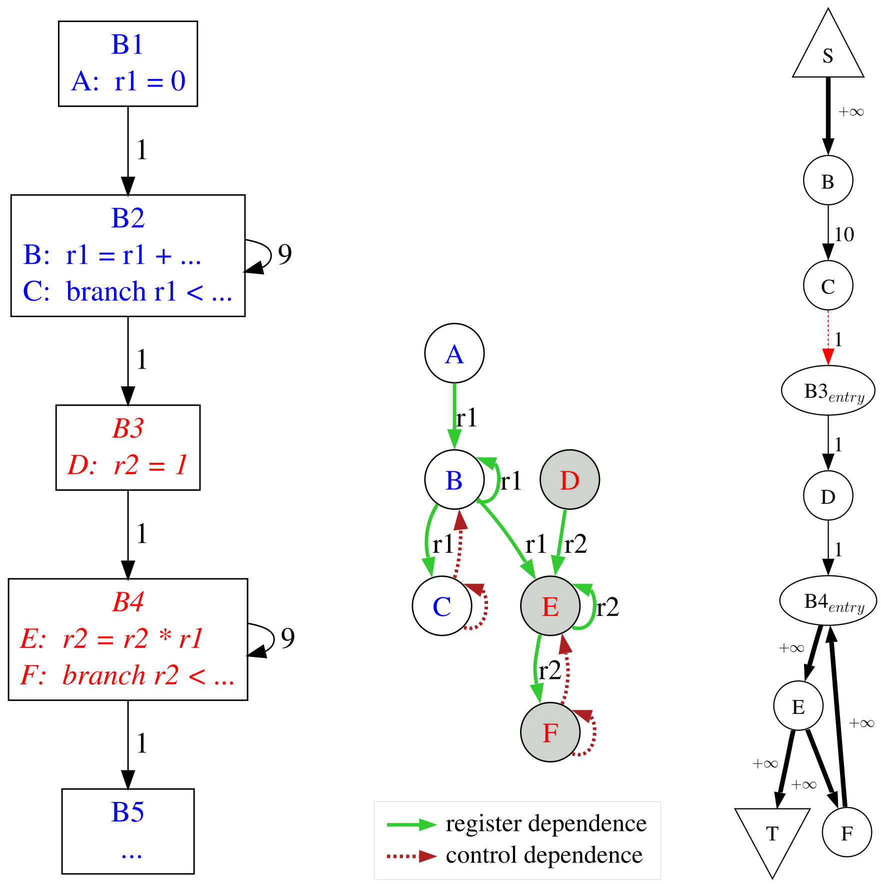
As an example, consider the code in Figure 4.1(a) with the partition $T_s = \{A, B, C, D, E, G\}$ and $T_t = \{F\}$. Figure 4.1(e) illustrates the graph G_f for register $r1$. The source and sink nodes are represented by a triangle and an inverted triangle, respectively. Node $B3_{entry}$ corresponds to the beginning of block B3, the only block that has $r1$ live at its entry.

When drawing special arcs to the target node T in G_f, besides the uses of r in instructions assigned to T_t, uses of r in relevant branches to T_t are also considered as uses in T_t. The reason for this is that, as mentioned earlier, relevant branches to a thread need to be included in it to properly implement its control flow. In effect, treating branches as belonging to all threads to which they are relevant allows the communication of branches' register operands to be optimized along with register data communications. This can result in better communication of branch operands compared to MTCG's strategy of implementing control dependences, which is to always communicating branch operands immediately before the branches (lines 24-27 in Algorithm 1).

Notice that, in order to satisfy the dependences involving r from T_s to T_t, communication instructions for r can be inserted at any subset of A_f that disconnects T from S. In other words, any *cut* in G_f corresponds to a valid placement of communication instructions, namely communicating r at each arc in this cut. This guarantees that r will be communicated from T_s to T_t along every path from a definition of r in T_s to a use of r in T_t. In particular, the original MTCG algorithm always uses the cut containing the outgoing arcs in G_f of the instructions defining r in T_s. This corresponds to the cut containing $(A \rightarrow B)$ and $(E \rightarrow B3_{entry})$ in Figure 4.1(e). Since S only has arcs to nodes corresponding to T_s's instructions defining r, this is clearly a cut in G_f.

By the construction of G_f, for a given cut C in G_f, the number of dynamic communications of r that will be executed corresponds to the cost of the arcs in C. Therefore, the problem of finding the minimum number of dynamic communications reduces to finding a *minimum cut* in G_f. In Figure 4.1(e), arc $(B3_{entry} \rightarrow F)$ alone forms a min-cut, with a cost of 10. This example also illustrates the role of the nodes in G_f corresponding to basic block entries, which is to allow the placement of communications before the first instruction in a basic block (B3 in this case).

In fact, the formulation described so far is still incomplete, because it allows any normal arc in G_f to be cut. However, there are arcs that must not participate in a cut because communicating r at those points violates one of Properties 2, 3 or 4. To prevent such arcs from participating in a cut, their costs are set to infinity. As long as there exists a cut with finite cost, these arcs (and also the special arcs involving S and T) are guaranteed not to be in a min-cut. Fortunately, a finite-cost cut always exists: the cut that the MTCG algorithm picks. That is true because the points right after the definitions of r in T_s are both safe (i.e. T_s has the latest value of r there) and relevant to T_s (since they have the same condition of execution of the definitions of r in T_s).



(a) Original CFG (b) PDG (c) G_f for $r1$

Figure 4.3: An example with loops.

To illustrate a more drastic case where the MTCG algorithm generates inefficient code, consider the example in Figure 4.3. The partition in this case is $T_s = \{A, B, C\}$ and $T_t = \{D, E, F\}$. The only inter-thread dependence is the register dependence $(B \rightarrow E)$. The MTCG algorithm communicates $r1$ right after instruction B, inside the first loop. For this reason, a transitive control dependence $(C \rightarrow E)$, not illustrated in Figure 4.3(b), also needs to be communicated. As a result, thread T_t will contain the first loop as well. In effect, T_t will consume the value of $r1$ each time B is executed, even though only the last value assigned to $r1$ is used by instruction E. Figure 4.3(c) shows the graph G_f constructed for $r1$. Notice that G_f does not contain nodes before B, including the arc $(C \rightarrow B2_{entry})$, since $r1$ is not live with respect to T_t at these points. Applying the register communication optimization, $r1$ can be communicated in either of the arcs with cost 1 in Figure 4.3(c). Any of these cuts essentially corresponds to communicating $r1$ at block B3. This drastically reduces the number of times $r1$ is communicated from the total number of B2's loop iterations, 10, down to 1. Furthermore, as a side effect, this completely removes the first loop from thread T_t, making it unnecessary to implement the transitive control dependence $(C \rightarrow E)$.

Fortunately, there are efficient algorithms to compute a min-cut in direct graphs. In fact, due to its duality to maximum flow [31], min-cut can be solved by efficient and practical max-flow algorithms based on preflow-push, with worst-case time complexity $O(n^3)$, where n is the number of vertices [21]. For our problem, given that G_f is limited to a register's live-range, even algorithms with worse time complexity run fast enough so as to not increase compilation time significantly, as observed in our experiments.

4.2.2 Reducing Control Flow

As illustrated in the example from Figure 4.3, the placement of data communication can also reduce the control flow in the target thread. In some cases, as in Figure 4.3, this comes for free simply by optimizing the data communications. However, there are cases where there are multiple cuts with the minimum cost, but some of them require more inter-thread control dependences to be implemented than others. Extra control flow in the target thread T_t is necessary whenever communication is placed at points currently not relevant to T_t. This forces these branches to be added to the set of relevant branches for T_t, so that they will need to be implemented in T_t.

In order to avoid branches unnecessarily becoming relevant to T_t, the costs of the arcs in G_f can be adjusted as follows. The idea is to penalize arcs that, if cut, will require additional branches to become relevant to T_t. Thus, we add to each arc A in G_f the profile weight of each currently irrelevant branch to T_t that will become relevant if communication is placed on A. The reasoning is that these branches would not be necessary otherwise, so we add the number of dynamic branches that would be executed to the cost of A. To illustrate this, consider the example in Figure 4.4(a), with $T_s = \{A, B, C, D, E, G\}$ and $T_t = \{F, H, I, J, K\}$. The corresponding PDG is shown in Figure 4.4(d). Consider the communication of $r1$ from T_s to T_t. This communication is only allowed to be placed in basic blocks B3, B4, and B6, since, from B7 on, it is not safe due to the definition of $r1$ in instruction F in T_t. The two alternatives then are to communicate $r1$ either in B6 or in B3 and B4. Looking at the profile weights, both alternatives look equally good. However, communicating at B3 and B4 makes the branch instruction B relevant to T_t, while communicating at B6 does not. Figure 4.4(b) illustrates the graph G_f for $r1$ with the costs adjusted to account for control flow costs. Assume branch B is currently not relevant to T_t. The arcs $(C \rightarrow D)$, $(D \rightarrow B6_{entry})$, and $(E \rightarrow B6_{entry})$ are control dependent on B, and thus have the profile weight of B, 8, added to their costs. With these penalties added, the min-cut in G_f is either arc $(B6_{entry} \rightarrow G)$ or arc $(G \rightarrow B7_{entry})$ in Figure 4.4(b). Both these cuts correspond to placing the communication of $r1$ in block B6, thus avoiding adding branch B to T_t's set of relevant branches.

Notice that, after adding these penalties to account for control flow, the problem is not precisely modeled anymore. For instance, multiple arcs including a penalty for one branch will include the cost of this branch, and thus a cut including two or more of these arcs will be over-penalized. However, since the arcs' costs are used to choose a cut, the information about which arcs will participate in the solution cut is unknown a priori to make the control-flow penalties more precise. Alternative approaches to address this limitation in modeling control flow are possible, but we are unaware of any that is efficient and that guarantees optimality.

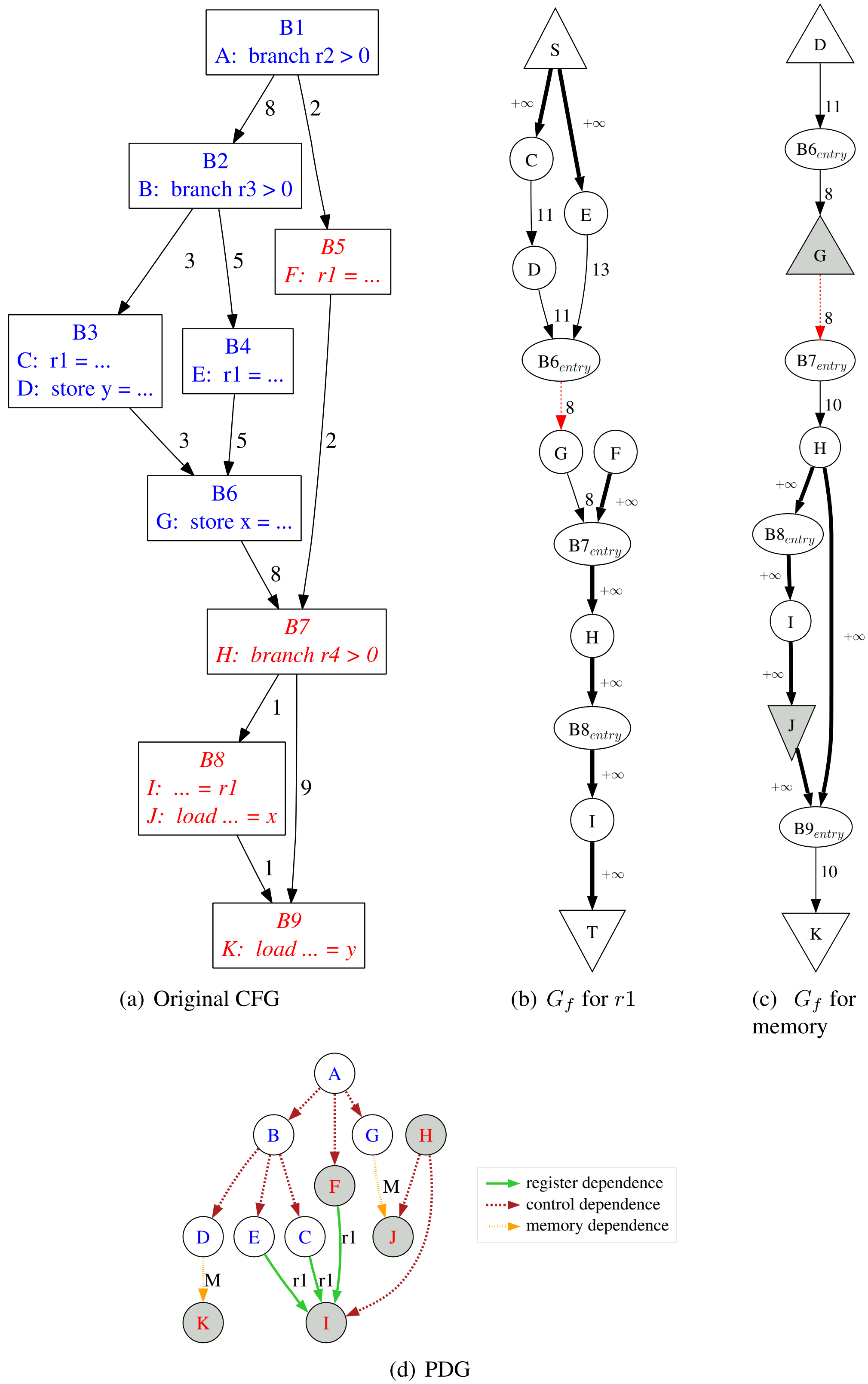


Figure 4.4: An example including memory dependences.

4.2.3 Optimizing Memory Synchronizations

This section describes how synchronization instructions, used to respect inter-thread memory dependences, can also be accurately modeled in terms of graph min-cut.

Although memory dependences are also implemented through queues and `produce` and `consume` instructions (Section 3.1.3), they differ from register dependences in several aspects. First, for memory dependences, no actual operand is sent through the queues, and only the synchronization matters. As a result, multiple memory dependence arcs involving unrelated memory locations can share the same synchronization instructions. This changes a fundamental characteristic of the optimization problem, as described shortly. Another difference compared to register communication is that, for memory, the `produce` and `consume` instructions must have the proper *release* and *acquire* semantics in the memory subsystem. That is, the `produce.rel` must ensure that previous memory-related instructions have committed and, analogously, the `consume.acq` must commit before successive memory-related instructions execute. Compared to the `produce` and `consume` instructions for register communication, the memory versions restrict reordering of instructions in the microarchitecture. This difference is also the reason why register communication instructions cannot be used to satisfy memory dependences.

As mentioned above, the fact that memory dependence arcs involving disjoint sets of memory locations can share synchronization instructions makes the problem different from the register communication one. The reason for this derives from the fact that, while the single-source, single-sink min-cut problem can be efficiently solved in polynomial time, min-cut is NP-hard for multiple source-sink pairs (also called *commodities*), where the goal is to disconnect each source from its corresponding sink [32]. For registers, the problem could be precisely modeled using a single source-sink pair by applying the standard trick of creating special source and sink nodes and connecting them to the rest of the graph appropriately. This was possible because, for a register r, it was necessary to disconnect *every* definition of r in T_s from *every* use in T_t. For memory, however, a similar trick does not lead to an optimal solution. Finding the optimal solution for memory requires optimizing all memory dependences from T_s to T_t simultaneously, since they all can be implemented by the same synchronization instructions. Nevertheless, it is *not necessary* for all sources of memory dependences in T_s to be disconnected from all targets of these dependences in T_t, since dependences can refer to disjoint sets of memory locations. A memory instruction in T_s needs only to be disconnected from its dependent memory instructions in T_t. To accurately model this optimization problem, it is necessary to use a graph min-cut with multiple source-sink pairs.

Notice that, although this difference requires using sub-optimal algorithms for memory optimization in practice, the possibility of sharing synchronization instructions makes the potential optimization impact larger for memory than for registers. In fact, this is confirmed in the experiments in Section 4.4.2.

We now describe how to construct the graph G_f to optimize memory dependences from thread T_s to thread T_t. Although the register optimization could be restricted to the region corresponding to the register's live-range, this is not always possible for memory due to weak memory updates and the impossibility of eliminating all false dependences through renaming. For this reason, the nodes in G_f for memory may need to correspond to the entire region being parallelized. Akin to what was described for registers, G_f here includes nodes corresponding to basic block entries. As explained above, to precisely model the memory optimization problem, it is necessary to use multiple source-sink pairs in G_f. Specifically, for each memory dependence arc from T_s to T_t in the PDG, a source-sink pair is created for its source and target instructions.

The costs on G_f's arcs for memory are the same as for registers, with two differences. First, there is no notion of safety for memory synchronization. In other words, since no operand is communicated for memory dependences, no arc is prohibited from participating in the cut because it is not safe to synchronize at that point. The second difference is that, since the source (sink) nodes here correspond to real instructions in the program, their outgoing (incoming) arcs are allowed to participate in the cut and thus do not have their costs set to infinity. Similar to what is used for register dependences, arcs dependent on irrelevant branches to either T_s or T_t have their costs set specially.

As an example, consider the placement of memory synchronization for the code in Figure 4.4. There are two cross-thread memory dependences from T_s to T_t: $(D \rightarrow K)$ involving variable y, and $(G \rightarrow J)$ involving variable x. Figure 4.4(c) illustrates the G_f graph constructed as described, with the two source-sink pairs distinguished by different shades. The arcs from node H all the way down to $B9_{entry}$ have infinite cost because they are control dependent on branch H, which is not relevant to T_s. The min-cut solution in the example is to cut the arc $(G \rightarrow B7_{entry})$, with a cost of 8.

Given the NP-hardness of the min-cut problem with multiple source-sink pairs, the following heuristic solution is used in this work. The optimal single-source-sink algorithm is successively applied to each source-sink pair. When an arc is cut to disconnect a pair, it is removed from the graph so that this can help disconnecting subsequent pairs. As illustrated in our experiments, this simple heuristic performs well in practice.

4.3 Optimizing Multiple Threads

We now turn to optimizing the communication for multiple threads. COCO tackles this more general problem by relying on the pairwise algorithms described above.

Algorithm 2 presents the pseudo-code for COCO. As input, COCO takes the original CFG and PDG for the region being parallelized, as well as the partition into threads specified by the partitioner. As output, this algorithm returns the set of inter-thread dependences annotated with the points in the program where the communication instructions should be inserted. These annotations can be directly used to place communications in a slightly modified version of the MTCG algorithm presented in Algorithm 1.

Algorithm 2 COCO

Require: *CFG, PDG*, $\mathcal{P}$
1: $G_T \leftarrow$ Build_Thread_Graph(*PDG*, $\mathcal{P}$)
2: $\mathit{relevantBr} \leftarrow$ Init_Relevant_Branches(G_T, $\mathcal{P}$)
3: $\mathit{deps} \leftarrow \emptyset$
4: **repeat**
5: $\quad \mathit{oldDeps} \leftarrow \mathit{deps}$
6: $\quad \mathit{deps} \leftarrow \emptyset$
7: $\quad$ **for each** arc $(T_s \rightarrow T_t) \in G_T$ [in topological order] **do**
8: $\quad\quad \mathit{stDeps} \leftarrow \emptyset$
9: $\quad\quad$ **for each** register r to be communicated from T_s to T_t **do**
10: $\quad\quad\quad G_f \leftarrow$ Build_Flow_Graph_for_Register($r, T_s, T_t, \mathit{relevantBr}$)
11: $\quad\quad\quad \mathit{commArcs} \leftarrow$ Min_Cut(G_f)
12: $\quad\quad\quad \mathit{stDeps} \leftarrow \mathit{stDeps} \cup \{(r, T_s, T_t, \mathit{commArcs})\}$
13: $\quad\quad$ **end for**
14: $\quad\quad G_f \leftarrow$ Build_Flow_Graph_for_Memory($T_s, T_t, \mathit{relevantBr}$)
15: $\quad\quad \mathit{commArcs} \leftarrow$ Min_MultiCut(G_f)
16: $\quad\quad \mathit{stDeps} \leftarrow \mathit{stDeps} \cup \{(\mathit{MEM}, T_s, T_t, \mathit{commArcs})\}$
17: $\quad\quad$ Update_Relevant_Branches($\mathit{relevantBr}[T_t]$, *stDeps*)
18: $\quad\quad \mathit{deps} \leftarrow \mathit{deps} \cup \mathit{stDeps}$
19: $\quad$ **end for**
20: **until** $\mathit{oldDeps} = \mathit{deps}$
21: **return** deps

The first step in COCO (line 1) is to build a *thread graph* G_T, representing the dependences between threads. For each thread, there is a node in G_T. There is an arc $(T_s \rightarrow T_t)$ in G_T if and only if there is a PDG dependence arc (including transitive control arcs) from an instruction in thread T_s to an instruction in another thread T_t. COCO successively optimizes the communications between each pair of threads connected by an arc in G_T.

The algorithm iteratively computes a set of inter-thread dependences (*deps*) annotated with their corresponding communication insertion points. Besides that, the algorithm maintains the set of relevant branches to each thread, computed according to Definition 3. At the beginning (line 2 in the algorithm), the sets of relevant branches are initialized following rules 1 and 3 in Definition 3. Later, as the insertion points for communication are computed, these sets grow using rules 2 and 3 in this definition. Although not illustrated in Algorithm 2, the set of relevant points to each thread, derived from the set of relevant branches according to Definition 4, is also maintained.

The algorithm iterates until the set of dependences with insertion points converges (*repeat-until* in lines 4-20). Iteration is necessary in general because, to satisfy the input dependences of a thread T_i, other branches may become relevant to it. However, changing the set of T_i's relevant branches can affect the best placement for T_i's output dependences. That is because, to satisfy Property 3, no communication is allowed on irrelevant points for the source thread. Iteration can, however, be avoided in the special case when G_T is acyclic, by computing the placement for a thread's input dependences before for its output dependences.

The *for* loop in lines 7-19 computes the placement of communication for each pair of threads connected by an arc in the thread graph. As mentioned previously, following a (quasi-)topological order of G_T's arcs here reduces the number of iterations of the *repeat-until* loop. For each arc $(T_s \rightarrow T_t)$ in G_T, the placement of communication is computed as described in Sections 4.2.1 through 4.2.3 above. That is, each register is optimized separately, and all memory dependences are optimized simultaneously. In each case, optimizing the communication placement involves creating a flow graph with costs on arcs, and then computing a min-cut in this graph. A tuple indicating the register involved in the dependence (or memory), its source and target threads, along with the communication insertion points computed, is then inserted in the set of dependences. Finally, on line 17, the set of relevant branches for the target thread is augmented to account for new branches that just became relevant to satisfy some dependences.

Algorithm 2 is guaranteed to converge because the sets of relevant branches are only allowed to grow, and the number of branches in the region is obviously finite.

Similar to code generated by the original MTCG algorithm, the code produced using COCO is also guaranteed to be deadlock-free. In both cases, this can be proved using the fact that pairs of `produce` and `consume` instructions are inserted at corresponding points in the original code. In fact, slightly more care is necessary with COCO because many communications may be chosen to be inserted at the same program point. For example, COCO may choose the same program point both to communicate a value from a thread T_1 and T_2, and to communicate a value from T_2 to T_1. In this case, if the communication instructions are inserted arbitrarily, both `consume`s may be inserted before the `produce`s, resulting in a deadlock. Two simple solutions can be used to avoid this problem. The first one is to impose an arbitrary total order among the dependences, and to enforce that, for all instructions to be inserted at a given program point, the communication instructions are inserted according to the order of the dependences they implement. An alternative solution is to put, for all instructions to be inserted at the same program point, all the `produce`s before all the `consume`s.

4.4 Experimental Evaluation

In this section, we describe our experimental setup and evaluate COCO. Our experimental setup includes the compiler infrastructure in which COCO was implemented, the target architecture, and the benchmark programs used. Based on these, we then present results that demonstrate COCO's effectiveness in reducing the communication instructions and improving performance of the applications.

4.4.1 Experimental Methodology

This section describes the general methodology used in the experiments, not only for COCO, but throughout this thesis. In the following, we describe the compiler in which our techniques were implemented, the simulator model utilized, as well as the benchmarks used for evaluation.

Compiler Infrastructure

The compilation techniques proposed in this work have been implemented both in the IMPACT compiler [96], from the University of Illinois, and in the VELOCITY compiler developed in our group [105]. For consistency, all the evaluations presented in this thesis use code produced by the VELOCITY compiler.

VELOCITY is a multi-threading research compiler. Currently, VELOCITY has an Itanium 2 [44] back-end, and it uses the front-end of the IMPACT compiler [96] to obtain a low-level intermediate representation (IR), called L-code. IMPACT's L-code IR is then translated into the equally low-level VELOCITY's X-code. Traditional machine-independent code optimizations are performed in VELOCITY, as well as some Itanium-2-specific optimizations. The GMT scheduling techniques in VELOCITY are performed after traditional optimizations, before the code is translated to Itanium 2's assembly, where Itanium-2-specific optimizations are performed, followed by register allocation and the final single-threaded instruction scheduling pass.

The PDG used by our GMT instruction scheduling framework is built from VELOCITY's low-level IR. Control and register data dependences are computed using VELOCITY's data-flow analysis framework. For memory dependences, we use the results of a context-sensitive, flow-insensitive pointer analysis performed in IMPACT [70]. The results of this analysis are annotated in the L-code and translated into VELOCITY's X-code. These memory aliasing annotations may be overly conservative due to both limitations in IMPACT's pointer analysis and the conservative propagation during code optimizations in VELOCITY [36]. Overly conservative memory annotations result in unnecessary synchronizations to be inserted by our MT code generation algorithms.

For each parallelized region, the compiler creates news functions containing the code to be executed by each of the auxiliary threads. Before entering a parallelized region, the main thread sends to each auxiliary thread the address of the corresponding auxiliary function on a specific queue (a *master queue*). Then each auxiliary thread, which is blocked on a `consume` operation on its master queue, wakes up and simply calls the function whose address it receives. Upon termination of a parallelized region, the corresponding auxiliary function returns to the master auxiliary function, which loops back to the `consume` instruction. The auxiliary threads then block again on their master queues, waiting for the next request from the main thread. The auxiliary threads terminate upon receiving through their master queues a special value, which consists of a NULL function pointer.

Hardware Model

To evaluate the performance of the code generated by VELOCITY, we used a cycle-accurate CMP model comprising a parameterized number of Itanium 2 cores. The core models are validated, with IPC and constituent error components accurate to within 6% of real hardware for measured benchmarks [79]. In our model, the cores are connected by the *synchronization array* communication mechanism proposed by Rangan et al. [85] and discussed in Section 1.2. In Figure 4.1, we provide details about the simulator model, which was built using the Liberty Simulation Environment [107].

Core	Functional Units: 6 issue, 6 ALU, 4 memory, 2 FP, 3 branch L1I Cache: 1 cycle, 16 KB, 4-way, 64B lines L1D Cache: 1 cycle, 16 KB, 4-way, 64B lines, write-through L2 Cache: 5,7,9 cycles, 256KB, 8-way, 128B lines, write-back Maximum Outstanding Loads: 16
Shared L3 Cache	$>$ 12 cycles, 1.5 MB, 12-way, 128B lines, write-back
Main Memory	Latency: 141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration

Table 4.1: Machine details.

The synchronization array (SA) in the model works as a set of low-overhead queues. In our base model, there is a total of 256 queues, each with 32 elements. Each of these elements can hold a single scalar value, which can be either an integer or a floating-point number. Our MT code generation algorithms guarantee the correctness of the produced code for any queue size greater than zero. The SA has a 1-cycle access latency, and it has four request ports that are shared between the two cores. The Itanium 2 ISA was extended with `produce` and `consume` instructions for inter-thread communication, and the GNU Assembler was extended accordingly. These instructions use the M pipeline, which is also used by memory instructions. This imposes the limit that only 4 of these instructions (minus any other memory instructions) can be issued per cycle on each core, since the Itanium 2 can issue only four M-type instructions in a given cycle. While the `consume` instructions can access the SA speculatively, the `produce` instructions write to the SA only on commit. As long as the SA queue is not empty, a `consume` and its dependent instructions can execute in back-to-back cycles.

The highly-detailed nature of the validated Itanium 2 model prevented whole-program simulation. Instead, detailed simulations were restricted to the parallelized region in each benchmark. We fast-forwarded through the remaining sections of the program while keeping the caches and branch predictors warm. The high accuracy of our simulation model comes at the cost of simulation time. This makes whole-program simulations impractical. For this reason, we use the sample-based simulation methodology developed by Rangan [82]. The complexity of our Itanium 2 core model also makes it impractical to build CMP models with many cores. The larger the number of cores, the larger are both the simulator binary and the simulation time.

Benchmark Programs

For our experiments, we used the set of benchmarks that VELOCITY is currently able to parallelize using the techniques proposed in this thesis. These techniques were applied to the applications from the MediaBench [58], SPEC-CPU [99], and Pointer-Intensive [6] benchmark suites that currently go through our tool-chain. To reduce simulation time, the parallelization and simulations were restricted to important functions in these benchmarks, generally corresponding to at least 25% of the benchmark execution. In Figure 4.2, we list the selected application functions along with their corresponding benchmark execution percentages.

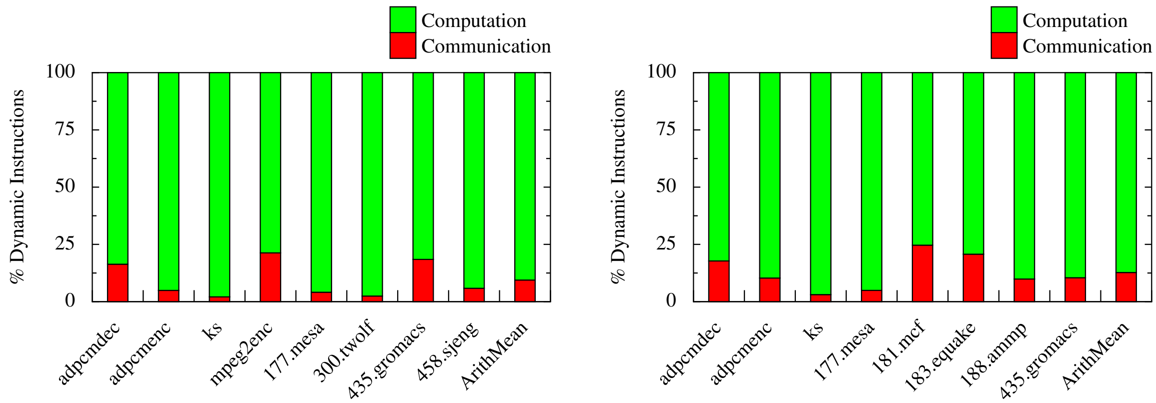
Benchmark	Description	Selected Function	Exec. %
adpcmdec	speech decompression	adpcm_decoder	100
adpcmenc	speech compression	adpcm_coder	100
ks	Kernighan-Lin graph partitioning	FindMaxGpAndSwap	100
mpeg2enc	video encoder	dist1	58
otter	theorem prover for first-order logic	find_lightest_geo_child	15
177.mesa	3-D graphics library	general_textured_triangle	32
179.art	image recognition using neural networks	match	49
181.mcf	combinatorial optimization	refresh_potential	32
183.equake	seismic wave propagation simulation	smvp	63
188.ammp	computational chemistry	mm_fv_update_nonbon	79
300.twolf	transistor placement and routing	new_dbox_a	30
435.gromacs	molecular dynamics simulation	inl1130	75
456.hmmer	hidden Markov model	P7_Viterbi	85
458.sjeng	chess program	std_eval	26

Table 4.2: Selected benchmark functions.

4.4.2 Experimental Results

This section presents an evaluation of the COCO framework. In particular, we illustrate the effectiveness of COCO in reducing the communication instructions and improving performance, compared to the basic MTCG algorithm. We evaluate COCO in scenarios that use two thread-partitioning techniques presented in the subsequent chapters: GREMIO (Chapter 5) and DSWP (Chapter 6). The specifics of these techniques are mostly irrelevant here, and they are just used to demonstrate the improvements of using COCO over the simple communication strategy described in Chapter 3. To demonstrate COCO's relevance in the face of different hardware support, the experiments for GREMIO use communication queues with a single element, instead of the 32-element default.

VELOCITY has profile-weight information annotated on its IR, which was used for the costs on the G_f graphs for min-cut. The profiles were collected on smaller, *train* input sets, while the results presented here were run on larger *reference* inputs.



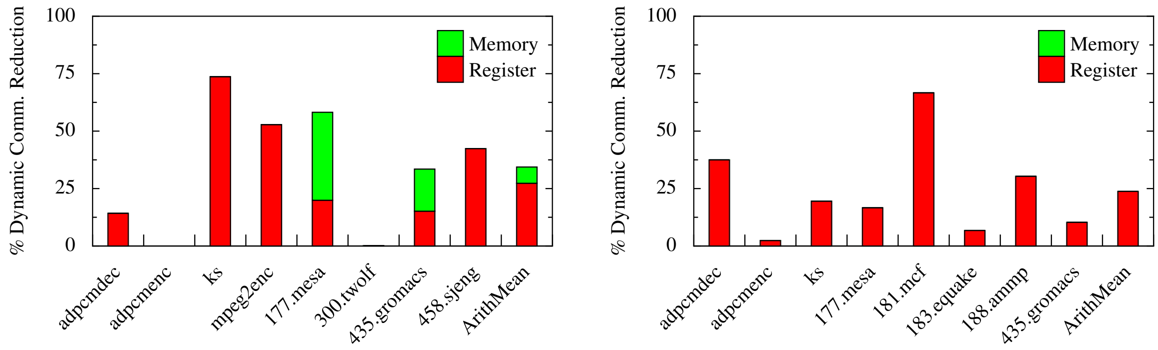
(a) GREMIO (b) DSWP

Figure 4.5: Breakdown of dynamic instructions in code generated by the basic MTCG algorithm (without COCO), for (a) GREMIO and (b) DSWP.

Results for Two Threads

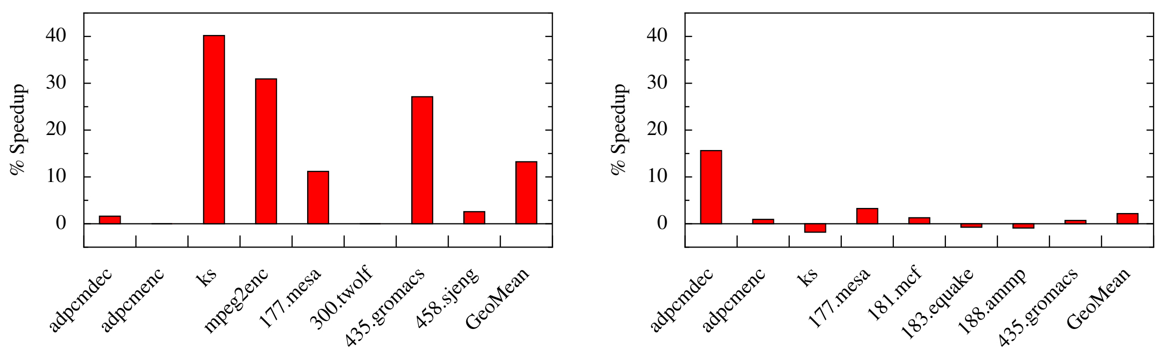
Because of the large number of dependences in general-purpose applications, the overhead of communication instructions can be quite significant in code generated by GMT instruction scheduling techniques. To illustrate this, Figure 4.5 shows the dynamic percentages of *communication* instructions compared to the original instructions in the program (the *computation*) for various benchmarks parallelized by GREMIO and DSWP using the basic MTCG communication placement. As illustrated, the communication instructions can account for up to one fourth of the total instructions. COCO's goal is to eliminate some of these communication instructions. This can improve performance not only by reducing the number of executed instructions, but also by eliminating synchronization points, thus enabling more TLP.

In Figure 4.6, we show the percentages of dynamic communication instructions that are eliminated when COCO is applied, relative to the codes using the original MTCG algorithm's communication strategy. The average reduction of the dynamic communication instructions was 34.4% for GREMIO, and 23.8% for DSWP. The largest reduction occurred for *ks* with GREMIO (73.7%), where an inner-loop whose only purpose was to consume a live-out could be completely removed from a thread, similar to the example in Figure 4.3.



(a) GREMIO (b) DSWP

Figure 4.6: Reduction in the dynamic communication / synchronization instructions by applying COCO, compared to the basic MTCG algorithm.



(a) GREMIO (b) DSWP

Figure 4.7: Speedup of using COCO over code generated by the basic MTCG algorithm.

In only one small case, *adpcmenc* with GREMIO, COCO had no opportunity to reduce communication. COCO never resulted in an increase in the number of dynamic communication instructions. Only two applications, *177.mesa* and *435.gromacs* with GREMIO, had inter-thread memory dependences. For both of these, COCO was able to remove more than 99% of the dynamic memory synchronizations. This confirms the great potential to eliminate memory synchronizations mentioned in Section 4.2.3.

COCO had a smaller impact on codes partitioned by DSWP, in part, because no inter-thread memory dependences can happen in this case. That is because the PDG employed in this work uses the results of a points-to static analysis [70]. And, since the instructions are inside a loop, any memory dependence is essentially bi-directional, thus forcing these instructions to be assigned to the same thread in order to form a pipeline [74]. With more powerful, loop-aware memory disambiguation techniques to eliminate false memory dependences, such as shape analysis or array-dependence analysis, DSWP can result in inter-thread memory dependences and thus benefit more from COCO.

Another reason for COCO's larger improvements for GREMIO, compared to DSWP, was the smaller queue sizes used in GREMIO's experiments. In some cases, a larger queue can compensate for inefficient communication placements. This happens, for example, if communications are unnecessarily placed inside loops. From another perspective, this means that COCO can alleviate the demand for more complex communication mechanisms.

In Figure 4.7, we present the speedups for the benchmarks parallelized with GREMIO and DSWP over their versions using the simple communication placement strategy. In general, the speedups correlate with the reduction of dynamic communication instructions shown in Figure 4.6. The average speedup for GREMIO improves by 13.2%, while the average improvement is 2.2% for DSWP. The maximum speedup is for *ks* with GREMIO, for which COCO provided a 40.2% speedup. In this benchmark, although the communication instructions were only a small percentage, they created critical synchronization points that hindered a lot of parallelism. For *mpeg2enc*, COCO optimized the register communication in various hammocks, also significantly reducing the control flow in the generated threads. In general, COCO improves performance not only by reducing the number of dynamically executed instructions, but also by increasing TLP through the removal of memory synchronizations and control dependences. For memory synchronizations, the reason is that the `consume.acq` instructions must wait for their corresponding synchronization token to arrive. For control dependences, the reason is that Itanium 2 uses a *stall-on-use* strategy, and control dependences are implemented as replicated branches that actually use their register operands. Removing register dependences has less effect because an outstanding `consume` instruction does not stall the pipeline until its consumed register is actually used.

In a couple of cases, COCO, despite reducing the communication instructions, degraded performance slightly. The reason for this was a bad interaction with the later single-threaded instruction scheduler, which plays an important role for Itanium 2. To reduce the number of communication instructions executed, COCO sometimes moves these instructions from program points where free scheduling slots are available to points that increase the schedule height. We envision two alternatives to avoid this problem. One is to add, in the graph used for min-cut, penalties to arcs so as to take scheduling restrictions into account. Another alternative is to change the priority of the `produce` and `consume` instructions in the single-threaded scheduler.

Results for More than Two Threads

In this section, we present the speedups resulting from COCO for the DSWP thread partitioning when targeting 4 and 6 threads. Figure 4.8 shows the results. These results only include the benchmarks for which the DSWP thread partitioner produced the requested number of threads. For 4 threads, the geometric-mean speedup is 1.6%, with a maximum of 3.9% for *458.sjeng*. For 6 threads, the geometric-mean speedup is 4.4% speedup, with a maximum of 15.9% for *188.ammp*. Notice that there is no correlation between the 2-thread, 4-thread, and 6-thread results for individual benchmarks. The reason for this is that the opportunities for COCO are completely dependent on the chosen partition. Once the partition changes for a benchmark, even for the same number of threads, it essentially becomes a completely different scenario for COCO. In Figure 4.8, we notice that the average speedup increases with the number of threads. This trend is expected because, as the number of threads increase, the ratio of the communication instructions over the computation instructions tends to increase. And, the higher the communication overhead, the larger the potential for COCO.

Compilation Time

Our current implementation of COCO uses Edmonds-Karp's min-cut algorithm [21], which has a worst-case time complexity of $O(n \times m^2)$, where n and m are the number of vertices

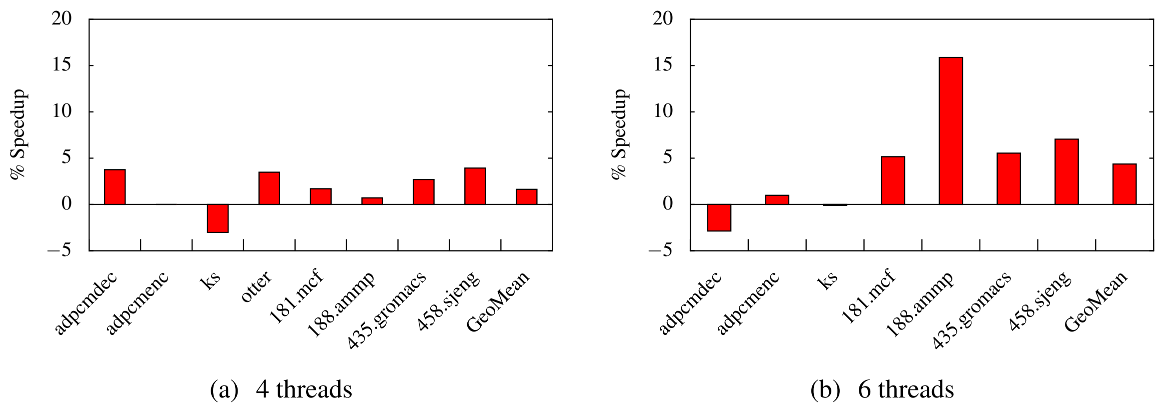


Figure 4.8: Speedup of using COCO over code generated by the basic MTCG algorithm for DSWP with (a) 4 threads, and (b) 6 threads.

and arcs in the graph, respectively. Since for CFGs m is usually $\Theta(n)$, this worst-case complexity approximates $O(n^3)$ in practice. In our experiments, this algorithm performed well enough not to significantly increase VELOCITY's compilation time. For production compilers, faster min-cut algorithms can be employed if necessary.

4.5 Related Work

This section compares and contrasts our communication optimizations to related work in the literature. Global instruction scheduling for multi-threaded architecture is a relatively new research topic. In this chapter, we evaluated the COCO framework for two GMT instruction scheduling techniques, DSWP [74] and GREMIO [72], both based on the MTCG algorithm. Although not evaluated in this work, COCO should also benefit a recently proposed technique that combines speculation with DSWP [109], which also uses MTCG.

Local MT instruction (LMT) scheduling techniques differ from the GMT in that they duplicate most of the program's CFG for each thread, thus mostly exploiting instruction-level parallelism within basic blocks. Similar to GMT, LMT techniques also need to insert communication instructions in order to satisfy inter-thread dependences. The Space-Time scheduling [59] LMT technique uses several simple invariants to make sure each thread gets the latest value of a variable before using it. First, each variable is assigned to a home node, which is intended to contain the latest value assigned to this variable. Second, each thread/node that writes to that variable communicates the new value to the home node right after the new value is computed. Finally, at the beginning of each basic block that uses a variable in a thread other than its home, a communication of this variable from its home node is inserted. This strategy is somewhat similar to the one used in the original MTCG algorithm, and other LMT techniques use similar approaches [60, 88]. Given their similarity to the original MTCG algorithm's strategy, they could also benefit from COCO to reduce the communication instructions inserted in the code.

For clustered single-threaded architectures, the scheduler also needs to insert communication instructions to move values from one register bank to another [11, 69, 118]. However, the fact that dependent instructions are executed in different threads makes the generation and optimization of communication more challenging for multi-threaded architectures. The technique of [11] also uses graph partitioning algorithms.

Another piece of related work is the compiler communication optimization proposed for Thread-Level Speculation (TLS) by Zhai et al. [119]. There are several differences between the communication optimizations for TLS and GMT scheduling. First, each thread in TLS operates on a different loop-iteration, and therefore there are clear notions of order and of which thread has the latest value of a variable. Second, the communication between the threads is always uni-directional for TLS. Third, each thread only receives values from one upstream thread and sends values to one downstream thread. All these differences make the problem for TLS significantly simpler, and Zhai et al. [119] propose a technique based on partial redundancy elimination (PRE) [53, 64] to minimize the communication. For GMT scheduling, we demonstrated that the optimization problem is statically undecidable, and proposed a profiling-based technique which uses graph min-cut algorithms. Furthermore, to deal with the possibility of arbitrary inter-thread communication patterns, we developed a notion of safety (Property 4), and proposed a thread-aware data-flow analysis to compute it.

Communication optimizations are also important for compiling data-parallel applications for distributed-memory machines [3, 12, 48, 50]. The main differences from the problem there and the one studied in this paper are the following. First, there is an enormous discrepancy in the parallelism available in the applications, and how the parallelism is expressed by the programmer. This allows message-passing compilers to concentrate on a more regular style of parallelism, SPMD (single program multiple data), where all processors execute the same code. The irregular structure and fine granularity of the parallelism available in general-purpose applications require GMT scheduling to exploit more general forms of parallelism. Furthermore, the main communication optimization for message-passing systems is *communication combination*, where multiple messages are combined in a larger message to amortize overhead. Since GMT scheduling uses a scalar communication mechanism, these optimizations are not applicable in this context. In spirit, the optimizations proposed in this paper are closer to *redundancy optimizations* for distributed-memory systems. However, the techniques for data-parallel codes are very different, being strongly based on loops and array accesses and frequently unable to handle arbitrary control flow [48]. Another optimization proposed for message-passing systems is *pipelining*, where the message is sent earlier than where it is consumed, in order to hide communication latency. This is somewhat accomplished in our techniques by a combination of choosing the *earliest* min-cut placement (i.e. closest to the source), and the stall-on-use implementation of the `consume` instruction.

4.6 Significance

This chapter presented the COCO framework, which consists of advanced algorithms to optimize the placement of communication instructions. By improving the MTCG algorithm from Chapter 3, COCO can reduce the communication overhead and improve the performance of every technique based on our global multi-threaded scheduling framework. The experiments in this chapter demonstrated that COCO can unlock large amounts of parallelism in some cases, particularly when it eliminates key synchronization points and control flow. Nevertheless, in many cases COCO resulted in small speedups when compared to the basic MTCG algorithm from Chapter 3. This indicates that the communication-placement strategy described in Chapter 3, despite its simplicity, generally produces well-performing code.

Part II

Thread-Partitioning Techniques

Based on our global multi-threaded instruction scheduling framework, this part describes three thread-partitioning techniques. Chapter 5 describes a thread partition based on list scheduling, called GREMIO. Based on the PDG representation, GREMIO generalizes list scheduling to operate on arbitrary, potentially cyclic code regions. Moreover, GREMIO is aware of multiple levels of resources, which is useful to balance thread-level and instruction-level parallelism.

In Chapter 6, we describe the Decoupled Software Pipelining (DSWP) thread partitioning. DSWP partitions the PDG nodes among threads with the goal of exploiting pipelined parallelism. In some sense, DSWP generalizes software pipelining [13] to exploit thread-level parallelism. Based on our global multi-threaded instruction scheduling framework, DSWP enables pipelined parallelism to be exploited from loop nests with arbitrary control flow. This overcomes a huge limitation of traditional software pipelining.

Despite its wide applicability, DSWP lacks performance scalability beyond a few threads. In Chapter 7, we describe an extension of DSWP that uses parallel stages in the pipeline to obtain better scalability.

Chapter 5

Global List-Scheduling-Based Thread Partitioning

This chapter describes a thread partitioning technique called GREMIO[1] (Global REgion Multi-threaded Instruction Orchestrator) [72]. GREMIO is based on list scheduling [35], which is a well-know scheduling technique applied in many areas. In compilers, list scheduling is noticeably used in single-threaded instruction scheduling [65].

GREMIO uses the PDG as an intermediate representation for scheduling decisions. Using the PDG to guide scheduling decisions is attractive because it makes explicit the communications that will be incurred. In other words, scheduling two dependent instructions to different threads will require an inter-thread communication. A problem that arises from using a PDG for scheduling decisions is the presence of cycles. The PDG for an arbitrary code region can have cycles due to loops in the CFG and loop-carried dependences. Scheduling cyclic graphs is more complicated than scheduling acyclic graphs. This is because the goal of a scheduler is to minimize the critical (i.e. longest) path through the graph. Although scheduling of acyclic graphs in the presence of resource constraints is NP-hard, at least finding the critical path in such graphs can be solved in linear time, through a topological sort. For cyclic graphs, however, even finding the longest path is NP-hard [32].

[1] In Portuguese, *grêmio* means club. The Grêmio Foot-Ball Porto-Alegrense, usually known simply as Grêmio, is one of the best soccer teams in Brazil.

5.1 GREMIO Algorithm

This section describes the GREMIO algorithm to choose a thread partition based on the PDG. Throughout this section, we illustrate the steps of GREMIO in the example in Figure 5.1. This is the same example from Section 2.3, which is replicated here for convenience. Figure 5.1(a) shows the C source code, and Figure 5.1(b) contains the corresponding low-level code. The CFG and PDG for this code are illustrated in Figures 5.1(c) and (d), respectively. As we will demonstrate in detail, for this example, GREMIO partitions the code into two threads as depicted by the vertical dashed line in Figure 5.1(d). This partition corresponds to scheduling each loop of Figure 5.1(a) into a separate thread.

Given the inherent difficulty of the global scheduling problem for cyclic code regions, GREMIO uses a simplifying approach that reduces this problem to an acyclic scheduling problem, for which well-known heuristics based on list scheduling exist [65]. In order to reduce the cyclic scheduling problem to an acyclic one, GREMIO uses two simplifications to the problem. First, when scheduling a given code region, each of its inner loops is coalesced to a single node, with an aggregated latency that assumes its average number of iterations (based on profiling or static estimates). Secondly, if the code region being scheduled is itself a loop, all its loop-carried dependences are disregarded. To deal with the possibility of irreducible code, a loop hierarchy that includes irreducible loops is used [39]. It is important to note that these simplifying assumptions are used for partitioning decisions only; for code generation, GREMIO relies on the MTCG algorithm (Chapter 3), which takes *all* dependences into account to generate correct code.

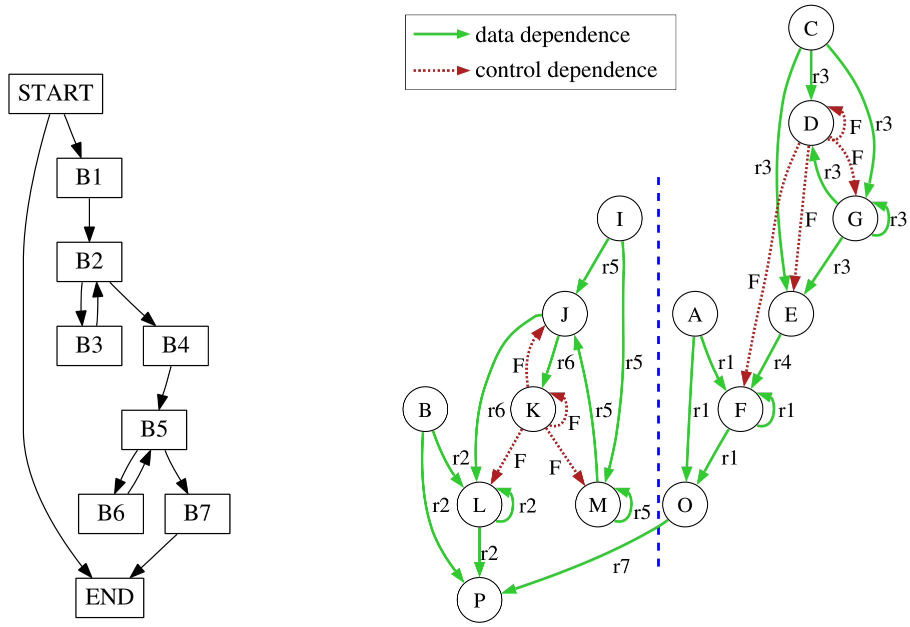
To distinguish from a full PDG, we call the dependence graph for a region with its inner loops coalesced and its loop-carried dependences ignored a *Hierarchical Program Dependence Graph* (HPDG). In a HPDG, the nodes represent either a single instruction, called a *simple node*, or a coalesced inner loop, called a *loop node*. To account for the possibility of irreducible code, a loop hierarchy that includes irreducible loops is used [39]. Figure 5.2(a) illustrates the HPDG corresponding to the PDG from Figure 5.1(d). The

```
s1 = 0;
s2 = 0;
for(p=head; p != NULL;
    p = p->next){
  s1 += p->value;
}
for(i=0; a[i] != 0; i++){
  s2 += a[i];
}
printf("%d\n", s1*s1/s2);
```

(a) Example code in C

```
(A) B1: move   r1 = 0        ;; s1 in r1
(B)     move   r2 = 0        ;; s2 in r2
(C)     load   r3 = [head]   ;; p in r3
(D) B2: branch r3 == 0, B4
(E) B3: load   r4 = [r3]     ;; p->value
(F)     add    r1 = r1, r4
(G)     load   r3 = [r3+4]   ;; p->next
(H)     jump   B2
(I) B4: move   r5 = @a       ;; &a[i] in r5
(J) B5: load   r6 = [r5]     ;; load a[i]
(K)     branch r6 == 0, B7
(L) B6: add    r2 = r2, r6
(M)     add    r5 = r5, 4
(N)     jump   B5
(O) B7: mult   r7 = r1, r1
(P)     div    r8 = r7, r2
```

(b) Low-level IR



(c) CFG

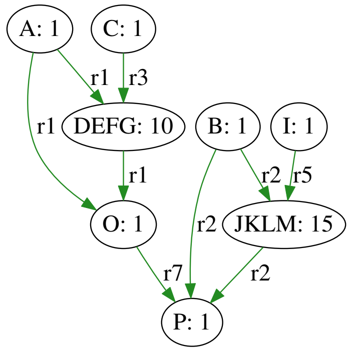
(d) PDG

Figure 5.1: Example code in: (a) C, (b) low-level IR, (c) CFG, and (d) PDG.

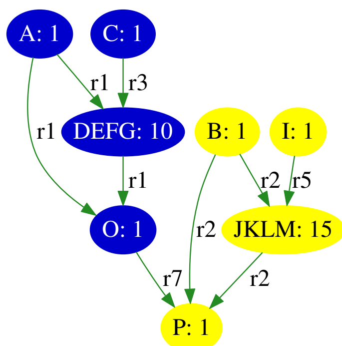
nodes are labeled by their corresponding nodes in the PDG, followed by their estimated execution latency. There are only two loop nodes in this example: $DEFG$ and $JKLM$.

Another complication intrinsic to MT scheduling is that the generated threads need to communicate to satisfy dependences, and so it is necessary to take the communication overhead into account while making scheduling decisions.[2] For instance, even though two

[2]This complication also arises in single-threaded instruction scheduling for architectures with partitioned



(a) HPDG



(b) Clustered HPDG

	Core 0		Core 1	
cycle	issue 0	issue 1	issue 0	issue 1
0	A	C	B	I
1	D E	F G	J K	L M
2	D E	F G	J K	L M
3	D E	F G	J K	L M
4	D E	F G	J K	L M
5	D E	F G	J K	L M
6	D E	F G	J K	L M
7	D E	F G	J K	L M
8	D E	F G	J K	L M
9	D E	F G	J K	L M
10	D E	F G	J K	L M
11	O		J K	L M
12	prod r7		J K	L M
13			J K	L M
14			J K	L M
15			J K	L M
16			cons r7	
17			P	

(c) Virtual Schedule

Figure 5.2: Operation of GREMIO on the example from Figure 5.1.

instructions can be executed in parallel on different threads, this may not be profitable due to the overhead to communicate their operands. To address this problem, GREMIO uses a *clustering* pre-scheduling pass on the HPDG, which takes into account the inter-thread communication overhead. The goal of this pass is to cluster together HPDG nodes that are likely to not benefit from schedules that assign them to different threads. Section 5.1.1 explains the clustering algorithm used by GREMIO, and Section 5.1.2 describes its partitioning heuristic.

5.1.1 Clustering Algorithm

There exist a variety of clustering algorithms in the parallel computing literature. These algorithms are used for task scheduling, being applicable to arbitrary directed acyclic

register files.

graphs (DAGs). Therefore, because we reduced the original cyclic scheduling problem (on a PDG) to an acyclic problem (on a HPDG), we can rely on previous research on DAG-clustering algorithms.

We chose to use the *Dominant Sequence Clustering (DSC)* algorithm [117], which has been shown to be very effective and efficient. Efficiency is important here because, given the fine granularity of the nodes in a HPDG, their number can be very large (on the order of thousands).

DSC, like other clustering algorithms, groups nodes in clusters so that nodes in the same cluster are unlikely to benefit from executing in parallel. Therefore, all nodes in the same cluster should be scheduled on the same processor (thread here). DSC also assumes that each cluster will be executed on a different processor. Later, the scheduling pass can assign multiple clusters on the same thread to cope with a smaller number of processors.

Briefly, DSC operates as follows. In the beginning, each node is assigned to its own cluster. The critical path passing through each node of the graph is then computed, considering both the execution latencies of nodes and the communication latencies. The communication latency is assumed to be zero if and only if the nodes are in the same cluster. DSC then processes each node at a time, following a topological order prioritized by the nodes' critical path lengths. At each step, the benefit of merging the node being processed with each of its predecessors is analyzed. The advantage of merging a node with another cluster is that the communication latency from nodes in that cluster will be saved. The downside of merging is that the nodes assigned to the same cluster are assumed to execute sequentially, in the order they are added to the cluster. Therefore, the delayed execution after a merge may outweigh the benefits of the saved communication. The node being processed is then merged with its predecessors' cluster that reduces this node's critical path the most. If all such merges increase the critical path, this node is left alone in its own cluster. For our running example, Figure 5.2(b) illustrates the clusters resulting from DSC assuming a 2-cycle communication latency.

5.1.2 Global Multi-Threaded List Scheduling

After the clustering pass on the HPDG, the actual scheduling decisions are made. Here again, because of the reduction to an acyclic scheduling problem, we can rely on well-known acyclic scheduling algorithms. In particular, GREMIO uses a form of *list scheduling* with resource constraints, with some adaptations to better deal with our problem. This section describes list scheduling and the enhancements used by GREMIO.

The basic list scheduling algorithm assigns priorities to nodes and schedules each node following a prioritized topological order. Typically, the priority of a node is computed as the longest path from it to a leaf node. A node is scheduled at the earliest time that satisfies its input dependences and that conforms to the currently available resources.

GREMIO uses a variation of list scheduling to partition the HPDG into threads. Even though the HPDG is acyclic, control flow still poses additional complications to GMT list scheduling that do not exist in local list scheduling. When scheduling a basic block, local schedulers have the guarantee that all instructions will either execute or not. In other words, all instructions being scheduled are *control equivalent*. Therefore, as long as the dependences are satisfied and resources are available, the instructions can safely be issued simultaneously. The presence of arbitrary control flow complicates the matters for GMT scheduling. First, control flow causes many dependences not to occur during the execution. Second, not all instructions being scheduled are control equivalent. For example, the fact that an instruction X executes may not be related to the execution of another instruction Y, or may even imply that Y will not execute. To deal with the different possibilities, we introduce three different *control relations* among instructions, which are used in GREMIO's list scheduling.

Definition 5 (Control Relations)**.** *Given two HPDG nodes X and Y, we call them:*

1. Control Equivalent*, if both X and Y are simple nodes with the same input control dependences.*

2. Mutually Control Exclusive*, if the execution of X implies that Y does not execute, and vice-versa.*

3. Control Conflicting*, otherwise.*

To illustrate these relations, consider the HPDG from Figure 5.2(a). In this example, A, B, C, I, O, and P are all control equivalent. Nodes $DEFG$ and $JKLM$ are control conflicting with every other node. No pair of nodes is mutually control exclusive in this example.

Although GREMIO uses list scheduling simply to decide the partition and relies on the MTCG algorithm to generate code, GREMIO still builds a schedule of HPDG nodes to cycles. This schedule is not realistic in that it includes all the nodes in a HPDG, even though some of them are mutually control exclusive. For this reason, we call it a *virtual schedule*, and we say the nodes are scheduled on *virtual cycles* in the virtual schedule.

For traditional, single-threaded instruction scheduling, the resources correspond to the processor's functional units. To simplify the discussion, although GREMIO can be applied in general, we assume a CMP with each core single-threaded. In this scenario, there are two levels of resources: the target processor contains multiple cores, and each core has a set of functional units. Instead of simply assuming the total number of functional units in all cores, considering these two levels of resources is important for many reasons. First, it enables us to consider the communication overhead to satisfy dependences between instructions scheduled on different cores. Furthermore, it allows us to benefit from key opportunities available in *multi-threaded* scheduling: the simultaneous issue of control-conflicting instructions. Because each core has its own control unit, control-conflicting instructions can be issued in different cores in the same cycle.

Thread-level scheduling decisions are made when scheduling the first node in a cluster. At this point, the best thread is chosen for that particular cluster, given what has already been scheduled. When scheduling the remaining nodes of a cluster, GREMIO simply schedules them on the thread previously chosen for this cluster.

The choice of the best thread to schedule a particular cluster to takes into account a number of factors. Broadly speaking, these factors try to find a good balance between two conflicting goals: maximizing the parallelism, and minimizing the inter-thread communication. For each thread, the total overhead of assigning the current cluster to it is computed. This total overhead is the sum of the following components:

1. *Communication Overhead*: this is the total number of cycles that will be necessary to satisfy dependences between this cluster and instructions in clusters already scheduled on different threads. This accounts for both overhead inside the cores (extra `produce` and `consume` instructions) and communication delay.
2. *Conflict Overhead*: this is the estimated number of cycles by which the execution of this cluster will be delayed when executing in this thread, considering the current load of unfinished instructions in clusters already assigned to this thread. This considers the both resource conflicts in terms of functional units, as well as control conflicts among instructions.

Once GREMIO chooses the thread to schedule a HPDG node to, it is necessary to estimate the virtual cycle in which that node can be issued in this core. The purpose of assigning nodes to virtual cycles within a thread is to guide the scheduling of the remaining nodes.

In order to find the virtual cycle in which a node can be issued in the chosen thread, it is necessary to consider two restrictions. First, it is necessary to make sure that the node's input dependences will be satisfied at the chosen cycle. For inter-thread dependences, it is necessary to account for the communication latency and corresponding `consume` instructions overhead. Second, the chosen cycle must be such that there are available resources in the chosen core, given the other nodes already scheduled on it. However, not all the nodes already scheduled on this thread should be considered. Resources used by nodes that are mutually control exclusive to this one are considered available since these nodes will never be issued simultaneously. On the other hand, the resource utilization of control equivalent nodes must be taken into account. Finally, the node cannot be issued in the same cycle as any previously scheduled node that has a control conflict with it. This is because each core has a single control unit, but control-conflicting nodes have unrelated conditions of execution. Notice that for target cores supporting predicated execution, however, this is not necessarily valid: two instructions with different execution conditions may be issued in parallel. But even for cores with predication support, loop nodes cannot be issued with anything else.

We now show how GREMIO's list scheduling algorithm works on our running example. For illustration purposes, we use as target a dual-core processor that can issue two instructions per cycle in each core (see Figure 5.2(c)). The list scheduling algorithm processes the nodes in the clustered HPDG (Figure 5.2(b)) in topological order. The nodes with highest priority (i.e. longest path to a leaf) are B and I. B is scheduled first, and it is arbitrarily assigned to core 1's first slot. Next, node I is considered and, because it belongs to the same cluster as B, the core of choice is 1. Because there is an available resource (issue slot) in core 1 at cycle 0, and the fact that B and I are control equivalent, I is scheduled on core 1's issue slot 1. At this point, either nodes A, C, or $JKLM$ may be scheduled. Even though $JKLM$ has the highest priority, its input dependences are not satisfied in the cycle being scheduled, cycle 0. Therefore, $JKLM$ is not a *candidate* node in the current cycle. So node A is scheduled next, and the overheads described above are computed for scheduling A in each thread. Even though thread 1 (at core 1) has lower communication overhead (zero), it has higher conflict overheads. Therefore, core 0 is chosen for node A. The algorithm then proceeds, and the remaining scheduling decisions are all cluster-based. Figure 5.2(c) illustrates the final schedule built and the partitioning of the instructions among the threads.

5.1.3 Handling Loop Nests

Although GREMIO's scheduling algorithm follows the clusters formed a priori, an exception is made when handling inner loops. The motivation to do so is that inner loops may fall on the region's critical path, and they may also benefit from execution on multiple threads.

GREMIO handles inner loops as follows. For now, assume that it has an estimate for the latency to execute one invocation of an inner loop L_j using a number of threads i from 1 up to the number N of threads on the target processor. Let $latency_{L_j}(i)$, $1 \leq i \leq N$, denote these latencies. Considering L_j's control conflicts, the algorithm computes the cycle in which each thread will finish executing L_j's control-conflicting nodes already scheduled on it. From that, the earliest cycle in which a given number of threads i will be available for L_j can be computed, being denoted by $cycle\_available_{L_j}(i)$, $1 \leq i \leq N$. With that, the algorithm chooses the number of threads k on which to schedule this loop node such that $cycle\_available_{L_j}(k) + latency_{L_j}(k)$ is minimized. Intuitively, this will find the best balance between the wait to have more threads available and the benefit from executing the loop node on more threads. If more than k threads are available at $cycle\_available_{L_j}(k)$ (i.e., in case multiple threads become available at this cycle), then the algorithm picks the k threads among them with which the loop node has more affinity. The affinity is computed as the number of dependences between this loop node and nodes already scheduled on each thread.

The question that remains now is: how are the $latency_{L_j}(i)$ values for each child loop L_j in the HPDG computed? Intuitively, this is a recursive question, since the same algorithm can be applied to the child loop L_j, targeting i threads, in order to compute $latency_{L_j}(i)$. This naturally leads to a recursive solution. Even better, dynamic programming can efficiently solve this problem in polynomial time. However, since this constrained scheduling problem is NP-hard, this dynamic programming approach may not be optimal because the list scheduling algorithm applied to each node is not guaranteed to be optimal.

More specifically, GREMIO's dynamic programming solution works as follows. First, it computes the loop hierarchy for the region to be scheduled. This can be viewed as a loop tree, where the root represents the whole region (which need not be a loop). We call such tree a *HPDG tree*. Figure 5.3 illustrates the HPDG tree for the example from Figure 5.1. The algorithm then proceeds bottom-up on the HPDG tree and, for each tree node L_j (either a loop or the whole region), it applies the GMT list scheduling algorithm to compute the latency to execute one iteration of that loop, with a number of threads i varying from 1 to N. The latency returned by the list scheduling algorithm is then multiplied by the average number of iterations per invocation of this loop, resulting in the $latency_{L_j}(i)$ values to be used for this loop node when scheduling its parent. In the end, the algorithm chooses the best schedule for the whole region by picking the number of threads k for the HPDG tree's root, R, such that $latency_R(k)$ is minimized. The corresponding partitioning of instructions onto threads can be obtained by keeping and propagating the partition $partition_{L_j}(i)$ of instructions corresponding to the value of $latency_{L_j}(i)$.

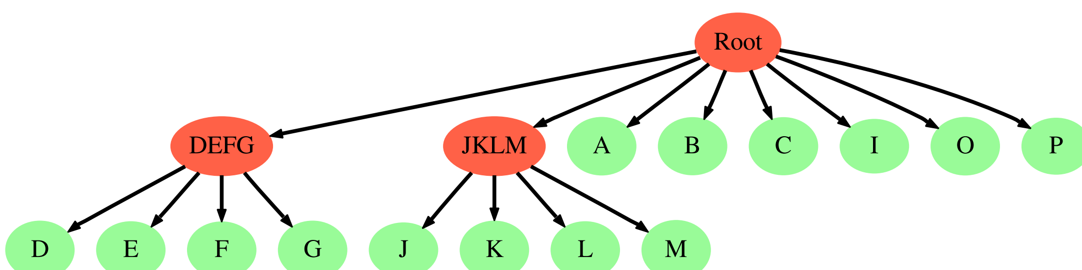
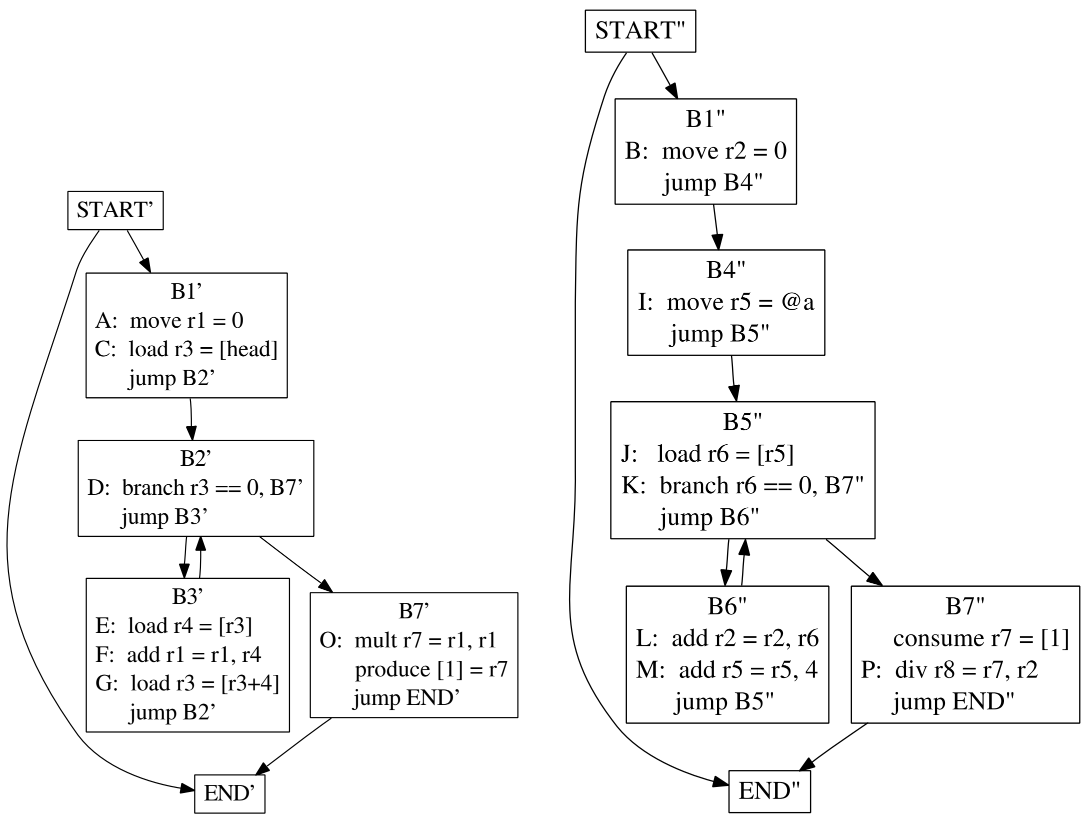


Figure 5.3: HPDG tree for the PDG in Figure 5.1.

5.1.4 Putting It All Together

After the partition into threads is chosen, the MTCG algorithm is applied. Figures 5.4(a)-(b) illustrate the generated code for the two threads corresponding to the global schedule depicted in Figure 5.2(c). As can be verified, each of the resulting threads contains only its relevant basic blocks, the instructions scheduled to it, the instructions inserted to satisfy the inter-thread dependences, and jumps inserted to connect the CFG. In this example, there is a single pair of `produce` and `consume` instructions, corresponding to the only cross-thread dependence in Figure 5.1(d).

By analyzing the resulting code in Figures 5.4(a)-(b), it is clear that the resulting threads are able to concurrently execute instructions in different basic blocks of the original code, effectively following different control-flow paths. The potential of exploiting such parallelization opportunities is unique to a *global* multi-threaded scheduling, and constitutes its key advantage over *local* multi-threaded scheduling approaches.



(a) Code for thread 1 (b) Code for thread 2

Figure 5.4: Resulting multi-threaded code.

5.1.5 Complexity Analysis

This subsection analyzes the complexity of GREMIO's partitioning algorithms. We first analyze the complexity of partitioning a single region with its inner loops coalesced, and then analyze the complexity of the hierarchical algorithm to handle loop nests. For the whole region's PDG, we denote n its number of nodes and e its number of arcs. By t we denote the target number of threads. Finally, we denote l the number of nodes in the HPDG tree, which is the number of loops in the region plus 1, and n_i and e_i the number of nodes and arcs in the HPDG for loop L_i, $0 \leq i \leq l$ ($i = 0$ for the whole region).

For a given loop L_i, the complexity of the DSC is $O(e_i + log(n_i))$ [117]. GREMIO's list scheduling, with checks for conflicts with currently scheduled nodes, has a complexity upper bound of $O(n_i^2)$.

In the dynamic programming algorithm, each node in the HPDG tree is processed exactly once. For each node, the clustering algorithm is applied once, and the list scheduling is applied t times, for each possible number of threads. Since the complexity of the clustering and list scheduling algorithms are more than linear, the worst case for the whole region's running time is when there are no loops. In this case, there is a single node in the HPDG tree ($l = 1$), and $n_0 = n$ and $e_0 = e$. Therefore, the total complexity for the whole region is $O(e \times log(n) + t \times n^2)$. This low complexity enables this algorithm to be effectively applied in practice to regions with up to several thousands of instructions.

5.2 Experimental Evaluation

This section presents an experimental evaluation of GREMIO, which was implemented as part of our GMT instruction scheduling framework in the VELOCITY compiler. The experimental methodology used here is the same described in Section 4.4.1. The results presented in this section include the selected benchmark functions from Table 4.2 for which GREMIO partitioned the code and estimated a speedup of at least 10%.

Figure 5.5 presents the speedups for the parallelized benchmark functions. For each benchmark, the two bars illustrate the speedup on the selected function, as well as the corresponding speedup for the whole application. The overall speedup per function is 34.6% on average, with a maximum of 69.5% for *ks*.

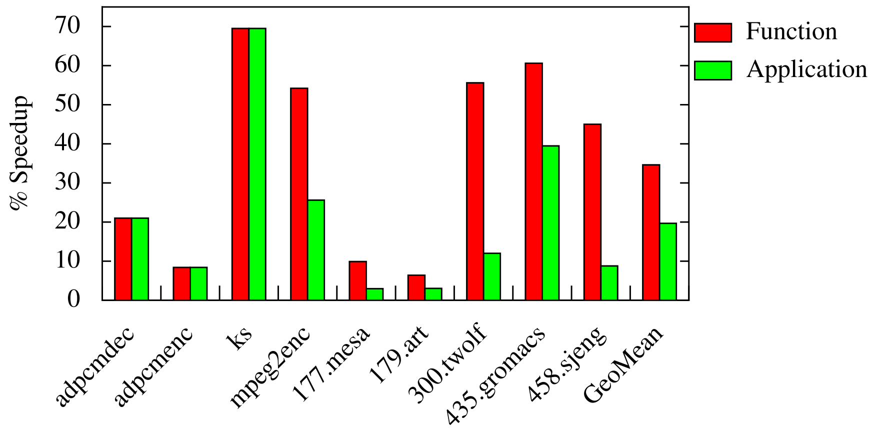


Figure 5.5: Speedup of dual-thread GREMIO over single-threaded.

Benchmark	Type of Parallelism
adpcmdec	CMT
adpcmenc	PMT
ks	PMT
mpeg2enc	CMT
177.mesa	CMT
179.art	CMT
300.twolf	PMT
435.gromacs	CMT
458.sjeng	PMT

Table 5.1: Type of parallelism extracted by GREMIO.

An interesting aspect of the parallelizations generated by GREMIO is the pattern of communication among the threads. We use the terminology introduced in [82], which classifies the parallelization patterns in three categories. If the threads are totally independent, except for initial and final communications, it is called *Independent Multi-Threading* (IMT). The typical example of IMT is DOALL parallelization. If the communication among the threads is unidirectional, then it is called *Pipelined Multi-Threading* (PMT). Finally, if the there is cyclic communication among the threads, then it is called *Cyclic Multi-Threading* (CMT). GREMIO is not restricted to a specific kind of parallelism. Nevertheless, GREMIO's hierarchical list-scheduling was designed to exploit parallelism within loop iterations. This may result in either of the types of parallelism described above. Table 5.1 shows the type of parallelism extracted by GREMIO for each benchmark. GREMIO produced CMT for five benchmarks, and PMT for the other four.

5.2.1 Comparison to Local Multi-Threaded Scheduling

In order to verify the amount of parallelism obtained by GREMIO that can be extracted by LMT techniques, the multi-threaded execution traces were analyzed and the cycles classified in two categories. The first one corresponds to cycles in which both threads are executing instructions that belong to the same extended basic block (EBB)[3] in the original code. This is parallelism that can be extracted by LMT instruction scheduling techniques such as [59, 60, 88]. The remaining cycles correspond to the portion of the execution in which GMT instruction scheduling is necessary in order to expose the parallelism. Figure 5.6 illustrates the execution breakdown for the benchmarks parallelized by GREMIO. These results illustrate that, for the majority of these benchmarks, less than 2% of the parallelism obtained by GREMIO can be achieved by LMT techniques. The function in the SPEC-CPU 2006 FP *435.gromacs* benchmark, which contains two nested loops with no other control flow, is the only one in which a good fraction (47%) of the parallelism extracted by GREMIO is within extended basic blocks.

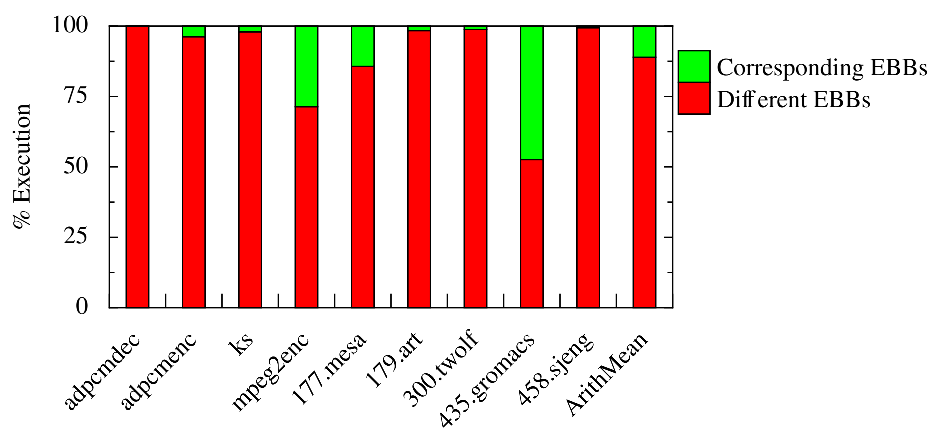


Figure 5.6: Percentage of execution on corresponding extended basic blocks.

[3] An extended basic block (EBB) is a sequence of instructions with a single entry (the first instruction) and potentially multiple exits.

5.2.2 Sensitivity to Communication Latency

In order to assess the effect of communication latency for code generated by GREMIO, we conducted experiments with the inter-core communication latency increased from 2 cycles in our base model to 10 cycles. Figure 5.7 contains the results. The average speedup from GREMIO dropped about 6%. Not surprisingly, the codes that are affected the most contain CMT-style parallelism (*adpcmdec* and *mpeg2enc*). However, not all benchmarks with CMT were slowed down by this increase in communication latency. In general, the CMT loops with small bodies are affected the most, since the communication latency represents a larger fraction of the loop body's execution.

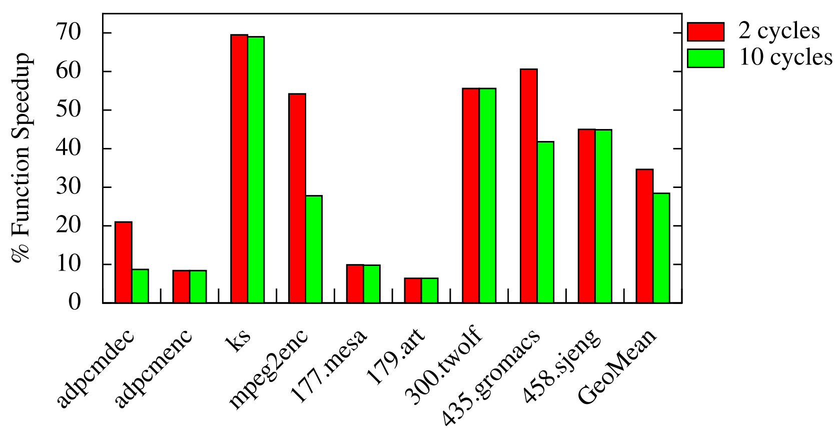


Figure 5.7: Speedup over single-threaded for different communication latencies.

5.2.3 Sensitivity to Queue Size

We also conducted experiments to measure how sensitive the parallelized codes are to the size of the communication queues. Figure 5.8 shows the resulting speedups on our base model, with 32-element queues, and with the size of the queues set to 1 element. The experiments show that most of the codes are not affected. In particular, for loops parallelized as CMT, one thread is never more than one loop iteration ahead of the other. As a result, single-entry queues are enough to obtain the maximum speedup in these cases. This means that a cheaper inter-core communication mechanism, with simple blocking

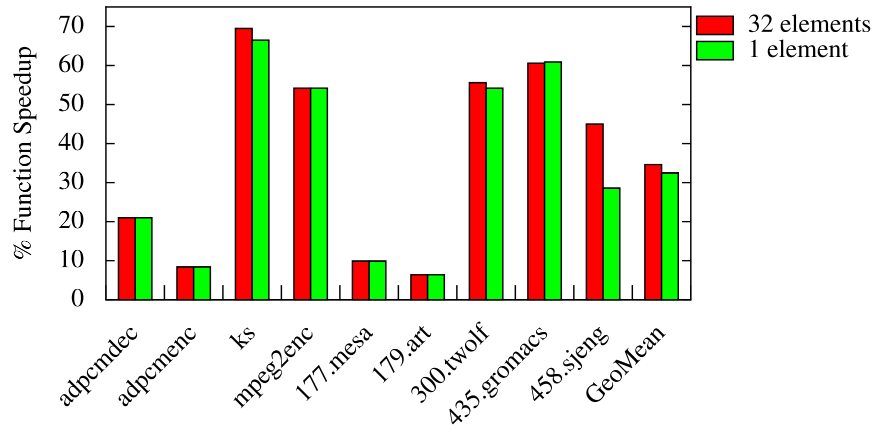


Figure 5.8: Speedup over single-threaded for different size of queues.

registers, is enough to get the most parallelism out of CMT loops. Longer queues can be more beneficial in other scenarios, as described in Chapter 6.

5.3 Related Work

There is a broad range of related work on instruction scheduling. Most of the related work, however, is more restricted than GREMIO in one or two aspects, which are discussed below.

First, many techniques perform scheduling for *single-threaded* architectures. Of course, this characteristic is highly dependent on the target architecture. Single-threaded scheduling is commonly used for a wide range of single-threaded architectures, from simple RISC-like processors to very complex ones such as VLIW/EPIC [8, 16, 30, 42, 55, 113] and clustered architectures [26, 69]. Besides scheduling the original program's instructions (the *computation*), multi-threaded schedulers must also generate *communication* instructions to satisfy inter-thread dependences. For clustered single-threaded architectures [26, 69], the scheduler also needs to insert communication instructions to move values from one register file to another. However, the fact that dependent instructions are executed in different threads makes the generation of communication more challenging for multi-threaded architectures.

Second, previously proposed multi-threaded instruction scheduling techniques have been restricted to *local multi-threaded* (LMT) scheduling, as discussed in Section 1.2. These techniques include the approaches proposed in the context of the RAW microprocessor [59, 60], and in the context of decoupled access/execute architectures [88]. Our approach enables truly thread-level parallelism, among control-unrelated instructions, by simultaneously scheduling a global code region.

Table 5.2 summarizes how various existing scheduling techniques are classified according to these two orthogonal dimensions: single-threaded versus multi-threaded, and regarding the scope of scheduling. Horizontally, the more a techniques is to the right, the more general is its handling of control flow. The techniques in bold are the ones proposed in this thesis.

Num. of Threads	Scope Basic Block	 Trace	 Loop	 Procedure
Single	List Sched. [65]	Trace [16, 26, 30] Superblock [42]	SWP [55, 69]	GSTIS [8] ILP [112]
Multiple	Space-time [59] Convergent [60] DAE Sched. [88]		**DSWP** **PS-DSWP**	**GREMIO**

Table 5.2: Instruction scheduling space.

Although operating at a different granularity, our work shares some similarities with task scheduling for parallel computers. Sarkar [90, 93] describes general algorithms to partition and schedule functional parallel programs on multiprocessors. In particular, the idea of using a clustering pre-pass used here was inspired by Sarkar's work. However, our problem differs from his on a number of aspects, including the granularity of tasks and the abundance of parallelism in the source programs. Furthermore, our algorithms differ from his in many ways. For example we use list scheduling with a virtual schedule, which is very useful for the granularity of the parallelism we exploit, and our dynamic programming approach allows GREMIO to handle larger regions of code. Finally, our MTCG algorithm is key to enable parallelization at the instruction granularity, by allowing multiple "tasks" to be assigned to a single thread.

5.4 Significance

This chapter described GREMIO, an extension of list-scheduling to simultaneously schedule instructions from a global region of code onto multiple threads. GREMIO uses two simplifying assumptions to reduce the cyclic PDG scheduling problem into an acyclic one. GREMIO schedules the instructions in the PDG hierarchically, using a dynamic programming technique. At each level, GREMIO employs a clustering pre-pass and a thread-aware list-scheduling targeting all possible number of threads.

Overall, the results obtained by GREMIO showed significant speedups for hard-to-parallelize benchmarks. Although only evaluated on two threads here, for some benchmarks, we have seen opportunities to obtain scalable results. As in any instruction scheduling technique, the opportunities for GREMIO can be increased by performing more aggressive loop unrolling and function inlining. However, these auxiliary techniques typically become prohibitive after some point due to increased code size. In Chapter 7, we describe an approach to get scalability without incurring excessive code growth.

Chapter 6

Decoupled Software Pipelining Thread Partitioning

In this chapter, we describe a thread partitioning that extends the idea of *software pipelining* (SWP) [13, 55, 65, 86, 87]. Software pipelining is a single-threaded instruction scheduling technique applicable to loop regions. A key property of software pipelining is that it allows the concurrent execution of instructions from different loop iterations. This contrasts with more restrictive techniques that schedule instructions of a loop by applying list scheduling to its loop body. Although our list-scheduling-based partitioner presented in Chapter 5 may achieve parallelism across loop iterations, it was not designed with such purpose. The thread partitioner described in this chapter aims at obtaining pipeline parallelism.

There are two key advantages of exploiting pipeline parallelism as TLP instead of ILP. First, with ILP, the presence of variable latencies, particularly in memory accesses, creates a well-know scheduling trade-off [96, 114]. Scheduling the use of the loaded value too close to the load itself may result in a stall in case of a cache miss, while scheduling the use too far can be suboptimal in case of a cache hit. With TLP, this problem can be avoided by scheduling the variable-latency instruction in a separate thread from its use [85]. Second, expressing pipeline parallelism as ILP has limitations due to control flow. This problem can be mitigated by aggressively applying if-conversion and relying on hardware support for predicated execution [5, 62]. However, this does not solve the problem for loop nests, which cannot be "if-converted". For this reason, all practical ILP software pipeline techniques are restricted to inner loops only. With TLP, the ability to execute completely unrelated control regions in different threads eliminates this problem.

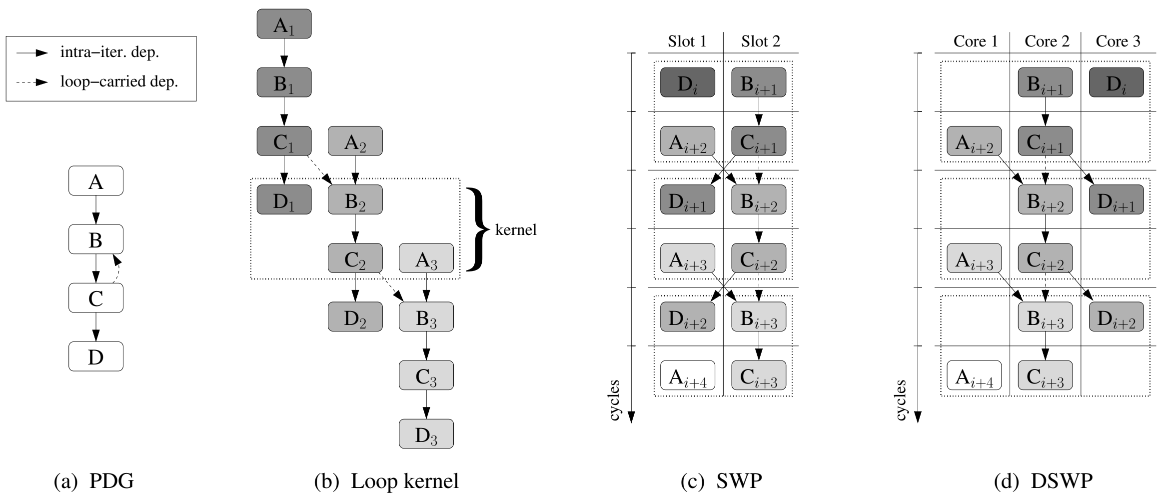


Figure 6.1: A simple example comparing Software Pipelining (SWP) and Decoupled Software Pipelining (DSWP).

This chapter describes the Decoupled Software Pipelining (DSWP) thread partitioning proposed in Ottoni et al. [74]. Based on the general GMT instruction scheduling framework we proposed, DSWP is a thread partitioner that focuses at extracting pipelined multi-threading (PMT) from arbitrary loop regions. Figure 6.1 contrasts DSWP with traditional SWP. The main difference lies on the fact that, to express parallelism, SWP uses ILP resources while DSWP uses TLP resources. For every loop to which SWP is applied, DSWP can be applied to express an equivalent parallelization in the form of TLP. This is achieved by assigning each group of instructions belonging to the same loop iteration in the loop kernel to its own thread (see Figure 6.1). Furthermore, by exploiting TLP, DSWP can express additional parallelizations because it does not require all instructions to be control equivalent. DSWP and SWP can also play complementary roles in the following way. DSWP can be applied first to extract TLP, thus producing a set of loops. Each of these loops is then a candidate for SWP, which can extract additional parallelism. This combination of DSWP and SWP can be especially beneficial when there are fewer processors/cores than the number of threads that DSWP can exploit.

In order to extract PMT, DSWP partitions the PDG nodes into threads such that, inside the parallelized region, the dependences flow in a single direction. In other words, there is no cyclic dependence among the threads inside the parallelized region. This unidirectional communication property may only be violated outside the parallelized region, because of the initial and final communications (Section 3.1.3). The unidirectional flow of communication created by DSWP has the key property of making this technique tolerant to communication latencies. This brings two advantages. The first is to make DSWP beneficial even in the face of larger communication latencies, which are likely to increase as the number of cores in a single chip grows. Second, this allows the partitioning algorithm to mostly ignore the communication latency while making partitioning decisions. Given the difficulty of the thread partitioning problem (discussed in Section 6.2), having one less parameter to worry about while making partitioning decisions simplifies the problem. However, focusing solely on PMT can lead to missed parallelization opportunities. We discuss this when we compare DSWP to GREMIO in Section 6.3.2.

In order to qualitatively compare DSWP to previously proposed loop-parallelization techniques, it is helpful to plot timelines illustrating how the code is executed by each technique. Here we compare DSWP with DOALL [2, 115], DOACROSS [22, 78], and DOPIPE [24, 78]. DOALL parallelization is only applicable to loops with no loop-carried dependence. In this scenario, each processor can execute a subset of the loop iterations, and all processors execute concurrently with no need for synchronization. This is illustrated in Figure 6.2(a), which contains a loop with two statements (C and X). The numbers in each node represent the iteration number, and the arcs depict dependences. For loops with recurrences, a popular parallelization technique is DOACROSS. In DOACROSS, the loop

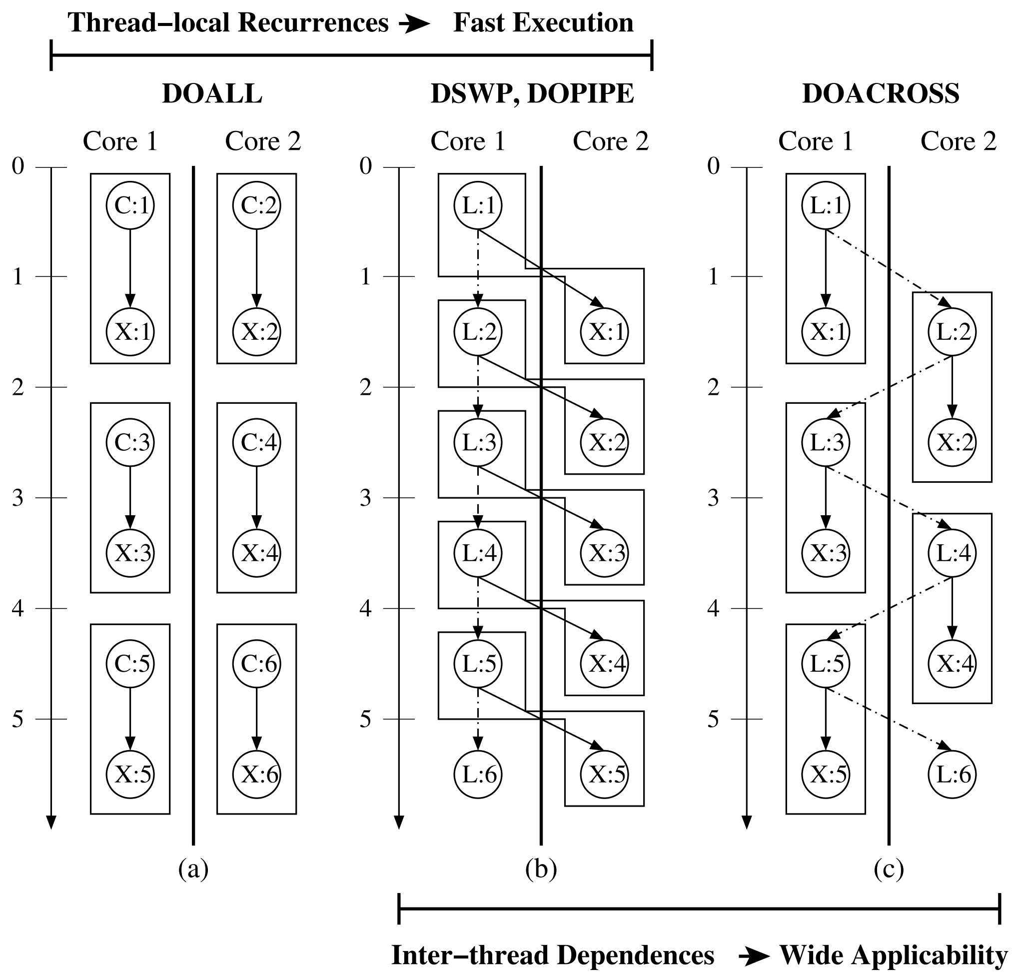


Figure 6.2: Execution timelines for DOALL, DSWP, DOPIPE, and DOACROSS.

iterations execute in a round-robin fashion among the processors, with synchronizations inserted in the code in order to respect loop-carried dependences. Figure 6.2(c) illustrates the execution of a simple loop with two statements, where the first one (L) is a recurrence. For loops containing recurrences and that execute a significant number of iterations, the critical path of execution will always contain loop-carried dependences. Since all loop-carried dependences are inter-thread dependences with DOACROSS, the inter-processor communication latency is inserted multiple times (once per iteration) in the critical path of execution. DSWP, akin to DOPIPE, takes a different approach, in which all loop-carried dependences are kept thread-local. Figure 6.2(b) illustrates the execution timeline for DSWP and DOPIPE, for the same loop illustrated for DOACROSS. DSWP and DOPIPE create unidirectional communication among the processors. This results in the communication cost being inserted in the critical path of execution only once per loop *invocation*, as opposed to once per loop *iteration* in DOACROSS. This makes DSWP and DOPIPE significantly more tolerant to inter-processor communication latencies than DOACROSS. Compared to DOPIPE, DSWP is superior because it is applicable to codes with irregular control flow and memory dependences. DOPIPE is restricted to counted loops with no other control flow and regular, array-based memory accesses [24].

Although having the same name as the approach proposed by Rangan et al. [82, 85], our Decoupled Software Pipelining is a much more general technique. More specifically, our technique eliminates the following limitations in [85]. First, their approach is limited to loops that traverse recursive data structures (RDS), and it consists in simply separating the RDS traversal from the remainder of the loop. As such, their approach can only extract two threads of execution, and it may result in greatly imbalanced threads. Furthermore, in [85], the benchmarks were manually parallelized, and no general algorithmic technique was proposed. Our DSWP is a much more general, fully automatic technique, which overcomes all these limitations by using the partitioning algorithms described in this chapter and our general GMT instruction scheduling framework. In fact, the evaluation in [82] uses the general and automatic DSWP technique proposed in this thesis.

In Section 6.1, we describe the general DSWP algorithm, whereas its key subproblem is studied in Section 6.2. An experimental evaluation of DSWP is presented in Section 6.3.

6.1 DSWP Algorithm

This section describes our DSWP algorithm and illustrates it on the code example of Figure 6.3(a), which is the same example used in Chapter 3. This code example traverses a list of lists of integers and computes the sum of all the integer values. After performing DSWP on the outer loop in Figure 6.3(a), it is transformed into two threads shown in Figures 6.3(d)-(e). In this example, the code in Figure 6.3(d) is executed as part of the main thread of the program, the one which includes the un-optimized sequential portions of the code.

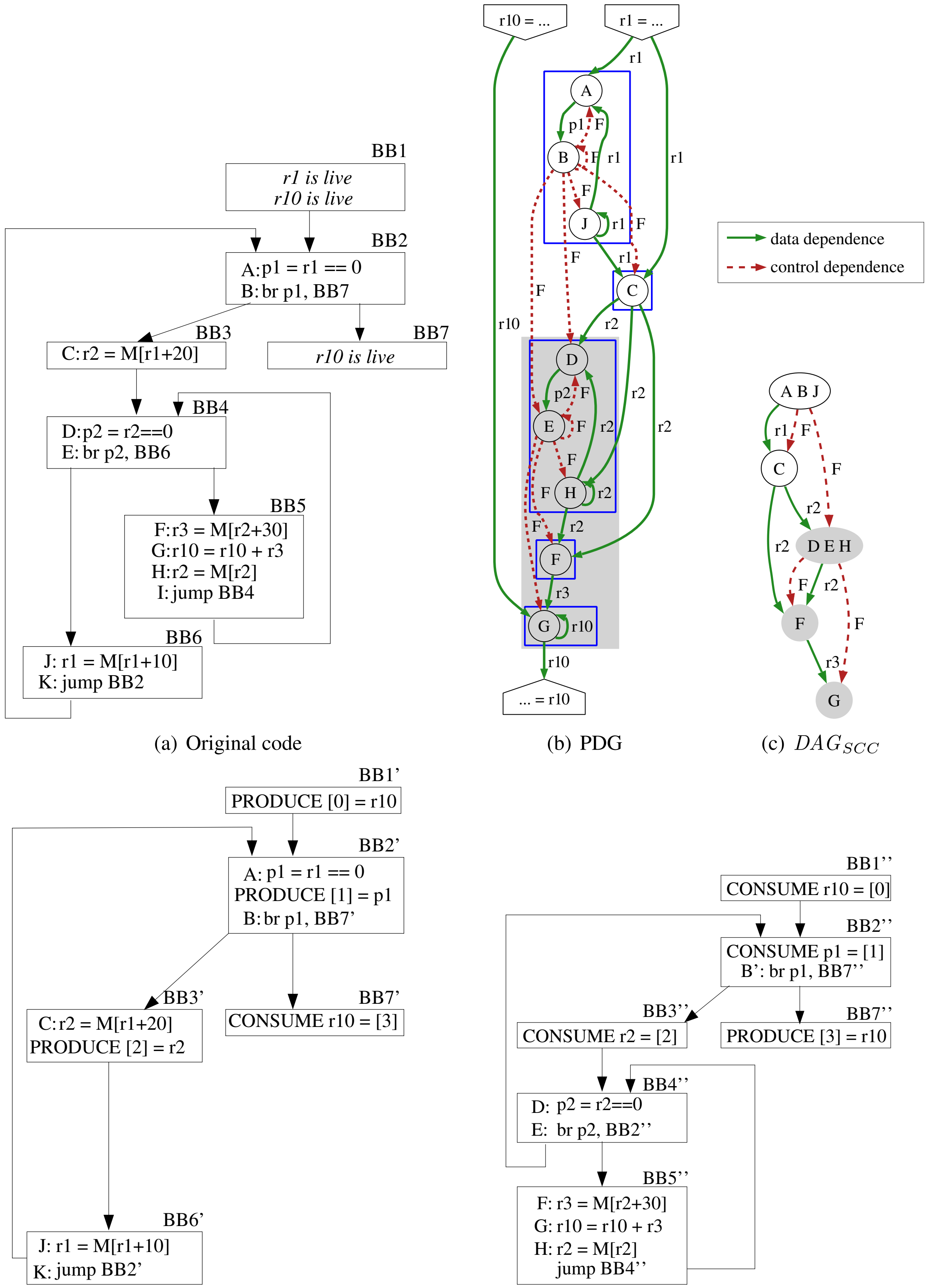
There are several important properties of the transformed code to be observed. First, the set of original instructions is partitioned between the two threads with one instruction in both (B as B and B'). Also notice that DSWP, being based on our GMT scheduling framework, does *not* replicate the control-flow graph completely, but only the parts that are relevant to each thread. In order to respect dependences, `produce` and `consume` instructions are inserted as necessary. For example, instruction C writes a value into `r2` that is then used by instructions D, F, and H in the other thread. Queue 2 is used to communicate this value as indicated in the square brackets. Note that, within the loop, the dependences only go in one direction, from the thread 1 to thread 2. This acyclic nature, along with the queue communication structures, provides the decoupling effect during the execution of the loop body. Outside the loop, this property need not be maintained; the main thread produces loop live-in values for the other thread and consumes loop live-out values after consumer loop termination.

Algorithm 3 shows the pseudo-code for the DSWP algorithm. It takes as input a loop L in an intermediate representation, and modifies it as a side-effect. The following paragraphs describe each step of the DSWP algorithm.

Algorithm 3 DSWP

Require: Loop L

$PDG \leftarrow build\_PDG(L)$
$SCCs \leftarrow find\_strongly\_connected\_components(PDG)$
if $|SCCs| = 1$ **then**
    return
end if
$DAG_{SCC} \leftarrow coalesce\_SCCs(PDG, SCCs)$
$\mathcal{P} \leftarrow thread\_partitioning\_algorithm(DAG_{SCC}, L)$
if $|\mathcal{P}| = 1$ **then**
    return
end if
$MTCG(L, PDG, \mathcal{P})$



(a) Original code (b) PDG (c) DAG_{SCC}

(d) Thread 1 (main thread) (e) Thread 2 (auxiliary thread)

Figure 6.3: DSWP example.

The first step in the DSWP algorithm is to build the PDG for loop L, as described in Chapter 2. Figure 6.3(b) illustrates the PDG for the loop in Figure 6.3(a). Data dependence arcs are annotated with the corresponding register holding the value, while control dependence arcs are labeled with their corresponding branch conditions. In this example, there are no memory dependences. Special nodes are included in the top (bottom) of the graph to represent loop live-in (live-out) registers.

The second step in the algorithm is to ensure an acyclic partition by finding the PDG's strongly connected components (SCCs) and creating the directed acyclic graph of them, the DAG_{SCC} [103]. The SCCs correspond to instructions collectively participating in a dependence cycle, the loop recurrences. As such, DSWP requires all instructions in the same SCC to remain in the same thread. Lines 3-5 in Algorithm 3 stop the transformation if the PDG has a single SCC, since such a graph is not partitionable into multiple threads by DSWP. The next step of the DSWP algorithm is to coalesce each SCC in PDG to a single node, obtaining the DAG_{SCC}. Figure 6.3(b) shows the SCCs delimited by rectangles, and Figure 6.3(c) shows the corresponding DAG_{SCC}.

After the DAG_{SCC} is built, the actual thread partition is chosen. In order to enforce the unidirectional flow of communication among the threads, not every partition of the DAG_{SCC} is valid. The following definition precisely characterizes the set of valid partitions for DSWP.

Definition 6 (Valid DSWP Partition)**.** *A valid partition $\mathcal{P}$ of the DAG_{SCC} is a sequence $T_1, T_2, \ldots, T_n$ of sets of DAG_{SCC}'s vertices (i.e. T_is are sets of SCCs) satisfying the following conditions:*

1. *$1 \leq n \leq t$, where t is the number of threads that the target processor can execute simultaneously.*

2. *Each vertex in DAG_{SCC} belongs to exactly one $T_i \in \mathcal{P}$.*

3. *For each arc $(u \rightarrow v)$ in DAG_{SCC}, with $u \in T_i$ and $v \in T_j$, we have $i \leq j$.*

A valid partition guarantees that all members of each $T_i \in \mathcal{P}$ can be assigned to a thread such that the communications inserted by the MTCG algorithm are unidirectional. Condition (3) in Definition 6 guarantees that each arc in the dependence graph G either flows forward from a thread T_i to a thread T_j, where $j > i$, or it is internal to one thread. In other words, this condition guarantees an order among the elements of $\mathcal{P}$ that permits the resulting threads to form a pipeline.

The DSWP *Thread-Partitioning Problem* (TPP) is the problem of choosing a *valid partition* that minimizes the total execution time of the resulting code. This optimization problem is at the heart of the DSWP algorithm. Section 6.2 demonstrates that this problem is NP-hard and describes a practical heuristic algorithm to solve it.

After a partition is chosen, the DSWP algorithm estimates whether or not it will be profitable by considering the cost of the communication instructions that need to be inserted. The $thread\_partitioning\_algorithm$ may indicate that no partition is desirable by returning a singleton partition. In such cases, Algorithm 3 simply terminates in line 9. Otherwise, it continues by invoking the MTCG algorithm (described in Chapter 3) to generate multi-threaded code according to partition $\mathcal{P}$. The code in Figures 6.3(d)-(e) corresponds to the partition represented by different shades in Figures 6.3(b)-(c).

6.2 DSWP Thread-Partitioning Problem

A key component of the DSWP algorithm presented in Section 6.1 is what we call the *thread-partitioning problem* (TPP). Informally, given a loop L, TPP consists of partitioning the SCCs in L's PDG among pipeline stages so as to obtain the maximum performance improvement. In this section, we formally describe TPP. In addition, we prove that TPP is NP-hard for the most common class of MT processors, namely those with scalar cores. Finally, we present the heuristic we use to solve TPP in practice.

6.2.1 Problem Formulation and Complexity

The goal of TPP is to split a loop into a sequence of loops, each one to be assigned to a different thread running on a different processor core, so that the total execution time is minimized. The input to TPP is a loop L to be partitioned, the corresponding DAG_{SCC}, the number t of threads that can execute simultaneously, as well as the information about the target processor that is necessary to compute a partition's cost. The output of TPP is a *valid partition* (as per Definition 6) of DAG_{SCC}, i.e. a sequence of blocks $T_1, T_2, \ldots, T_n$ ($n \leq t$) corresponding to the new threads.

For a loop L, we denote $exec\_cycles(L)$ the number of cycles that it takes to execute L on a given processor. $exec\_time(L)$ accounts for the execution time of instructions in L, as well as communication instructions that the chosen partition requires. For a large enough number of iterations of the original loop L,[1] the initial fill time can be amortized so that the execution time of the threaded version of the code is dominated by the execution time of the slowest loop. This situation is analogous to a processor pipeline, whose execution cycle is dominated by its longest stage, and in both cases the optimal scenario occurs when the imbalance between the stages (or loops) is minimum. The following definition formalizes this idea.

Definition 7 (DSWP Thread-Partitioning Problem (TPP))**.** *Given a loop L, its corresponding DAG_{SCC}, the maximum number t of threads that can be executed simultaneously, and the processors' characteristics necessary to compute $exec\_time$, choose a partition $\mathcal{P}$ = $(T_1, T_2, \ldots, T_n)$ of the SCCs in DAG_{SCC} such that:*

1. *$\mathcal{P}$ is a* valid partitioning *(according to Definition 6); and*

2. $\max\{exec\_time(T_i) \mid 1 \leq i \leq n\}$ *is minimized.*

   (We assume that $exec\_time(T_i)$ includes the time for the necessary intra-region communication instructions.)

[1] We avoid applying DSWP to loops with small trip counts due to the costs of initialization and finalization.

In Definition 7, the characteristics of the specific target processor necessary to compute $exec\_time$ are informally described, and they are dependent on the target architecture. However, because most MT processors are built from simple, scalar cores, it is particularly important if one could find an efficient algorithm to solve TPP for this class of processors. Unfortunately, as the following theorem shows, this problem is NP-hard for this class of processors, and therefore it is unlikely that such efficient algorithm exists. We assume that instruction $instr$ in a scalar processor has a known latency, $lat(instr)$, and the $exec\_time(l)$ of a loop l is the sum of the latencies of the instructions in l. Therefore, a MT processor with scalar cores can be described by its set of instructions I and the function $lat\colon I \to \mathbb{N}$.

Theorem 4. *TPP is NP-hard for MT processors with scalar cores.*

Proof. We prove this by a reduction from the *bin packing* problem, which is known to be NP-complete [32]. The input to bin packing is a sequence of numbers S, a number of bins b, and the capacity c of each bin. The problem is to decide if there exists an assignment of the numbers in S to the b bins such that the sum of the numbers in each bin is at most c.

A reduction from an instance of bin packing with input S, b and c, to an instance of TPP with input L, DAG_{SCC}, t, I and lat can be obtained as follows. For each number $s_k \in S$, we add an instruction $instr_k$ to I with $lat(instr_k) = s_k$, and also add $instr_k$ to L. In addition, we make all instructions in L independent, so that its DAG_{SCC} will contain one vertex for each instruction and no edges. Finally, we choose $t = b$. It is fairly straightforward to verify that the bin packing instance has a solution if and only if the TPP instance has a partitioning with cost at most c. □

6.2.2 Load-Balance Heuristic

Given the inherent difficulty of the TPP, we propose a simple heuristic algorithm to solve this problem efficiently in practice. This heuristic, called *Load-Balance Heuristic* (LBH), tries to equally divide among the threads the estimated total number of cycles C to execute the input loop. The value of C is computed by summing up the number of cycles that each instruction takes to execute. For variable-latency instructions, we estimate the expected (average) number of cycles. In addition, the number of cycles of each instruction is multiplied by the profile weight of the basic block containing it. The algorithm computes the corresponding expected number of cycles, $load(s)$, for each node s in DAG_{SCC} as well.

Having computed C and $load$ for each SCC, the goal of the LBH is to assign DAG_{SCC}'s vertices to the threads $T_1, \ldots, T_t$ so that, for each thread T_i, the sum of the expected cycles taken by the SCCs assigned to it is as close to C/t as possible. As a secondary metric, LBH tries to minimize the number of communication instructions that the resulting partition will require.

The algorithm chooses the SCCs to put in each thread at a time, from T_1 to T_t. It keeps a set of current *candidate nodes*, which are the SCCs whose all predecessors have already been assigned to a thread (including nodes with no predecessors in DAG_{SCC}). At each step, a new candidate is chosen for the thread T_i being constructed, or the algorithm decides to stop adding nodes to T_i and to start building T_{i+1}. The candidate c chosen at each step is the one with the highest $load(c)$ such that, when added to T_i, the load of T_i will not exceed C/t (in fact, C and t are updated accordingly after finishing the creation of each thread). In case of ties, we choose the node with the largest balance $|in\_edges(c)| - |out\_edges(c)|$ in DAG_{SCC}, as an attempt to reduce the number of synchronizations[2]. After choosing a candidate to put in a thread, the set of candidates is updated accordingly, and the process is repeated. The algorithm terminates when $t-1$ threads have been created, and the remaining nodes are assigned to the last thread.

Finally, to take the cost of communication into account, the resulting load assigned to each SCC is increased by the expected number of cycles corresponding to the communications required by the chosen partition. In case the readjusted load for a SCC exceeds C, the

[2]In our DAG_{SCC} we have multiple edges between two SCCs u and v, one per instruction in u on which some node in v depends.

partition is not applied to the code.

This heuristic was designed specifically for resource-bound cores, such as simple RISC cores. However, as the experimental results show, it is also very effective for less restricted MT processors with VLIW cores.

6.3 Experimental Evaluation

In this section, we experimentally evaluate the DSWP thread partitioning. The results reported here use an implementation of DSWP in the VELOCITY compiler. The experimental methodology used here is the same described in Section 4.4.1.

6.3.1 Dual-Thread Results

Figure 6.4 presents the results for DSWP targeting two threads. Among the benchmarks described in Section 4.4.1, Figure 6.4 shows the ones for which DSWP obtained a dual-thread partition, using the load-balance heuristic from Section 6.2.2. The overall function speedup in Figure 6.4 is 27.9%. The maximum speedup is 143.7% for *435.gromacs*. This benchmark has a large memory footprint, and therefore it effectively benefited from the doubled L2 cache capacity (the cores have private L2).

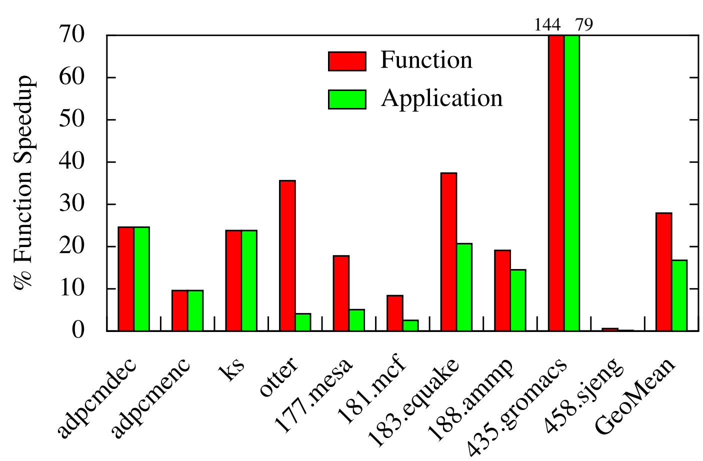


Figure 6.4: Speedups for dual-thread DSWP over single-threaded execution.

6.3.2 Comparison to GREMIO

Figure 6.5 compares the function speedups achieved by GREMIO and DSWP, for the benchmarks from Table 4.2 that were parallelized by at least one of these techniques. On average, DSWP achieves 20.9% speedup, compared to 22.9% for GREMIO. As can be seen, each of GREMIO and DSWP outperforms the other on several benchmarks. As mentioned above, due to caching effects, DSWP results in 2.44× speedup for *435.gromacs*. A similar behavior was not observed with GREMIO because it unluckily kept the instructions responsible for most L2 misses in the same thread. Figure 6.5 also shows a bar for each benchmark indicating the speedup of the best performing version. This is the performance a compiler combining only these two MT techniques can ideally obtain with two threads. This best-of speedup averages 36.9%.

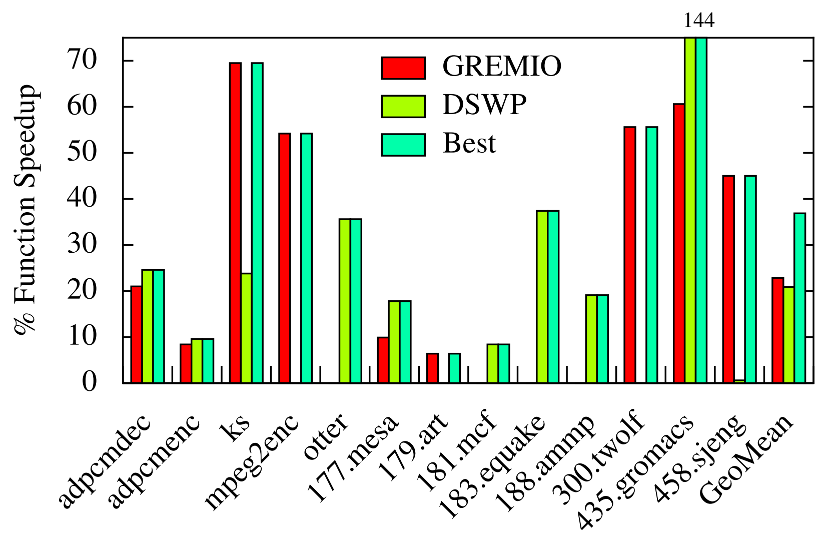


Figure 6.5: Speedups for dual-thread GREMIO and DSWP over single-threaded execution.

By the nature of DSWP, the parallelism it extracts is *pipelined multi-threading* (PMT). GREMIO, as discussed in Section 5.2, is not restricted to a specific kind of parallelism. In our experiments, GREMIO produced CMT for five benchmarks, and PMT for the other four (Table 5.1). As can be noticed, CMT is applicable in cases where DSWP is not (*mpeg2enc* and *179.art*). In other cases, DSWP outperforms the CMT extracted by GREMIO (*adpcmdec*, *177.mesa* and *435.gromacs*). In the cases GREMIO extracted PMT, it is better than the PMT extracted by DSWP using the load-balance heuristic described in [74] in some cases (*ks*, *300.twolf*, and *458.sjeng*). This shows potential for studying better partitioning algorithms for DSWP.

6.3.3 Sensitivity to Communication Latency

Figure 6.6 shows the effect of increasing the communication latency of our base model from 2 cycles to 10 cycles. The average speedup drops only 0.2%. As mentioned earlier, this tolerance to communication latency is a key property of PMT. In fact, in Ottoni et al. [74], we conducted more experiments and observed the same latency-tolerance even for communication latencies of 100 cycles. The communication overhead that affects DSWP is only the extra instructions that execute inside the cores.

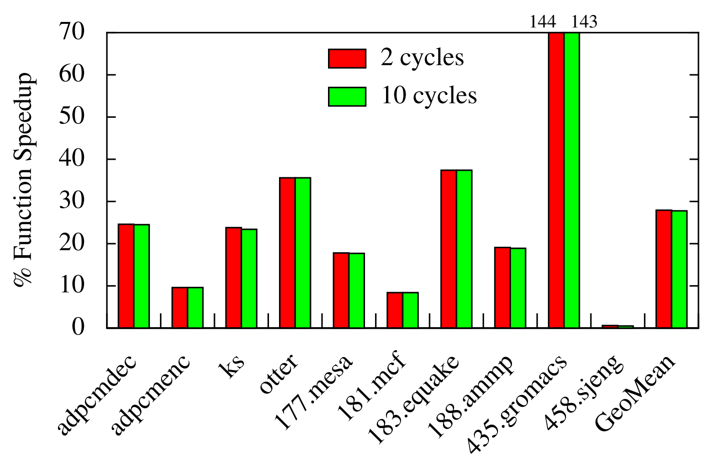


Figure 6.6: Speedups for dual-thread DSWP over single-threaded execution, with different communication latencies.

6.3.4 Sensitivity to Queue Size

We also conducted experiments with reduced queue sizes. In these experiments, the size of each queue in the model was reduced from 32 down to a single element. Figure 6.7 illustrates the results. Most benchmarks are affected, and the average speedup drops about 6%.

The reason for this is that larger communication queues enable better decoupling among the threads. The better decoupling hides variability in the threads across different loop iterations, which is more important for benchmarks with lots of control flow and irregular memory accesses (e.g. *ks* and *181.mcf*).

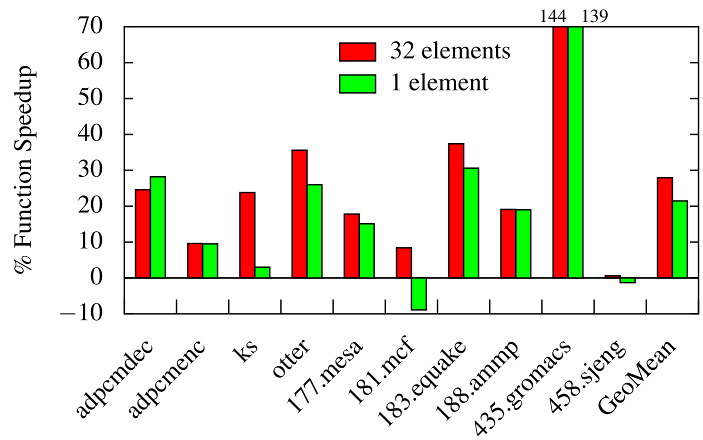


Figure 6.7: Speedups for dual-thread DSWP over single-threaded execution, with different queue sizes.

6.3.5 Scalability

This section analyzes the scalability of DSWP when targeting more than 2 threads. Figure 6.8 shows the speedups obtained by DSWP for 2, 4, and 6 threads. The average speedups are 27.1% for 2 threads, 33.9% for 4 threads, and 42.2% for 6 threads. These results show some performance scalability. However, the performance of DSWP is not very scalable in general. The main reason for this is that the performance of DSWP is limited by the slowest SCC in the PDG. After this SCC is isolated in its own pipeline stage, although the other stages can be further partitioned, this will not result in any performance improvement. Another reason for the lack of scalability of DSWP is the overhead of communication instructions. As the number of threads increase, the amount of communication that is necessary also tends to increase.

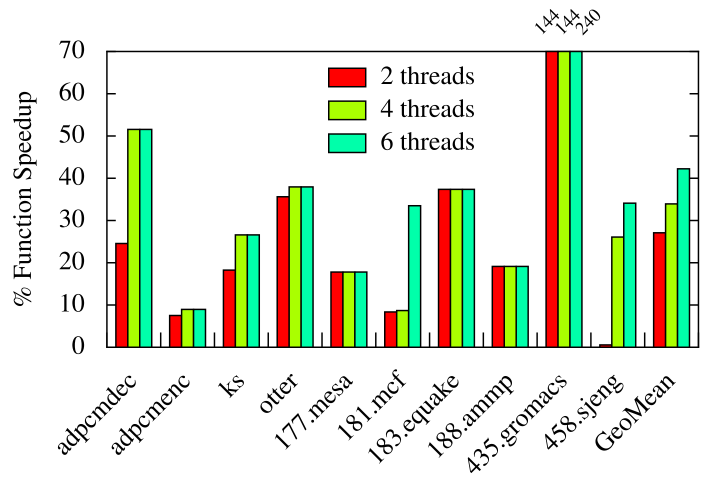


Figure 6.8: Speedups for DSWP over single-threaded execution.

6.4 Related Work

As discussed earlier in the beginning of this chapter, the DSWP partitioning technique presented here was inspired by the work of Rangan et al. [82, 85]. Nevertheless, our DSWP is a significantly more general. The technique in [85] is only applicable to loops traversing recursive data structures, and it consists of putting the traversal code in a separate thread to hide cache misses. On the other hand, our technique focuses on the recurrences in the code, and it tries to balance the load among the threads while keeping the recurrences thread-local. As such, our DSWP is applicable to general loops (not only traversals of recursive data structures), and can generate multiple threads of execution. Furthermore, by utilizing our framework based on the PDG and the code-generation algorithms we proposed, our DSWP is a general compilation technique that can handle irregular dependences and arbitrary control flow.

From a different perspective, the DSWP presented in this chapter shares similarities with loop distribution (or loop fission) [2, 115]. Loop distribution consists of partitioning a loop into a sequence of loops, and it is generally used to expose DOALL loops and to improve memory locality. Compared to DSWP, the main difference is that the loops generated by DSWP are executed concurrently, while the loops obtained by loop distribution run one after the other. This difference is also the source of a limitation of loop distribution, which is generally restricted to counted loops, or loops for which an upper bound on the number of iterations can be statically determined. The reason for this is the necessity to allocate storage to communicate values produced by one thread and used by another. DSWP, on the other hand, can be applied to loops with arbitrary number of iterations, precisely because the loops execute concurrently. The threads will naturally block if the communication queues become full, thus avoiding to have to pre-allocate an unbounded amount of intermediate storage.

A different approach to parallelize loops with recurrences is DOACROSS [22, 78]. With DOACROSS, loop iterations are executed in multiple processors in a round-robin fashion. To respect loop-carried dependences, synchronization primitives are inserted in the code, such that instructions in one iteration wait for their dependent instructions in the previous iterations to complete. Despite some success, the benefits of DOACROSS are generally limited by two factors. First, the number and position of loop-carried dependences lead to synchronizations in the code that limit the amount of parallelism. Second, by executing loop iterations in a round-robin fashion among the processors, DOACROSS inserts the synchronizations in the critical path to execute the loop. In other words, the synchronization cost is paid between every pair of consecutive iterations, essentially multiplying this cost by the number of iterations of the loop. Together, these two issues generally negate the benefits of DOACROSS. DSWP avoids these two main problems of DOACROSS. First, DSWP isolates large, problematic recurrences in separate threads, which reduces the latency to complete the critical dependence cycles of the loop. To some extent, this benefit can be obtained by techniques that combine scheduling with DOACROSS [14]. Second, DSWP avoids the problem of inserting the communication latency on the critical path to execute the loop. This is achieved by creating a unidirectional flow of the dependences among the threads.

Despite its limitations, DOACROSS has found applicability when combined with speculation, in techniques generally known as *Thread-Level Speculation* (TLS) [9, 47, 98, 101, 106, 120]. Unfortunately, even in this scenario, the amounts of parallelism obtained have been limited, hardly justifying the complex hardware support required for speculation. Furthermore, recent research has shown that, by adding speculation support to DSWP [109], much larger performance gains can be obtained [10].

A much less popular parallelization technique is called DOPIPE [24, 78]. Similar to DSWP, DOPIPE partitions a loop into a sequence of loops to be executed concurrently. Davies [24] even describes the combination of DOPIPE with DOALL, which is very similar to the Parallel-Stage DSWP described in Chapter 7. The advantages of DSWP over DOPIPE lie in the generality and applicability of the technique. DOPIPE is restricted to structured programs and counted loops. DSWP, by relying on our general MTCG algorithm, is able to handle arbitrary control flow.

6.5 Significance

This chapter presented the Decoupled Software Pipelining (DSWP) thread partitioning. DSWP generalizes the idea of single-threaded software pipelining, which exploits ILP across loop iterations, to exploit TLP. The DSWP technique proposed in this chapter exploits pipeline parallelism from loop nests with arbitrary control flow. In this work, we demonstrated that the problem of finding the best partition for DSWP is NP-hard, and we described a simple heuristic that tries to balance the load among the threads. The experimental results demonstrated, for selected benchmarks, an average speedup of 27% on 2 threads and 42% on 6 threads. In Chapter 7, we describe an extension of DSWP that increases its scalability.

Chapter 7

Parallel-Stage Decoupled Software Pipelining Thread Partitioning

The work presented in this chapter is the fruit of a strong collaboration with Easwaran Raman, based on ideas introduced in [77] and fully developed in [81]. More specifically, this chapter focuses on the author's main contributions, which are related to the extension to the MTCG algorithm (Section 7.3) to support parallel stages. The experimental results presented here use a partitioning heuristic proposed by Easwaran Raman, which is briefly described in Section 7.2.

In Chapter 6, we presented the Decoupled Software Pipelining (DSWP) thread partitioning. Although widely applicable, the results in Section 6.3.5 demonstrate that DSWP does not scale well with the number of threads. In this chapter, we describe an extension of DSWP, called Parallel-Stage DSWP, that trades applicability for increased scalability [77, 81].

Before describing PS-DSWP, it is important to understand the main limiting factor of DSWP. Since DSWP treats strongly connected components (SCCs) in the PDG as atomic blocks for partitioning, the latency of the slowest SCC is a lower bound on the latency that

DSWP can achieve for a loop. In other words, DSWP reaches its maximum performance when the slowest stage of the pipeline is exactly the largest SCC.

Our key observation is that, although SCCs cannot be split among multiple threads to form a strict pipeline, under some conditions, a pipeline stage can be executed by multiple threads concurrently. In particular, if the bottleneck stage can run concurrently on multiple threads, it is possible to achieve performance beyond the limit of basic DSWP. This chapter investigates and exploits opportunities for *parallel stages* in the pipeline, i.e. stages executed in parallel by multiple threads.

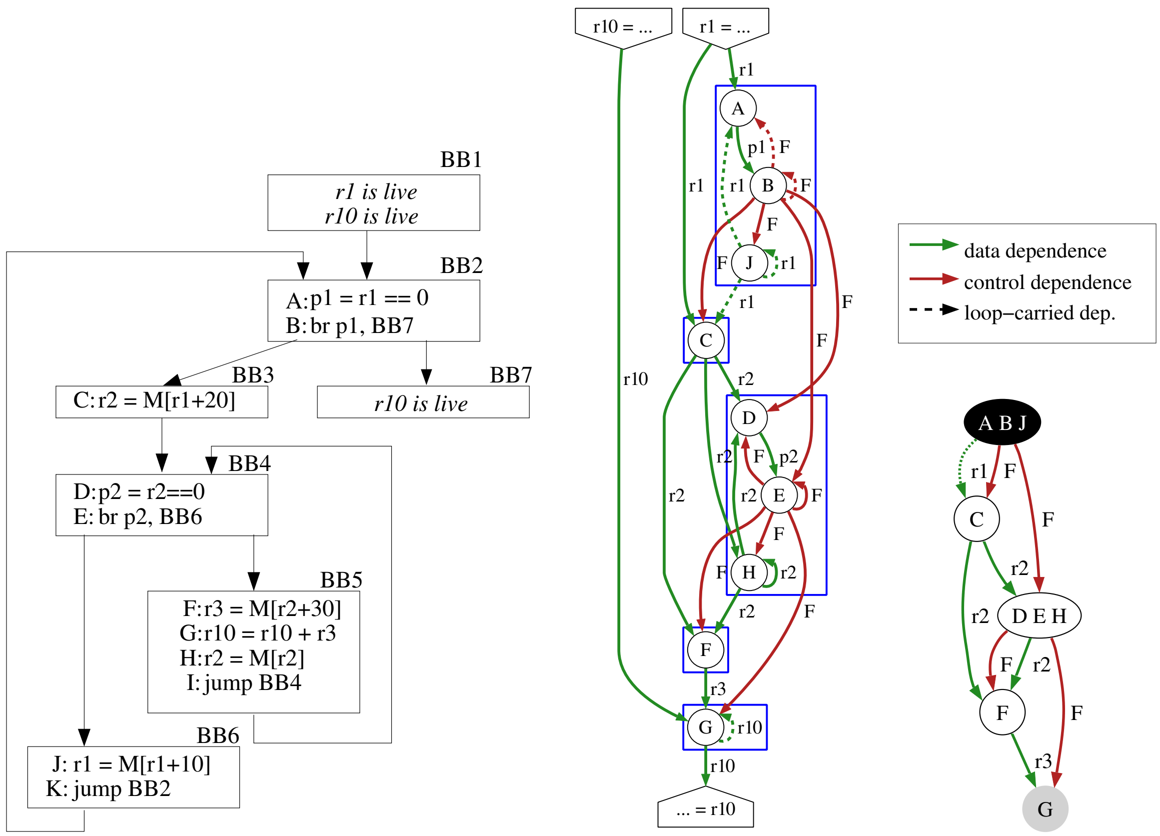
The question that arises is: when can we make a stage parallel, especially if it contains a non-trivial SCC? Naturally, SCCs represent recurrences, or dependence cycles, so at first it may seem that those must be executed sequentially. Our key insight is that, although a non-trivial SCC must include loop-carried dependences, this SCC can be safely executed by multiple threads if none of its loop-carried dependences are carried by the outer loop to which DSWP is being applied. In other words, a large SCC may exist because of dependences carried only by an inner loop. In this case, although each invocation of the inner loop must be executed by a single thread, different invocations of this inner loop can be concurrently executed in separate threads. Another example of a large SCC that may be assigned to a parallel stage is in case of function calls. If multiple calls to a function may execute concurrently, then the call instructions to this function will constitute trivial but heavy-weight SCCs. In some cases, this problem can be mitigated by function inlining. However, inlining may not be practical because of code size. Furthermore, executing this SCC in a parallel thread provides better scalability than doing inlining and subdividing it in multiple stages.

As an example, consider the code in Figure 7.1, which is the same example from Chapters 3 and 6. This code traverses a list of lists, implemented as a linked data structure, and adds up the `cost` fields of all the elements. Figure 7.2(a) shows this example in low-level code, and its corresponding PDG and DAG_{SCC} are illustrated in Figures 7.2(b)-(c). The arcs for intra-iteration dependences with respect to the outer loop are represented by solid lines; dependences carried by the outer loop are represented by dashed lines. Data dependence arcs are annotated with the corresponding register holding the value, while control dependence arcs have no label. Since there are five SCCs in Figure 7.2(c), DSWP can extract up to five threads for this example. However, due to imbalance among the SCCs, it is unprofitable to use five threads in this case. In fact, assuming the lists traversed in the inner loop contain at least a few elements, the SCC DEH will be the bottleneck. However, its two loop-carried dependences, $E \rightarrow D$ and $H \rightarrow D$, are carried only by the inner loop. Therefore, this SCC can be assigned to a parallel stage.

```
while (list != NULL) {
  for (node = list->head; node != NULL;
       node = node->next) {
    total += node->cost;
  }
  list = list->next;
}
```

Figure 7.1: Motivating example.

Similar to DSWP, PS-DSWP can benefit from any analysis or transformation that proves the absence of or eliminates loop-carried dependences. For both DSWP and PS-DSWP, these analyses and transformations can break SCCs, thus exposing more parallelization opportunities. In the case of PS-DSWP, these can also expose larger pieces of code that can be assigned to a parallel stage. In addition, for PS-DSWP, it is important to differentiate which loops carry each dependence. There are several analyses and transformations in the literature that can be applied to disprove or eliminate loop-carried dependences, thus greatly improving the applicability of PS-DSWP. We discuss some of these techniques in the next section. After that, we discuss thread partitioning for PS-DSWP, present the extensions to the MTCG algorithm to support parallel stages, and provide an experimental evaluation.



(a) Original code (b) PDG (c) DAG_{SCC}

Figure 7.2: (a) Low-level code, (b) PDG, and (c) DAG_{SCC} for the example in Figure 7.1.

7.1 Breaking Loop-Carried Dependences

Many memory analysis techniques have been proposed to disprove false memory dependences. For regular array-based codes, a variety of complex analyses have been proposed to find DOALL loops, including the GCD and the Omega tests [2, 115]. Although these techniques are rarely sufficient to find truly DOALL loops beyond scientific applications, they can eliminate some dependences and thus expose larger parallel stages for PS-DSWP.

Programs using pointer-based, recursive data structures are generally much harder to analyze. Nevertheless, several existing techniques can prove the absence of loop-carried dependences in these scenarios. Notably, memory-shape analyses [34, 37, 89] can be used for this purpose. By proving that a data structure is acyclic, shape analyses can be used to determine that different iterations of a loop traversing such data structure can execute in parallel. Complementally, the navigator analysis proposed by Ghiya et al. [33] can prove that a traversal loop follows an acyclic path even if the data structure is cyclic.

In several cases, even though true loop-carried dependences exist, transformations can be applied to eliminate these dependences. The classic examples are *reductions*, such as *sum reduction* [2]. Reduction transformations benefit from the commutative and associative properties of operations to allow them to execute in a different order. For example, a sequence of additions can be performed in any order and the result is the same. To be part of a reduction, a variable must have no use inside the loop other than in the reduction instructions. An example of sum reduction is operation G in Figure 7.2 (denoted in gray in Figure 7.2(c)). Even though this instruction has a loop-carried self dependence cycle, it can be assigned to a parallel stage. In the parallelized code, each thread executing this stage will have its own local copy of the reduction variable, and the values of all these variables will be added together at the end. Other types of reductions include multiplication, minimum, maximum, and complex minimum and maximum (with other associated values) [2].

Other common sources of memory dependences are calls to memory-management routines, such as `malloc` and `free` in C. Typically, the calls to these functions are conservatively executed sequentially. However, under some circumstances, these dependences can be eliminated. To do so, it is necessary to analyze the code to determine that the addresses and the order of the allocated memory chunks do not affect the program's semantics. Under this common circumstances, the calls to memory-management functions can be replaced by calls to corresponding thread-safe versions, if available.

7.2 PS-DSWP Partitioning

For PS-DSWP, the thread-partitioning problem is extended with one more dimension. Besides partitioning the nodes of the DAG_{SCC} among the pipeline stages, a *replication factor* must be chosen for each stage. For a particular stage S_i, its replication factor $f(S_i)$ corresponds to the number of threads that will execute S_i. Therefore, if a stage S_i has $f(S_i) > 1$, then S_i is a *parallel stage*. Otherwise, $f(S_i) = 1$ and S_i is a *sequential stage*. Being s the the number of stages in the pipeline and t the target number of threads, the replication factors must satisfy: $\sum_{i=1}^{s} f(S_i) \leq t$. This guarantees that the number of threads needed is at most the number of target threads.

As mentioned earlier, not every SCC is assignable to a parallel stage. In order to determine whether a set of SCCs can be assigned to a parallel stage, SCCs are classified into two categories. If a SCC has no dependence carried by the outer loop (i.e., the one being parallelized), then it is classified as a *doall* SCC. Furthermore, for a given SCC, if all its dependences carried by the outer loop correspond to reductions (as described in Section 7.1), then this SCC is *doall*. Otherwise, the SCC contains dependences carried by the outer loop that are not subject to reduction transformations, and the SCC is *sequential*. In the example in Figure 7.2(c), the only sequential SCC is ABJ (denoted in black). SCC G, despite having a dependence carried by the outer loop, is a sum reduction.

Based on the definitions above, a pipeline stage S_i may be made parallel if two conditions are satisfied. First, all S_i's SCCs must be doall. Second, all the arcs in the PDG's subgraph induced by the instructions in S_i must not be carried by the outer loop. Together, these two conditions ensure that a parallel stage has no dependence carried by the outer loop, except for reduction-related dependences.

Similar to DSWP, the primary goal of a PS-DSWP partitioner should be to balance the load among the threads. In the presence of parallel stages, this means to minimize $MAX_{i=1}^{s}(exec\_time(S_i)/f(S_i))$, where $exec\_time(S_i)$ is the execution time for stage S_i. Clearly, this problem is at least as hard as the DSWP partitioning problem, which was demonstrated to be NP-hard in Section 6.2.

To solve this problem in practice, Raman et al. [81] proposes a simple heuristic, which focuses on catching cases where a single, large parallel stage is possible. This heuristic starts with each SCC in its own stage, and then greedily merges parallel stages as long as the resulting stage is also parallel and no cycle among the stages is formed. When it is no longer possible to merge parallel stages, the parallel stage with the largest weight is selected as the only final parallel stage. All the other stages are merged, still ensuring the unidirectional communication between stages, and made sequential stages. For non-DOALL loops, this can result in three configurations: (1) a sequential stage followed by a parallel stage; or (2) a parallel stage followed by a sequential stage; or (3) a sequential stage, followed by a parallel stage, followed by another sequential stage. The replication factor f for the final parallel stage is then selected as the target number of threads minus the number of sequential stages. As we show in Section 7.4, this simple heuristic is able to find good partitions for a few benchmarks. However, there are several drawbacks with this heuristic, such as not balancing the load among the threads and allowing a single parallel stage. Studying better partitioning algorithms for PS-DSWP is left as future work and, in fact, it is the subject of ongoing research.

7.3 Code Generation

In order to support parallel stages in the pipeline, it is necessary to extend the MTCG algorithm in several ways. This section describes these extensions, as we proposed in [81]. In general, these extensions can be applied as a post-pass to the MTCG algorithm, which need not be aware of parallel stages.

A very naïve code generation strategy to support parallel stages is to initially apply loop unrolling. This way, multiple copies of corresponding code could be assigned to different stages, therefore obtaining the effect of parallel stages. However, this approach does not scale well because of code growth. We avoid this problem with a slightly more elaborate code generation scheme, which allows all the threads executing the same stage to share the same code.

The key to avoid code replication is to be able to access the communication queues using *indexed addressing mode*. In this mode, a queue is specified by a constant offset plus the value contained in a register. By simply changing the value of this register for different loop iterations, the communications between sequential and parallel stages can be coordinated in a simple manner. Indexed addressing mode is trivial to implement in case of software queues. And, even for hardware queues, which typically only allow immediate addressing to queues [85, 104], this requires just a slight hardware change. The trickiest part is to avoid having to encode a register value in the special `produce` and `consume` instructions, which may require more bits to encode than what is available. However, we can work around this problem by adding one architectural register to contain a base queue value, which is added to the immediate queue number encoded in the instructions. We call this register the *queue-base register*. The only other addition is a special instruction to set the value of these queue-base register. We call this instruction `queue.set`.

The next subsections describe the transformations that need to be performed after the MTCG algorithm is applied. Notice that these are independent from the communication placement strategy employed by MTCG, thus naturally allowing COCO to be used in conjunction with parallel stages.

7.3.1 Communication Insertion

This section explains how to adjust the communication instructions inserted by the MTCG algorithm to deal with the presence of parallel stages.

Loop Communications

Inside the parallelized loop, it is necessary to adjust the communication instructions to enable proper communication between sequential and parallel stages. As mentioned above, one of the goals is to have all threads executing a parallel stage share the same code.

One possible scheme is for all threads to share the same communication queues, with parallel threads dynamically pulling work out of the queues as they finish their previous iterations. This scheme has the advantage of dynamically balancing the load among the parallel threads, and it is somewhat similar to the OpenMP workqueuing or taskqueuing model [95, 102]. However, since PS-DSWP can generate multiple parallel threads communicating to a single downstream sequential thread, some complications arise with this scheme. First, this scheme requires a more complex design for communication queues, where multiple producers can write to the same queue. Second, with the scalar communication queues that we have been assuming so far, not all values are read at the same time. Due to timing issues, this brings the possibility of different threads consuming the values for the same loop iteration, which can result in incorrect execution. Therefore, to guarantee correctness, a mechanism to enforce that each parallel thread consumes all values corresponding to the same loop iteration is necessary. Third, this dynamic work allocation scheme brings complications because the parallel threads may need to communicate in the same order with both an upstream and a downstream sequential threads. In other words, the order in which the threads executing a parallel stage consume values from their upstream sequential stage may have to match the order in which they produce values to their downstream sequential thread. More precisely, this is necessary when the sequential threads also communicate directly. This can, however, be avoided by forcing all values from the upstream sequential stage be communicated to the downstream sequential stage through the intermediate parallel stage. Nevertheless, this requires adapting the MTCG algorithm to enforce a pattern of strictly linear pipeline communication.

To avoid the issues of dynamic work assignment mentioned above, we opted for a simpler scheme, in which iterations are statically assigned to threads executing a parallel stage. In this approach, each thread executing the parallel stage both consumes from and produces to a unique set of queues. Since all threads for a parallel stage execute the same code, uniqueness can be achieved by setting the base-queue register with a different value upon creation of the auxiliary threads. The value of the base-queue register should be chosen so that the sets of queues used by the parallel threads do not overlap. For the threads executing sequential stages, the value of the base-queue register is adjusted every iteration of the outer loop, so that they communicate with the parallel threads in a round-robin fashion.

Initial Communications

As described in Section 3.1.3, the MTCG algorithm inserts communication primitives before the region's entry for dependences from instructions before the parallelized region to instructions inside this region. In PS-DSWP, if the target of the dependence is part of a parallel thread, special handling is necessary. There are two cases. First, if this is a loop-invariant register dependence, then this dependence must be communicated to every parallel thread before entering the region. Second, if this is a loop-variant register dependence, then it is necessary to move the communication instructions inside the loop region. The reason is that, in this case, the register involved in the dependence may be redefined in some loop iterations. Nevertheless, as per the MTCG algorithm, the new value assigned to the register will only be communicated to the parallel thread executing that particular loop iteration. The other parallel threads would still have the stale value of the register. Since the communication of this new register value may be nested inside a deep control region, we simply communicate the value of such register at the loop header. In effect, this will communicate the latest value of conditionally defined registers to the parallel threads in every loop iteration. A similar problem may occur due to conditionally exercised memory dependences from a sequential to a parallel stage. The MTCG algorithm will only enforce a memory synchronization between the sequential thread and the parallel thread executing that particular iteration. Nevertheless, this synchronization needs to be enforced to every parallel stage, which is also achieved by adding a synchronization at the loop header.

Final Communications

In the MTCG algorithm, all registers written inside the parallelized region and live at the exit of the region need to be communicated back to the main thread. As discussed in

Section 3.1.3, a problem arises if multiple threads may contain the latest value of a given register. A similar problem may occur if a register is conditionally assigned inside a parallel stage. The same solutions discussed in Section 3.1.3 are applicable here. Nevertheless, to enable larger parallel stages, the time-stamp solution is definitely preferable for PS-DSWP.

Reductions

For reduction SCCs assigned to parallel stages, the corresponding reduction transformation is applied. These transformations are similar to standard reduction transformations applied to DOALL loops [2]. More precisely, each thread keeps its local copy of the reduced value, which is initialized to the neutral value for that reduction operator (e.g. zero for sum). At the exit of the loop, all the parallel threads communicate their local reduction values to the main thread, which then applies the reduction operation to obtain the correct final value.

7.3.2 Loop Termination

The last code generation problem that arises from parallel stages is loop termination. In the basic DSWP, loop termination is naturally handled due to the control dependences corresponding to the loop exits. More specifically, once a loop exit is taken in the first thread, the corresponding control dependence is communicated to every other thread, thus causing them to terminate the loop as well. However, this scheme does not work directly in the presence of parallel stages. With PS-DSWP, once a loop exit is taken, only the parallel thread executing the last loop iteration will terminate.

Our solution is to add one special communication to terminate the execution of threads executing a parallel stage. At the header of the loop, the parallel-stage threads consume a *continue* token, and exit the loop if this is `false`. At the header of the first-stage thread, `true` is passed as the *continue* token to the parallel thread that will execute the current iteration, thus enabling it to continue. At the loop exit, the first-stage thread sends `false` as the *continue* token to every thread executing the parallel stage, thus causing them all to leave the loop through the exit created at the header. The parallel-stage thread executing the last iteration, which will take the normal loop exit created by MTCG, must consume the *continue* token once it terminates the loop.

In order to enable a parallel first stage, loop termination is handled differently. To have a parallel first stage, it is necessary to have a counted loop. In this case, instead of having a sequential first-thread that sends *continue* tokens to the parallel-stage threads, all threads know upfront how many iterations they have to execute. This information actually makes loop termination for counted loops straightforward to implement [2, 81].

7.4 Experimental Evaluation

In this section, we present an evaluation of the PS-DSWP technique. The experimental methodology is the same described in Section 4.4.1, except for the additional base-queue register and `queue.set` instruction discussed in Section 7.3.

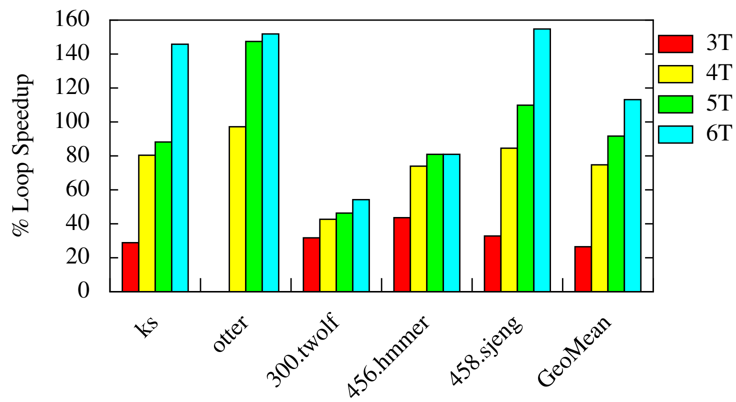
Despite the importance of array-dependence and memory-shape analysis for PS-DSWP mentioned in Section 7.1, the VELOCITY compiler currently lacks these techniques. To compensate for that, we used an alternative approach for the benchmarks evaluated in this section. The approach was to use an in-house, loop-aware memory profiler to detect overly conservative memory dependences based solely on the available pointer analysis. Such dependences were then manually inspected for the feasibility of eliminating them through static array-dependence and shape analysis. The dependences determined to be feasible to eliminate through such analyses were then left out of the PDG.

7.4.1 PS-DSWP Results

Among the benchmarks described in Section 4.4.1, the simple partitioning heuristic described in Section 7.2 obtained a parallel stage for five benchmarks.[1] For these five bench-

[1] The compiler only parallelized loops if the estimated speedup was at least 50% over sequential execution.

marks, Figure 7.3 presents their speedups using up to 6 threads, over the single-threaded execution. PS-DSWP obtains speedups of up to 155% (for *458.sjeng*), and a overall speedup of 113% among these five loops. Figure 7.3(b) describes the scheme of pipeline stages generated by PS-DSWP. In this table, p means a parallel stage and s means a sequential stage. For example, for *ks*, there is a first sequential stage followed by a parallel stage.



(a)

Benchmark	Pipeline stages
ks	$s \rightarrow p$
otter	$s \rightarrow p \rightarrow s$
300.twolf	$s \rightarrow p$
456.hmmer	$p \rightarrow s$
458.sjeng	$s \rightarrow p$

(b)

Figure 7.3: (a) Speedups for PS-DSWP over single-threaded execution. (b) Resulting pipeline stages (*s*equential or *p*arallel).

7.4.2 Comparison to DSWP

In this section, we compare the results of PS-DSWP with the basic DSWP. This comparison includes all the benchmarks described in Section 4.4.1 that were parallelized by at least one of these two techniques. For the benchmarks that had loop-carried dependences manually removed for PS-DSWP, the same version of the benchmark without such dependences was used for DSWP in this evaluation.[2]

Figure 7.4 shows the results for these benchmarks using up to 6 threads. As expected, DSWP is applicable to many more benchmarks than PS-DSWP. On the other hand, when applicable, PS-DSWP generally results in significantly better performance. The only exception is *300.twolf*, for which both techniques are applicable and DSWP results in better

[2]This explains the differences between speedups for DSWP reported here and in Chapter 6.

performance. The main reason for this is that the loop in *300.twolf*, although invoked many times, iterates just a few times per invocation. In this scenario, the parallel stages are not fully utilized.

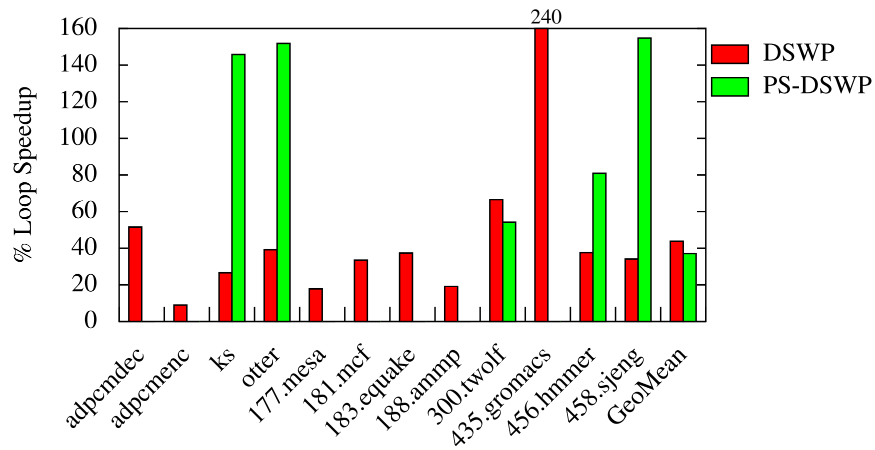


Figure 7.4: Speedups for PS-DSWP and DSWP using up to 6 threads, over single-threaded execution.

7.5 Significance

This chapter presented an extension of the DSWP technique (Chapter 6) that enables the presence of parallel stages in the pipeline. This new technique is called Parallel-Stage DSWP (PS-DSWP). Compared to DSWP, PS-DSWP trades applicability for increased scalability. PS-DSWP is generally beneficial in quasi-DOALL loops, which, despite having some recurrences, contain a large portion that can benefit from iteration-level parallelism.[3]

Akin to DSWP, PS-DSWP shares similarities with loop distribution. Like loop distribution, PS-DSWP is beneficial when a large portion of the original loop contains iteration-level parallelism. Like DSWP, PS-DSWP offers more benefits than loop distribution by executing all loops concurrently. This also increases the applicability of DSWP and PS-DSWP over loop distribution, by enabling the parallelization of uncounted loops. Further-

[3]We call iteration-level parallelism what is typically called loop-level parallelism in the literature [2]. We adopt this terminology to avoid confusion with pipeline parallelism, which is also extracted at the loop level.

more, by relying on our MTCG algorithm, DSWP and PS-DSWP are able to handle loops with arbitrary control flow.

PS-DSWP is also very similar to the combination of DOALL and DOPIPE [24]. The main advantage of PS-DSWP comes again from its ability to handle applications with irregular memory accesses and control flow. The technique proposed by Davies [24] is only applicable to very structured programs, with counted loops, and it only creates parallel stages for DOALL inner loops.

Chapter 8

Conclusions and Future Directions

The shift in the microprocessor industry towards multi-threaded and multi-core architectures has brought the necessity of exploiting thread-level parallelism (TLP) to mainstream computing. Given the extra burden and complications that parallel programming imposes to the programmer, it is desirable to create tools to automatically extract TLP as much as possible. This is particularly important for fine-grained TLP enabled by modern multi-threaded architectures.

Previous research in compilation techniques attacked the problem of extracting parallelism in mainly two different granularities. At the high-level, loop-parallelization techniques have focused at extracting coarse-grained parallelism from regular, mostly scientific applications. At the low-level, instruction scheduling techniques have focused at exploiting fine-grained, instruction-level parallelism for a single processor. This thesis bridges the gap between these two fronts.

8.1 Summary

The central idea in this thesis is to extend low-level instruction scheduling techniques to exploit fine-grained thread-level parallelism. A key advantage brought by multi-threaded architectures is the possibility of concurrently executing different, unrelated control regions. To truly exploit this potential, it is imperative to have a large scope for scheduling. Typically, instruction scheduling techniques are applied to each basic block or trace independently. However, this can only exploit local parallelism among instructions with very related or equivalent conditions of execution.

One of the main contributions of this thesis was to propose a general compilation framework that enables global instruction scheduling for multi-threaded architectures. By global we mean an *arbitrary* intra-procedural scope. This framework is based on a program dependence graph (PDG) intermediate representation, and it provides a powerful separation of concerns into *thread partitioning* and *code generation*.

A key component of our framework is the multi-threaded code generation (MTCG) algorithm described in Chapter 3. This algorithm takes an arbitrary thread partition of the PDG and generates efficient multi-threaded code. In particular, the code generated by the MTCG algorithm does not replicate the entire control flow in all threads, a huge limitation of previous work on multi-threaded instruction scheduling. Instead, the MTCG algorithm only replicates the control-flow regions that are relevant to each thread, given its subset of the PDG nodes. This is an important property for generating truly multi-threaded code. To the best of our knowledge, the MTCG algorithm is the first method to satisfy this property and to support arbitrary control flow and thread partitions. Furthermore, our MTCG algorithm is elegant and simple to implement. The key to this simplicity is MTCG's uniform handling of dependences, particularly of control dependences.

This thesis also investigated two different approaches to placing communication instructions in the MTCG algorithm. The first approach uses a very simple property, which is to communicate an inter-thread dependence at the point corresponding to its source instruction. The second approach, called COCO and described in Chapter 4, is significantly more elaborate. COCO employs thread-aware data-flow analysis to determine all the safe points to communicate each dependence, and then uses graph min-cut algorithms to choose communication points that reduce the frequency of communications. Our experiments showed that, although the simple communication placement method proposed generally does a good job, COCO's improvement can be quite substantial in some cases.

For thread partitioning, this thesis investigated three different approaches. The GREMIO technique, described in Chapter 5, extends traditional list scheduling to operate globally and to be aware of different levels of resources (functional units in a core, and multiple cores in a processor). GREMIO schedules instructions based on the control relations among them, which determine whether or not they can be issued simultaneously in the same core/thread. GREMIO operates hierarchically based on the loop tree, using a dynamic programming approach.

Another thread partition technique proposed in this thesis, called Decoupled Software Pipelining (DSWP) and described in Chapter 6, extends the idea of software pipelining to schedule instructions of an arbitrary loop region. DSWP partitions the nodes of the PDG into multiple stages of a pipeline. The pipeline is formed by enforcing a unidirectional flow of dependences among the stages (threads). To obtain maximum efficiency, the goal of a DSWP partitioner is to balance the load among the threads, while minimizing inter-thread communication.

The final thread partitioner described in this thesis is an extension of DSWP that allows multiple threads to execute the same pipeline stage in parallel. This extension is called Parallel-Stage Decoupled Software Pipelining (PS-DSWP), and it was described in Chapter 7. In essence, PS-DSWP combines the pipeline parallelism of DSWP with iteration-level (or DOALL) parallelism to improve scalability.

All the techniques proposed in this thesis were implemented in the VELOCITY compiler, and evaluated on a cycle-accurate multi-core model based on Itanium 2 cores. This multi-core model has a scalar communication hardware support, which provides low intra-core overhead for inter-thread communication and synchronization. Our experiments evaluated the three thread partitioners describe in this thesis. These techniques were able to extract significant parallelism for a set of applications that are very hard to parallelize, especially with non-speculative techniques like ours. These results are very encouraging, and they demonstrate the power of our techniques to extract thread-level parallelism from sequential general-purpose applications to benefit from modern microprocessors.

8.2 Conclusions

Overall, this thesis proposed a new way of looking at and exploiting thread-level parallelism. Modern multi-threaded and multi-core processors are typically regarded as a curse because of the difficulty of utilizing them to improve the performance of a single application. Contradicting this general belief, this work demonstrated that the ability to concurrently execute multiple threads actually enables novel opportunities to exploit parallelism. In particular, our work demonstrated that traditional instruction scheduling techniques can be extended to extract parallelism in the form of *thread-level parallelism* that cannot be practically exploited as *instruction-level parallelism*. This insight motivated the techniques developed in this work and enabled the automatic parallelization of notoriously sequential programs. Furthermore, followup work on combining speculation and other enhancements (described in Section 8.3) with the techniques developed in this thesis have demonstrated even higher applicability and scalability [10, 109]. Altogether, the resulting approach provides a viable alternative to attain performance our of modern, highly parallel microprocessors.

8.3 Future Directions

This thesis provides the groundwork for global multi-threaded instruction scheduling research. As such, it opens new avenues to be explored with the goal of improving performance on modern multi-threaded and multi-core processors. In the following, we summarize some of these opportunities for future research in this area, several of which are already being pursued in our group.

- *Novel Thread Partitioning Algorithms*

  The thread partitioning algorithms proposed in this thesis are a first step in obtaining parallelism. They were inspired by instruction-level parallelism schedulers, and designed to have an efficient execution time. They, by no means, extract the maximum amount of parallelism. In fact, as our experiments illustrated, for different benchmarks, a different partitioning algorithm resulted in the best performance. These results motivate more research in this area, not only by integrating the partitioners proposed here in one capable of achieving the best among them, but also in proposing more elaborate, completely new algorithms.

- *Extensions to the Multi-Threaded Code Generation Algorithm*

  Despite its generality, there are several ways in which the MTCG algorithm can be extended. Chapter 4 presented algorithms to improve the inter-thread communication. However, these algorithms are not optimal either, and therefore there may be room for improvement in this area.

  A limitation of the MTCG algorithm described here is that each PDG node can only be assigned to a single block in the partition. Only necessary branches are replicated in multiple threads, which is required to implement the flow of control. There is potential to improve the generated code by duplicating other instructions as well. For example, when we have a simple instruction, say the load of an immediate value into a register, which feeds two uses, MTCG will generate an inter-thread communication if the uses are in different threads. However, by assigning the load-immediate instruction to both threads containing the uses, no communication is necessary. Although the possibility of assigning a node to multiple threads can be enabled by applying node splitting upfront in the PDG, a more general solution is to allow a thread partition to assign the same instruction to multiple threads. In this case, a communication is only required if the thread containing the use does not have a copy of the definition feeding it as well. This ability to assign a single instruction to multiple threads creates an interesting tradeoff between recomputing or communicating a value, which can be explored to improve parallelism. Of course, not every instruction is amenable to being replicated in multiple threads, e.g. calls to I/O libraries. Such restrictions must be enforced in the thread partition.

Another possible extension to the MTCG algorithm is specific to DSWP. The experiments in [82] demonstrate that a strictly linear pipeline can alleviate the necessity for larger communication queues for DSWP. The MTCG algorithm can be extended to enforce this communication pattern for DSWP, by ensuring that a dependence between two non-consecutive stages is always communicated through the intervening stages.

Another restriction in our compiler framework is that it assumes the initial program is sequential. To support parallel programs, the PDG representation, which is central in our work, should be extended to encode the semantics of parallel programs. One possibility is to use the Parallel Program Graph (PPG) representation [92]. Moreover, to obtain better parallelism, the thread partitioning and MTCG algorithms need also be made aware of the semantics of parallel programs.

For simplicity, the MTCG algorithm utilizes a different communication queue for each inter-thread dependence. Clearly, this is not optimal and may require a large number of communication queues. Analogous to register allocation, a queue-allocation technique may be devised to reuse communication queues.

- *Optimization of Multi-Threaded Code*

  In our compiler, we are able to apply many single-threaded optimizations to each of the generated threads. However, some optimizations are not safe, because they do not understand the semantics of the multi-threaded code. For example, register promotion may incorrectly promote a variable from memory to a register even though it may be accessed by another thread. Such optimizations must be made thread-aware in order to be applicable to multi-threaded code.

In addition to enabling single-thread optimizations to properly operate on code generated by the MTCG algorithm, there is also room to apply novel inter-thread optimizations. The example mentioned above, where a constant value is communicated from one thread to another, could be solved by applying inter-thread constant propagation. Similarly, other traditional code optimizations can be applied to improve the multi-threaded code. To enable inter-thread analysis and optimizations in general, many of the compiler fundamentals need to be revisited and extended. One step in this direction was done by Sarkar [92], based on the Parallel Program Graph (PPG), but a lot more work is necessary in this area.

- *Inter-Procedural Extensions*

The framework described in this thesis is applicable to an arbitrary intra-procedural region. In this scenario, function calls are treated as atomic nodes in the PDG: they can only be assigned to one thread (or one stage in case of PS-DSWP). Of course, this restricts the possible thread partitions, which can result not only in imbalance among the threads but also in the impossibility of isolating instructions inside the called function that participate in a problematic dependence chain. In this work, function inlining was used to mitigate these problems. A more general solution is to extend our framework to operate inter-procedurally, potentially with a whole-program scope. There are several challenges to enable this extension. First, the PDG representation needs to be extended to model inter-procedural code regions. For this, we can rely on the System Dependence Graph (SDG) representation [41], which essentially has the same properties of the PDG. A SDG is constructed by building the PDG for each procedure and connecting the call-sites to the called functions. Special data-dependence arcs are inserted to represent parameter passing, and special control dependence arcs, called *call arcs*, are added to connect the call-site with the callee procedure's entry [41]. Using the SDG representation, the MTCG algorithm works mostly unchanged. In particular, any thread that is assigned an instruction inside a called function will have the call-sites of that function as relevant (by Definitions 2 and 3) because of the call arcs. Therefore, call-sites will naturally be replicated for each thread as needed by the MTCG algorithm. Small changes to the MTCG algorithm are necessary both to rename the new procedures resulting from partitioning instructions in one procedure among multiple threads and to specialize the new procedures and their call-sites to only have the necessary parameters.

Another problem that arises from partitioning procedures among multiple threads is related to stack-allocated variables. Since the code accessing these variables may now be split into multiple procedures executed on different threads, these variables should be promoted to the heap. Furthermore, the allocation and deallocation of this heap area must be placed in points in the code that guarantee that they will happen, respectively, before and after all accesses to these variables. This may require extra synchronizations to be added to the code.

We notice that there are some drawbacks in naively utilizing the SDG. The main problem is the lack of context sensitivity, which may create dependence chains among independent instructions. As with any context-insensitive analysis, dependence chains from one call-site will appear to reach uses following another call-site to the same procedure. This problem does not happen with the intra-procedural approach because the call-sites are atomic and represented by different nodes in the PDG. To avoid this problem with the inter-procedural extension, we can use frameworks for inter-procedural analysis with some level of context-sensitivity (e.g. [105]) and duplicate problematic procedures with multiple call-sites. This approach also enables a procedure to have its instructions partitioned differently among the threads for different call-sites. Another problem that may arise from operating with a whole-program scope is the size of the SDG and the scalability of the algorithms used. For really large applications, two alternatives are to seek a good tradeoff between treating procedure calls as atomic or not, and to explore a hierarchical approach.

- *Use of Better Memory Disambiguation Techniques*

  A major source of conservativeness in our framework, and compiler optimizations in general, is the inaccuracy of memory disambiguation techniques. In this work, we use the results of a flow-insensitive, context-sensitive pointer analysis implemented in the IMPACT compiler [70]. The results of this analysis are propagated into VELOCITY's IR. In addition to imprecisions inherent to this specific pointer analysis [70], our framework also suffers from the fact that many optimizations are done in VELOCITY before we apply GMT instruction scheduling. As we demonstrated in [36], the propagation of pointer-aliasing information throughout code optimizations deteriorates the quality of this information even further.

  To improve the accuracy of memory disambiguation, other techniques from the literature, such as memory-shape analysis [33, 34, 37] and advanced array-dependence analyses [2], could be employed. Furthermore, there may be room to improve memory disambiguation analysis by benefiting from specific properties of our parallelization techniques.

- *Speculative Extensions*

  Another avenue for research is the combination of speculation with the techniques devised in this thesis. We have done initial work in this area with the Speculative DSWP extension [75, 109], and there is room for adding speculation to the other techniques as well. The benefit of speculation is that it frees the compiler from being overly conservative in many cases. Given the difficulties of memory disambiguation, speculation can be particularly useful for memory dependences. Furthermore, it can also help in case of infrequently executed control paths. With speculation, the compiler can parallelize the code optimistically and, in case of mis-speculation, only a performance penalty is paid and correctness is not affected. The major downside of speculation is that it generally requires very complex and expensive hardware designs to obtain good performance.

- *Run-time Extensions*

  The evaluation presented in this work utilized a static compiler. Akin to other code optimizations, there is potential to improve the parallelism resulting from our techniques by applying them at run-time. In general, run-time optimizations can benefit from dynamic information that is not available to a static compiler. For instance, run-time information can be used to determine whether or not the threads are well balanced, and to repartition the threads to obtain a better balance if necessary. Furthermore, for speculative extensions, run-time information can be used to determine dependences that either are infrequent or do not manifest at run-time. This information can be used to obtain a better thread partition and to reduce the mis-speculation rates.

- *Improvements to Other Code Optimizations*

  Finally, we believe that some ideas in the COCO framework may be applicable to other code optimization problems. Akin to many code optimizations, the problem of finding the best placement for communication is undecidable and input-dependent. Nevertheless, there is a notion of optimality in presence of profiling information. Many compiler optimizations, however, do not use profiling information to achieve a better solution. Instead, they rely on simplistic estimates only. For example, loop-invariant code motion generally assumes that a loop iterates many times per invocation and that all the code inside the loop is executed, therefore moving code outside the loop is beneficial. Similarly, partial redundancy elimination techniques [17, 51, 53, 64] are only "computationally optimal" under a "safety" assumption that is unnecessary for most computations. Specifically, the safety property aims at avoiding triggering exceptions in cases where the original program would not. For computations that cannot trigger exceptions, this condition can be violated to reduce the number of computations even further. In this case, the idea introduced in the Chapter 4, of using graph min-cut to minimize communications, can be used to minimize the number of computations based on profile weights.

8.4 A Final Word

The techniques proposed in this thesis provided the groundwork for research on global multi-threaded instruction scheduling. A recent study of the potential of these techniques, when combined with some of the extensions mentioned in Section 8.3, has demonstrated an enormous potential for our approach [10]. Specifically, this study showed that extensions of the compilation techniques proposed in this thesis, using the PS-DSWP partitioner, can achieve scalable parallelism for most of the applications in the SPEC CPU 2000 Integer benchmark suite. To achieve these results, it is necessary to extend our framework to operate inter-procedurally and to aggressively use profiling, speculation, and some simple user annotations to eliminate dependences. Although the results reported in [10] did not use compiler-generated code, many of the necessary extensions are being integrated in the VELOCITY compiler, which is currently able to obtain parallelizations similar to those in [10] for several applications. These initial results provide significant evidence of the enormous power and viability of our approach to extract useful parallelism from sequential general-purpose programs to benefit from modern, highly parallel processors.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[3] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 126–138, 1993.

[4] A. W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[5] D. I. August. *Systematic Compilation for Predicated Execution*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 2000.

[6] T. Austin. Pointer-Intensive Benchmark Suite. University of Wisconsin, Madison, Web site: http://pages.cs.wisc.edu/ austin/ptr-dist.html, 1995.

[7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[8] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[9] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[10] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–81, December 2007.

[11] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: a preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, December 1992.

[12] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, May 1996.

[13] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(9):18–27, September 1981.

[14] D.-K. Chen. *Compiler Optimizations for Parallel Loops with Fine-grained Synchronization*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1994.

[15] B. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIG-*

PLAN Conference on Programming Language Design and Implementation, pages 57–69, June 2000.

[16] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, April 1987.

[17] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., 2004.

[18] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[19] K. D. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 308–319, June 1997.

[20] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1990.

[22] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–884, August 1986.

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM*

Transactions on Programming Languages and Systems, 13(4):451–490, October 1991.

[24] J. R. B. Davies. Parallel loop constructs for multiprocessors. Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, May 1981.

[25] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.

[26] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1985.

[27] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, 1988.

[28] J. Ferrante, M. Mace, and B. Simmons. Generating sequential code from parallel code. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 582–592, July 1988.

[29] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[30] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[31] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, 1979.

[33] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 159–173, March 1998.

[34] R. Ghiya and L. J. Hendren. Is it a Tree, DAG, or Cyclic Graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[35] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[36] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[37] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[38] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[39] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[40] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 146–157, 1988.

[41] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[42] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[43] The IBM XL C/C++ compilers. http://www.ibm.com/software/awdtools/ccompilers.

[44] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, 2002.

[45] Intel Corporation. Intel Pentium 4 Processor Family. Online documentation: http://www.intel.com/design/Pentium4/documentation.htm, 2002.

[46] The Intel C/C++ Compiler. http://www.intel.com/software/products/compilers.

[47] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 59–70, 2004.

[48] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Trans. Program. Lang. Syst.*, 21(6):1251–1297, 1999.

[49] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing*, pages 407–416, November 1990.

[50] K. Kennedy and A. Sethi. A communication placement framework with unified dependence and data-flow analysis. In *Proceedings of the 3rd International Conference on High Performance Computing*, pages 201–208, December 1996.

[51] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.

[52] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, January 1998.

[53] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.

[54] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[55] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[56] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 46–57, May 1992.

[57] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[58] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[59] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw Machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

[60] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002.

[61] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[62] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[63] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[64] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, pages 96–103, February 1979.

[65] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[66] C. J. Newburn. *Exploiting Multi-Grained Parallelism for Multiple-Instruction-Stream Architectures*. PhD thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, United States, November 1997.

[67] C. J. Newburn. Personal Communication, January 2008.

[68] C. J. Newburn and J. P. Shen. Automatic partitioning of signal processing programs for symmetric multiprocessors. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pages 269–280, October 1996.

[69] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 103–114, December 1998.

[70] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium*, August 2004.

[71] The OpenMP API specification for parallel programming. http://www.openmp.org.

[72] G. Ottoni and D. I. August. Global multi-threaded instruction scheduling. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–68, December 2007.

[73] G. Ottoni and D. I. August. Communication optimizations for global multi-threaded instruction scheduling. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–232, March 2008.

[74] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, November 2005.

[75] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining, November 2005. Presentation slides for [74]: http://www.cs.princeton.edu/˜ottoni/pub/DSWP-MICRO38.ppt.

[76] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. A new approach to thread extraction for general-purpose programs. In *Proceedings of the 2nd Watson Conference on Interaction between Architecture, Circuits, and Compilers*, September 2005.

[77] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August. From sequential programs to concurrent threads. *IEEE Computer Architecture Letters*, 4, June 2005.

[78] D. A. Padua. *Multiprocessors: Discussion of some theoretical and practical problems*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, United States, November 1979.

[79] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[80] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[81] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, 2008.

[82] R. Rangan. *Pipelined Multithreading Transformations and Support Mechanisms*. PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ, United States, June 2007.

[83] R. Rangan and D. I. August. Amortizing software queue overhead for pipelined inter-thread communication. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism (PMUP)*, pages 1–5, September 2006.

[84] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 259–269, December 2006.

[85] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

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

[87] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pages 183–198, October 1981.

[88] K. Rich and M. Farrens. Code partitioning in decoupled compilers. In *Proceedings of the 6th European Conference on Parallel Processing*, pages 1008–1017, Munich, Germany, September 2000.

[89] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[90] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge, MA, 1989.

[91] V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 1992.

[92] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[93] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 17–26, June 1986.

[94] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[95] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in OpenMP. In *Proceedings of the 1st European Workshop on OpenMP*, September 1999.

[96] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu. Field-testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 26–37, June 2004.

[97] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 112–119, April 1982.

[98] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[99] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[100] B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, WA, 1993.

[101] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[102] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen. Compiler support of the workqueuing execution model for Intel SMP architectures. In *Proceedings of the 4th European Workshop on OpenMP*, September 2002.

[103] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[104] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[105] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[106] J. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[107] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.

[108] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. A. Connors. Chip multi-processor scalability for single-threaded applications. In *Proceedings of the Workshop on Design, Architecture, and Simulation of Chip Multi-Processors*, November 2005.

[109] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[110] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[111] M. Wilkes. *Automatic Digital Computers*. John Wiley & Sons, 1956.

[112] S. Winkel. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2004.

[113] S. Winkel. Optimal versus heuristic global code scheduling. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–55, December 2007.

[114] S. Winkel, R. Krishnaiyer, and R. Sampson. Latency-tolerant software pipelining in a production compiler. In *Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 104–113, April 2008.

[115] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[116] Y. Wu and J. R. Larus. Static branch prediction and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 1–11, December 1994.

[117] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[118] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 160–169, December 2001.

[119] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, October 2002.

[120] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, November 2002.