Software Orchestration of Instruction Level Parallelism on Tiled Processor Architectures

by

Walter Lee

B.S., Computer Science
Massachusetts Institute of Technology, 1995

M.Eng., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1995

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

[June 2005]

Signature of Author: _____
Department of Electrical Engineering and Computer Science
May 16, 2005

Certified by: _____
Anant Agarwal
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by: _____
Saman Amarasinghe
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: _____
Arthur C. Smith
Chairman, Departmental Graduate Committee

Software Orchestration of Instruction Level Parallelism on Tiled Processor Architectures

by

Walter Lee

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2005 in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy
in Electrical Engineering and Computer Science

ABSTRACT

Projection from silicon technology is that while transistor budget will continue to blossom according to Moore's law, latency from global wires will severely limit the ability to scale centralized structures at high frequencies. A tiled processor architecture (TPA) eliminates long wires from its design by distributing its resources over a pipelined interconnect. By exposing the spatial distribution of these resources to the compiler, a TPA allows the compiler to optimize for locality, thus minimizing the distance that data needs to travel to reach the consuming computation.

This thesis examines the compiler problem of exploiting instruction level parallelism (ILP) on a TPA. It describes Rawcc, an ILP compiler for Raw, a fully distributed TPA. The thesis examines the implication of the resource distribution on the exploitation of ILP for each of the following resources: instructions, registers, control, data memory, and wires. It designs novel solutions for each one, and it describes the solutions within the integrated framework of a working compiler.

Performance is evaluated on a cycle-accurate Raw simulator as well as on a 16-tile Raw chip. Results show that Rawcc can attain modest speedups for fine-grained applications, as well speedups that scale up to 64 tiles for applications with such parallelism.

Thesis Advisors: A. Agarwal, Professor, Computer Science & Engineering
S. Amarasinghe, Associate Professor, Computer Science & Engineering

Acknowledgments

I feel privileged to have worked with so many talented people during my time at MIT, starting at the top with my advisors Anant Agarwal and Saman Amarasinghe. Anant has been advising me throughout my graduate career. He is a visionary with amazing technical acumen. My research has greatly benefited not only from his technical guidance, but from his vision as well. When I get bogged down with details, he keeps me focused on the big picture. He also makes the pressures of graduate student live easier to endure with his steady stream of enthusiasm and praises.

Saman became my co-advisor toward the beginning of the Raw project. He has taken a very active role in my research, especially early on. He was responsible for encouraging me to formalize many research ideas, such as convergent scheduling. He has a lot of insights about the role of compilers in computer architectures, and I have tried to learn as much from him as possible. Saman is also very patient and detailed with his feedback and comments in both my research and writing. The quality of my work is dramatically better due to his guidance.

Srinivas Devadas was on my thesis committee. He provides valuable feedback on drafts of my thesis from the view of an external reader. Several of his organizational comments greatly improved the clarity of the presentation.

I would like to give special thank to several close collaborators. Michael Taylor is my favorite and most frequent collaborator, and a good friend as well. One thing I learn at MIT is that there are perfectionists, and then there are *perfectionists.* Michael is a *perfectionist,* and he brings his high standards to everything he is involved in. He and I spent many all nighters together working on a variety of projects, including starsearch, landscaper, rawgrok, medicare, and SON (I swear they are all clean and research related!). Michael also lead the Raw chip design efforts, wrote the Raw cycle accurate C simulator, and had a hand in many parts of the software tool chain. In particular, his awesome extensible debugger helps make software development on Raw much more manageable that it could have been.

I worked very closely Rajeev Barua during the beginning of the Raw compiler efforts. Rajeev is one of the most focused persons I know, and we got a lot of stuff done in a short amount of time. We wrote the first version of the Raw compiler together. Rajeev developed most of the techniques used in Maps, the memory managing component of the Raw compiler.

Matt Frank was like a third advisor to me. My research has benefited greatly from many discussions with him. When Matt clears his throat, the world stops and listens. He has an uncanny ability to always be supportive and state his point of view at the same time. He is also great at finding interesting insights from the most mundane, and his comments are always salient, concise, and thought provoking.

David Wentzlaff was an officemate and great friend. Though we did not have many chances to work together, the few times we did work together was a blast. Dave is one of the most creative, productive, and responsible people I know. David is also an all around great guy to talk to about anything because he is genuinely interested in the opinion of others.

System research requires an immense amount of human capital. My work would not have been possible without contributions from the following. On the compiler side, Vivek Sarkar had a big early influence on the Raw compiler. He was the one who pointed out the similarities between ILP orchestration on Raw and task graph scheduling on MIMD machines. Vivek also served as my mentor during my internship at IBM.

Several people have contributed directly to the Raw compiler infrastructure. Niko Matsakis implemented the Mips backend on which the Raw backend was based. Radu Rugina implemented the pointer analysis used in Rawcc. Sam Larsen implemented the congruence analysis and transformation package. Shane Swenson implemented a part of the Raw convergent scheduler. Devabhaktuni Srikrishna wrote some early transformation passes.

Other things being equal, research on real systems carries more weight than research on paper systems. Thus, I am thankful for the people who dedicated many man years to the chip and prototype board development efforts: Fataneh Ghodrat, Benjamin Greenwald, Paul Johnson, Jason Kim, Albert Ma, Jason Miller, Arvind Sarif, Nathan Schnidman, Michael

Taylor, and David Wentzlaff.

I am grateful for members of the Raw team who built the Raw infrastructure that my research depends upon. Elliot Waingold wrote the first Raw simulator. Jason Miller wrote the software icaching system. Benjamin Greenwald ported, wrote, and maintained many of the binary utilities.

Before the Raw project, I began my research career gang scheduling on Fugu and garbage collecting on Alewife. I would like to thank Ken Mackenzie, Larry Rudolph, and David Kranz for taking the time to train a clueless graduate student during that period.

I have had the pleasure of collaborating with many others. The convergent scheduler was joint work with with Diego Puppin and Mark Stephenson, who implemented the scheduler for clustered VLIW machines. Other collaborators include Jonathan Babb, Ian Bratt, Michael Gordon, Henry Hoffman, Michael Karczmarek, Theodoros Konstantakopoulos, Jasper Lin, David Maze, Andras Moritz, James Psota, Rodric Rabbah, Jeffrey Sheldon, Ken Steele, Volker Strumpen, Bill Thies, Kevin Wilson, and Michael Zhang.

Finally, I thank my family and friends for their support over the years. I would like to express my deepest love and gratitude to my parents, Kai Fong and Alice, for being the best parents one can possibly hope for. They have always been supportive of me both emotionally and financially, and they instilled in me the importance of education and work ethics. Thanks Dad for putting Steve, Amy, and me through MIT on a professor's salary. Thanks Mom for devoting your life to your children – I can't love you enough for it. And thank you Steven Lee, Amy Lee, Cehuan Yong, Janice Lee, Sinming Cheung, Yitwah Cheung, Warren Lam, Rain Lan, Brian Lee, Alex Perlin, and Jonathan Weinstein for adding spice to my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computer architecture is driven by two forces. On the one hand, the emergence of multimedia applications have helped fuel the demand for processors with more computational power. On the other hand, exponential increase in transistor budget is projected to hold for the near future, with billion-transistors chips in view within the next five years. On the surface, these two forces seem to combine nicely to map out a simple course of action for computer architects: the market has a demand for more compute power, and the technology supplies the transistors that can be used to implement such power.

Unfortunately, modern architectures are ill suited to convert the copious silicon resources into copious compute resources. Consider the Intel processor family. It first sported a multi-ALU processor in 1993 when it incorporated superscalar technology into the Pentium. During the past decade, the transistor count of the family processor has increased 70 times, from 3.2M in the Pentium to 221M in the Itanium 2. Yet today, the widest issue processor produced by Intel, the Itanium 2, still only has six ALUs. Furthermore, as shown in Figure 1-1, those six ALUs only occupy 3% of the die area.

The fundamental problem with scaling modern architectures is that most resources in these architectures are centralized. To add more functional units to an architecture, one needs to make a corresponding increase in the support structures and resources in virtually every part

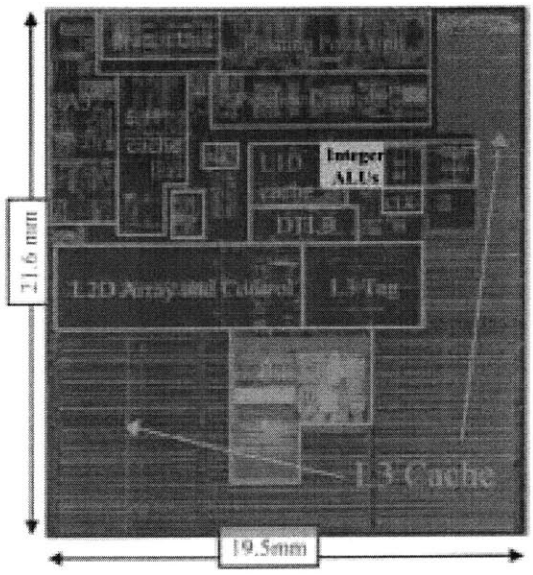


Figure 1-1. A die photo of the Itanium 2. The Itanium sports 221M transistors and a 441 mm^2 die, yet it only has 6 ALUs that occupy 3% of the die area.

of the architecture, such as the bypass network, the register file, the issue unit, or numerous hardware structures employed by superscalars to coordinate instruction level parallelism at run-time. Many of these structures scale worse than linearly in both area and speed with the number of functional units. For example, the bypass network is a crossbar between N ALUs that grows quadratically with the number of functional units. On the Itanium 2, this network services six ALUs, yet it is already consuming 50% of the cycle time [32]. For the register file, supporting an additional functional unit involves a corresponding increase in three resources: the number of read ports, the number of write ports, as well as the number of registers. Asymptotically, given N functional units, this implies that a register file scales in the order of N^3 [43]. In practice, although circuit tricks can reduce this cubic order of growth, they are not sufficient to prevent the register file from becoming a source of bottleneck in a processor's design. One commercial example is the Alpha 21264. Although it only has four ALUs, its register file has trouble meeting the timing goals of the processor [14]. Other scaling issues can be found throughout the design of a centralized processor.

Wire delay Centralized structures also lead to designs with long wires, whose latency does

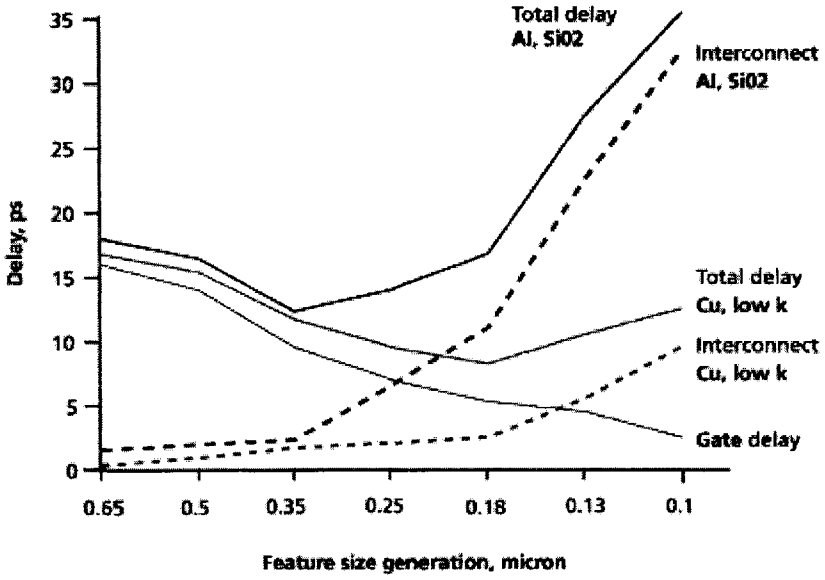


Figure 1-2. Composition of overall delay in a typical design.

not scale with technology. It wasn't long ago that wires had the abstraction of occupying zero space and propagating at instantaneous speed. This abstraction is far from reality today. Figure 1-2 shows the composition of the total delay in a typical design as technology scales, as estimated by Cadence [8]. As transistors become smaller, gate delay decreases, but wire delay decreases more slowly than gate delay and eventually even increases. For aluminum wires, overall wire delay surpasses gate delay at .18 micron process. For copper wires, the crosspoint is at .13 micron. In today's 90nm process, wire delay accounts for more than 75% of the overall delay in a design.

The increasing fraction of overall delay due to wire delay can be attributed to the pervasiveness of *global wires* in designs. As technology scales, wires in a design can be roughly divided into two types. Global wires are wires that travel over a fixed distance, while local wires are wires that travel over a fixed number of gates. It is these global wires whose latency does not scale with technology [15]. For a high frequency processor design, the implication is that as technology scales, the portion of chip area that a signal can travel in one clock cycle is getting smaller and smaller. Figure 1-3 shows the projected number of cycles it takes for a signal for travel across the chip as feature size shrinks.

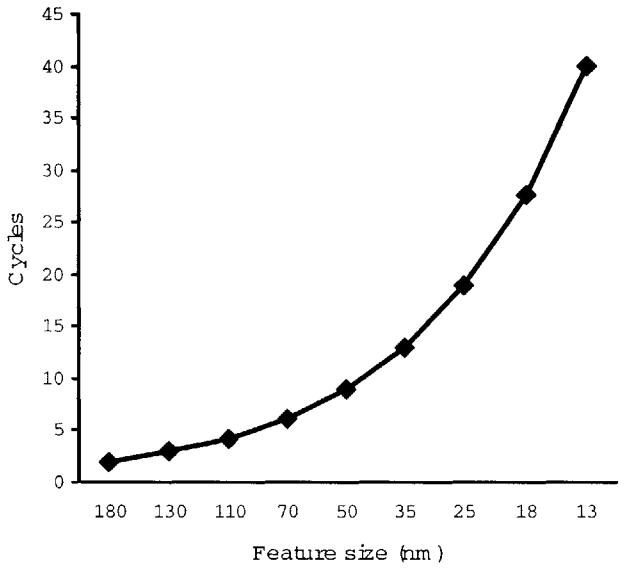


Figure 1-3. Number of cycles it takes to cross a chip as feature size shrinks.

1.1 Multimedia extensions: a short term solution

One sign that the demand for compute resources cannot be met by the ability of existing architectures to take on more ALUs is the existence of multimedia SIMD extensions such as SSE or AltiVec. In these extensions, an instruction specifies multiple instances of the same operation that are to be executed in parallel. To use an SSE instruction, the source operands feeding it must be packed in software into registers, and the destination operands must also be unpacked in software. The cost of packing and unpacking can be amortized if the output of one SSE instruction feeds into the input of another SSE instruction.

Multimedia extensions provide a modest increase in compute power to existing architectures while bypassing some of aforementioned scalability issues. By encoding the parallelism *within* a single instruction, they lessen the strain on resources such as fetch bandwidth, register ports, and registers. By explicitly specifying the parallelism in the instruction encoding, they avoid the strain on superscalar resources normally used to detect parallelism.

On the other hand, multimedia extensions are only a short term solution to adding compute resources to an architecture. They can only provide a small number of extra functional units. In addition, they can only specify SIMD parallelism, a restrictive form of parallelism.

Furthermore, they incur overhead from register packing and unpacking. As a result, multimedia extensions are difficult to compile to, and in practice are usually confined to assembly coding of libraries or intrinsic functions [37].

1.2 An evolutionary path toward scalability

In order to be able to leverage Moore's law into delivering a high frequency microprocessor that can scale to thousands of functional units, ideally one needs an architecture where there is no single centralized structure that grows worse than linearly with functional units, and where every wire is a local wire. Seven years ago, I sketched out one possible evolutionary path that leads from an existing architecture to such a scalable architecture [25]. This path consists of the following two steps:

1. Decentralize through replication and distribution.

2. Incorporate a scalable scalar operand network.

Each step is elaborated below.

Decentralize through replication and distribution A common technique to address hardware structures that scale poorly is decentralization through replication. Rather than scaling a single monolithic structure, an architect replicates smaller instances of the same structure to meet the demand for a particular resource. These smaller replicated structures can then be distributed along with the functional units that they serve. For example, the Alpha 21264 was unable to incorporate a centralized register file that met timing and provided enough register ports needed by its dual-ported cache and four functional units. Instead, as shown in Figure 1-4, the Alpha replicated its register file. Each physical register file provided half the required ports. A cluster was formed by organizing two functional units and a cache port around each register file. Communication within a cluster occurred at normal speed, while communication across clusters took an additional cycle. Another example is the

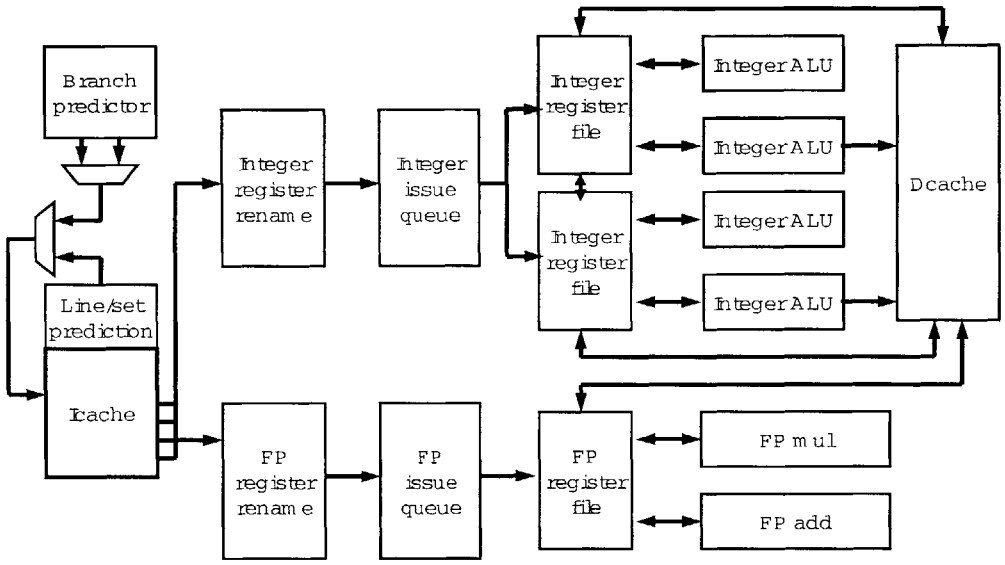


Figure 1-4. A block diagram of the Alpha 21264. To provide enough register ports while meeting timing, the integer execution core is organized into two clusters, each consisting of two ALUs, a cache port, and its own copy of the register file.

Multiflow Trace computer, a commercial clustered VLIW built in the 80s that can execute up to 28 operations per cycle [28]. To meet cycle time, it employs a separate register file for each ALU as well as each memory and branch unit.

By systemically replicating and distributing *all* microprocessor resources, an architecture can avoid the scalability issues associated with centralized structures. We use the term *tile* to describe the unit of replication.

Incorporate a scalable scalar operand network After the processor resources are organized as tiles and distributed across the chip, the next step is to make the interconnect between the tiles scalable. A *scalar operand network* (SON) is the collection of communication mechanisms used to deliver scalar values between ALUs [?]. Initially, an architecture may consist of very few tiles, so it may be able to use a crossbar or one or more busses for a SON. Both of these interconnects, however, are not scalable because they require global wires. As the number of tiles increases, a pipelined, point-to-point interconnect can be used to keep the wires local while providing the required latency and bandwidth – a progression

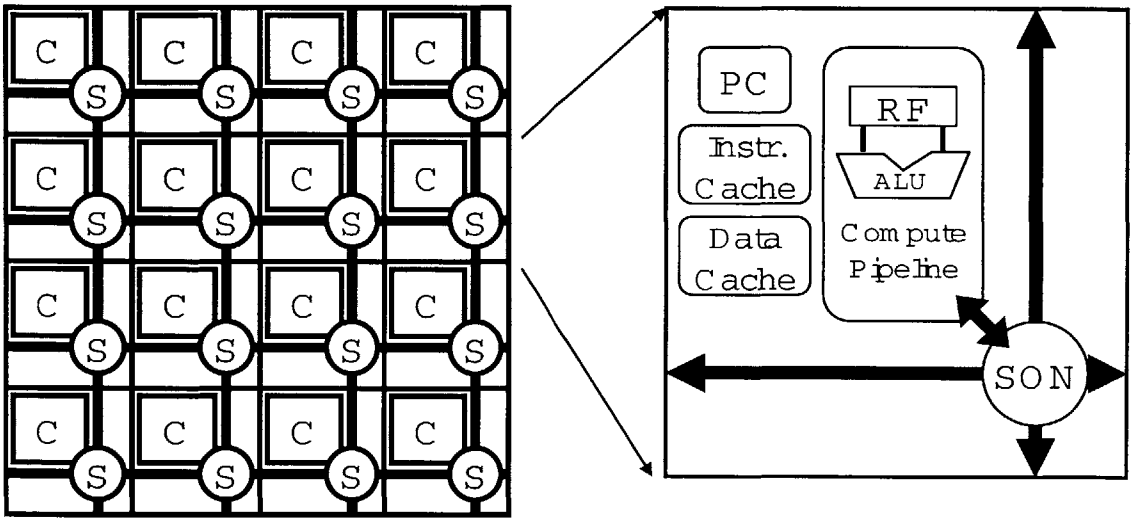


Figure 1-5. A tiled processor architecture consists of replicated tile, each with its own compute element and switch. Each tile has its own program counter, compute pipeline, registers, ALUs, data memory, and instruction memory. Tiles communicate with each other via a pipelined, point-to-point, mesh scalar operand network (SON).

reminiscent of multiprocessor evolution.

1.3 Tiled processor architectures

A tiled processor architecture (TPA) lies at the end of the above evolutionary path. Figure 1-5 shows a picture of a TPA. It is defined as follows: A TPA is an ILP processor whose resources are fully distributed across the chip in units called tiles. Tiles are connected together via a point-to-point mesh scalar operand network. The spatial distribution of resources is exposed to the compiler for locality management.

This definition includes four key concepts: ILP, tiles, scalar operand network, and spatial exposure. Let us take a look at each of them in turn:

ILP A TPA is an ILP processor. This means that all of its resources can be brought to bear on a single sequential program. Therefore, a chip multiprocessor is not a TPA because its communication mechanism is not fast enough to exploit ILP across its processors.

Tiles On a traditional architecture, adding functional units implies a major design effort to support the corresponding scaling of other supporting resources. A TPA avoids this problem by defining a replicable unit called *a tile*. Each tile consists of a set of functional units and a complete set of support resources needed to operate those functional units. This includes the processor pipeline, program counter, instruction caches, data caches, and the register file. Given a transistor budget, a TPA fills that budget by replicating as many tiles as possible. The exact composition of a tile may vary across different instances of a TPA, and may be a simple RISC core, a VLIW, or a superscalar.

Scalar Operand Network A scalar operand network (SON) is the collection of communication mechanisms used to deliver scalar values between ALUs. To support ILP between tiles, the SON of a TPA needs to satisfy the following three requirements:

Scalable It has to be able to scale to many tiles (from one to many hundreds).

Scalar It must support the delivery of *scalar* values.

Speed The end-to-end latency between functional units on neighboring tiles has to be low (one to a few cycles).

To be scalable, a SON employs a mesh, point-to-point interconnect, rather than a bus or a crossbar. To provide fast scalar transport, the interconnect is integrated into the processor pipeline, and a functional unit has direct, register-level access to the interconnect.

Spatial Exposure The tiled, mesh organization of a TPA makes it easy to limit the length of wires to no longer than the distance required to cross a tile. On a TPA with many tiles, however, this organization alone is not sufficient to prevent the latency of wires from degrading application performance. If an application is mapped onto a TPA such that tiles on the opposite corners of the chip end up communicating a lot, it will incur much wire latency and thus execute slowly.

A TPA addresses this issue by exposing the spatial distribution of resources to the compiler. This means that a compiler has some control over the where operations, operands,

Architecture	Fully Distributed	Register-level Communication	Scalable Network	Spatial Exposure
Industry Multicores	yes	no	no	yes
Clustered VLIWs	no	yes	no	yes
Partial TPAs	no	yes	yes	yes
TPAs	yes	yes	yes	yes

Table 1.1. Comparison of TPAs with other existing commercial and research architectures.

and memory data are placed. The compiler uses this interface to manage locality and thus minimizes the distance between the operands/data and the operations that need them.

1.4 Status of TPA evolution

The concept of a TPA was proposed by my colleagues and I in the Raw architecture group in 1997, in the special issue of IEEE Computer on billion transistor microprocessors [42]. Let's take a look at how that TPA vision relates to architectures today in 2005. Table 1.1 lists several classes of commercial and research architectures, along with the essential features of a TPA that the each architecture possesses. A more detailed discussion of each architecture in relation to TPA is included below.

Industry Multicores A Multicore is a single chip microprocessor that integrates multiple full processor cores onto the same die. Recently, there is a noticeable shift in focus in the microprocessor industry from traditional, centralized unicore to multicores. There are two high profile examples of this shift. First, Sun canceled the unicore Ultrasparc V and is instead touting duo cores Olympus and Niagara for its server market [18]. Second, Intel canceled Tejas, a superpipelined unicore that was supposed to replace the Pentium 4 [20]. Instead, Intel's technology roadmap is heavily focused on multicores, with upcoming duo cores products in server (Montecito, duo Itanium 2) and even the mobile (Yonah, duo Pentium M) market. Already existing in the market are duo-core processors such as the IBM Power 5, the AMD Opteron, and the Intel Smithfield.

It is interesting to note that multicores are consistent with the first step of the evolutionary

path described in Section 1.2. Their prevalence in industry suggests that the TPA evolution may be becoming a commercial reality. So far, however, the communication mechanisms between cores in existing Multicores are neither fast nor scalable yet as in a TPA. In Smithfield, for example, communication between its two cores needs to travel through two level of caches and the front side bus, with latency of tens of cycles.

Clustered VLIWs A clustered VLIW distributes its compute resources in units called clusters. Each cluster has its own set of ALUs and portion of the register file. Like a TPA, a clustered VLIW provides register-level communication between ALUs. Unlike a TPA, however, processing resources including instruction fetch, control, and memory still remain centralized, and most clustered VLIWs typically connect clusters to each other via busses or crossbars.

Clustered VLIWs exist in both commercial and research community, and it is a popular research topic. In fact, one active area of research is to make VLIW more scalable. One such attempt at decentralizing both instruction and data memories results in an architecture that looks very much like a TPA [30].

Partial TPAs We define a partial TPA to be an architecture that shares all the features of a TPA, except that only a subset of its processor resources is distributed in a tile. Two examples of partial TPAs that have been proposed in the research community are Trips [33] and WaveScalar [40]. In Trips, a tile includes the ALUs and a portion of the registers to store temporary state. However, the rest of the state, including the persistent portion of the register file, remain centralized. WaveScalar distributes almost all its resources along with its tile, including instruction memory, control flow, and register file. Only the load store queues and data memory system remain centralized.

To summarize, many of the key concepts that make a TPA scalable can be found in current industrial and research architectures. We anticipate that complexity and wire delay issues will continue to drive architectures toward truly scalable designs, eventually causing future

generations of the architectures above to evolve into full blown instances of TPAs with all their defining features.

1.5 Changing application landscape

Given the technological constraints, TPA is a logical way to build processors. It brings the exciting prospect of the ability to ride Moore's law so that every doubling of transistor count translates directly into double the issue rate and double the functional units. But architectures are only as useful as the applications find them to be. Therefore, it is important to examine how a TPA benefits applications.

The traditional approach to evaluate the performance of a general purpose microprocessor is to run SPEC benchmarks on it. SPEC consists of applications in two domains: desktop integer and desktop scientific floating point. These domains correspond reasonably well to what people used to use computers for 15 years ago. The application landscape today, however, is far more varied. Application domains such as wireless, networking, graphics, signal processing, or server are poorly represented the standard microprocessor evaluation based on SPEC. A more updated and complete reflection of the range of microprocessor applications is the Versabench benchmark suite [36]. In additional to desktop integer and floating application domains, Versabench also includes applications for server, embedded streaming, and embedded bit-level.

With a much larger pool of potential applications, there is increasing desire for an architecture to be *versatile*. Rather than being excellent in one or two applications domain but mediocre at the remaining, an architecture is better off being good across a wide spectrum of application domains.

One common trait in many applications in the newer application domains is that they have a high demand for parallelism that do not require heroic compiler efforts to detect. For example, in the server domain, there are many independent applications that can each be run completely in parallel. Another common source of parallelism come from the emergence

of applications that fall under the *streaming* model of computation. In streaming programs, computation is applied to elements of potentially infinite input streams. The long input stream gives rise to much potential for parallelism. Many audio/video encoding algorithms, such as beamforming, GSM, or MPEG are streaming in nature.

Given the increasing demand for parallelism in many application domains, one drawback that hurts the versatility of traditional microprocessor architectures is the lack of parallelism in the hardware. While these architectures contain many centralized structures that are pretty effective for boosting the performance of low-parallelism, desktop applications, the centralized structures end up taking a lot of space – an opportunity cost for the space that otherwise could have been used to add more functional units. This use of space helps low-parallelism applications, but it hurts high-parallelism ones.

For TPAs, the hardware situation is the exact opposite: TPAs have lots of parallelism in the hardware but no centralized structures. But how versatile is this type of architecture?

In the Versabench list of application domains, TPA is a clear winner for server applications. One can compile and execute a different application on each tile. With little efforts, the throughput of TPAs can scale linearly with Moore's law. But what about the single threaded applications in the other domains? If these applications are mapped to one tile, they will not speed up at all as TPA uses the increasing transistor budget to add more tiles. This performance stagnation would give a TPA a very poor versatility.

1.6 Compiling ILP to TPAs

My thesis is on using the extra tiles to speed up single threaded applications. In particular, it studies the compile-time problem of exploiting instruction level parallelism (ILP) across the tiles of a TPA. I choose ILP because it is the most general form of parallelism. Other types of parallelism such as loop level parallelism, vectorization, or pipeline parallelism can often be converted to ILP, but the converse is not true. In addition, Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is

limited by the fraction of the time the faster mode can be used. ILP is one of the few forms of parallelism that can be detected and exploited for the entire length of the program.

This thesis examines the problem of how to build an ILP compiler on a TPAs. I examine the issues related to the distribution of ALUs, registers, memory, control, and wires, and I devise solutions for each one. I will describe the solutions both in the context of a unified framework, as well as clearly distinguish which part of the framework relates to which resource. The intention is to make it easy for other TPAs or partial TPAs to identify the techniques relevant to their architectures. Special attention is paid to the distribution of ALUs, because they are distributed universally by all TPAs and because they are so important to performance.

I have written an ILP compiler for the Raw architecture, a TPA consisting of simple RISC cores as tiles, with a programmable SON whose routing decisions are performed at compile time. It is the only TPA that has actually been built – a 16-tile prototype system has been in operation since early 2003.

My thesis statement is the following: *A compiler can profitably exploit instruction level parallelism across the tiles of a TPA, with a scalable SON that has realistic latencies and machine configurations of up to 64 tiles.*

The thesis make the following contributions:

1. It describes the design and implementation of the first ILP compiler for TPAs.

2. It provides solutions for the compiler management of each distributed resource: ALUs, registers, memories, control, wires.

3. Because instruction assignment is so important, it describes two assignment techniques. One leverages research from task scheduling on MIMD machines. The other is a novel, assignment/scheduling framework that is designed to be modular and easily adaptable to architectures with special constraints.

4. It evaluates the compiler on a machine that has been implemented in silicon. The

evaluation includes results for up to 16 tiles on the actual hardware, as well as results for up to 64 tiles on a cycle-accurate simulator.

The remaining of the thesis is organized as follows. Chapter 2 overviews the problem of compiling ILP to a TPA. Chapter 3 introduces the Raw architecture, overviews our compilation problem and its solution. Chapters 4 and 5 describe Rawcc, the compiler that I implemented to exploit ILP on Raw. Chapter 4 describes on Maps, the component of Rawcc that manages memory, while Chapter 5 describes Spats, the "Spatial Assignment Temporal Scheduling" component of Rawcc that manages all distributed resources except for memory. Chapter 6 describes Convergent Scheduling, a flexible framework for performing assignment and scheduling. Chapter 7 presents evaluation results. Chapter 8 presents related work. Chapter 9 concludes.

Chapter 2

Compiling ILP on TPAs

This chapter overviews the problem of compiling ILP on TPAs. It gives a flavor of the general type of problems that needs to be solved to compile ILP for a general TPA. In Chapter 3, I will consider a specific TPA and examine in depth all the relevant ILP issues.

To ground the discussion, this chapter first examines the analogous problem of compiling ILP on a traditional, centralized ILP processor. It identifies how the compiler maps program constructs to hardware resources, and it identifies the hardware mechanisms that are used to exploit ILP. The chapter then moves on to TPA and uses this framework to overview the issues with compiling ILP on it. It concludes by explaining a simple performance metric for the scalar operand network that can be used to determine how suitable a TPA is for exploiting ILP.

2.1 ILP on centralized ILP processors

Consider how the compiler maps a program onto the hardware of a centralized ILP processor such as a superscalar or a VLIW. Figure 2-1a shows a sample C program, and Figure 2-1b shows the same program translated into three operand form, which makes the operations and the program state explicit. As shown in the figure, an input program can be divided four parts: computation, control, scalars, and data structures. Computation consists of in-

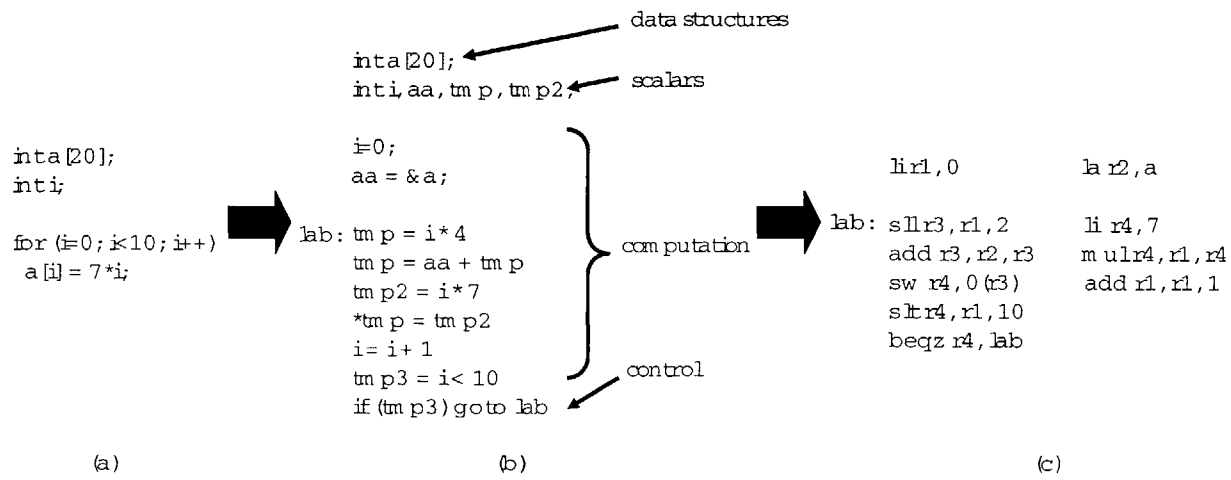


Figure 2-1. Anatomy of a sample input program. (a) is the original program. (b) is the same program rewritten so that the operations and program state are made explicit. A program consists of computation, control, scalars, and data structures. (c) shows the assembly output of the program after it compiled onto a hypothetical two-way VLIW. Each column represents an issue slot in the machine.

Program constructs	Hardware resources
Computation	Functional units
Control	Hardware branches
Scalars	Registers
Data structures	Caches/Memory

Table 2.1. A mapping from program constructs to the hardware of a centralized ILP processor.

structions that specify actual work. It includes arithmetic operations as well as accesses to memory. Control are instructions that redirects the flow of a program. Scalars are program states that communicate values between instructions. A scalar may be *explicit*, such as the program variable i, or it may be *implicit* as part of an expression, such as the result of the expression $a[i]$ which gets fed into the address of the store. Data structures are aggregate objects such as arrays, unions, or structs.

The compiler problem of mapping a program to an architecture is a resource management problem. Table 2.1 shows the correspondence between the program constructs and the hardware they are mapped to. Computation gets mapped to functional units; control gets mapped to the branch unit; scalars get mapped into the register file; and data structures are

mapped into the memory system.

Let's take a look at what the compiler has to do for each of these program construct/ hardware resources pairs:

Computation The most important compiler task is the mapping of computation to functional units. This task is the one that corresponds directly to exploiting ILP. On a centralized ILP processor, the task consists of scheduling the instructions onto the functional units. On the VLIW, the compile-time schedule is used directly at run-time. On a superscalar, however, the hardware does its own scheduling at run-time, but the compile-time scheduling is still useful because it can find parallelism across a larger window of instructions.

Because of the presence of dynamic control flow, it is not possible for the compiler to schedule the entire program as a single unit. Instead, all instruction scheduling algorithms have the concept of a scheduling region that defines the scope of scheduling. The compiler divides the control flow graph of a program into multiple scheduling regions. Each region is then scheduled independently for ILP. Though cyclic regions of the control flow graph are possible, the scope of this thesis is restricted to scheduling of acyclic regions.

Control Since there is only one program counter on a centralized machine, the management of control is very simple. A program control transfer instruction is mapped to a branch unit that manipulates the single program counter in the machine.

Scalars Scalars correspond to registers in hardware. [1] To manage this resource, the compiler performs register allocation, which decides at each point in the program what scalars are best placed in a limited number of hardware registers. For scalars that are not permanently in registers, the compiler also inserts code at the appropriate point to spill and reload the scalars from memory.

Data structures Data structures are mapped to memory in hardware. The compiler tries to optimize the placement of data for reuse and minimize cache misses.

[1] Note that only scalars whose pointers are not taken are candidates to be mapped to registers.

To summarize, mapping a program on a centralized processor consists of three tasks: ILP scheduling, register allocation, and cache optimizations. Figure 2-1c shows result of compiling our sample program onto a hypothetical 2-way VLIW. The figure includes the result of instruction scheduling and register allocation, but no cache optimization is shown.

2.2 ILP on TPAs

Having reviewed ILP compilation on a centralized architecture in the last section, this section now considers the same problem on a TPA. In comparing the two problems, we will find that the distribution of resources leads to the following new themes. First, the management of many resources now includes an additional spatial aspect. In a centralized architecture, the compiler addresses questions such as: "when should instruction i be executed?"; or "should this scalar value be should allocated to a register?" On a TPA, the compiler not only addresses the when or the whether, but it also has to address the *where*. Locality becomes a critical resource management criteria. Second, the distribution of resources on a TPA forces us to create new decentralized mechanisms to handle what was taken for granted in centralized hardware. For example, TPA does not have a global branch unit in hardware that a branch in the program can directly be mapped onto. For these new mechanisms, different TPAs may make different decisions as to how much hardware they should provide versus how much responsibility they should offload to the software. Third, communication between the ALUs is exposed through the pipelined scalar operand network and may be managed by the compiler. In contrast, in a centralized superscalar, this communication is hidden deep in the architecture within the scoreboarding mechanism, bypass network, and register file – there is no opportunity for compiler to manage this communication.

Here, I highlight two specific resources that are the focus of my research: functional units and branches; the remaining resources will be discussed in the next chapter.

Functional units Recall that on a centralized ILP processor, instruction management is a resource management problem in time. All functional units can be treated identically;

there is no reason to prefer one functional unit over another. In contrast, on a TPA, the management of functional units is a problem in both time and space. Spatially, it takes time to communicate between functional units on different tiles, and it takes more time for a functional unit to communicate with distant tiles than neighboring tiles. Thus, a haphazard assignment of instructions to tiles is likely to incur much communication and lead to poor performance. We call this collective instruction management problem in both space and time the *space-time scheduling* of instructions. The spatial component is called instruction assignment, while the temporal component is called instruction scheduling.

To perform good instruction assignment, the compiler needs to strike the proper balance between two conflicting issues. On the one hand, a tile only has a limited number of functional units. Exploiting parallelism beyond that requires instructions to be distributed across tiles. On the other, dependent instructions that have been distributed across tiles will incur the cost of communication. Given a parallel but communicating block of computation, the compiler has to analyze whether the parallelism is worthwhile to exploit across tiles in spite of the communication cost.

Branches On a centralized ILP processor, a program branch corresponds directly to a branch in hardware, which serves the role of transitioning the resources of the entire processor from exploiting ILP in one computation block to another computation block. On a TPA, however, there is no equivalent hardware to performs this transition. Instead, the only hardware branches that are available are branches local to a tile, each of which controls the local program counter and implements control flow that operates independently of the other tiles.

We call the general problem of supporting program branches in an architecture *control orchestration.* While control orchestration on a centralized ILP processor is trivial, control orchestration on a TPA requires mechanisms beyond just the independent branches on each tile, either in the form of software mechanisms or extra hardware.

2.3 Evaluating a TPA for ILP

In order to exploit ILP across tiles on a TPA, operands need to travel across the SON. Thus, a fast SON is essential to the ability of the TPA to exploit ILP. This section presents a five-tuple performance metric [?] that can be used (1) to evaluate the suitability of an SON for ILP, and (2) to compare different SON implementations.

The five-tuple captures the end-to-end cost of transporting an operand through the SON, starting immediately after the cycle the operand is produced, and ending at the cycle the operand is consumed. This five-tuple of costs <SO, SL, NHL, RL, RO> consists of the following:

Send occupancy	average number of cycles that an ALU wastes in transmitting an operand to dependent instructions at other ALUs.
Send latency	average number of cycles incurred by the message at the send side of the network without consuming ALU cycles.
Network hop latency	average transport network hop latency, in cycles, between ALUs on adjacent tiles.
Receive latency	average number of cycles between when the final input to a consuming instruction arrives and when that instruction is issued.
Receive occupancy	average number of cycles that an ALU wastes by using a remote value.

For reference, these five components typically add up to tens to hundreds of cycles on a multiprocessor [21], which makes it unsuitable for ILP. In contrast, all five components in conventional superscalar bypass networks add up to *zero* cycle. The architectural challenge of a TPA is to explore the design space of efficient scalar operand networks that also scale.

Chapter 3

Compiling ILP on Raw

This chapter overviews the compilation of ILP on Raw. It begins by introducing the Raw microprocessor. The rest of the chapter overviews Rawcc, the ILP compiler for Raw. First, it describes the compilation framework, which includes an execution model as well as descriptions of how register and data are mapped onto tiles. Next, it overviews the list of resource management problems faced Rawcc in this framework. Finally, it overviews the solution for each problem.

3.1 Raw microprocessor

The Raw microprocessor is designed and built by the Raw architecture group with two primary research objectives. The first objective is to come up with a scalable microprocessor architecture that properly addresses wire delay issues as well as other VLSI scalability issues. This objective is accomplished by obeying the set of design principles articulated in Chapter 1, which collectively define a new class of architectures known as tiled processor architectures. The second objective is to research opportunities to implement architectural mechanisms in software. In addition to ILP orchestration, our research group has explored primarily software solutions to data caching, instruction caching, as well as memory dependence speculation.

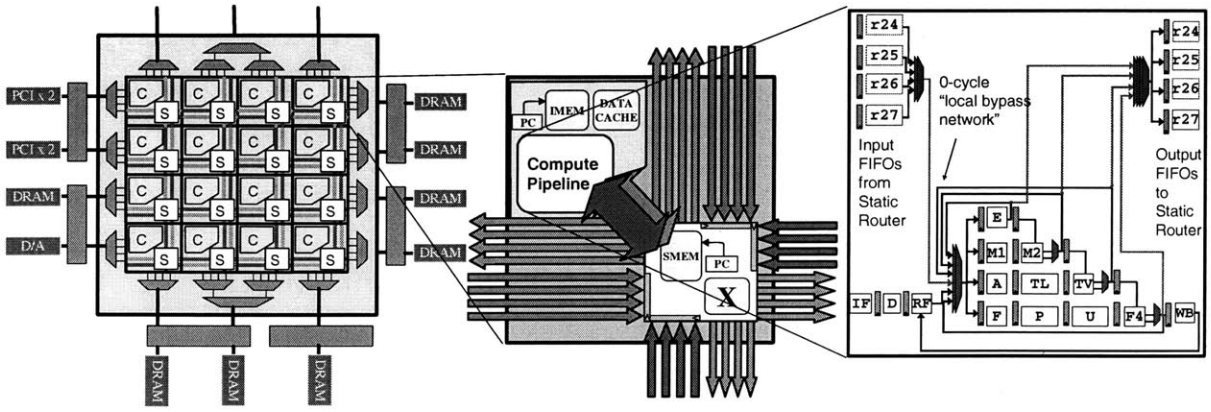


Figure 3-1. A Raw microprocessor is a fully distributed TPA. It consists of a mesh of tiles. Each tile has its on instruction memory, program counter, compute pipeline, data cache, and switch. The data caches are backed by off-chip DRAMs. The switch is programmable and contains its program counter and own instruction memory. It is register mapped and integrated directly into the processor pipeline.

The architecture Figure 3-1 shows a diagram of the Raw microprocessor [42] [41]. Raw is a tiled processor architecture. Each Raw tile has its own processing element, cache memory, and network switch. The processor element is a full, eight-stage in-order single-issue MIPS-derived pipeline, which includes its own instruction cache, fetch unit, program counter, register file, local bypass path, and data cache. The cache memories are backed by off-chip DRAMs.

A point-to-point SON is directly integrated into the processor pipeline to provide fast transport of scalar operands between tiles. It is register mapped – an instruction sends or receives a value on the SON simply by reading or writing a reserved register. The SON is programmable and directly under compiler control – this means that all routing decisions are made statically at compile time. Latency on the SON is very fast: three cycles between neighboring tiles and one extra cycle for each extra unit of manhattan distance. The 5-tuple performance metric for this SON is <0, 0, 1, 1, 0>.

The SON has blocking semantics that provides near-neighbor flow control – a processor or switch stalls if it is executing an instruction that attempts to access an empty input port or a full output port. This specification ensures correctness in the presence of timing variations

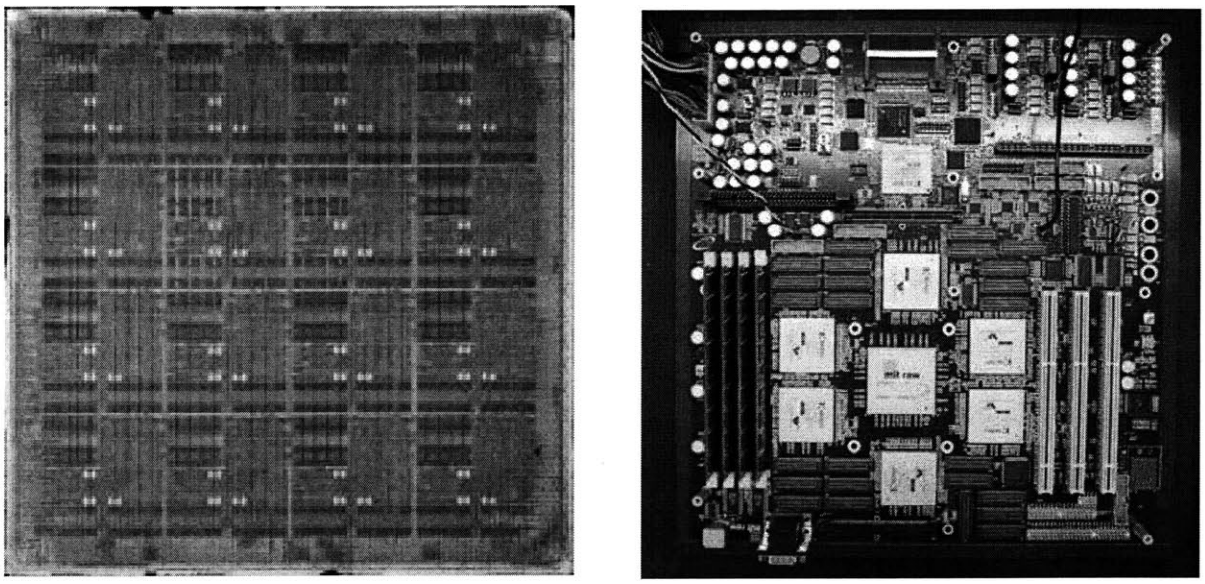


Figure 3-2. Photos of the Raw chip and Raw prototype motherboard respectively.

introduced by dynamic events such as cache misses and I/O operations, and it obviates the lock-step synchronization of program counters required by many statically scheduled machines.

In addition, Raw also provides two worm-hole dynamically-routed networks called the memory dynamic network (MDN) and the general direct network (GDN). Like the static SON, they are directly integrated into the processor pipeline to provide easy access by functional units. The MDN is primarily used for data cache transactions, while the GDN supports message passing programming model as found in traditional multiprocessors. For this research, the GDN is used as part of the compiler managed memory system.

Raw chip We have built a prototype chip and motherboard of the Raw architecture. The Raw chip is a 16-tile prototype implemented in IBM's 180 nm 1.8V 6-layer CMOS 7SF SA-27E copper process. Although the Raw array is only 16 mm x 16 mm, we used an 18.2 mm x 18.2 mm die to allow us to use the high pin-count package. The 1657 pin ceramic column grid array package (CCGA) provides us with 1080 high speed transceiver logic (HSTL) I/O pins. Our measurements indicate that the chip core averages 18.2 watts at 425MHz. We quiesce unused functional units and memories and tri-state unused data I/O pins. We

targeted a 225 MHz worst-case frequency in our design, which is competitive with other 180 nm lithography ASIC processors, like VIRAM, Imagine, and Tensilica's Xtensa series. The nominal running frequency is typically higher – the Raw chip core, running at room temperature, reaches 425MHz at 1.8V, and 500 MHz at 2.2V. This compares favorably to IBM-implemented microprocessors in the same process; the PowerPC 405GP runs at 266-400 MHz, while the follow-on PowerPC 440GP reaches 400-500 MHz.

With our collaborators at ISI-East, we have designed a prototype motherboard (shown in Figure 3-2) around the Raw chip that we use to explore a number of applications with extreme computation and I/O requirements. A larger system, consisting of four Raw chips, connected to form a virtual 64 tile Raw processor, has also being fabricated in conjunction with ISI-East and is being tested.

ILP features To exploit ILP on Raw, the compiler maps the existing parallelism of computation across the tiles and orchestrates any necessary communication between the computation using the on-chip networks. The following architectural features are key to exploiting ILP on Raw.

Effective scalar operand communication One of the more interesting architectural innovations that Raw demonstrated for ILP on TPA is its scalar operand network. Raw's combined hardware-software solution is rather elegant and quite unique. A scalar operand network must have the following three non-trivial functionalities. First, it has to efficiently match the operands coming off the network with the corresponding operations. Second, it has to be free of deadlocks. Third, it has to be able to tolerate dynamic timing variations due to cache misses. The Raw SON is able to implement these functionalities using two simple ideas. At compile-time, the software *statically orders* the processing of messages at each switch. Then at run-time, this order is enforced by the near neighbor flow control available in the network. The Raw SON also neatly integrates the receive mechanisms into the processor pipeline. This integration gives two benefits. First, it provides a fast path where an operand can come directly

off the network and be consumed directly by the operation that needs it. Second, for incoming operands that are not immediately consumed, the Raw SON can use the register file to store them. This obviates the need for extra hardware to store these operands.

In addition, the Raw SON has several features that makes it very efficient at transmitting scalar operands. It allows single-word register-level transfer without the overhead of composing and routing a message header, and without the quadratic complexity of superscalar scoreboarding. Its use of compile-time routing provides the compiler with the mechanism for the precise orchestration of communication. This means the compiler can use its full knowledge of the network status to minimize congestion and route data around hot spots.

Control decoupling Control decoupling refers to the ability of each tile to follow a flow of control that is independent of the other tiles. The feature has proven to be a key mechanism that serves many useful purposes. First, it significantly enhances the potential amount of parallelism a machine can exploit [23] by allowing the machine to follow multiple flows of control. Second, it enables *asynchronous global branching* as described in Section 5.3, a means of implementing global branching on Raw's distributed interconnect. Third it enables *control localization*, a technique introduced in Section 5.3 to allow ILP to be scheduled across branches. Finally, it gives Raw better tolerance of long latency events compared to lock-step execution machines such as a VLIW, as shown in Section 7.

A compiler interface for locality management Consistent with one of the main motivations for TPAs, Raw fully exposes its hardware to the compiler by exporting a simple cost model for communication and computation. The compiler, in turn, is responsible for the assignment of instructions, registers, and data to Raw tiles. These assignments are better performed at compile time because they critically affect both the parallelism

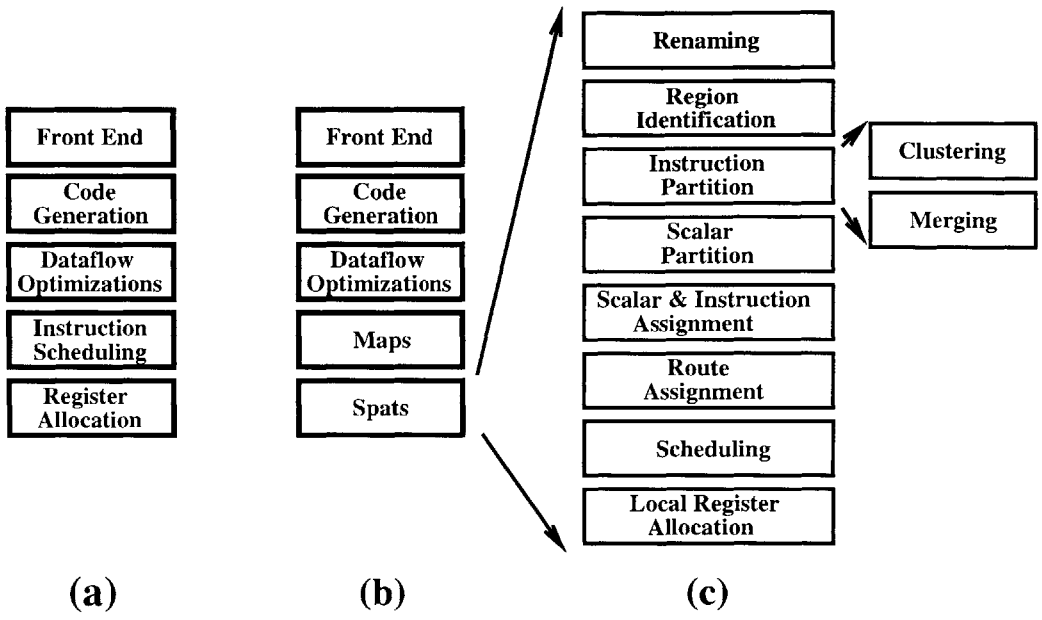


Figure 3-3. Phases of (a) an ILP compiler for a traditional centralized architecture, (b) Rawcc, and (c) Spats.

and locality of the application, and the computational complexity is greater than can be afforded at run-time.

Simple, scalable means of expanding the register space Each Raw tile contains a portion of the register space. Because the register set is distributed along with the functional units and memory ports, the number of registers and register ports scales linearly with total machine size. Each tile's individual register set, however, has only a relatively small number of registers and register ports, so the complexity of the register file will not become an impediment to increasing the clock rate. Additionally, because all physical registers are architecturally visible, the compiler can use all of them to minimize the number of register spills.

3.2 Compilation Framework

Rawcc is the ILP compiler that maps the ILP of sequential C or Fortran programs onto multiple Raw tiles. Figure 3-3 depicts the structure of Rawcc. For comparison, it also shows

the structure of an ILP compiler for a traditional, centralized architecture.

To manage the compiler complexity, Rawcc's ILP-related functionality is divided into two main components: Maps ans Spats. Maps is the component that manages the distribution of program data such as arrays and structures across the tiles. Spats, an acronym for *Spatial Assignment and Temporal Scheduling*, is the component that manages all other aspects of mapping a sequential program onto Raw.

The underlying hardware of Raw is a decentralized set of resources scattered across the chip, connected via a scalar operand network. There is no single way to map a sequential program onto this substrate. Even if the target is narrowed down to exploiting ILP, the Raw hardware still affords a compiler writer much flexibility and opportunities. This section describes the specific framework that Rawcc uses to map a sequential program onto Raw. The framework answers two questions:

1. How do the independent instruction streams on the individual processors and switches coordinate to exploit the ILP of the input program?

2. How are the program entities (computation, branches, and data) distributed across the tiles?

The compilation framework consists of three parts: an execution model, a scalar framework, and a data object (arrays and structures) framework.

3.2.1 Execution Model

Figure 3-4 depicts the execution model. The basic unit of compiler orchestration and execution is an *extended basic block*, a connected subgraph of the control flow graph with a single entry point, and whose internal edges are all forward edges. At compile time, Rawcc partitions the input program into a series of extended basic blocks. Each extended basic block is then orchestrated separately by the compiler. For each extended basic block, the compiler emits a collection of instruction sequences, one for the processor and one for

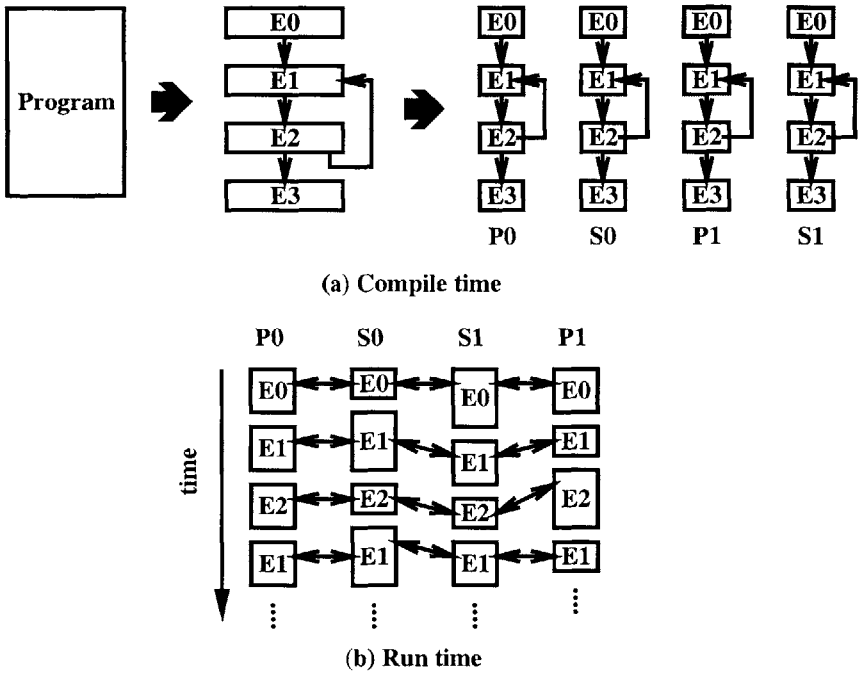


Figure 3-4. Raw execution model. At compile time, Rawcc divides the input program into extended basic blocks (E0, E1, E2, E3). For each block, it orchestrates the parallelism within it, producing corresponding code sequence that executes on each processor (P0, P1) and switch (S0, S1). For each processor and switch, the code sequences for each extended basic block are stitched back together, with control flow between blocks mirroring that of the original program.

At run-time, Raw collectively executes one extended basic block at a time, but in a loosely coordinated fashion. While Raw is performing the computation in an extended basic block in parallel, processors communicate with each other through the SON.

the switch of each tile, that encodes the computation and the necessary communication to execute that block. Each of these code sequences is assigned to a specific processor or switch at compile time. We call this collection of instruction sequences corresponding to one extended basic block an *execution unit.* For a given processor or switch, its code image for the entire program then consists of the collection of instruction sequences that have been assigned to it, with one such sequence from each execution unit.

When the Raw processor executes a program, all the tile resources – the processors, the switches, the registers, and the memories – are pooled to execute one execution unit at a time. In this execution model, there are two types of control flows, those internal to an execution

unit and those between execution units. Within a single execution unit, the processor on a tile may take branches independently of the other tiles to exploit parallelism that originates from different basic blocks, similar to how predicated execution exploits such parallelism in other architectures. After completing the work in one execution unit, tiles proceed to the next execution unit in a loosely coordinated branching mechanism called *asynchronous global branching*. Somewhere within the execution unit, the branch condition for the global branching is generated on one tile and broadcasted to the processors and switches of all the other tiles through the static network. When a processor or switch finishes its work on that execution unit and has received the branch condition, it can branch to the local target corresponding to the next execution unit without any additional synchronization. Due to this lack of explicit synchronization, it is possible for some tiles to begin executing the next execution unit before all tiles have completed the previous execution unit.

3.2.2 Scalars

Scalars are program variables with no aliasing that can be register allocated by the compiler. In the compiler back end, they correspond to virtual registers. Rawcc distinguishes between two types of scalars. *Transient scalars* are those that are only used within a single extended basic block. *Persistent scalars* are those that are live across more than one extended basic blocks.

Scalars are distributed as follows. Transient scalars are implicitly distributed as Rawcc distributes the instructions that define and use them. A transient scalar is created and resides on the tile on which the defining instruction resides, and it is directly forwarded to any tiles with instructions that use the scalar. A persistent scalar, however, requires special compiler attention. Between extended basic blocks, the compiler needs to know the which tile to look for a persistent scalar. This issue is resolved by specifying a *home tile* for each persistent scalar. In each extended basic block where a persistent scalar is defined, Rawcc forwards the value of its last definition to corresponding home tile. In each extended basic

block where a persistent scalar is used, Rawcc forwards the value stored at the home tile to any tiles that use it.

Once scalar values have been assigned to tiles, Spats can decide which scalar values to allocate to registers by applying a traditional register allocator to the code for each individual tile.

Note that because Spats performs renaming early on, many persistent scalars in the original program are each converted to multiple transient scalars. This conversion limits the adverse effects that home tiles of persistent scalars may have on locality.

3.2.3 Data objects

A data object refers to either an array, a structure, or an aliased program variable that cannot be register allocated. Maps is responsible for handling the distribution of these objects onto Raw's memory system hardware, which consists of a data cache on each tile backed by off-chip DRAMs.

Maps manages the Raw caches by implementing the following caching policy. The address space is statically partitioned across the caches, so that each address can only be cached on one tile: there is no data sharing between caches. By partitioning data this way, Maps avoids the complications and overheads associated with maintaining coherent caches. Instead, any data that is needed on multiple tiles is stored in one cache, communicated through one of the on-chip networks, and shared at the register level. The low latency of the networks, particularly that of the SON, makes this scheme practical.

Once the data is in a cache, Maps provides two ways for remote tiles to access the data, one for static references and one for dynamic references. A reference is called a *static reference* if every invocation of it can be determined at compile-time to refer to data on a specific, known tile. We call this property the *static residence property.* Such a reference is handled by placing it on the corresponding tile at compile time, so that every run-time instance of the memory reference occurs on that tile. A non-static or *dynamic reference* is a reference

that cannot be determined to refer to data on a known tile at compile-time. These references are handled by disambiguating the address at run-time in software, using a combination of the SON and the GDN to handle any necessary communication. To ensure proper ordering of dynamic memory references, Maps implements a mechanism called *turnstiles*, which can be likened to logically centralized but physically distributed load-store queues.

Maps does not guarantee that a single object ends up on a single tile. In practice, as I will explain in Section 4, Maps interleaves many arrays across tiles so that their elements can be accessed in parallel.

3.3 Problem Summary

Given the framework for compiling ILP on Raw in Section 3.2, what set of problems does the compiler need to address? In general, all architectural compilers can be considered compiler that specializes in resource management. Compared to an ILP compiler on a traditional centralized architecture, the resource management problem facing an ILP compiler for a TPA like Raw is more complex in two ways. First, there are many more resources to manage. For a superscalar or a VLIW, the compiler only needs to manage two resources: instructions (functional units) and registers. For Raw, the compiler needs to manage instructions, registers, as well as three additional resources: memories, branches, and wires (*e.g.*, the SON). Second, the tiled organization of the resources gives resource management an additional spatial aspect. For the register resource, this means register allocation consists of both a global component (assigning a value to a tile) as well as a local component (assigning the value to a specific register). For functional units, it means managing instructions is both a tile assignment and a scheduling problem.

Here is the list of resource management issues, organized by resources. Each issue is italicized for emphasis.

Instructions Instructions need to be *assigned* to tiles and *scheduled.*

Resource	Tasks		
Instructions		Tile Assignment	
Caches		Tile Assignment	Scheduling
Registers		Tile Assignment	Register Allocation
Branches (Predicate)	Selection	Tile Assignment	Scheduling
Branches (Global)	Selection		Scheduling
Wires		Route Assignment	Scheduling

Table 3.1. A summary of resource management tasks faced by Rawcc.

Caches Data objects need to be *distributed* across the caches on the tiles.

Registers Scalars need to be *assigned* to tiles and then *register allocated* on the local tiles. Specifically, tile assignment applies to persistent scalars, which needs to be assigned to a home tile.

Branches Program branches need to be converted to architectural branches. This process takes multiple steps.

First, for each program branch the compiler needs to *select* the type of branch it has to become. A program branch may either become a predicate branch or a global branch. Predicate branch then needs to be *assigned* to tiles and *scheduled*. A predicate branch may become multiple architecture branches. A global branch is by definition assigned to an architectural branch on each tile, so it only needs to be *scheduled*.

Wires Communication of remote operands need to be *routed* and *scheduled* on the wires (*e.g.*, the SON).

Table 3.1 summarizes the list of resource management problems.

3.4 Solution Overview

This section overviews the techniques Rawcc employ to manage each of the distributed resources.

3.4.1 Caches

Maps attempts to distribute objects such that it can identify as many static references as possible. Static references are attractive for the following reasons. First, because Maps uses a general dynamic network to implement dynamic references, the software overhead of dynamic accesses is high.[1] Second, static references can proceed without any dynamic disambiguation and synchronization overhead. Third, static references can better take advantage of the aggregate cache bandwidth across multiple tiles. Finally, static accesses give the compiler the opportunity to place computation close to the data that it accesses.

Maps creates and identifies static references through intelligent data mapping and code transformation. It employs different techniques for arrays and non-array objects. Arrays are distributed element-wise across the tiles. For array references that are affine functions of loop indices, Maps employs two techniques that use loop transformations to satisfy the static residence property: *modulo unrolling* and *congruence transformations*. For references to non-array objects, Maps employs *equivalence class unification (ECU)*. In ECU, objects that may be aliased through pointers are mapped onto the same tile. This approach trades off memory parallelism for the ability to identify static accesses. In practice, Maps is successful at making almost all memory references static using these two techniques.

3.4.2 Instructions

Of the resources managed by Spats, the one most critical to performance is the management of instructions. Instruction management consists of two tasks: assigning instructions to tiles and scheduling. During instruction assignment, Spats has to account for constraints imposed be *preplaced* instructions, which are static memory references that must be mapped to specific tiles.

Spats performs assignment in three steps: clustering, merging, and placement. *Clustering* groups together instructions into clusters, such that instructions within a cluster have no

[1]This software overhead is not fundamental and can largely be eliminated with some hardware support.

parallelism that is profitable to exploit given the cost of communication. *Merging* combines clusters together so that the number of clusters reduces to the number of tiles. Placement performs a bijective mapping from the merged clusters to the processing units, taking into account the topology of the interconnect. Scheduling of instructions is then performed simultaneously across all the tiles with a list scheduler.

3.4.3 Registers

Like instruction assignment, Spats divides the task of scalar home assignment into scalar partitioning and scalar placement. First, persistent scalars that tend to be accessed together are partitioned into sets. Each of these sets is then mapped onto a different tile. Spats then attempts to map scalars onto the same tile as the instructions that access them.

After scalar values have been assigned to tiles, Spats decides which scalar values to allocate to registers by applying a traditional register allocator to the code for each individual tile.

3.4.4 Branches

Spats handles branches as follows:

- For branch selection, Spats selects as many program branches to be predicated branches as possible. All forward branches in the control flow graph can become predicated branches.

- A predicated branch may have many instructions that are control dependent on it. Tile assignment of that branch is guided by the tile assignment of those instructions. Spats allows those instructions the freedom to be assigned to whichever tiles it deems best. One or more copies of the predicate branch are then made on each tile where the instructions have been mapped. During scheduling, Spats tries to amortize the cost of a predicate branch over as many instructions as possible.

- Spats translates a global branch into a broadcast of the branch condition followed by

	Phase	Resource Management Task
	Front End	-
	Maps	Cache Tile Assignment
	Code Generation	-
	Dataflow Optimizations	-
	Renaming	-
	Region Identification	Branch Selection
S	Instruction Partition	Instruction Tile Assignment Predicate Branch Tile Assignment
P	Scalar Partition	Scalar Tile Assignment
A	Scalar & Instruction Assignment	Scalar Tile Assignment Instruction Tile Assignment Predicate Branch Tile Assignment
T	Route Assignment	Wire Route Assignment
S	Scheduling	Instruction Scheduling Branch Scheduling Wire Scheduling
	Register Allocation	Register Allocation

Table 3.2. Correspondence between resource management task and compiler phases.

a local branch on each tile. After translation, individual pieces of the global branch is scheduled just like any other instructions and communication, with no special treatment.

3.4.5 Wires

A remote operand is routed on the SON by choosing the path of switches along which the operand will move. Since congestion has been observed to be low, Spat simply selects the switches in dimension ordered. To ensure that the computation is in sync with communication, the scheduling of routes on the switch is performed by the list scheduler at the same time as the scheduling of computation instructions.

3.4.6 Summary by Phases

Table 3.2 lists the correspondence between resource management task and compiler phases given in Figure 3-3. Note that some phases handle multiple tasks, while some tasks require the coordinated efforts of multiple phases.

Chapter 4

Compiler managed distributed memory

This section gives a summary of Maps, the part of Rawcc that manages the cache memory resources on Raw. A more extensive description of this system can be found in [6].

4.1 Overview

4.1.1 Motivation

Like all other resources on Raw, the on-chip caches are distributed on chip. Much of the motivation for distributing any resource on Raw applies to cache memories as well. Compared to a centralized cache found in a conventional architecture, Raw's distributed caches offer the following advantages. First, distribution makes it easy to scale the physical resources to match demand of the increasing number of functional units. Such resources include the number of cache ports as well as the cache itself. Second, the Raw system provides a means for controlling impact of on-chip wire delay on performance. On a traditional memory system, the average distance between a processing element and the SRAM cells containing the data it needs will grow in proportion to the dimensions of the cache. This increasing distance translates to increasing wire delay incurred. On Raw, however, the cache is spatially

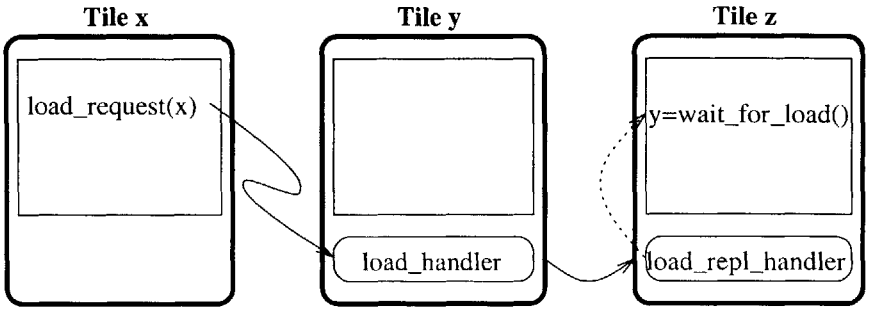


Figure 4-1. Anatomy of a dynamic load. A dynamic load is implemented with a request and a reply dynamic message. Note that the request for a load needs not be on the same tile as the use of the load.

distributed so that each tile has both a processing element and a portion of the on-chip cache. As more tiles (and thus more caches) are added to the architecture, an intelligent compiler can still map computation and the data it needs onto the same tile, thus keeping constant the communication cost incurred from wire delay.

4.1.2 Memory mechanisms

The Raw hardware provides three ways of accessing the data memory on chip: local access, remote static access, and dynamic access, in increasing order of cost. A memory reference can be a local access or a remote static access if it satisfies the *static residence property* — that is, (a) every dynamic instance of the reference must refer to memory on the same tile, and (b) the tile has to be known at compile time. The access is local if the Raw compiler places the subsequent use of the data on the same tile as its memory location; otherwise, it is a remote static access. A remote static access communicates through the static scalar operand network (SON) and works as follows. The processor on the tile with the data performs the load, and it places the load value onto the output port of its static switch. Then, the pre-compiled instruction streams of the static network route the load value through the network to the processor needing the data. Finally, the destination processor accesses its static input port to get the value.

If a memory reference fails to satisfy the static residence property, it is implemented as a

Distance	0	1	2	3	4
Dynamic store	17	20	21	22	23
Static store	1	4	5	6	7
Dynamic load	28	34	36	38	40
Static load	3	6	7	8	9

Table 4.1. Cost of memory operations.

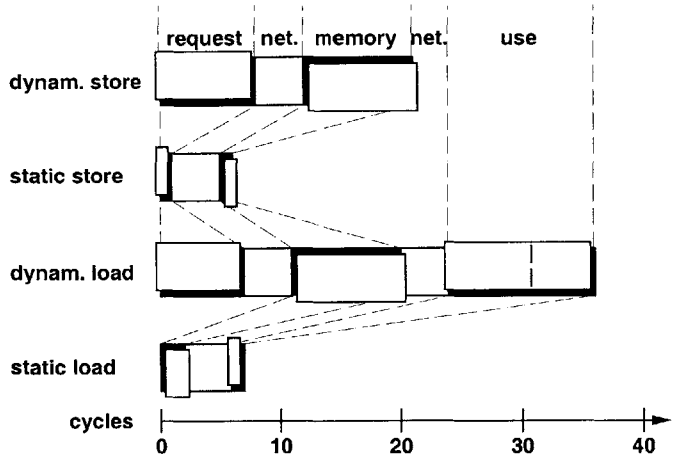


Figure 4-2. Breakdown of the cost of memory operations between tiles two units apart. Highlighted portions represent processor occupancy, while unlifted portions represents network latency.

dynamic access. A load access, for example, turns into a split-phase transaction requiring two dynamic messages: a load-request message followed by a load-reply message. Figure 4-1 shows the components of a dynamic load. The requesting tile extracts the resident tile and the local address from the global address of the dynamic load. It sends a load-request message containing the local address to the resident tile. When a resident tile receives such a message, it is interrupted, performs the load of the requested address, and sends a load-reply with the requested data. The tile needing the data eventually receives and processes the load-reply through an interrupt, which stores the received value in a predetermined register and sets a flag. When the resident tile needs the value, it checks the flag and fetches the value when the flag is set. Note that the request for a load needs not be on the same tile as the use of the load.

Table 4.1 lists the end-to-end costs of memory operations as a function of the tile distance.

The costs include both the processing costs and the network latencies. Figure 4-2 breaks down these costs for a tile distance of two. The measurements show that a dynamic memory operation is significantly more expensive than a corresponding static memory operation. Part of the overhead comes from the protocol overhead of using a general dynamic network that is not customized for this specific task, but much of the overhead is fundamental to the nature of a dynamic access. A dynamic load requires sending a load request to the proper memory tile, while a static load can optimize away such a request because the memory tile is known at compile time. The need for flow control and message atomicity to avoid deadlocks further contributes to the cost of dynamic messages. Moreover, the inherent unpredictability in the arrival order and timing of messages requires expensive reception mechanisms such as polling or interrupts. In the SON, its blocking semantics combine with the compile-time ordering and scheduling of static messages to obviate the need for expensive reception mechanisms. Finally, only static accesses provides the compiler with the ability to manage the locality of a memory reference. When the tile location of a memory access is known, the compiler can map the computation that needs the access onto the same tile. This knowledge is only available for a static access but not a dynamic access.

4.1.3 Objectives

The goal of Maps is to provide efficient use of hardware memory mechanisms while ensuring correct execution. This goal hinges on three issues, identification of static accesses, support for memory parallelism, and efficient enforcement of memory dependences. The primary goal of Maps is to identify static accesses. As shown in Table 4.1, static accesses are much faster than dynamic accesses. In addition, Maps attempts to provide memory parallelism by distributing data across tiles. Not only does it distribute different objects to different tiles, it also divides up aggregate objects such as arrays and structs and distributes them across the tiles. This distribution is important as it enables parallel accesses to different parts of the aggregate objects.

Maps actively converts memory references into static references in a process called *static promotion*. Static promotion employs two techniques: *equivalence class unification*, which promotes references through intelligent placement of data objects guided by traditional pointer analysis; and *modulo unrolling/congruence transformations*, which promotes references through loop transformations and intelligent placement of arrays. Maps provide memory parallelism by mapping data to different tiles whenever doing so does not interfere with static promotion.

For correctness, Maps must ensure that the memory accesses occurring on different tiles obey the dependences implied by the original serial program. Three types of memory dependences need to be considered: those between static accesses, those between dynamic accesses, and those between a static and a dynamic access. Dependences between static accesses are easily enforced. References mapped to different tiles are necessarily non-conflicting, so the compiler only needs to avoid reordering potentially dependent memory accesses on each tile. The real difficulty comes from dependences involving dynamic accesses, because accesses made by different tiles may potentially be aliased and require serialization. Maps uses a combination of explicit synchronization and a technique called software serial ordering to enforce these dependences. By addressing these central issues, Maps enables fast accesses in the common case, while allowing efficient and parallel accesses for both the static and dynamic mechanisms.

4.2 Static promotion

Static promotion is the act of making a reference satisfy the static residence property. Without analysis, Maps is faced with two unsatisfactory choices: map all the data to a single tile, which makes all memory accesses trivially static at a cost of no memory parallelism; or distribute all data, which enables memory parallelism but requires expensive dynamic accesses. This section describes three compiler techniques for static promotion that preserve some memory parallelism. Section 4.2.1 describes equivalence class unification, a general

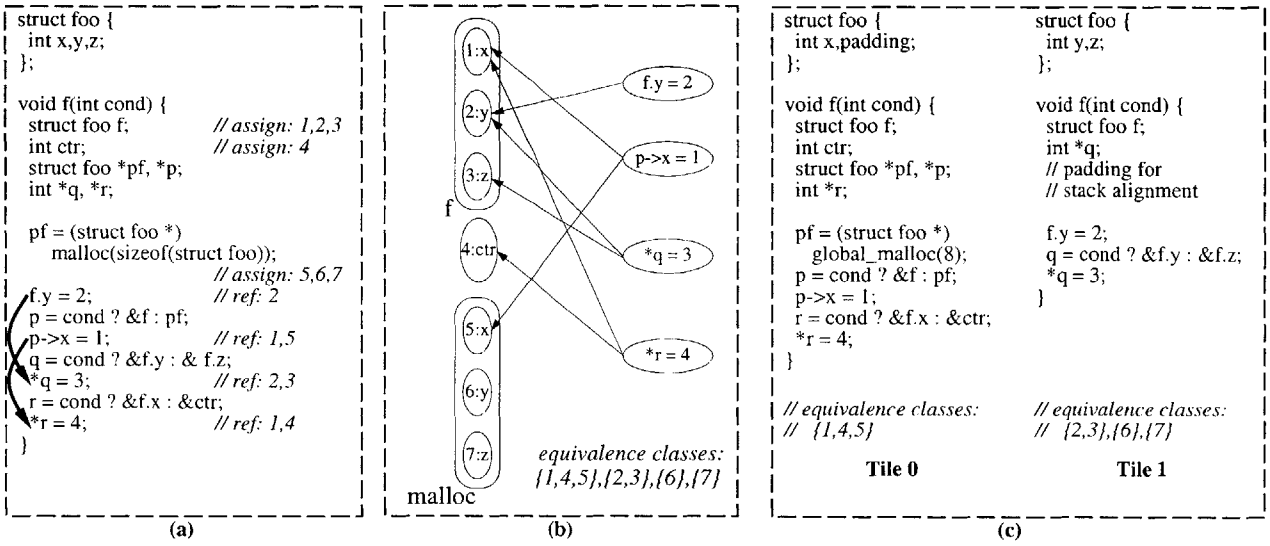


Figure 4-3. A sample program processed through pointer analysis and ECU. (a) shows the program annotated with the information provided by pointer analysis. The arrows represent memory dependences derived from pointer analysis. (b) shows its bipartite graph and its equivalence classes. (c) shows the program after it is distributed through ECU and space-time scheduling.

promotion technique based on the use of pointer analysis to guide the placement of data. Section 4.2.2 describes modulo unrolling, a code transformation technique applicable to most array references in the loops of scientific applications. Section 4.2.3 describes congruence transformations, which increases the applicability of modulo unrolling.

4.2.1 Equivalence class unification

Equivalence class unification (ECU) is a static promotion technique that uses pointer analysis to help guide the placement of data. This section first describes what information pointer analysis provides. Then it describes how ECU uses that information.

Figure 4-3(a) shows pointer analysis applied to a sample program. For each memory access, pointer analysis identifies its *points to set*, which is the list of abstract objects it may point to during the dynamic execution of the program. The point to set is conservative – some objects on the list may not be pointed to, but no objects not on the list will be pointed to. An abstract object is either a static program object, or it is a group of dynamic

objects created by the same memory allocation call in the static program. An entire array is considered a single object, but each field in a struct is considered a separate object. In the figure, each abstract object is identified by a *location set number*. To avoid clutter, location set numbers have only been listed for non-pointer objects; in actuality pointers are assigned location set numbers as well.

Maps defines the concept of *alias equivalence classes* from the program's points to sets. Alias equivalence classes form the finest partition of the point to sets such that each memory access refers to location set numbers in only one class. Maps derives the equivalence classes as follows. First, it constructs a bipartite graph. A node is constructed for each abstract object and each memory reference. Edges are constructed from each memory reference to the abstract objects corresponding to the reference's location set list. Then, Maps finds the connected components of this graph. The location set numbers in each connected component then form a single alias equivalence class. Note that references in the same alias class can potentially alias to the same object, while references in different classes can never refer to the same object. Figure 4-3(b) shows the bipartite graph and the equivalence classes in our sample program.

ECU can promote all memory references in a single alias equivalence class by placing all objects corresponding to that class on the same tile. By mapping objects for every alias equivalence class in such a manner, all memory references can be statically promoted. By mapping different alias equivalence classes to different tiles, memory parallelism can be attained.

Elements in aggregate objects such as arrays and structs are often accessed close together in the same program. Distribution and static promotion of arrays are addressed in Section 4.2.2. For structs, pointer analysis differentiates between accesses to different fields, so that fields of a struct can be in different alias equivalence classes and distributed across the tiles. Figure 4-3(c) shows how equivalence class unification is applied to our sample program.

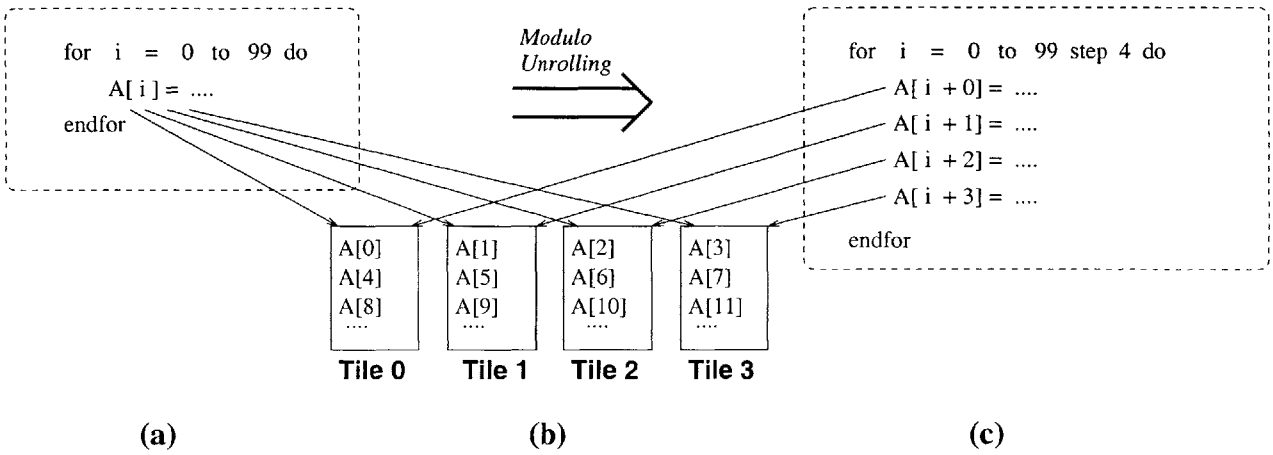


Figure 4-4. Example of modulo unrolling. (a) shows the original code; (b) shows the distribution of array A on a 4-tile Raw; (c) shows the code after unrolling. After unrolling, each access refers to memory on only one tile.

4.2.2 Modulo unrolling

The major limitation of equivalence class unification is that an array is treated as a single object belonging to a single equivalence class. Mapping an entire array to a single tile sequentializes accesses to that array and destroys the parallelism found in many loops. Therefore, Maps provides an alternate strategy to handle the static promotion of array accesses. In this approach, arrays are laid out in memory through *low-order interleaving*, which means that consecutive elements of an array are interleaved in a round-robin manner across the caches on the Raw tiles. Maps then applies transformations that statically promote array accesses in loops.

Modulo unrolling is a framework for determining the unroll factor needed to statically promote all array references inside a loop. This technique is illustrated in Figure 4-4. Consider the source code in Figure 4-4(a). Using low-order interleaving, the data layout for array A on a four-tile Raw is shown in Figure 4-4(b). The A[i] access in Figure 4-4(a) refers to memories on all four tiles. Thus it is not a static access.

Intelligent unrolling, however, can enable static promotion. Figure 4-4(c) shows the result of unrolling the code in Figure 4-4(a) by a factor of four. Each access in the resultant loop

then refers to elements on only one tile – the act of unrolling has statically promoted these accesses.

It can be shown that this technique is always applicable for loops with array accesses having indices which are affine functions of enclosing loop induction variables. These accesses are often found in dense matrix applications and multimedia applications. For a detailed explanation and the symbolic derivation of the unrolling factor, see [7].

4.2.3 Congruence transformations

Maps also employs congruence transformations to statically promote array accesses. The congruence of an access is characterized by a stride a and an offset b [24]. An access is said to have stride a and offset b if all the addresses it targets has the form $an + b$, where $n \geq 0$. For arrays that have been low-order interleaved, the congruence of an access can be used to determine whether it is a static access. An access to a low-order interleaved array with element size e is static if and only if its stride a and offset b satisfy the following:

$$a \ mod \ (N * e) = 0 \tag{4.1}$$

$$b \ mod \ e = 0 \tag{4.2}$$

For example, given an array of four-byte elements distributed across 4 tiles, an access to it with congruence $16n + b$ is static and resides on tile $b/4 \ mod \ 4$, provided that b is a multiple of 4.

Congruence transformations are code transformations that increase the stride of accesses. Maps employs a suite of such transformations developed by Larsen [24]. The transformations aim to increase the run-time count of the number of accesses whose strides satisfy equation 3.1.

The primary transformation deals with loops with array accesses whose initial congruence relations are unknown, as is possible with pointers into arrays. Modulo unrolling requires

that the initial congruence relations of accesses be known. For this type of loops, a pre-loop is inserted before the main loop. Each iteration of the pre-loop executes one iteration of the original loop, and it keeps executing until each array access in the loop satisfies a predetermined stride and offset. This way, when execution enters the main loop, the congruence relations of the accesses become known, and modulo unrolling can be applied.

The exit condition is selected by the compiler using profiling data. When a loop has multiple array accesses, a careful selection of the exit condition is needed to ensure that the exit condition is satisfiable.

4.3 Dynamic accesses

This section describes Maps's support for dynamic accesses. First, it explains the limitations of static promotion and motivates the need for a dynamic fall-back mechanism. Then, it describes how Maps enforce dependences involving dynamic accesses, using static synchronization and *software serial ordering*.

4.3.1 Uses for dynamic references

A compiler can statically promote all accesses through equivalence-class unification alone, and modulo unrolling/congruence transformations help improve memory parallelism during promotion. There are several reasons, however, why it may be undesirable to promote all references. First, modulo unrolling sometimes requires unrolling of more than one dimension of multi-dimensional loops. This unrolling can lead to excessive code expansion. To reduce the unrolling requirement, some accesses in these loops can be made dynamic. In addition, static promotion may sometimes be performed at the expense of memory parallelism. For example, indirect array accesses of the form $A[B[i]]$ cannot be promoted unless the array $A[]$ is placed entirely on a single tile. This placement, however, yields no memory parallelism for $A[]$. Instead, Maps can choose to forgo static promotion and distribute the array. Indirect accesses to these arrays would be implemented dynamically, which yields better parallelism

at the cost of higher access latency. Moreover, dynamic accesses can improve performance by not destroying static memory parallelism in critical parts of the program. Without it, arrays with mostly affine accesses but a few irregular accesses would have to be mapped to one tile, thus losing all potential memory parallelism to the arrays. Finally, dynamic accesses can increase the resolution of equivalence class unification. A few isolated "bad references" may cause pointer analysis to yield very few equivalence classes. By selectively removing these references from promotion consideration, more equivalence classes can be discovered, enabling better data distribution and improving memory parallelism. The misbehaving references can then be implemented as dynamic accesses.

For these reasons, it is important to have a good fall-back mechanism for dynamic references. More importantly, such mechanism must integrate well with the static mechanism. The next section explains how these goals are accommodated.

For a given memory access, the choice of whether to use a static or a dynamic access is not always obvious. Because of the significantly lower overhead of static accesses, the current Maps system makes most accesses static by default, with one exception. Arrays with any affine accesses are always distributed, and two types of accesses to those arrays are implemented as dynamic accesses: non-affine accesses, and affine accesses that require excessive unroll factors for static promotion.

4.3.2 Enforcing dynamic dependences

Maps handles dependences involving dynamic accesses with two separate mechanisms, one for the type of dependences between a static access and a dynamic access, and one for the type of dependences between two dynamic accesses. A static-dynamic dependence can be enforced through explicit synchronization between the static reference and either the initiation or the completion of the dynamic reference. When a dynamic store is followed by a dependent static load, this synchronization requires an extra dynamic store acknowledgment message at the completion of the store. Because the source and destination tiles of the synchronization

message are known at compile-time, the message can be routed on the static network.

Enforcing dependences between dynamic references is a little more difficult. To illustrate this difficulty, consider the dependence which orders a dynamic store before a potentially conflicting dynamic load. Because of the dependence, it would not be correct to issue their requests in parallel from different tiles. Furthermore, it would not suffice to synchronize the issues of the requests on different tiles. This is because there are no timing guarantees on the dynamic network: even if the memory operations are issued in correct order, they may still be delivered in incorrect order. One obvious solution is complete serialization as shown in Figure 4-5(a), where the later memory reference cannot initiate until the earlier reference is known to complete. This solution, however, is expensive because it serializes the slow round-trip latencies of the dynamic requests, and it requires store completions to be acknowledged with a dynamic message.

Software serial ordering (SSO) is a technique that efficiently enforces such dependences. Figure 4-5(b) illustrates this technique. Software serial ordering leverages the in-order delivery of messages on the dynamic network between any source-destination pair of tiles. It works as follows. Each equivalence class is assigned a *turnstile* node. The role of the turnstile is to serialize the request portions of the memory references in the corresponding equivalence class. Once memory references go through the turnstile in the right order, correct behavior is ensured from three facts. First, requests destined for different tiles must necessarily refer to different memory locations, so there is no memory dependence which needs to be enforced. Second, requests destined for the same tile are delivered in order by the dynamic network, as required by the network's in-order delivery guarantee. Finally, a memory tile handles requests in the order they are delivered.

Note that in order to guarantee correct ordering of processing of memory requests, serialization is inevitable. SSO keeps this serialization low, and it allows the exploitation of parallelism available in address computations, latency of memory requests and replies, and processing time of memory requests to different tiles. For efficiency, SSO employs the

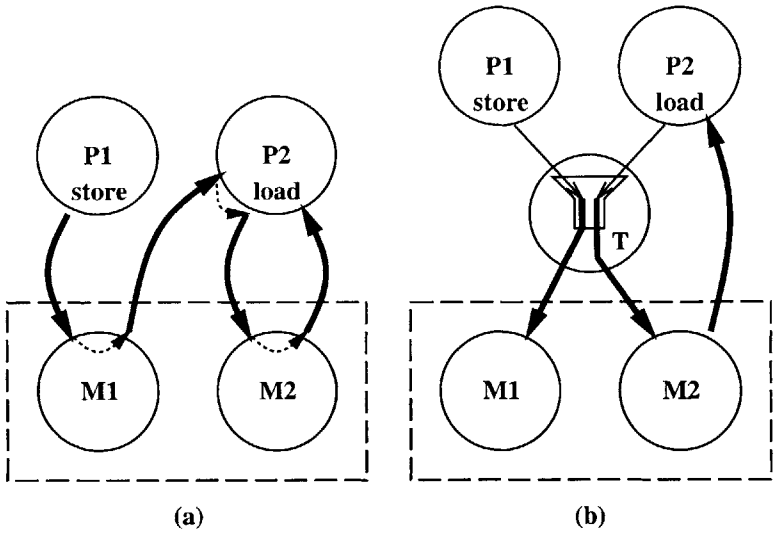


Figure 4-5. Two methods for enforcing dependences between dynamic accesses. P1 and P2 are processing nodes initiating two potentially conflicting dynamic requests; both diagrams illustrate an instance when the two requests don't conflict. M1 and M2 are the destinations of the memory requests. The light arrows are static messages, the dark arrows are dynamic messages, and the dashed arrows indicate serialization. The dependence to be enforced is that the store on P1 must precede the load on P2. In (a), dependence is enforced through complete serialization. In (b), dependence is enforced through software serial ordering. T is the turnstile node. The only serialization point is the launches of the dynamic memory requests at T. Note that Raw tiles are not specialized; any tile can serve in any or all of the following roles, as processing node, memory node, or turnstile node.

static network to handle synchronization and data transfer whenever possible. Furthermore, different equivalence classes can employ different turnstiles and issue requests in parallel. Interestingly, though the system enforces dependences correctly while allowing potentially dependent dynamic accesses to be processed in parallel, it does not employ a single explicit check of run-time addresses.

Chapter 5

Space-time scheduling

This Chapter describes Spats, the part of Rawcc that manages the instruction, register, branch, and wire resources of Raw. Spats is more easily explained by considering the control related functionality separately. We term the collective management of instructions, registers, and wires *ILP orchestration*, whose details are given in Section 5.1. Section 5.2 proofs the static ordering property, a key property that allows Rawcc to generate a valid program with statically ordered communication, even in the presence of dynamic events such as cache misses. Section 5.3 then explains control orchestration. Section 5.4 reviews the design decisions.

5.1 ILP orchestration

Figure 5-1 repeats the compiler flow of Spats, which was also shown in Figure 3-3. Spats begins by performing SSA renaming on a procedure body. Renaming exposes available parallelism by removing anti and output dependences. Then, Spats partitions a procedure body into scheduling regions, which becomes the unit of compilation within which the compiler orchestrates parallelism. The next four phases focus on spatial aspect of the orchestration, as it assigns instructions, scalars, communication routes to tiles. After assignment, Spats performs a single coordinated scheduling pass that schedules both computation on each of the

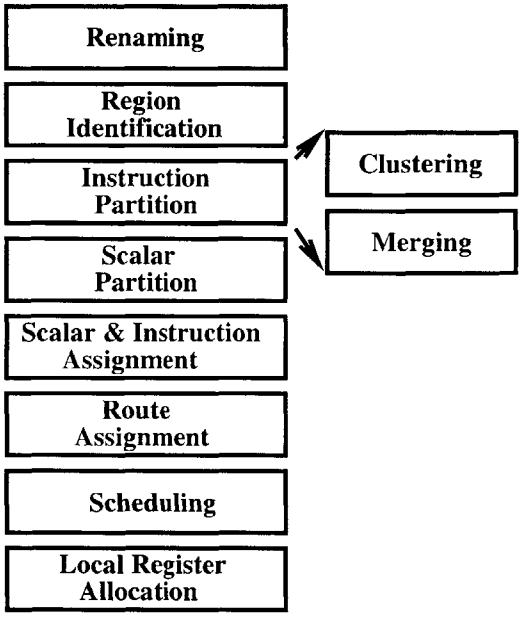


Figure 5-1. Phases of Spats.

tiles as well as communication on the switches. After scheduling, Spats finishes by applying a traditional register allocator to the code on each tile.

The components of Spats operate at one of two levels. Renaming, region identification, and the scalar partitioning/assignment phases operate on an entire procedure at a time. The rest of the phases, which deal primarily with management of instructions, operate on a region at a time. Logically, each of these phases iterate through all the regions in a procedure before proceeding to the next phase.

Each of the phases is described in more details below. To facilitate the explanation, Figure 5-3 shows the transformations performed by Spats on a sample piece of code.

Renaming Spats begins by converting the procedure representation to single static assignment (SSA) form [10]. SSA form was developed as intermediate format for dataflow analysis; it is used here because it is a good representation of parallelism. Figure 5-2 shows a sample program being converted to SSA form. In SSA form, every variable is defined exactly once. A program is converted to SSA form by renaming multiple-defined variables so that each definition gets a unique variable name. The renaming removes anti and output dependences

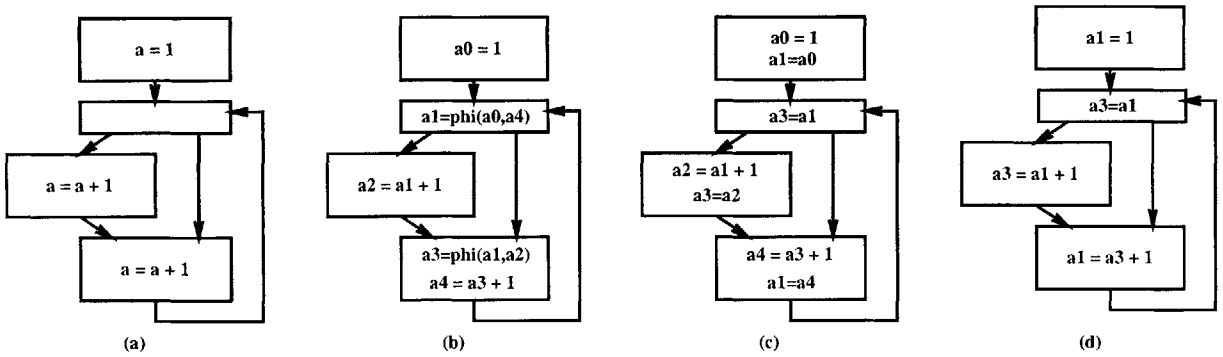


Figure 5-2. An example of SSA renaming. (a) shows the initial code in its control flow graph. (b) shows the renamed SSA form. (c) shows the result after phi nodes have been converted to copy statements. (d) shows the final code after the copy statements have been coalesced.

and exposes all the natural ILP that is contained in the program.

At join points in the control flow graph, a variable in pre-SSA form may have multiple reaching definitions. SSA form represents this situation by the use of phi nodes. A phi node is a like a select operator, where the output variable takes on the value of one of the input variables. Here, the output variable is a fresh renamed variable, while each input variable corresponds to a reaching definition at that program point.

Phi nodes are not constructs that can be executed in hardware. Spats actually converts them into copy statements. A phi node is spit into multiple copy statements, one per input variable. Each copy statement is then placed below the corresponding reaching definition. When possible, a copy statement is merged with its reaching definition.

Region identification The region identification phase partitions each procedure into distinct scheduling regions. Each scheduling region is a single-entry, single-exit portion of the control flow graph containing only forward control flow edges, such as those generated by if-then-else constructs. Within each scheduling region, if-conversion [29] is applied to convert the code to predicated form. Most subsequent phases of Spats are then applied to each scheduling region separately.

To prepare for the communication of persistent scalars between regions, two types of dummy instructions are inserted. *Read* instructions are inserted at the beginning of the

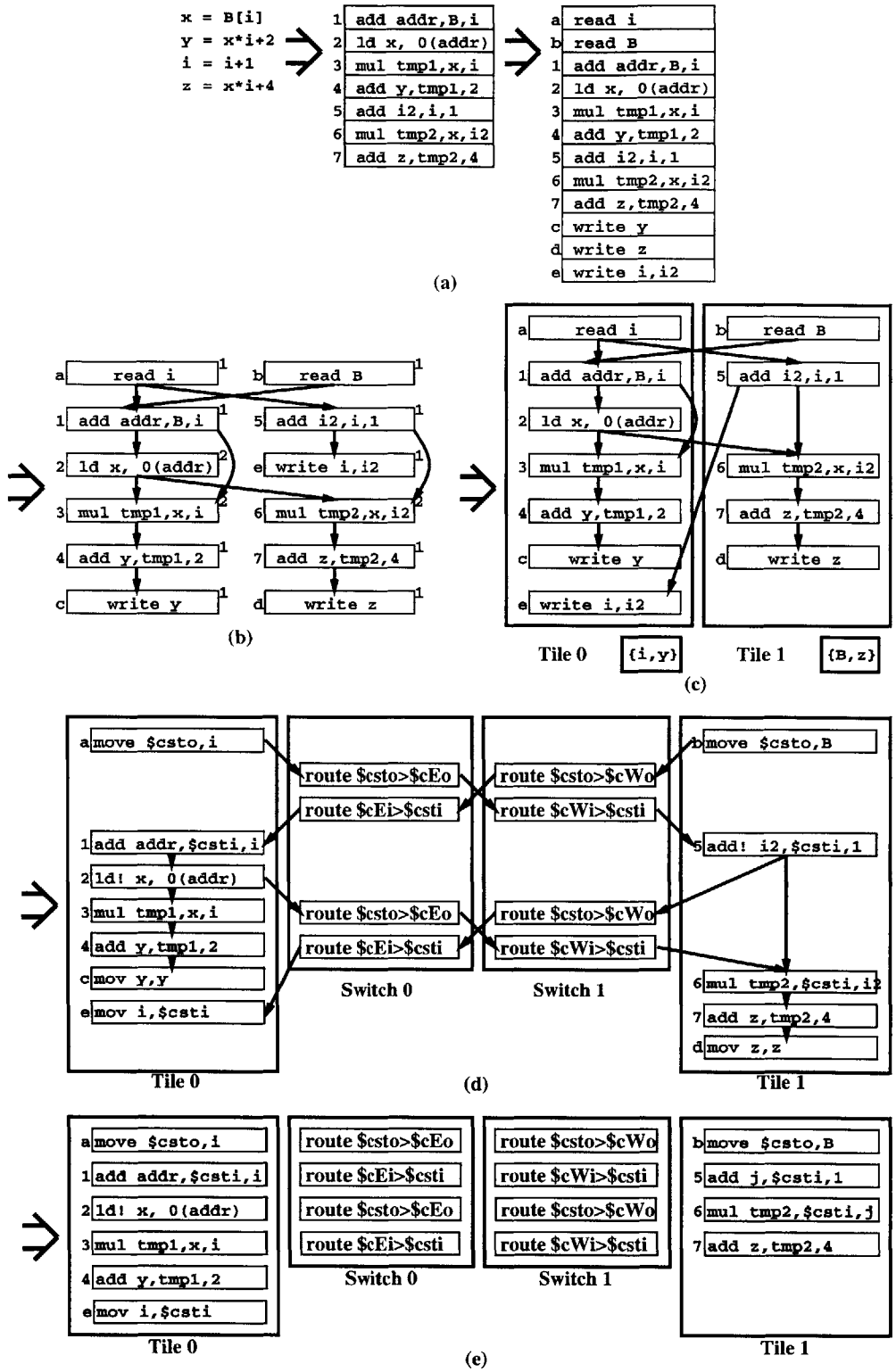


Figure 5-3. An example of the program transformations performed by Spats on a single region. (a) shows the initial program undergoing transformations made by *renaming* and *region identification*; (b) shows the data dependence graph corresponding to the final code sequence in (a); (c) shows the result after *instruction partition*, *scalar partition*, and *scalar and instruction assignment*, (d) shows the result of *route assignment*; (e) shows the result after *scheduling*.

code sequence for any persistent scalars that are accessed. *Write* instructions are inserted at the end of the code sequence for any persistent scalars that are written. These instructions simplify the eventual representation of *stitch code*, the communication needed to transfer values between scheduling regions. This representation in turn allows the event scheduler to overlap the stitch code with other work in the scheduling region. Dummy instructions that are not involved in inter-tile communication are eliminated later in the compiler.

Figure 5-3a shows the transformations performed by renaming. Figure 5-3b shows the same output code sequence in (a), but in the form of a data dependence graph. In the graph, a node represents an instruction, and an edge represents a true data dependence between two instructions. Each node is labeled with the execution time of the instruction. We will continue to use this graph representation for the rest of the example.

Region identification also helps manage the control resource on Raw by performing control selection. See Section 5.3 for more details.

Instruction partition The instruction partitioner partitions the original list of instructions into multiple lists of instructions, one for each tile. It does not bind the resultant instruction lists to specific tiles — that function is performed by the instruction placer. The partitioner attempts to balance the benefits of parallelism against the overheads of communication.

Certain instructions have constraints on how they can be partitioned and where they can be placed. Read and write instructions to the same scalar variable must be mapped to the tile on which the data resides (see *scalar partition* and *scalar and instruction assignment* below). Similarly, preplaced instructions (static loads and stores) must be mapped to specific tiles. The instruction partitioner performs its duty without considering these constraints. They are taken into account during the scalar partition and the scalar and instruction placement phases.

Partitioning consists of two sub-phases, clustering and merging. Each is described in turn below.

Clustering Clustering attempts to partition instructions to minimize run-time, assuming

non-zero communication cost but infinite processing resources. The cost of communication is modeled to be three cycles, the latency of adjacent tile communication on Raw. The sub-phase groups together instructions that either have no parallelism, or whose parallelism is too fine-grained to be exploited given the communication cost. Subsequent phases guarantee that instructions in the same cluster will be mapped to the same tile.

Clustering is a two step process. In the first step, Spats employs a greedy technique based on the estimation of completion time called Dominant Sequent Clustering (DSC) [45]. Initially, each instruction node belongs to a unit cluster. Communication between clusters is assigned a uniform cost. The algorithm visits instruction nodes in topological order. At each step, it selects from the list of candidates the instruction on the longest execution path. It then checks whether the selected instruction can merge into the cluster of any of its parent instructions to reduce the estimated completion time of the program. Estimation of the completion time is dynamically updated to take into account the clustering decisions already made, and it reflects the cost of both computation and communication. The algorithm completes when all nodes have been visited exactly once.

In the second step, Spats post-processes the output clusters from DSC to ensure that at most one preplaced instruction is in any one cluster. Each DSC cluster with multiple preplaced instruction is split further into multiple clusters, with one preplaced instruction in each resultant cluster. During the splitting, each node is assigned to the new cluster containing the preplaced instruction that is closest to it.

Merging Merging combines clusters to reduce the number of clusters down to the number of tiles. Spats uses a temporal-aware load balancing technique. Each tile is represented by a partition holding the clusters assigned to it. Initially, this partition is empty. As clusters are assigned to partition, the algorithm dynamically keeps of the amount of computation assigned to the partition at each depth of the data dependence graph. The depth of a node in the data dependence graph is defined to be length of the longest path from any root of the graph to that node.

Merging visits each cluster in decreasing order of size. As it visits each cluster, it selects a partition to assign to the cluster. Partition assignment is performed as follows. If a cluster contains a preplaced instruction, it is assigned to the partition corresponding to the tile of the preplaced instruction. Otherwise, the algorithm selects the partition with the lightest load during the range of depth spanned by the cluster. Note that this load metric ensures that the clusters distributed across the tiles can likely be executed in parallel. It is superior to a simple load balancing criteria without temporal-awareness, because a temporal-unaware load balancing technique may end up distributing dependent clusters that cannot be executed in parallel.

Here are more details on the algorithm. The input to merging phase is the following:

```
T = number of tiles

N = set of nodes the dependence graph

E = set of edges in dependence graph

C = set of existing unmerged clusters
```

The output is a mapping from unmerged clusters to new merged clusters.

Nodes, edges, and clusters have predefined attributes. Figure 5-4 shows those attributes and the functions to access the attributes. It is worth pointing out the how the occupancy of a node and latency of an edge relate to the latency of executing an instruction. Suppose a node represents an instruction that has a latency of 4, and the functional unit it executes on is not pipelined. This node has an occupancy of 4, and its outgoing true dependence edges have latencies of 0. If the functional unit is pipelined, however, the node occupancy becomes 1, and the outgoing true dependence edges have latencies of $4 - 1 = 3$.

Figure 5-5 shows the main function of the merging algorithm, which returns a mapping from original clusters to T merged clusters. First, the function initializes some load information for each cluster. Then it initializes T empty merged clusters. Using the load information, it iterates through the clusters and merge each cluster into the most suitable

succ: Node $n \rightarrow$ EdgeSet
return $\{ e \mid e \in E;\ \mathsf{head}(e) = n \}$

occupancy: Node $n \rightarrow$ int
return number of cycles a node takes to execute

clusterof: Node $n \rightarrow$ Cluster
return cluster n is mapped to

tile: Node $n \rightarrow$ int
return tile requirement of n. -1 means no requirement.

head: Edge $e \rightarrow$ Node
return head node of e

tail: Edge $e \rightarrow$ Node
return tail node of e

latency: Edge $e \rightarrow$ Int
return latency of e

tile: Cluster $c \rightarrow$ Int
return tile requirement of c. -1 means no requirement.

Figure 5-4. Node, edge, and cluster attributes.

```
DoMerge: int T × NodeSet N × EdgeSet E × ClusterSet C → ClusterMap
    InitClusterLoad(N, E)
    ClusterSet MC ← CreateEmptyClusters(T)
    ClusterMap M ← nullmap
    foreach Cluster c ∈ { c | c ∈ C; tile(c) ≠ −1 }
      Cluster mc ← mc ∈ MC | tile(mc) = tile(c)
      Update(mc, c)
      M(c) ← mc
    foreach Cluster c ∈ { c | c ∈ C; tile(c) = −1 } sorted in decreasing size(c)
      Cluster mc ← LeastLoadedCluster(MC, timeBegin(c), timeEnd(c))
      Update(mc, c)
      M(c) ← mc
    return M
```

Figure 5-5. Main function of merging algorithm

merged cluster. There are two criteria for merging. Clusters with the same tile requirements are merged into the same merged cluster. Otherwise, a cluster is merged into the merged cluster that has the least load during the range of depth when the cluster has work.

InitClusterLoad() initializes a set of functions that summarize the loads of the original clusters. Figure 5-6 shows the set of load functions, as well as InitClusterLoad() itself. The functions use depth of a node to approximate time. The computation of depth is given in Figure 5-7.

Figure 5-8 shows the remaining helper functions for the merging algorithm, which creates, selects, and updates the merged clusters.

Scalar partition The purpose of the scalar partitioner is to group persistent scalars into sets, each of which is to be mapped to the same home tile. To minimize communication, scalars that tend to be accessed by the same merged cluster of instructions should be grouped together.

The scalar partitioner operates across all the scheduling regions of a procedure body after instruction partitioning. It tries to find a partition of scalars and instruction clusters across virtual tiles that minimizes the amount of cross-tile communication. The algorithm is as

```
load: Cluster c × int t → int
    return load of c at time t

timeBegin: Cluster c → int
    return min { t | load(c, t) ≠ 0 }

timeEnd: Cluster c → int
    return max { t | load(c, t) ≠ 0 }

computeLoad: Cluster c × int t1 × int t2 → int
    return Σ_{t1≤t≤t2} load(c)

InitClusterLoad: NodeSet N × EdgeSet E
    NodeIntMap depth ← computeDepth(N, E)
    foreach Node n ∈ N
      Cluster c ← clusterof(n)
      foreach int t ∈ [depth(n), depth(n) + occupancy(n) − 1]
        load(c, t) += 1
      timeBegin(c) = min(timeBegin(c), depth(n))
      timeEnd(c) = max(timeEnd(c), depth(n) + occupancy(n))
```

Figure 5-6. Load initialization functions for merging algorithm.

```
computeDepth: NodeSet N × EdgeSet E → NodeIntMap
    NodeList S ← topoOrder(N, E)
    NodeIntMap depth ← nullmap
    while S ≠ ∅
      Node n ← pop(S)
      foreach Edge e ∈ succ(n)
        Node n' ← tail(e)
        depth(n') ← max(depth(n'), depth(n) + occupancy(n) + latency(e))
    return depth

topoOrder: NodeSet N × EdgeSet E → NodeList
    NodeIntMap p ← nullmap
    foreach Node n ∈ N
      p(n) ← | { edge e ∈ E | tail(e) = n } |
    NodeSet R ← { node n ∈ N | p(n) = 0 }
    NodeList L ← {}
    while R ≠ ∅
      Node n ← pop(R)
      L ← L + n
      foreach Edge e ∈ succ(n)
        Node n' ← tail(e)
        p(n') ← p(n') − 1
        if p(n') = 0 then R ← R ∪ {n'}
    return L
```

Figure 5-7. Depth computation.

CreateEmptyClusters: int $T \rightarrow$ ClusterSet
    ClusterSet C $\leftarrow \emptyset$
    foreach int i $\in [0, T-1]$
        C $\leftarrow$ C $\cup$ { Cluster(Tile=i) }
    return C

LeastLoadedCluster: ClusterSet C $\times$ int $t1$ $\times$ int $t2 \rightarrow$ Cluster
    Cluster c $\leftarrow c \in C \mid \forall$ Cluster $c' \in C, \mathsf{computeLoad}(c, t1, t2) \leq \mathsf{computeLoad}(c', t1, t2)$
    return c

Update: Cluster mc $\times$ Cluster c
    foreach int t $\in [TimeBegin(c), TimeEnd(c) - 1]$
        $\mathsf{load}(mc, t)$ += $\mathsf{load}(c, t)$
    $\mathsf{timeBegin}(mc) = \mathsf{min}(\mathsf{timeBegin}(mc), \mathsf{timeBegin}(c))$
    $\mathsf{timeEnd}(mc) = \mathsf{max}(\mathsf{timeEnd}(mc), \mathsf{timeEnd}(c))$

Figure 5-8. Helper functions related to merged clusters for merging algorithm.

follows. Initially, both scalars and instruction clusters are arbitrarily assigned to these virtual tiles, with the constrain that in each scheduling region, exactly one instruction cluster is mapped to each virtual tile. In addition, static accesses to memory are also partitioned across virtual tiles in accordance with their known tile locations. The algorithm then incrementally improves on the mapping to reduce the number of remote data accesses between instruction clusters and data. First, it remaps instruction clusters to virtual tiles given a fixed mapping of the scalars. Then, it remaps scalars to virtualized tiles given the fixed mappings of instruction clusters. The remapping is performed by making randomly swaps that reduce overall communication. Static memory accesses are kept fixed and not remapped. This remapping repeats until no incremental improvement in locality can be found. The final mapping of scalars to virtual tiles form the desired partition.

Scalar and instruction assignment The scalar and instruction assignment phase assigns sets of persistent scalars and instruction clusters to physical tiles. Figure 5-3c shows a sample output of this phase. The assignment phase removes the assumption of the idealized

interconnect and takes into account the non-uniform network latency. Assignment of each scalar set is currently driven by the preference of static memory accesses: if a scalar set ends up on the same virtual tile as a static memory access, that set is mapped to the same tile as the memory access. In addition to mapping scalar sets to tiles, the scalar assignment phase also locks the dummy read and write instructions to the home tiles of the corresponding persistent scalars.

Scalar assignment is performed before instruction assignment to allow cost estimation during instruction assignment to account for the location of those scalars. For instruction assignment, Spats uses a swap-based greedy algorithm to minimize the communication bandwidth. It initially assigns clusters to arbitrary tiles, and it looks for pairs of mappings that can be swapped to reduce the total number of communication hops.

Route assignment The route assignment phase translates each non-local edge (an edge whose source and destination nodes are mapped to different tiles) in the dependence graph into a set of communicating instructions that route the necessary data value from the source tile to the destination tile. Figure 5-3d shows an example of such transformation. Communication instructions include *send* and *receive* operations on the processors as well as *route* instructions on the switches. New nodes are inserted into the graph to represent the communication instructions, and the edges of the source and destination nodes are updated to reflect the new dependence relations arising from insertion of the communication nodes. These dependence relations are later enforced during scheduling to ensure a correct ordering of communication events. Note that the Raw ISA allows *send* and *receive* operations to be merged with existing ALU operations, as shown by instructions 2 and 6 in Figure 5-3d (the ! after the ld is an implicit send). To minimize the volume of communication, edges with the same source are serviced jointly by a single *multicast* operation, though this optimization is not illustrated in the example.

Currently, route assignment is performed via dimension-ordered routing. Network con-

tention has generally been low, so there has been no need to use a more intelligent method.

Scheduling The scheduler schedules the computation and communication events within a scheduling region with the goal of producing the minimal estimated run-time. Because routing on Raw is itself specified with explicit switch instructions, all events to be scheduled are instructions. Therefore, the scheduling problem is a generalization of the traditional instruction scheduling problem.

The job of scheduling communication instructions carries with it the responsibility of ensuring the absence of *deadlocks* in the network. If individual communication instructions are scheduled separately, Spats would need to explicitly manage the buffering resources on each communication port to ensure the absence of deadlock. Instead, Spats avoids the need for such management by treating a single-source, multiple-destination communication path as a single scheduling unit. When a communication path is scheduled, contiguous time slots are reserved for instructions in the path so that the path incurs no delay in the static schedule. By reserving the appropriate time slot at the node of each communication instruction, Spats automatically reserves the corresponding channel resources needed to ensure that the instruction can eventually make progress.

Though scheduling is a static problem, the schedule generated must remain deadlock-free and correct even in the presence of dynamic events such as cache misses. The Raw system uses the *static ordering property*, implemented through near-neighbor flow control, to ensure this behavior. The static ordering property states that if a schedule does not deadlock, then any schedule with the same order of communication events will not deadlock. A proof of this property is presented in Section5.2. Because dynamic events like cache misses only add extra latency but do not change the order of communication events, they do not affect the correctness of the schedule.

The static ordering property also allows the schedule to be stored as compact instruction streams. Timing information needs not be preserved in the instruction stream to ensure

correctness, thus obviating the need to insert no-op instructions. Figure 5-3e shows a sample output of the event scheduler. Note that the dummy read and write instructions only turn into explicit instructions if they are involved in communication. Also, on the switches, the route instructions that correspond to different paths are freely interleaved.

The scheduling algorithm is as follows. Before scheduling, Spats applies reverse if-conversion to the code on each tile to convert the predicated code back to branch code (See Section 5.3). Then, Spats uses a single greedy list scheduler to schedule both computation and communication instructions. The algorithm keeps track of a ready list of events. An event is either a computation instruction or a communication path. As long as the list is not empty, it selects and schedules the task on the ready list with the highest priority. The priority scheme is based on the following observation. The priority of a task should be directly proportional to the impact it has on the completion time of the program. This impact, in turn, is lower-bounded by two properties of the task: its *level*, defined to be its critical path length to an exit node; and its *average fertility*, defined to be the number of descendant nodes divided by the number of tiles. Therefore, we define the priority of a task to be a weighted sum of these two properties.

Local register allocation The final phase of Spats is register allocation. Spats treats this problem as multiple independent instantiations of the analogous problem on a traditional RISC machine. It simply applies a traditional, graph-coloring based register allocator to the code of each tile.

5.2 The Static ordering property

Dynamic events such as cache misses prevent one from statically analyzing the precise timing of a schedule. Rawcc relies on the static ordering property of the Raw architecture to generate correct code in the presence these dynamic events. The static ordering property states that the result produced by a static schedule is independent of the specific timing of

the execution. Moreover, it states that whether a schedule deadlocks is a timing independent property as well. Either the schedule always deadlocks, or it never does.

To generate a correct instruction schedule, Rawcc orders the instructions in a way that obeys the instruction dependencies of the program. In addition, it ensures that the schedule is deadlock free assuming one set of instruction timings. Static ordering property then ensures that the schedule is deadlock free and correct for any execution of the schedule.

This section provides an informal proof of the static ordering property. The section restricts the static ordering property to the practical case: given a schedule that is deadlock free for one set of instruction timings, then for any set of instruction timings,

1. it is deadlock free.

2. it generates the same results.

First, we show (1). A deadlock occurs when at least one instruction stream on either the processor or the switch has unexecuted instructions, but no instruction stream can make progress. A non-empty instruction stream, in turn, can fail to make progress if it is attempting to execute a blocked communication instruction. A communication instruction blocks when either its input port is empty, or its output port is full. Computation instructions do not use communication ports; they cannot cause deadlocks and are only relevant in this discussion for the timing information they represent.

Consider the generic scenario in Figure 5-9. In the figure, each large long rectangle represents an execution node (which are all switches in the figure, but a node may also be a processor), and each wide rectangle represents a communication instruction. There are three switches: S1, S2, and S3. Port A connects Switch S1 to Switch S2, while Port B connects Switch S2 to Switch S3. The focus of this setup is Instruction I_y, which represents a generic communication instruction that executes on Switch S2 and sends a value from Port A to Port B.

Let's derive the conditions under which Instruction I_y can execute. The necessary con-

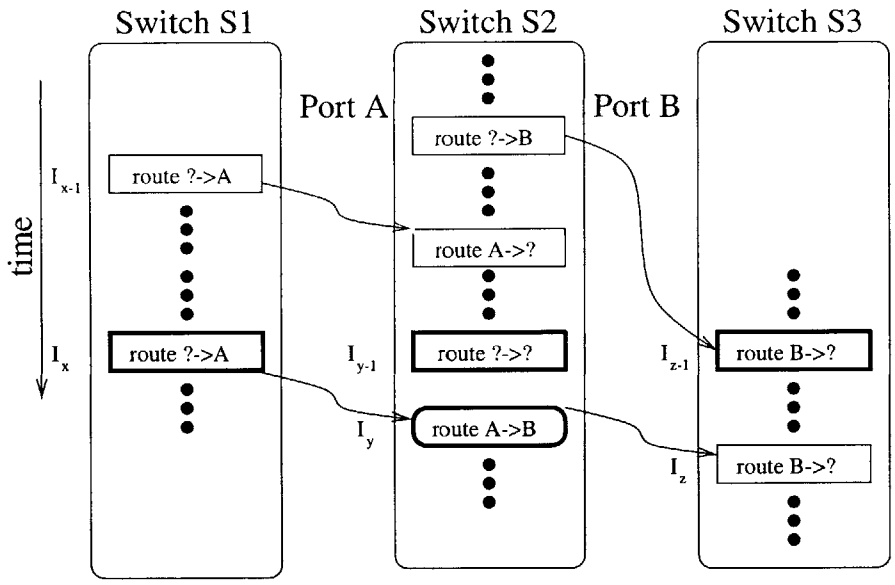


Figure 5-9. Dependent instructions of a communication instruction. Each long rectangle represents an execution node, and each wide rectangle represents a communication instruction. Spaces between execution nodes are ports. Edges represent flow of data. A communication instruction is of the form *Route* $X -> Y$, where X is the source port and Y is the destination port. Since we only care about Port A and Port B in this example, other ports are labeled with "?". The focal instruction is the thick rounded rectangle labeled I_y. Its dependent instructions are in thick regular rectangles.

ditions for its ability to execute are the following: its input value must have been sent, its switch S2 must be ready to execute it, and the destination of its value (Port B) must be available.[1] These conditions can also be represented by execution of a set of instructions. Note that ports are dedicated connections between two fixed nodes, so that each port has exactly one reader node and one writer node. Let I_y be the x^{th} instruction that reads from port A, the y^{th} instruction that executes on its node N, and the z^{th} instruction that writes to port B. Then before I_y can execute, the following instructions must have executed:

1. the x^{th} instruction that writes port A.
2. the $y-1^{th}$ instruction that executes on switch S2.
3. the $z-1^{th}$ instruction that reads (and *flushes*) port B.

Next, we argue that these conditions are also sufficient for I_y to execute. The key observation is that once a resource becomes available for instruction I_y, it will *forever* remain available until the instruction has executed. The value on the input port cannot disappear; the execution node cannot skip over I_y to run other instructions; the output port cannot be full after the previous value has been flushed. The reservation of the resources is based on three properties: the single-reader/single-writer property of the ports, the blocking semantics of the communication instructions, and the in-order execution of instructions.

Therefore, a communication instruction can execute whenever its dependent instructions, defined by the enumeration above, have executed.

Now, consider the schedule that is deadlock-free for one known set of timings. Plot the execution trace for this set of timings in a two dimensional grid, with node-id on the x-axis and time on the y-axis. Each slot in the execution trace contains the instruction (if any) that is executed for the specified node at the specified time. The plot is similar to Figure 5-9, except that real execution times, rather than the static schedule orders, are used to place the instructions temporally.

[1] In general, the three resources need not all be applicable. A send by a processor only requires an output port and the execution node, while a receive by a processor only requires the input value and the node.

Finally, consider a different set of timings for the same schedule. Let t_{new} be a point in time for the new timings when the schedule has not been completed, and let $E_{new}(t_{new})$ be the set of instructions that have executed before time t_{new}. We use the above deadlock-free execution trace to find a runnable instruction at time t_{new}. Find the smallest time t in the deadlock-free execution trace that contains an instruction not in $E_{new}(t_{new})$. Call the instruction c. The dependent instructions of c must necessarily be contained in $E_{new}(t_{new})$.[2] Therefore, c must be be runnable at time t_{new} for the new set of timings.

Having found a runnable instruction for any point in time when the schedule is not completed, the schedule must always make progress, and it will not deadlock.

The second correctness condition, that a deadlock-free schedule generates the same results under two different sets of timings, is relatively easy to demonstrate. Changes in timings do not affect the order in which instructions are executed on the same node, nor do they change the order in which values are injected or consumed at individual ports. The blocking semantics of communication instructions ensures that no instruction dependence can be violated due to a timing skew between the sender and the receiver. Therefore, the values produced by two different timings must be the same.

5.3 Control orchestration

Spats provides a two-tiered support for branches in a program: global branches and local branches. A global branch is a branch that requires a global transfer of control across all the tiles. A local branch is a branch that is executed independently on only one or a few tiles. Spats's employs this two-tiered support in order to provide both generality and performance. Branches between scheduling regions require the generality of global coordination, but branches within a scheduling region only need to be executed independently on one or

[2]This statement derives from two facts:

1. All dependent instructions of c must execute before c in the deadlock-free execution trace.
2. Since c executes at time t and all instructions executed before time t are in $E_{new}(t_{new})$, all instructions executed before c in the deadlock-free execution trace are in $E_{new}(t_{new})$.

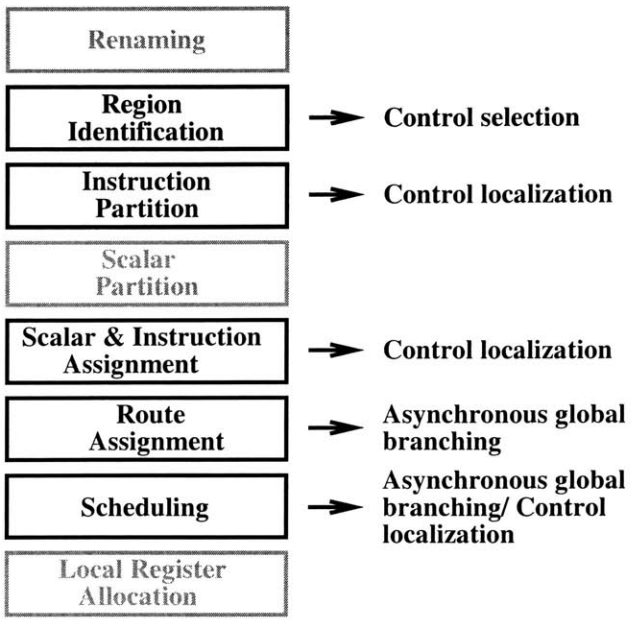


Figure 5-10. A mapping of Spats phases to the control functions they help implement.

a few tiles. Furthermore, local branches allow Spats to operate on scheduling regions that encompass multiple basic blocks, which in turn increases the scope within which Spats can schedule parallelism.

This section describes control orchestration in Spats, which consists of three parts. *Control selection* decides whether each branch in the program is a local branch or a global branch. *Asynchronous global branching* is the software mechanism for global branches. *Control localization* is the collection of software techniques to support efficient execution of local branches.

Figure 5-10 shows the phases of Spats involved in supporting control orchestration.

Control selection Control selection decides whether each branch in the program is a local branch or a global branch. It is the indirect byproduct of region identification. During region identification, a procedure is divided into multiple scheduling regions. This process also divides branches into two types: internal branches and external branches. An internal branch is a forward branch whose target is in the same scheduling region. These branches are temporarily replaced by predicated code, and they eventually become local branches.

An external branch is either a backward branch, or a forward branch whose target is in a different scheduling region. This type of branches becomes global branches.

Asynchronous global branching Spats implements global branching asynchronously in software by using the SON and local branches. First, the branch value is broadcast to all the tiles through the SON. This communication is exported and scheduled explicitly by the compiler just like any other communication. Therefore, it can overlap with other computation in the basic block. Then, each tile and switch individually performs a branch without synchronization at the end of the execution of a scheduling region. Correct execution is ensured despite the introduction of this asynchrony because of the static ordering property. As shown in Figure 5-10, the synthesis and scheduling of the branch's communication are performed by the route assignment and scheduling phases of Spats.

The overhead of asynchronous global branching is explicit in the broadcasting of the branch condition. This overhead contrasts with the implicit overhead of global wiring incurred by global branching in VLIWs and superscalars. Making the overhead explicit has the following advantages. First, the compiler can hide the overhead by overlapping it with useful work. Second, this branching model does not require dedicated wires used only for branching. Third, the approach allows the global coordination to be performed without any global wires, which is consistent with the decentralized design methodology of a TPA and helps enable a faster clock.

Control localization Control localization is the application of a collection of software techniques to support efficient execution of local branches. The current implemented set of techniques are designed to achieve two goals. The first goal is flexibility during instruction assignment. During instruction assignment, Spats uses a predicated code representation and treats each instruction as an individual unit, so that instructions with the same control dependence can freely be mapped onto different tiles. This representation has two advantages. First, it allows the compiler to exploit the parallelism available among predicated

instructions. Second, it naturally supports multiple loads/stores that have the same control dependence but are static memory references on different tiles. The actual assignment of branches to tiles mirrors that of instruction management, and it is supported by the same partitioning and assignment phases as shown in Figure 5-10.

The second goal of control localization is to minimize the number of local branches that have to execute once individual predicated instructions have been assigned to tiles. Spats achieves this goal by converting post-assignment predicated code on each tile back to branch code, in a process called reverse if-conversion [3]. Then during scheduling, code control dependent on the same branch is treated as a single unit. This conversion reduces the number of local branches in two ways. First, instructions that share the same predicate can amortize the cost of executing the local branch. Second, instructions with complementary predicates can also share the same local branch.

5.4 Design decisions

This section discusses the design decisions made in two key areas of Spats: instruction management and control localization. They are two areas that we have observed empirically to have a high impact on performance.

Instruction management Spats decomposes instruction management into multiple phases and sub-phases: clustering, merging, placement, and scheduling. There are two reasons for this decomposition strategy. First, given a machine with a non-uniform network, empirical results have shown that separating assignment from scheduling yields superior performance [44]. Furthermore, given a graph with fine-grained parallelism, having a clustering phase has been shown to improve performance [12].

In addition, the instruction management problem, as well as each of its sub-problems, is NP complete [38]. Decomposing the problem into a set of greedy heuristics enables us to develop an algorithm that is computationally tractable. The success of this approach, of course,

depends heavily on carefully choosing the problem decomposition. The decomposition should be such that decisions made by an earlier phase should not inhibit subsequent phases from making good decisions.

We believe that separating the clustering and scheduling phases was a good decomposition decision. The benefits of dividing merging and placement have been less clear. Combining them so that merging is sensitive to the resource topology may be preferable, especially if the target TPA has preplaced memory instructions like Raw does.

Spats integrates its additional responsibilities relatively seamlessly into this basic framework. By inserting dummy instructions to represent home tiles, inter-region communication can be represented the same way as intra-region communication. Communication for global branching is also represented similarly. The need for explicit communication is identified through edges between instructions mapped to different tiles, and communication code generation is performed by replacing these edges with a chain of communication instructions. The resultant graph is then presented to a vanilla greedy list scheduler, modified to treat each communication path as a single scheduling unit. This list scheduler is then able to generate a correct and greedily optimized schedule for both computation and communication.

Control localization Our approach to control localization went through three iterations. Initially, we try to group all instructions under the same local branch into a single atomic block of instructions [25]. This block, called a *macroinstruction*, then becomes an atomic unit on which the compiler performs assignment and scheduling. Thus, the entire macroinstruction gets assigned to a single tile, and its instructions get scheduled in contiguous time slots. When there are multiple preplaced instructions under the same branch, we break the instructions into the minimum number of macroinstructions needed to satisfy the preplacement constraints. This scheme minimizes the number of local branches, but the disadvantage is that the parallelism available under local branches cannot be exploited across tiles.

To address the parallelism issue, we next use a fully predicated representation of the

program. Each predicated instruction is then independently assigned and scheduled. During scheduling, we augment the priority function with a factor that accounts for the cost of extra branches. For example, when the previously scheduled instruction had a predicate p, each instruction in the ready list with the same predicate p gets a bump in its priority.

After experimenting with this approach, we find that while the flexibility provided by the predicate form is useful during assignment, it is rarely useful during scheduling. Note that during scheduling, no parallelism can be gained by spitting up two instructions with the same predicate. The cost of the local branches on Raw is sufficiently high that it the scheduler almost always does better by having as few of them as possible. Thus, for simplicity we settle on a control localization strategy where reverse if-conversion is applied to the predicate code on each tile before scheduling begins.

Chapter 6

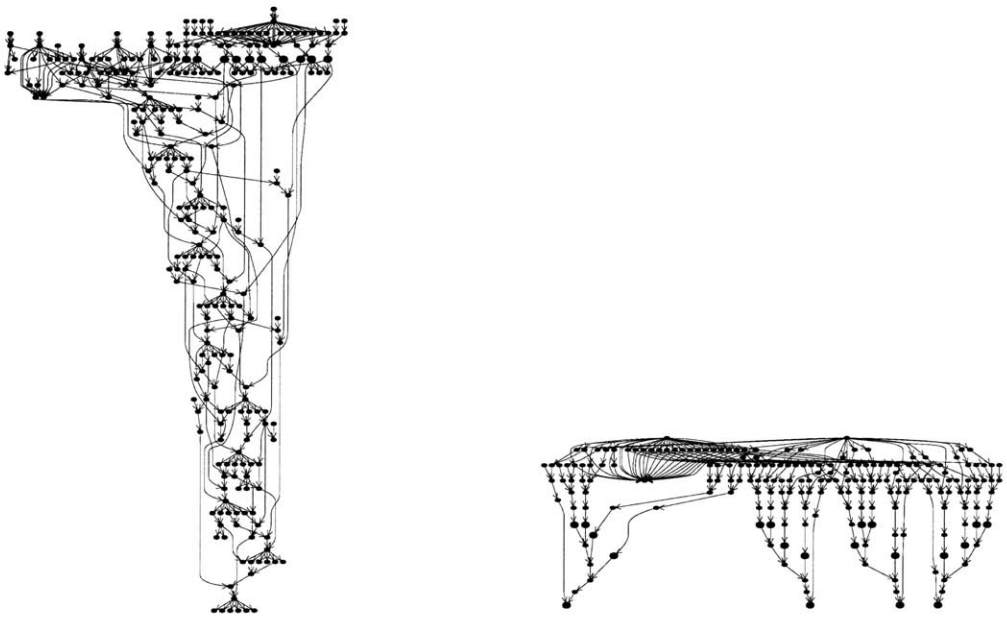
Convergent scheduling

This chapter presents *convergent scheduling*, a general framework for resource management that makes it easy to specify arbitrary constraints and scheduling heuristics.

6.1 Introduction

Convergent scheduling is motivated by two early observations from instruction assignment by Spats on Raw. First, different assignment heuristics work well for different types of graphs. Figure 6-1 depicts representative data dependence graphs from two ends of a spectrum. In the graphs, nodes represent instructions and edges represent data dependences between instructions. Graph (a) is typical of graphs seen in non-numeric programs, while graph (b) is representative of graphs coming from applying loop unrolling to numeric programs. Consider the problem of mapping these graphs onto a TPA. Long, narrow graphs are dominated by a few critical paths. For these graphs, critical-path based heuristics are likely to work well. Fat, parallel graphs have coarse grained parallelism available and many critical paths. For these graphs it is more important to minimize communication and exploit the coarse-grain parallelism. To perform well for arbitrary graphs, the assignment algorithm may require multiple heuristics in its arsenal.

Second, preplaced instructions have a significant impact on assignment, especially for



(a) (b)

Figure 6-1. Different data dependence graphs have different characteristics. Some are thin and dominated by a few critical paths (a), while others are fat and parallel (b).

dense matrix codes. In Spats, instructions are assigned are partitioned into threads and then placed onto tiles. Though the placement process accounts for the preference of the preplaced instructions, the decisions made by earlier by the instruction partitioner limits the flexibility of that latter phase.

In general, it is typical for resource allocation problems to be faced with many conflicting constraints. In addition the issues above, proper assignment needs to understand the tradeoff between parallelism and locality. To preserve good locality, related computation should be mapped to the same tile. On the other hand, to exploit parallelism within computation, the computation needs to be distributed across tiles.

Traditional resource allocation frameworks handle conflicting constraints and heuristics in an ad hoc manner. One approach is to direct all efforts toward the most serious problem. For example, on a hypothetical clustered VLIW with many clusters connected by a single-cycle bus, a compiler would focus on keeping as many of the clusters busy and not be as concerned about locality. Another approach is to address the constraints one at a time in a sequence

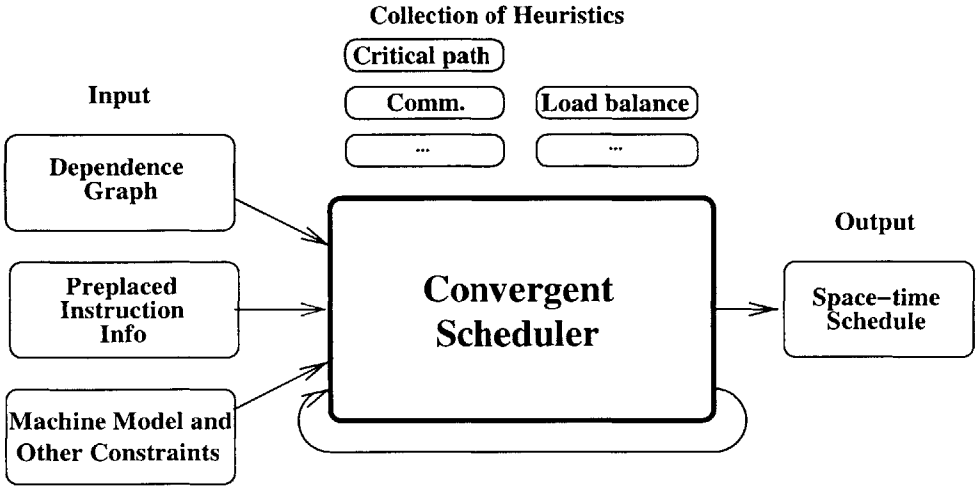


Figure 6-2. Convergent schedule framework.

of phases. This approach, however, introduces phase ordering problems, as decisions made by the early phases are based on partial information and can adversely affect the quality of decisions made by subsequent phases. A third approach is to attempt to address all the constraints together. This makes it difficult to handle additional constraints existing algorithms.

Convergent scheduling is a general resource allocation framework that makes it easy to specify arbitrary constraints and heuristics. Figure 6-2 illustrates this framework. A convergent scheduler is composed of independent passes. Each pass implements a heuristic that addresses a particular problem such as parallelism or locality. Multiple heuristics may address the same problem.

All passes in the convergent scheduler share a common interface. The input or output to each pass is a collection of spatial and temporal preferences of instructions. A pass operates by analyzing the current preferences and modifying them. As the scheduler applies the passes in succession, the preference distribution converges to a final schedule that incorporates the preferences of all the constraints and heuristics. Logically, preferences are specified as a three-input function that maps an instruction, space, and time triple to a weight.

Convergent scheduling has the following features:

1. Its scheduling decisions are made *cooperatively* rather than *exclusively.*

2. The interface allows a pass to express confidence about its decisions. A pass needs not make a poor and unrecoverable decision just because it has to make a decision. On the other side, any pass can strongly affect the final choice if needed.

3. Convergent scheduling can naturally recover from a temporary wrong decision by one pass.

4. Most compilers allow only very limited exchange of information among passes. In contrast, the weight-based interface to convergent scheduling is very expressive.

5. The framework allows a heuristic to be applied multiple times, either independently or as part of an iterative process. This feature is useful to provide feedback between passes and to avoid phase ordering problems.

6. The simple interface (preference maps) between passes makes it easy for the compiler writer to handle new constraints or design new heuristics. Passes for different heuristics are written independently, and the expressive, common interface reduces design complexity. This offers an easy way to retarget a compiler and to address peculiarities of the underlying architecture. If, for example, an architecture is able to exploit auto-increment on memory-access with a specific instruction, one pass could try to keep together memory-accesses and increments, so that the scheduler will find them together and will be able to exploit the advanced instructions.

Convergent scheduling has been used to perform both assignment and scheduling on clustered VLIWs [26]. For my research, I use it to better handle instruction assignment on Raw. Because of the ease of adding and removing heuristics, I use it as an experimental testbed for new heuristics. The final resulting algorithm is described in Sections 6.3 and 6.4. I also look for insights from heuristics that work well in convergent scheduling and incorporate those

ideas into the version of Spats described Chapter 5. In particular, the handling of preplaced instructions in Spats is motivated by the convergent scheduling framework.

The remaining sections are organized as follows. Section 6.2 describes the convergent scheduling interface between passes, while Section 6.3 describes the collection of passes I have implemented for Raw. Section 6.4 shows how a quality instruction assignment algorithm is composed from these passes.

6.2 Convergent interface

At the heart of the convergent scheduler is the expressive interface between passes. This interface is a collection of preference maps, one per instruction. The spatial and temporal preferences of each instruction are represented as weights in its preference map; a pass influences the assignment and scheduling of an instruction by changing these weights.

Initially, an instruction has no preference for any specific tile or time slot, so all the weights in its preference map is equal. A pass examines the dependence graph and the weight matrix to determine the characteristics of the preferred schedule so far. Then, it expresses its preferences by manipulating the preference maps. A Pass is not required to perform changes that affect the preferred schedule. If it is indifferent to one or more choices, it can keep the weights the same.

When convergent scheduling completes, the slot in the map with the highest weight is designated as the *preferred slot*, which includes both a preferred tile and a preferred time. The instruction is assigned to the preferred tile; the preferred time is used as the instruction priority for list scheduling.

Formally, the preference maps are represented by a three dimensional matrix $W_{i,c,t}$, where i spans over all instructions in the scheduling unit, c spans over the tiles, and t spans over time. The scheduler allocates as many time slots as the critical-path length (CPL).

Let i spans over instructions, c over tiles, t over time-slots. The following invariants are

maintained:

$$\forall i, t, c : 0 \leq W_{i,t,c} \leq 1$$

$$\forall i : \sum_{t,c} W_{i,t,c} = 1$$

Given an instruction i, we define the following:[1]

$$\begin{aligned}
\text{preferred_time(i)} &= \arg\max \left\{ t : \sum_{c} W_{i,t,c} \right\} \\
\text{preferred_tile(i)} &= \arg\max \left\{ c : \sum_{t} W_{i,t,c} \right\} \\
\text{runnerup_tile(i)} &= \arg\max \left\{ \begin{array}{l} c : \sum_t W_{i,t,c}; \\ c \neq \text{preferred_tile(i)} \end{array} \right\} \\
\text{confidence(i)} &= \frac{\sum_t W_{i,t,\text{preferred_tile(i)}}}{\sum_t W_{i,t,\text{runnerup_tile(i)}}}
\end{aligned}$$

Preferred values are those that maximize the sum of the preferences over time and tiles.

The confidence of an instruction measures how confident the convergent scheduler is about its current spatial assignment. It is computed as the ratio of the weights of the top two tiles.

The following basic operations are available on the weights:

- Any weight $W_{i,t,c}$ can be increased or decreased by a constant, or multiplied by a factor.

- The matrices of two or more instructions can be combined linearly to produce a matrix of another instruction. Given input instructions $i_1, i_2, ..., i_n$, an output instruction j, and weights for the input instructions $w_1, ..., w_n$where $\sum_{k \in [1,n]} w_k = 1$, the linear combination is as follows:

$$\text{for each } (c,t), W_{j,t,c} \leftarrow \sum_{k \in [1,n]} w_k * W_{i_k,t,c}$$

[1]The function arg max returns the value of the variable that maximizes the expression for a given set of values (while max return the value of the expression). For instance $\max\{0 \leq x \leq 2 : 10 - x\}$ is 10, and $\arg\max\{0 \leq x \leq 2 : 10 - x\}$ is 0.

My implementation uses a simpler form of this operation, with $n = 2$ and $i_1 = j$:

$$\text{for each } (c,t), W_{i_1,t,c} \leftarrow w_1 W_{i_1,t,c} + (1 - w_1) W_{i_2,t,c}$$

The convergent scheduler never performs this full operation because it is expensive. Instead, the scheduler only does the operation on part of the matrices, *e.g.*, only along the space dimension, or only within a small range along the time dimension.

- The system incrementally keeps track of the sums of the weights over both space and time, so that they can be determined in $O(1)$ time. It also memoizes the preferred_time and preferred_tile of each instruction.

- The preferences can be normalized to guarantee our invariants; the normalization simply performs:

$$\text{for each } (i,c,t), W_{i,t,c} \leftarrow \frac{W_{i,t,c}}{\sum_{t,c} W_{i,t,c}}$$

6.3 Collection of Heuristics

This section presents a collection of heuristics I have implemented for convergent scheduling. Each heuristic attempts to address a single constraint and only communicates with other heuristics via the weight matrix. There are no restrictions on the order or the number of times each heuristic is applied. For the Raw convergent scheduler, the following parameters are selected by trial-and-error: the set of heuristics we use, the weights used in the heuristics, and the order in which the heuristics are run. It is possible to use machine learning to tune these parameters [35].

Whenever necessary, the map values are normalized at the end of a pass to enforce the invariants given in Section 6.2. For brevity, this step is not included in the description below.

A pass similar to this one can address the fact that some instructions cannot be scheduled in all tiles in some architectures, simply by squashing the weights for the unfeasible tiles.

Preplacement (PLACE) This pass increases the weight for preplaced instructions to be placed in their home tile. Since this condition is required for correctness, the weight increase is large. Given preplaced instruction i, let $cp(i)$ be its preplaced tile. Then,

$$\text{for each } (i \in PREPLACED, t),$$
$$W_{i,t,cp(i)} \leftarrow 100 W_{i,t,cp(i)}$$

Critical path strengthening (PATH) This pass tries to keep all the instructions on a critical path (CP) in the same tile. If instructions in the paths have bias for a particular tile, the path is moved to that tile. Otherwise the least loaded tile is selected. If different portions of the paths have strong bias toward different tiles (*e.g.*, when there are two or more preplaced instructions on the path), the critical path is broken in two or more pieces and kept locally close to the relevant home tiles. Let $cc(i)$ be the chosen tile for the CP.

$$\text{for each } (i \in CP, t, c),\ W_{i,t,cc(i)} \leftarrow 3 W_{i,t,cc(i)}$$

Communication minimization (COMM) This pass reduces communication load by increasing the weight for an instruction to be in the same tiles where most of neighbors (successors and predecessors in the dependence graph) are. This is done by summing the weights of all the neighbors in a specific tile, and using that to skew weights in the correct direction.

$$\text{for each } (i,t,c), W_{i,t,c} \leftarrow W_{i,t,c} \sum_{n \in \text{neighbors of } i} W_{n,t,c}$$

We have also implemented a variant of this that considers *grand-parents* and *grand-children*, and we usually run it together with COMM.

$$\text{for each } (i), W_{i,t_i,c_i} \leftarrow 2 W_{i,t_i,c_i}$$

Load balance (LOAD) This pass performs load balancing across tiles. Each weight on a tile is divided by the total load on that tile:

$$\text{for each } (i,t,c), W_{i,t,c} \leftarrow W_{i,t,c}/load(c)$$

Level distribute (LEVEL) This pass distributes instructions at the same *level* across tiles. Given instruction i, we define $level(i)$ to be its distance from the furthest root. Level distribution has two goals. The primary goal is to distribute parallelism across tiles. The second goal is to minimize potential communication. To this end, the pass tries to distribute instructions that are far apart, while keeping together instructions that are near each other.

To perform the dual goals of instruction distribution without excessive communication, instructions on a level are partitioned into bins. Initially, the bin B_c for each tile c contains instructions whose preferred tile is c, and whose confidence is greater than a threshold (2.0). Then, we perform the following:

LevelDistribute: int l, int g

I_l = Instruction i | level(i) = l

foreach Tile c

$I_l = I_l - B_c$

$I_g = \{i \mid i \in I_l; \text{distance}(i, \text{find\_closest\_bin}(i)) > g\}$

while $I_l \neq \phi$

B = round_robin_next_bin()

$i_{closest} = \arg\max\{i \in I_g : \text{distance}(i, B)\}$

$B = B \cup i_{closest}$

$I_l = I_l - i_{closest}$

Update I_g

The parameter g controls the minimum distance granularity at which we distribute instructions across bins. The distance between an instruction i and a bin B is the minimum distance

between i and any instruction in B.

LEVEL can be applied multiple times to different levels. Currently we apply it every four levels on Raw. The four levels correspond approximately to the minimum granularity of parallelism that Raw can profitably exploit given its communication cost.

Path propagation (PATHPROP) This pass selects high confidence instructions and propagates their convergent matrices along a path. The confidence threshold t is an input parameter. Let i_h be the selected confident instruction. The following propagates i_h along a downward path:

find $i \mid i \in \text{successor}(i_h); \text{confidence}(i) < \text{confidence}(i_h)$
while $(i \neq nil)$
  for each $(c, t), W_{i,t,c} \leftarrow 0.5W_{i,t,c} + 0.5W_{i_h,t,c}$
  find $i_n \mid i_n \in \text{successor}(i); \text{confidence}(i_n) < \text{confidence}(i_h)$
  $i \leftarrow i_n$

A similar function that visits predecessors propagates i_h along an upward path.

Emphasize critical path distance (EMPHCP) This pass attempts to help the convergence of the time information by emphasizing the level of each instruction. The level of an instruction is a good time approximation because it is when the instruction can be scheduled if a machine has infinite resources.

$$\text{for each } (i, c), W_{i,level(i),c} \leftarrow 1.2W_{i,level(i),c}$$

6.4 Raw convergent scheduler

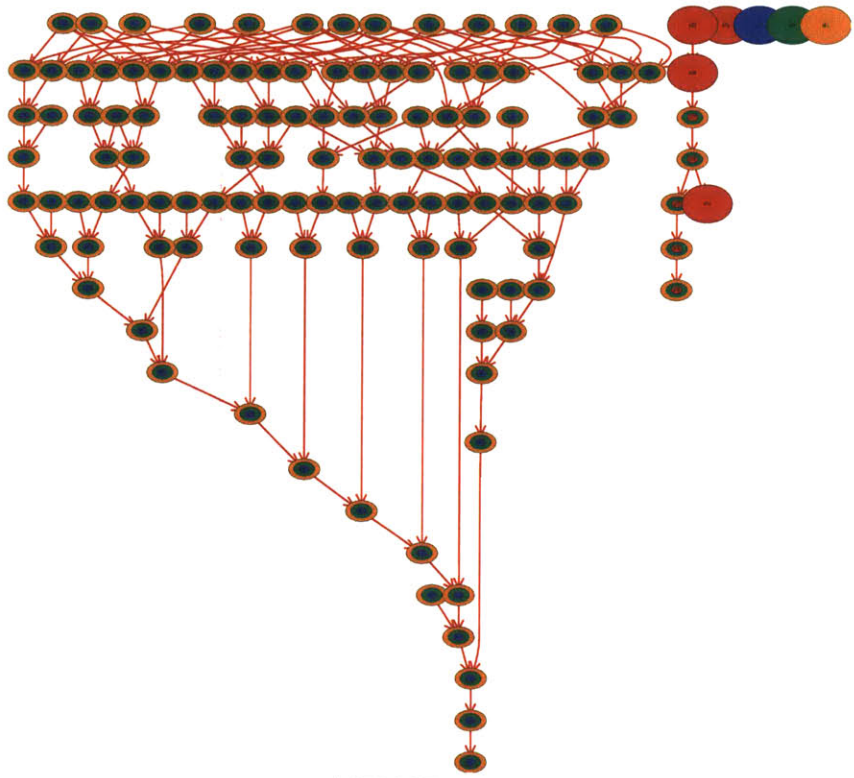
Figure 6-3 shows the passes that make up the Raw convergent scheduler, which performs instruction assignment. Passes are listed in the order they are executed. Two kinds of passes make up the convergent scheduler. One kind of passes biases a selective subset of

```
PLACE
PATHPROP
LOAD
PATH
PATHPROP
LEVEL
PATHPROP
COMM
PATHPROP
EMPHCP
```

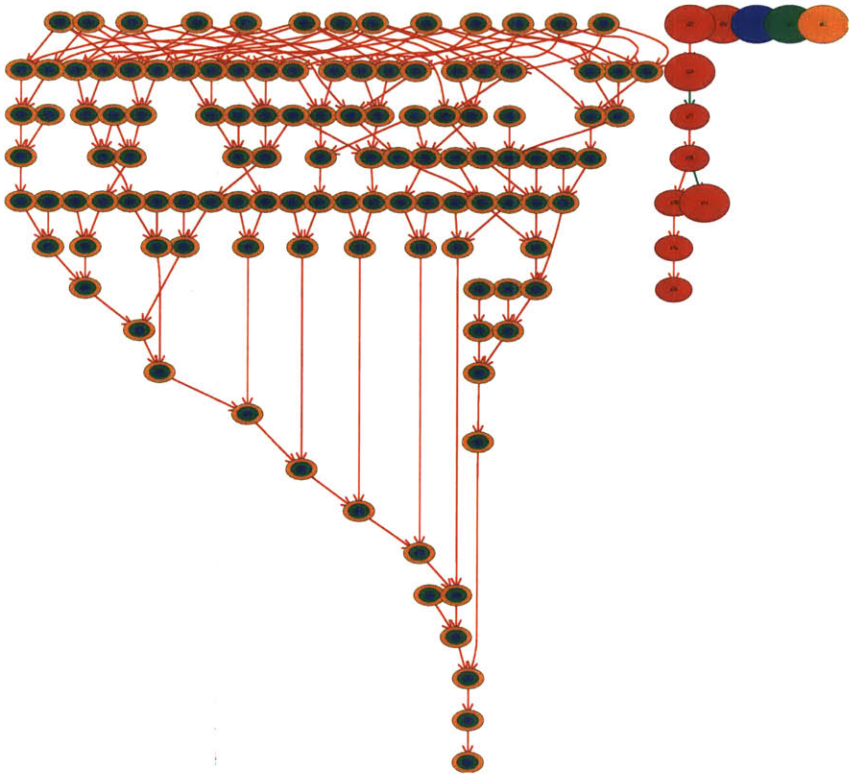
Figure 6-3. Sequence of heuristics used by the convergent scheduler for Raw.

nodes toward a particular tile. For example, LEVEL looks for nodes at the same depth in the dependence graph and spreads them across tiles, while PATH biases a critical path toward a tile. The other kind of passes propagates the preferences of nodes with high confidence toward neighboring nodes with low confidence. This propagation helps reduce communication and improve locality. COMM and PATHPROP fall into this category.

Figures 6-4 to 6-7 show the scheduler in action as it performs tile assignment of a data dependence graph onto four tiles. The figures show the state of the preference map after selected passes, encoded by the colors of the nodes. Each color represents a different tile, and the amount of color in a node represents the level of preference the node has for the corresponding tile. Thus, a node with a strong preference for a particular tile will be displayed as a big, solid-color node. A node with a weak preference is either a small node or a node with rainbow colors. The final assignment is shown in Figure 6-8.

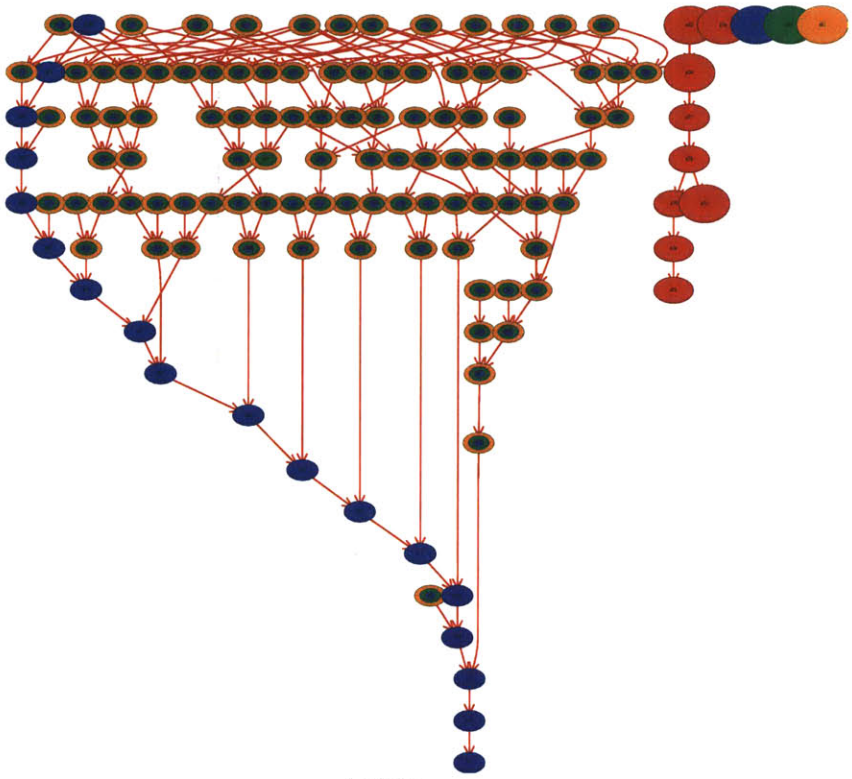


(a)PLACE

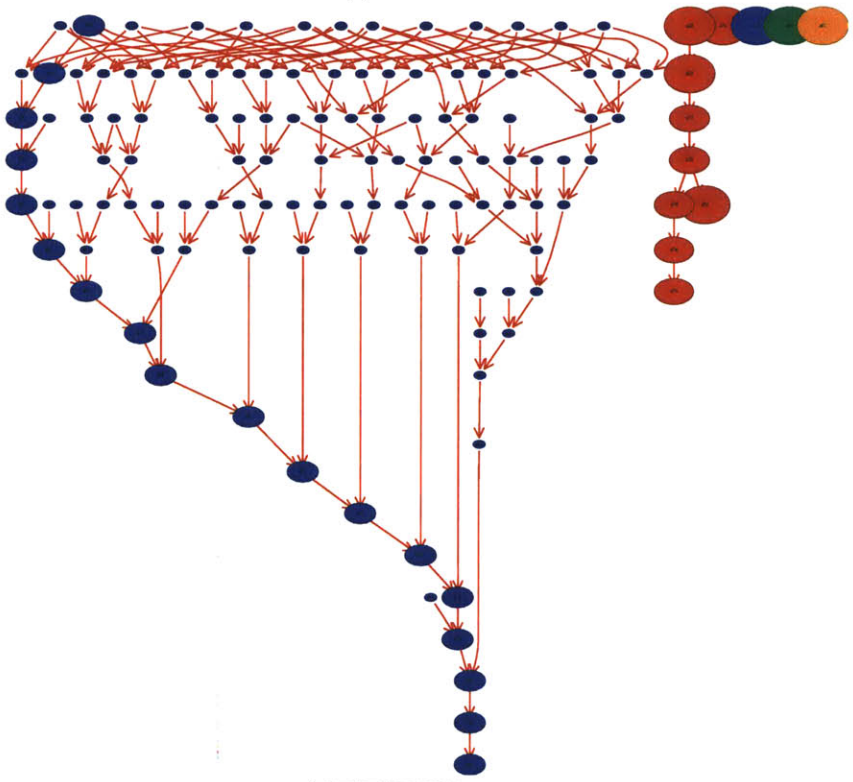


(b)PATHPROP

Figure 6-4. A convergent scheduling example; passes a-b.

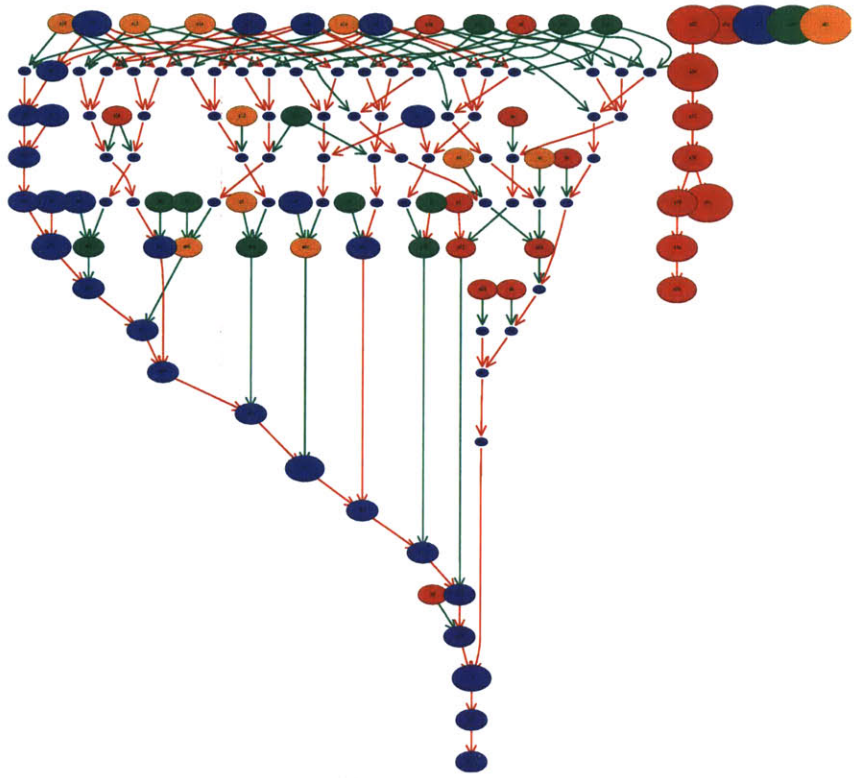


(c)PATH

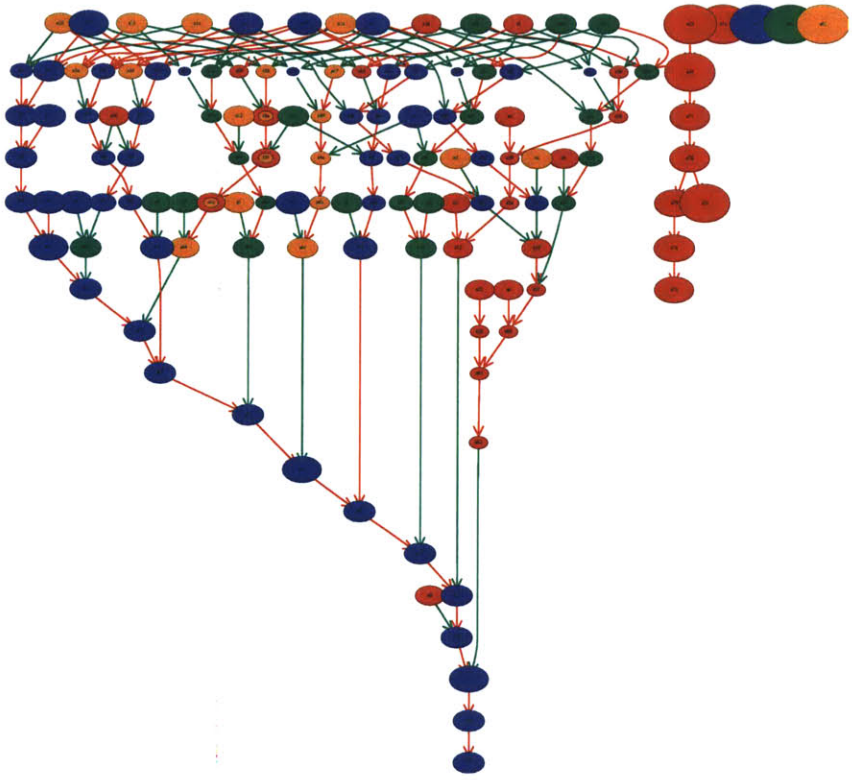


(d)PATHPROP

Figure 6-5. A convergent scheduling example; passes c-d.

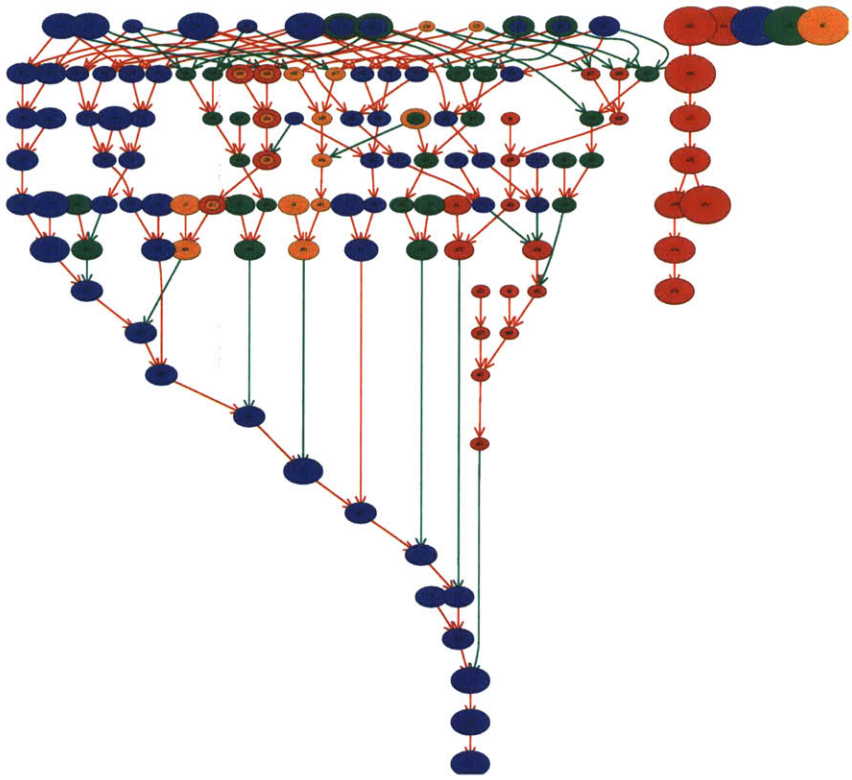


(e)LEVEL

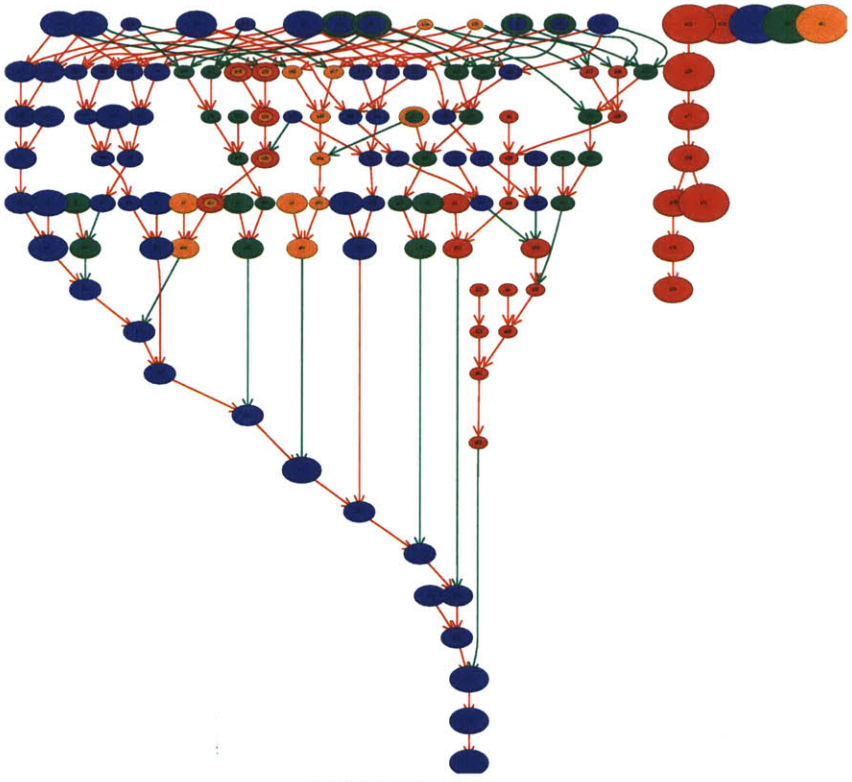


(f)PATHPROP

Figure 6-6. A convergent scheduling example; passes e-f.



(g)COMM



(h)PATHPROP

Figure 6-7. A convergent scheduling example; passes g-h.

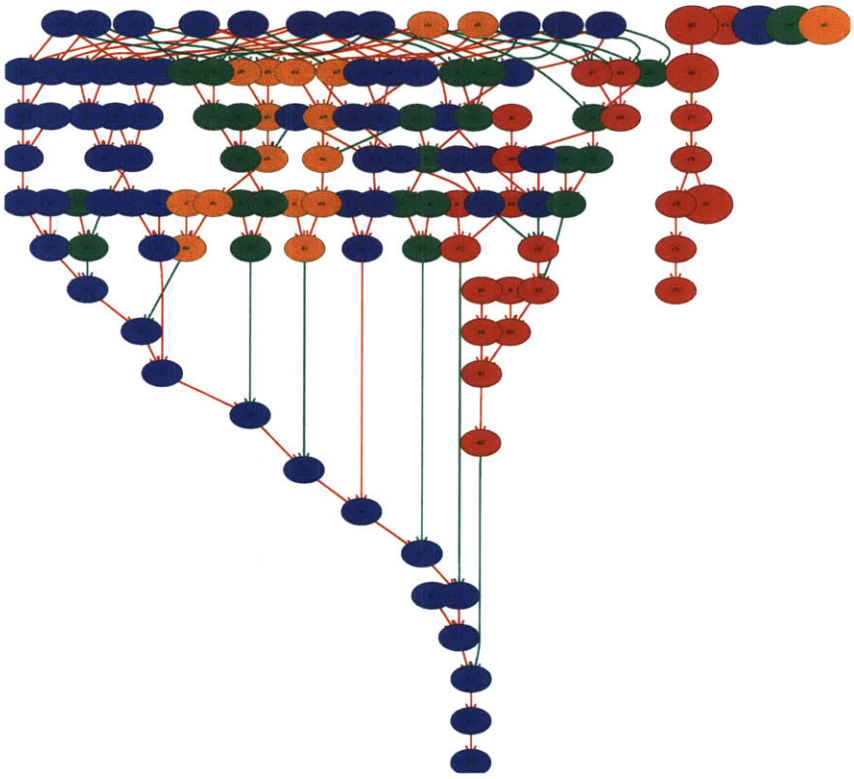


Figure 6-8. A convergent scheduling example: final assignment.

Chapter 7

Evaluation

This section presents some evaluation of Rawcc, which implements all the ILP techniques described in this paper. Unless noted otherwise, the results are from the version of Spats described in Section 5. The section begins by describing the methodology. Data is gathered on the validated, cycle-accurate Raw simulator. The section presents data showing that the simulator produces results that are on average within 8% of the 16-tile hardware prototype, despite the fact that the two systems have different the instruction memory systems.

The section includes the following evaluation results. First, it presents the performance of Rawcc on Raw machine as the number of tiles is scaled from one to 64 tiles. The results show that Rawcc is able to attain modest speedup for fine-grained applications, as well speedups that scale up to 64 tiles for applications with such parallelism. Then, the section shows how performance scales with problem sizes, and it demonstrates that unlike traditional coarse-grained parallelization techniques used for multiprocessors, Rawcc is able to attain speedup for small as well as large problem sizes. Next, the section compares three versions of Spats that use different algorithms for instruction assignment. To help understand the degree of locality exhibited by the parallelized applications, the section then presents statistics on operand usage, measuring the fraction of operands consumed that are locally generated versus those that are remotely generated. The section concludes by examining the effects

Type	Benchmark	Source	Lines of Code	Primary Array Size	Sequential Run-time
Irregular	sha	Perl Oasis	626	-	1.58M
	aes		102	-	1.16M
	fpppp-kernel	Spec95	735	-	1.14M
Sparse Matrix	moldyn	Chaos	815	64x3	133M
	unstructured	Chaos	1030	3136x3	21M
Dense Matrix	btrix	Nasa7:Spec92	236	256x1x2x5	17.2M
	cholesky	Nasa7:Spec92	126	1024	34M
	vpenta	Nasa7:Spec92	157	512x16	23M
	mxm	Nasa7:Spec92	64	32x1024	23M
	tomcatv	Spec95	254	64x64	18.9M
	swim	Spec95	618	512x32	137M
	life	Rawbench	118	32x1024	42M
	jacobi	Rawbench	59	512x512	40M

Table 7.1. Benchmark characteristics.

of long latency dynamic events on performance. It finds that control decoupling gives the architecture a much higher tolerance for these events, especially in machines with many tiles or high memory latency.

A more comprehensive list of result figures and tables can be found in Appendix A.

7.1 Methodology

Experiments are conducted by compiling a set of benchmarks with Rawcc and running them on Beetle, a Raw simulator. Table 7.1 gives some basic characteristics of the benchmarks, which include irregular, sparse matrix, and dense matrix codes. *Fpppp-kernel* is the kernel basic block that accounts for 50% of the run-time in fpppp of Spec95. Since Raw currently does not support double-precision floating point, all floating point operations in the original benchmarks are converted to single precision. In some cases the size of the datasets and the iteration counts have been modified to reduce simulation time.

Latencies of the basic instructions on Raw are as follows: 1-cycle integer add or subtract, 2-cycle integer multiply, 36-cycle integer divide, 2-cycle load, 1-cycle store, 66-cycle average cache miss latency, 4-cycle floating add or subtract, 4-cycle floating multiply, and 10-cycle floating divide. All operations except for divides are pipelined.

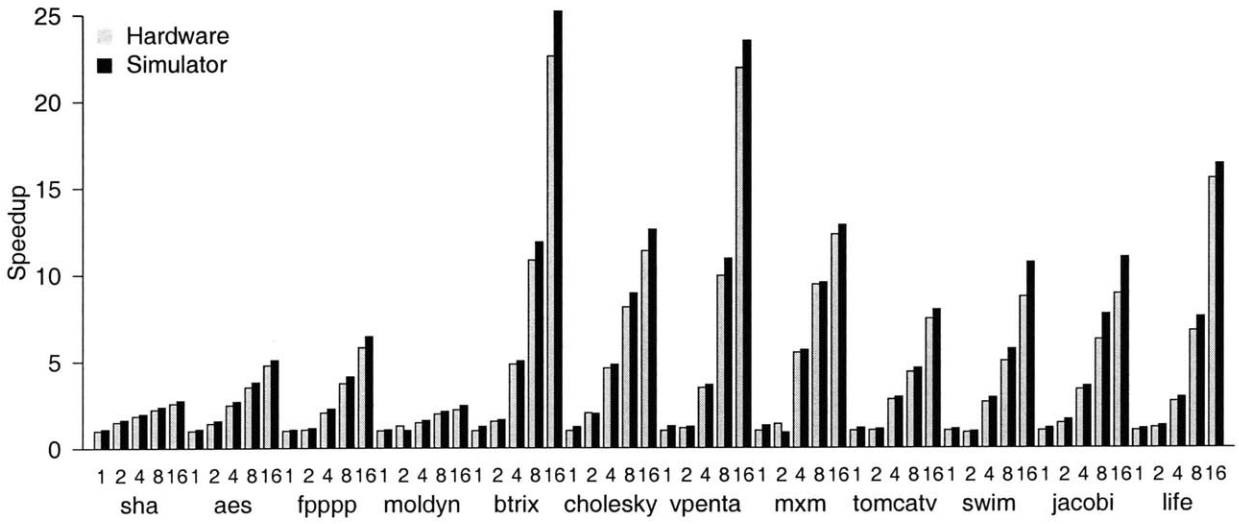


Figure 7-1. Performance of Rawcc parallelized applications on the hardware and on the simulator, with the targeted number tiles varying from one to 16. Each datapoint is normalized against the single-tile execution time of the corresponding benchmark on the hardware.

The Beetle simulator has been verified against a gate-level RTL netlist of a 16-tile Raw prototype chip and found to have *exactly* the same timing and data values for all 200,000 lines of our hand-written assembly test suite. With the above benchmarks, we further validate the simulator against a prototype Raw motherboard that consists of a single Raw chip, SDRAM chips, I/O interfaces and interface FPGAs. This prototype only differs from the simulation environment in one respect: the instruction memory system. The prototype contains 32 KB of instruction memory per tile, and it relies on a fully software system to perform instruction caching [31]. The simulator, on the other hand, can simulate up to 128 KB instruction memory per tile, enough to fit the code for any of the benchmarks in our experiments.

Figure 7-1 shows the performance comparison between the simulator and the hardware system as the targeted number of tiles is varied from one to 16.[1] Magnified versions of these graphs can be found in the appendix. In the graph, each datapoint is normalized against the single-tile run-time of the corresponding benchmark on the hardware. Despite the difference in the instruction memory systems, the run-times on the simulator are very close to that

[1]Currently *unstructured* cannot be run on the hardware because it uses specialized I/O for initialization.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32	N=64
sha	1.00	1.50	1.81	2.17	2.52	1.93	1.86
aes	1.00	1.45	2.46	3.49	4.66	4.36	4.22
fpppp-kernel	1.00	1.09	2.13	3.89	6.07	7.92	10.03
unstructured	1.00	1.65	2.97	3.32	3.57	3.44	3.32
moldyn	1.00	0.98	1.52	2.01	2.31	2.28	2.43
btrix	1.00	1.32	4.06	9.57	20.30	40.34	89.51
cholesky	1.00	1.63	3.93	7.29	10.28	10.14	9.80
vpenta	1.00	0.97	2.86	8.56	18.40	37.19	73.03
mxm	1.00	0.70	4.36	7.37	9.91	11.06	12.35
tomcatv	1.00	0.95	2.55	4.01	6.92	8.21	10.24
swim	1.00	0.86	2.62	5.16	9.67	18.68	28.58
jacobi	1.00	1.42	3.10	6.70	9.54	22.86	21.99
life	1.00	1.16	2.64	6.82	14.78	27.78	55.67

Table 7.2. Benchmark speedup versus performance on a single-tile.

on the hardware. The differences vary between 1% and 20% of the execution time on the hardware, with an average difference of 8.3%. On 16 tiles, the average difference is 7.8% with a worst case difference of 15%.

Having performed this simulation validation, the rest of the experiments will be conducted on the simulator. The simulator gives us the flexibility to collect results for up to 64 tiles, collect detailed operand statistics, as well as evaluate control decoupling in the presence of long latency dynamic events.

7.2 Performance

I measure the speedup attained by Rawcc as we vary the number of tiles from one to 64. Table 7.2 shows these results. The results show that the Rawcc is able to exploit ILP profitably across the Raw tiles for all the benchmarks. The average speedup on 64 tiles is 23.6.

The sparse matrix and irregular applications have parallelism that stresses the locality/parallelism tradeoff capabilities of Rawcc. For these applications, the results show that Rawcc is able to profitably exploit a modest amount of parallelism on four to eight tiles despite the communication latencies.

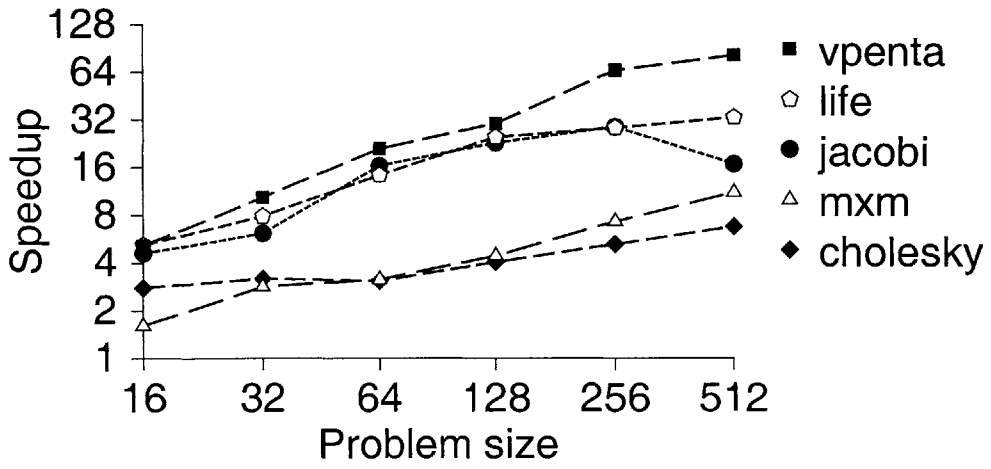


Figure 7-2. Rawcc speedup on 64 tiles for varying problem sizes.

As expected, the dense matrix applications perform particularly well because arbitrarily large amount of parallelism can be exposed by loop unrolling. In some cases the speedup is superlinear because cache capacity increases with the number of tiles, which in turn leads to fewer cache misses. Currently, Rawcc unrolls loops by the minimum amount required to guarantee the static residence property referred to in Section 3.2.3, which in most of these cases expose as many copies of the inner loop for scheduling of ILP as there are tiles.

The dense matrix benchmarks have been parallelized on multiprocessors by recognizing do-all parallelism and distributing such parallelism across the processors. Rawcc detects the same parallelism by partially unrolling a loop and distributing individual instructions across tiles. The Rawcc approach is more flexible, however, because it can schedule do-across parallelism contained in loops with loop carried dependences. For example, several loops in *tomcatv* contain reduction operations, which are loop carried dependences. In multiprocessors, the compiler needs to recognize a reduction and handle it as a special case. The Raw compiler handles the dependence naturally, the same way it handles any other arbitrary loop carried dependences.

Another advantage of Rawcc's ILP approach is that it has lower overhead than multiprocessor parallelization, which often requires large datasets before it can amortize away the

overhead and attain speedups [2]. Figure 7-2 plots the speedup attained by Rawcc for a selected subset of dense matrix applications as the problem size is varied. In the figure, the problem size is the size of the primary array along its largest dimension. The figure shows that Rawcc is able to attain decent speedups for both large and small datasets. Speedup decreases gracefully as problem size decreases, and Rawcc is able to attain speedup even for problem sizes as small as 16 data elements.

7.3 Comparison of instruction assignment algorithms

A focus of this thesis is instruction assignment, as it has been empirically observed to have a high impact on performance. This section presents comparison of results for three assignment algorithms: pSpats, Spats, and convergent scheduling. pSpats is an early version of Spats originally described in [25]. Its assignment algorithm differs from Spats in two ways. First, the clustering phase simply employs the clusters output from DSC. It does not perform the cluster refinement step, which enforces the constraint that at most one preplaced instruction is in a single cluster. Second, the merging phase uses a communication minimizing, load-balancing algorithm that does not directly focus of parallelism distribution. Spats is the algorithm described in Section 5. Convergent scheduling is the algorithm described in Section 6.

Figures 7-3 and 7-4 show the results of these comparisons. The data can also be found in tabular form in Tables A.1, A.2, and A.3 in the Appendix. The comparison between pSpats and Spats are as follows. Spats perform much better on the dense matrix applications. On 16 tiles the improvement is 52%, while on 64 tiles the improvement is 82%. For non-dense matrix applications, Spats performs a little worse, with a performance loss of about 11% on 64 tiles. However, the performance of these applications peeks at 16 tiles as they do not have enough parallelism to sustain 64 tiles. If we compare the performance at optimal configuration of 16 tiles, the performance loss is only 2.4%.

Overall, I find the the algorithmic improvements from Spats to be worthwhile, with an overall improvement of 46% over pSpats on 64 tiles.

As Spats draws some of its algorithmic improvement for convergent scheduling, it is not surprising that the performance of the two algorithms are very similar. For dense matrix codes, convergent scheduling outperforms Spats by 10%, with a sigificant portion of the advantage coming from the single application vpenta. For non-dense matrix codes, it performs 5% worse. Overall, convergent scheduling performs 4.5% better than Spats.

7.4 Analysis of operands and communication

To help understand the degree of locality exhibited by the parallelized applications, the section presents statistics on operand usage, measuring the fraction of operands consumed that are locally generated versus those that are remotely generated.

7.4.1 Analysis of Operand Origins

Figure 7-5a analyzes the origins of operands on the tiles they are used. *Fresh* operands are computed locally and make use of either the 0-cycle local bypassing or the local register file; *remote* operands arrive over the inter-tile transport network; and *reload* operands originate from a previous spill to the local data cache. The figure presents these results for each benchmark, varying the number of tiles from 2 to 64. An operand is counted only once per generating event (calculation by a functional unit, arrival via transport network, or reload from cache) regardless of how many times it is used.

For the dense matrix applications, a large fraction of the operands, between 75% to 90%, is local across all tile configurations. This fraction stays relatively constant as the number of tiles increases. For these applications, this analysis suggests that the applications have much inherent locality that Rawcc is able to exploit. In contract, the sparse matrix and irregular applications have a much larger fraction of remote operands, and the fraction increases with

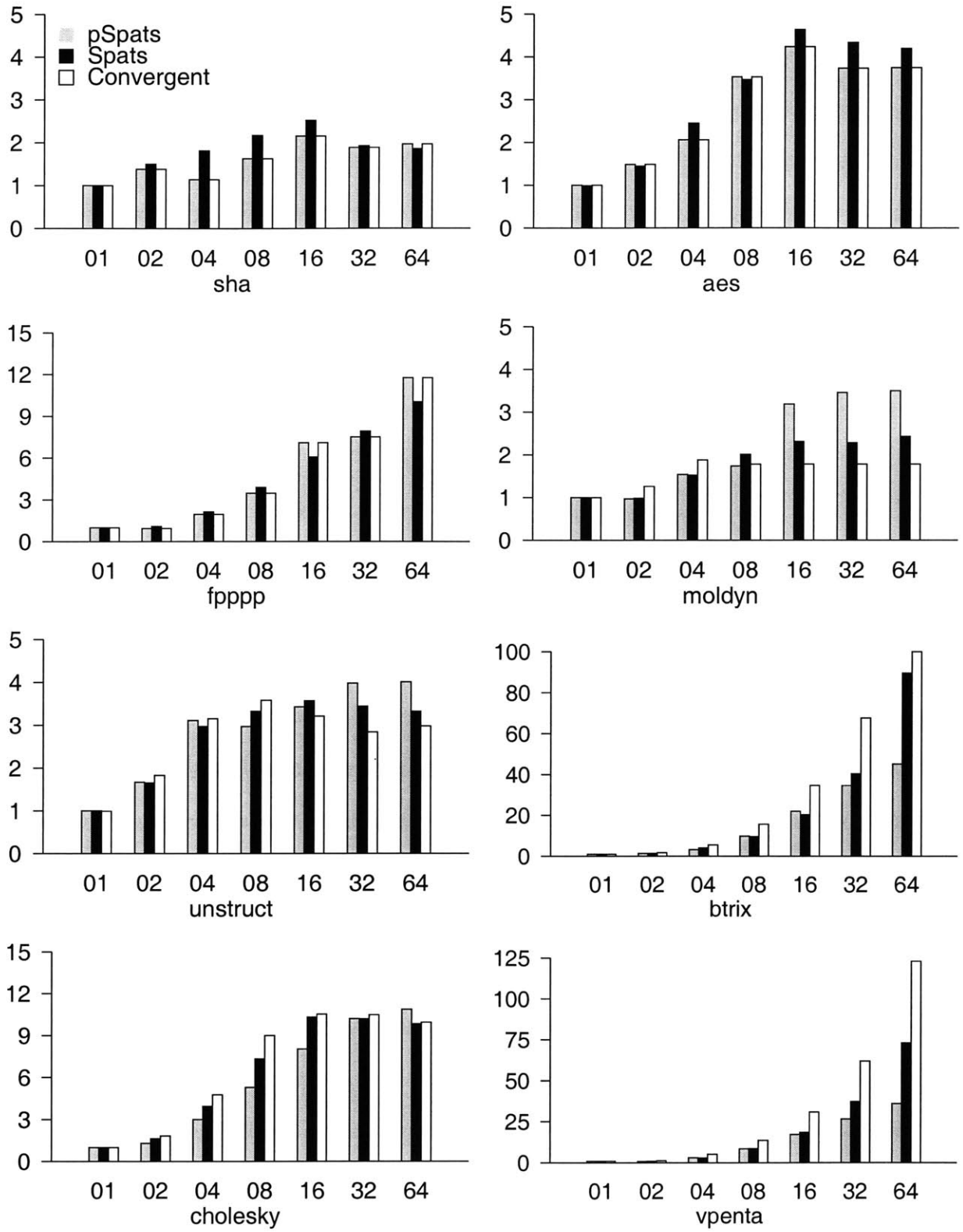


Figure 7-3. Performance of Rawcc parallelized applications using various instruction assignment algorithms, part I.

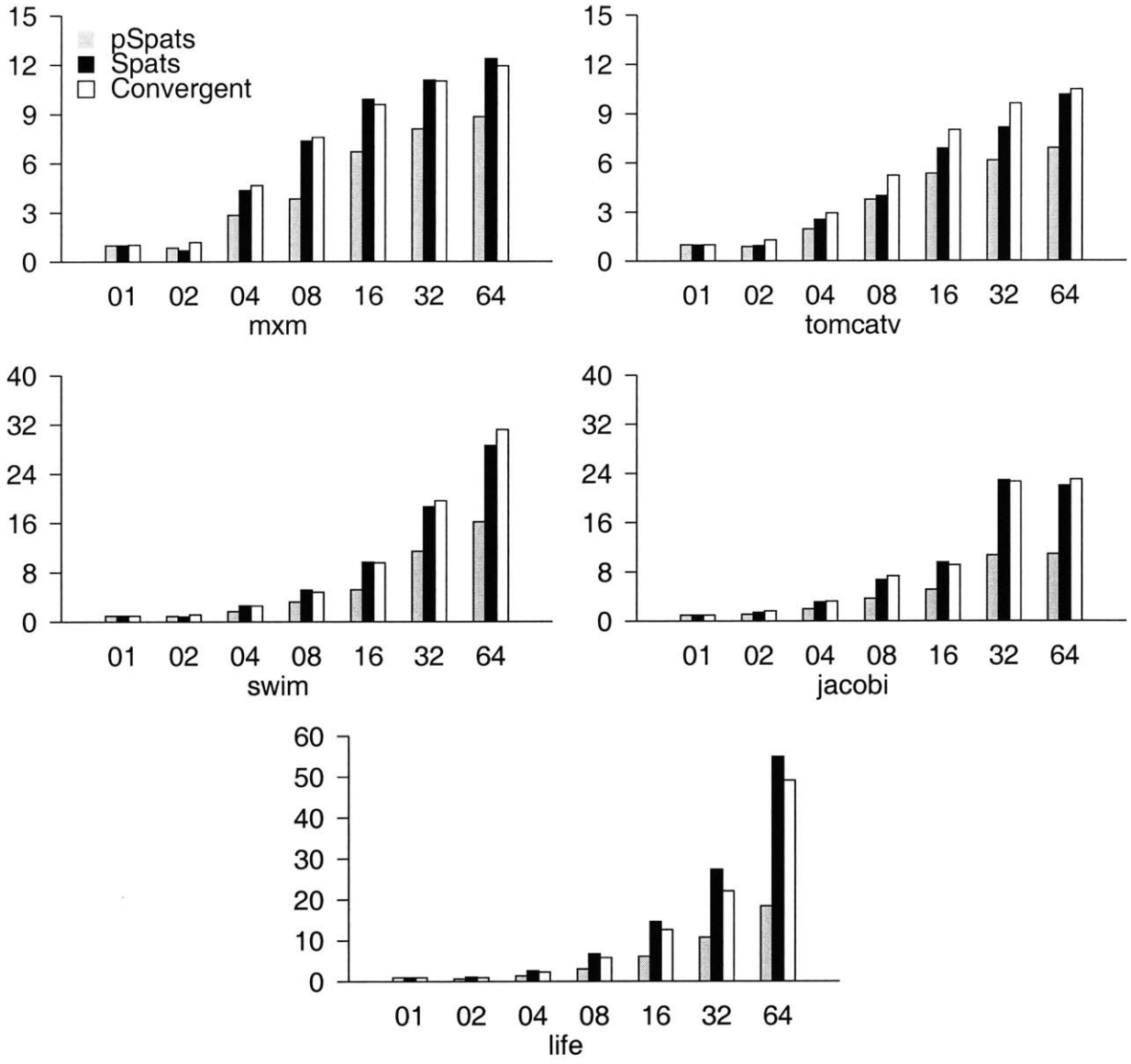


Figure 7-4. Performance of Rawcc parallelized applications using various instruction assignment algorithms, part II.

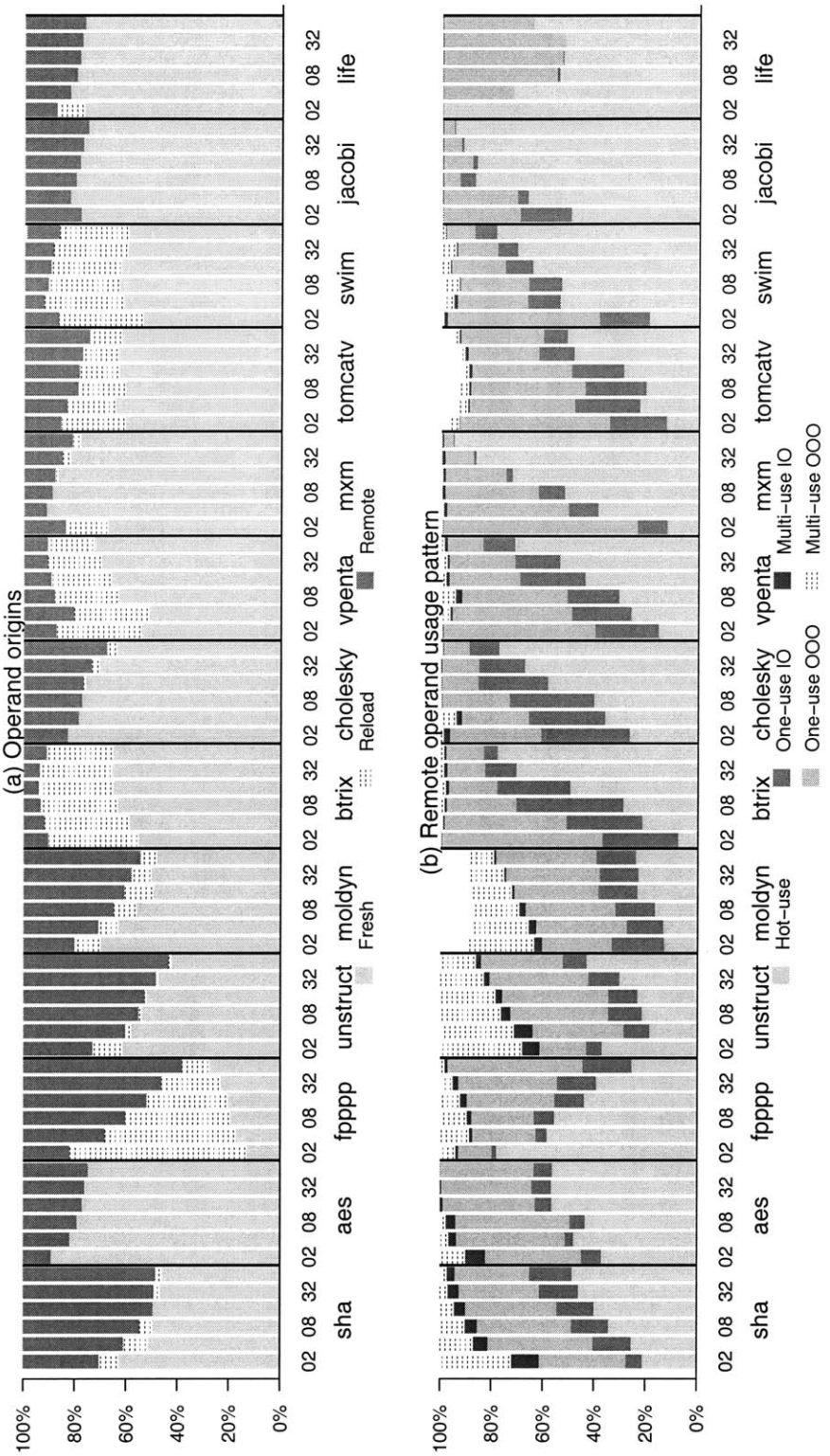


Figure 7-5. a) Analysis of operand origins. b) Analysis of remote operand usage pattern. For each benchmark there are six columns, one for each power of two configuration from two to 64.

the number of tiles. On 64 tiles, this fraction ranges from 20% (*aes*) to 60% (*moldyn*). For these applications, the analysis suggests that Rawcc had to sacrifice a significant amount of locality to exploit parallelism.

7.4.2 Analysis of Remote Operand Usage Pattern

I further analyze the remote operands from Figure 7-5a by their usage pattern. The data measures the ability of the compiler to schedule sender and receiver instruction sequences so as to allow values to be delivered just-in-time.

The remote operands at the receiver node are classified as follows. *Hot-uses* are directly consumed off the network and incur no extra instruction cost. *One-uses* and *Multi-uses* are first moved into the register file and then used — *One-uses* are subsequently used only once, while *Multi-uses* are used multiple times. The study also considers whether the operand arrives in the same order in which a tile uses them. A remote operand is *in-order (IO)* if all its uses precede the arrival of the next remote operand; otherwise it is *out-of-order (OOO)*.

Figure 7-5b shows the relative percentages of each operand class.[2] Interestingly, a large fraction of remote operands is single-use. Even for the few benchmarks that have a large fraction of multi-use operands, that fraction rapidly decreases as the number of tiles increases. This trend is consistent with the expectation that as Rawcc parallelizes across more tiles, a multi-use operand on a machine with few tiles will likely become multiple single-use operands on multiple tiles on a machine with more tiles.

7.5 Benefit of control decoupling

Rawcc attempts to statically orchestrate all aspects of program execution. Not all events, however, are statically predictable. Dynamic events include cache misses, I/O operations, and dynamic memory operations with unknown tile locations. This section measures the

[2]Due to how the compiler handles predicated code, a few of the benchmarks have remote operands that are not consumed at all, which explains why a few of the frequency bars sum up to less than 100%.

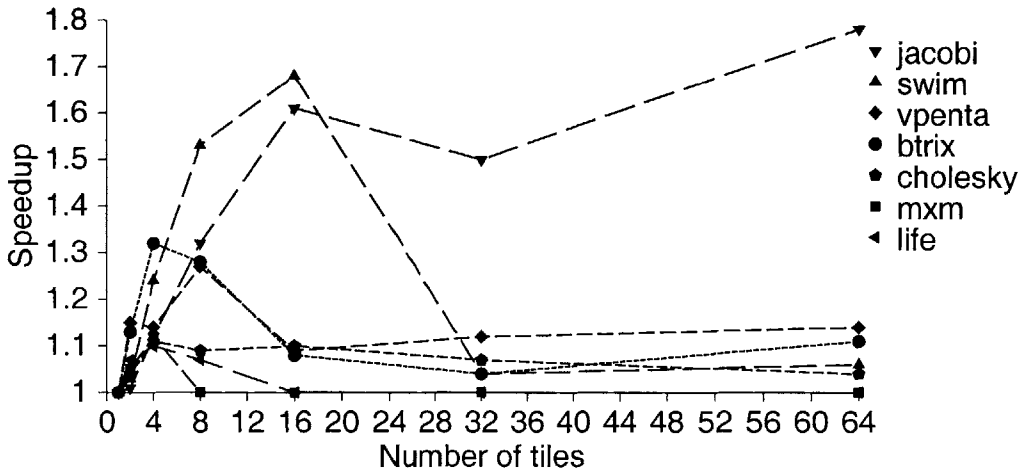


Figure 7-6. Speedup ratio of control-decoupled machine (Raw) over a lock-step machine in the presence of cache misses, for the seven affected applications.

benefit of control decoupling on performance in the presence of these long-latency dynamic events.

The study compare the effects of these events on two machine models. One is a faithful representation of the Raw machine, which has control decoupling. The other machine is identical to the Raw machine except that control on each tile executes in lock-step, mirroring the execution style of a traditional VLIW machine. On the control decoupled machine, a dynamic event only directly affects the tile on which the event occurs. Other tiles can proceed independently until they need to communicate with the blocked tile. On the lock-step machine, however, a dynamic event stalls the entire machine immediately. [3]

Figure 7-6 plots the speedup ratio of a control-decoupled machine over that of a lock-step machine. The figure only plots the seven applications whose differences in performance between the two machine model are significant. For the remaining applications, the frequency of cache misses is low and the performance ratio is less than 5% for any number of tiles.

If cache misses were randomly distributed events, one would expect the lock-step execu-

[3]Many VLIWs support non-blocking stores, as well as loads that block on use instead of blocking on miss. These features reduce but do not eliminate the adverse effects of stalling the entire machine, and they come with a potential penalty in clock speed.

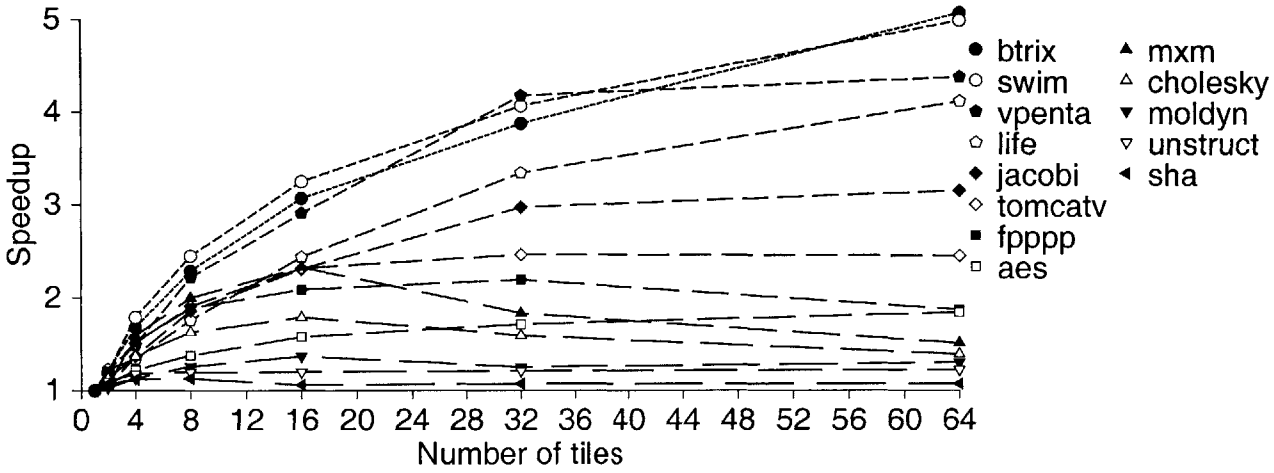


Figure 7-7. Speedup of a control decoupled machine over a lock-step machine, with insertion of artificial cache misses with 5% probability and 100 cycles delay.

tion to perform worse progressively with increasing number of tiles. However, none of the applications exhibits this behavior. For *jacobi*, the performance ratio saturates at 16 tiles. This saturation effect occurs because for dense matrix applications, the instruction schedule on each tile tend to be similar, so cache misses tend to occur at around the same time even on control decoupled machines. Therefore, the performance ratio stops increasing after the machine reaches a certain size.

The behavior for the other six applications all experiences a peak ratio for an intermediate sized configuration and negligible performance difference at 64 tiles. This is because on 64 tiles, those applications have datasets that can fit in memory; thus they do not incur any cache misses. On smaller number of tiles, however, cache misses do occur and their performance on a lock-step machine experiences the expected degradation.

To get a feel for how much performance degrades on a lock-step machine for applications with larger data sets and thus more cache misses, additional artificial dynamic disturbances are inserted into the simulation. The disturbances are modeled as random events that happen on loads and stores, with a 5% chance of occurrence and an average stall time of 100 cycles.

Figure 7-7 shows the speedup ratio of a control-decoupled machine over that of a lock-step

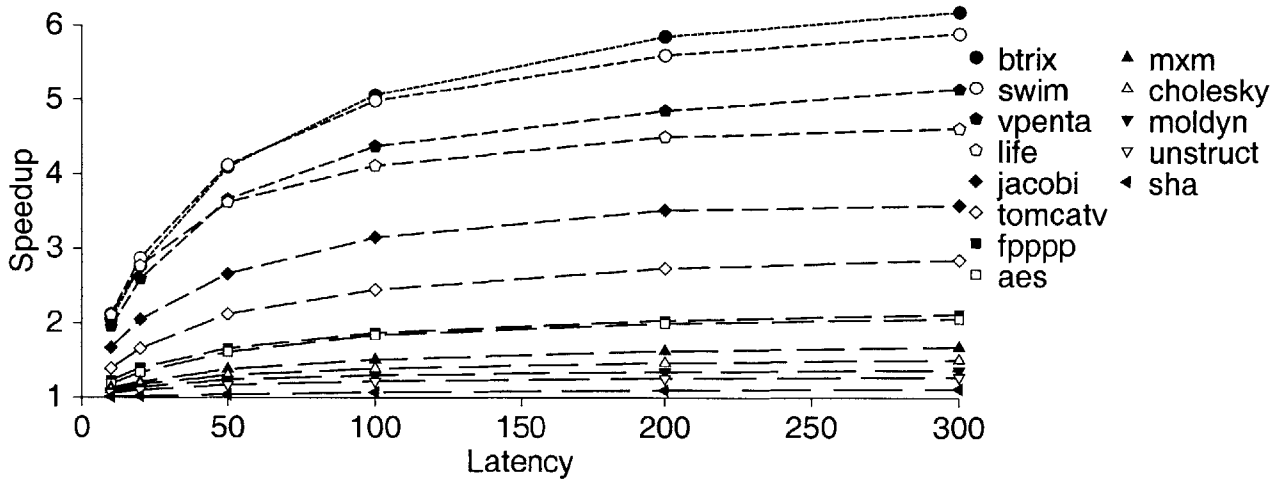


Figure 7-8. Speedup of a control decoupled machine over a lock-step machine for 64 tiles, with insertion of artificial cache misses with 5% probability and varying cycles delay.

machine in the presence of these disturbances. It shows the modest miss rate of 5% has severe effects on lock-step performance. On 16 tiles, the average performance ratio is 2.1x. On 64 tiles, the average performance ratio worsens to 2.6x.

An expected trend is that memory latency will continue to increase in terms of clock cycles. Therefore, I vary the memory latency and measure the speedup of a control decoupled machine over a lock-step machine for 64 tiles. Figure 7-8 plots the results. The trend is that the gap in performance between decoupled machine and lock-step machine increases with increasing memory latency. The average performance ratio is 2.6x at 100 cycles and 3.0x at 300 cycles.

Chapter 8

Related work

This section presents related work. First, it discusses the applicability of Spats to partial TPAs. It then discusses related work on ILP orchestration and control orchestration.

8.1 Partial TPAs

Though Spats is developed for a full TPA, its techniques may be applicable to partial TPAs as well. For a partial TPA with a specific set of distributed resources, Table 3.2 can be used to look up the phases that are relevant to those resources. In particular, instruction management is the most performance critical part of Spats, and all partial TPAs distribute their functional units. Therefore, the instruction partition and assignment algorithms of Spats are applicable to partial TPAs.

Recent partial TPAs include Trips [33], and WaveScalar [40]. Table 8.1 summarizes the level of distribution of these architectures as well as of Raw.

8.2 ILP orchestration

Raw shares much similarities with clustered VLIWs, which also statically schedule ILP across distributed clusters of registers and functional units. In fact, Raw resembles the result

	Instructions	Registers	Control	Dcache
Raw	yes	yes	yes	yes
Trips	yes	hierarchical	no	no
WaveScalar	yes	yes	yes	no

Table 8.1. A comparison of the level of resource distribution between Raw, Trips, and WaveScalar. Each box answers whether a resource is distributed on an architecture.

of one effort to make clustered VLIW architectures more scalable via distributed instruction and data caches and allowing decoupled execution between the clusters [30]. Therefore, it is not surprising that much prior work in ILP orchestration can be found in VLIW research.

Of all prior work on ILP compilers, the Bulldog compiler [13] faces a problem that most resembles that of Rawcc. Bulldog targets a VLIW machine that distributes not only functional units and register files but memory as well, all connected together via a partial crossbar. Therefore, it too has to handle preplaced instructions. Unlike Rawcc, however, Bulldog does not have to handle distributed control or schedule communication.

Bulldog adopts a two-step approach, with an assignment phase followed by a scheduling phase. Assignment is performed by an algorithm called Bottom-Up Greedy (BUG), a critical-path based mapping algorithm that uses fixed memory and data nodes to guide the placement of other nodes. Like the approach adopted by the clustering algorithm in Rawcc, BUG visits the instructions topologically, and it greedily attempts to assign each instruction to the processor that is locally the best choice. Scheduling is then performed by greedy list scheduling.

There are two key differences between the Bulldog approach and the Rawcc approach. First, BUG performs assignment in a single step that simultaneously addresses critical path, data affinity, and processor preference issues. Rawcc, on the other hand, divides assignment into clustering, merging, and placement. Second, the assignment phase in BUG is driven by a greedy depth-first traversal that maps all instructions in a connected subgraph with a common root before processing the next subgraph. As observed in [28], such a greedy

algorithm is often inappropriate for parallel computations such as those obtained by unrolling parallel loops.

Most other prior work in ILP orchestration is on the management of instructions, the most performance critical resources, on clustered VLIW architectures. Much of this research focuses specifically on instruction assignment and relies on a traditional list scheduler to perform scheduling. These algorithms do not account for preplaced instructions, and they assume a uniform crossbar interconnect and thus do not perform placement like Rawcc does. The following three algorithms fall under this category. The Multiflow compiler [28] uses a variant of BUG described above. PCC [11] is an iterative assignment algorithm targeting graphs with a large degree of symmetry. It bears some semblance to the Raw partitioning approach, with a clustering-like phase and a merging-like phase. The clustering phase has a threshold parameter that limits the size of the clusters. This parameter is adjusted iteratively to look for the value that yields assignment with the best execution times. Because of this iterative process that requires repeated assignment and scheduling, the algorithm is time consuming. RHOP [9] is a hierarchical, multi-level assignment scheme. It consists of two stages: coarsening and refinement. Coarsening attempts to group instructions on the critical path, while offloading non-critical instructions to other clusters. During coarsening, the algorithm forms a hierarchical grouping of instructions. At the lowest level, pairs of dependent instructions are grouped together. Each subsequent level pairs up dependent instruction groupings from the previous level. This grouping continues recursively until the number of instruction groups is equal to the number of VLIW clusters. Refinement takes the result of coarsening and tries to improve its load balance by repeatedly moving instruction groups from one cluster to another.

Other instruction management algorithms combine instruction assignment with other tasks. UAS [34] performs assignment and scheduling of instructions in a single step, using a greedy, list-scheduling-like algorithm. CARS [17] uses an even more unifying approach that

performs assignment, scheduling, and register allocation in one step. This goal is achieved by integrating both assignment and register allocation into a mostly cycle-driven list scheduler. Convergent Scheduler [26] constructs a compiler from many simple passes that may be applied repeatedly. Each pass either addresses a specific issue (*e.g.*, distributing parallelism) or applies a specific heuristic (*e.g.*, critical path first). Leupers [27] describes an iterative approach based on simulated annealing that performs both scheduling and assignment on a VLIW DSP.

The MIMD task scheduling problem is similar to Raw's ILP orchestration problem, with tasks at the granularity of instructions. Rawcc draws its decomposed view of the problem from this source. Sarkar [38], for example, employs a three step approach: clustering, combined merging/placement, and temporal scheduling. Similarly, Yang and Gerasoulis [44] use clustering, merging, and temporal scheduling, without the need for placement because they target a machine with a symmetric network. Overall, the body of work on MIMD task scheduling is enormous; readers are referred to [1] for a survey of some representative algorithms. One major distinction between the problems on MIMD and on Raw is the presence of preplaced instructions on Raw.

The structure of Rawcc is also similar to that of the Virtual Wires compiler [4] for mapping circuits to FPGAs, with phases for partitioning, placement, and scheduling. The two compilation problems, however, are fundamentally different from each other because a Raw machine multiplexes its computational resources (the processors) while an FPGA system does not.

Many ILP-enhancing techniques have been developed to increase the amount of parallelism available within a basic block. These techniques include control speculation [19], data speculation [39], superblock scheduling [16], and predicated execution [3]. These techniques can be used to increase the size of the scheduling regions in Spats.

8.3 Control orchestration

Raw's techniques for handling control flow are related to several research ideas. Asynchronous global branching, first described in [25], is similar to autonomous branching, a branching technique for a clustered VLIW with an independent instruction cache on each cluster [5]. Both techniques eliminate the need to broadcast branch targets by keeping a branch on each tile or cluster.

It is useful to compare the result of control localization with predicated execution [29]. Control localization enables the Raw machine to perform predicated execution without extra hardware or ISA support, but at the cost of executing local branches. The cost of the local branches, however, can be amortized across instructions with the same predicates. In addition, control localization utilizes its fetch bandwidth more efficiently than predicated execution. For IF-THEN-ELSE constructs, the technique fetches only the path that is executed, unlike predicated execution which has to fetch both paths.

Control localization shares some techniques with hyperblock scheduling [3]. They both employ if-conversion so that parallelism can be scheduled across branches. They also both employ reverse if-conversion, but for different reasons. Control localization uses it to amortize the cost of local branches across multiple instructions, while hyperblock scheduling uses it to avoid an excessive amount of predicated code, which ties up compute resources and is potentially of no value.

The idea of localizing the effects of branches to a subset of compute nodes can be found in other research. Hierarchical reduction [22] collapses control constructs into a single abstract node in software, so that the compiler can software-pipeline loops with internal control flow on a VLIW. The idea can also be found in Multiscalar's execution model [39], where tasks with independent flows of control are assigned to execute on separate processors.

Chapter 9

Conclusion

This thesis describes techniques to compile ILP onto a TPA. It individually studies the implication of resource distribution on ILP, for each the following resources: instructions, registers, control, data memory, and wires. It designs novel solutions for each one, and it describes the solutions within the integrated framework of a working compiler. The techniques are implemented in the Rawcc compiler to target the Raw microprocessor, but they are also applicable to other full or partial TPAs.

Rawcc manages all the distributed resources, including ALUs, register files, cache memories, branches (control), and wires. Because the resources are distributed, the spatial management or resources is critical. Of particular importance is the spatial assignment of instructions to tiles, and the compiler handles this task by combining borrowed techniques from MIMD task scheduling with novel techniques that account for the non-uniform spatial distribution of resources and the presence of preplaced instructions. Rawcc also explicitly orchestrates the communication required for both operand transfer and control coordination.

Rawcc has been evaluated on both a simulator and the actual hardware. Results show that Rawcc is able to attain modest speedup for fine-grained applications, as well speedups that scale up to 64 tiles for applications with such parallelism. In addition, because Raw supports control decoupling, it is relatively tolerant to variations in latency that the compiler

is unable to predict.

TPAs are a solution to an increasingly important architectural problem: how to satisfy the increasing demand for resources without relying on large centralized resources or global wires that would lead to a slower clock. I envision that they will become more popular with existing technological trends. By showing that the compiler can orchestrate ILP on TPAs, this thesis helps make the case that TPAs are good candidates for general purpose computing.

Appendix A

Result tables and figures

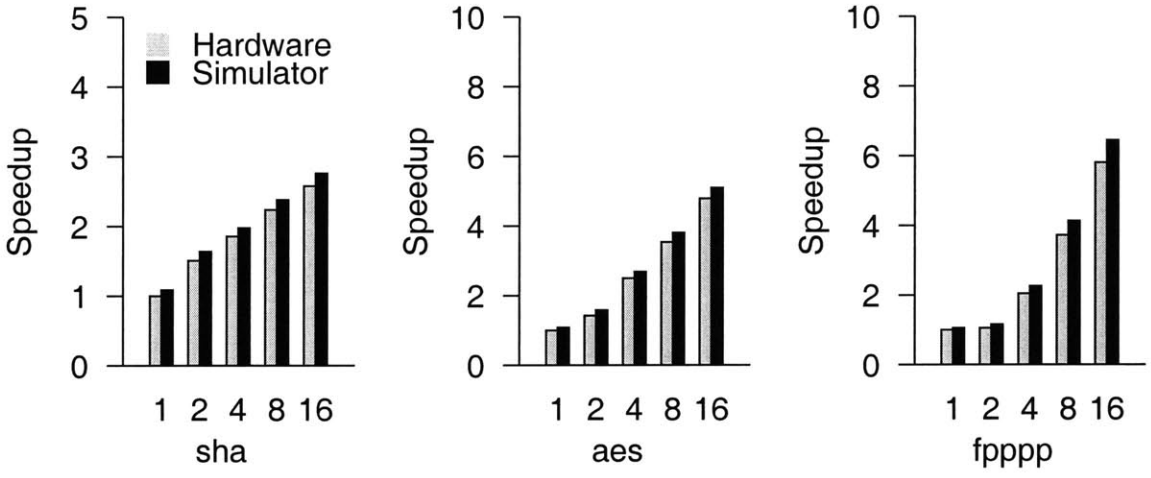


Figure A-1. Performance of Rawcc parallelized applications on the hardware and on the simulator, part I. This is a magnification of Figure 7-1.

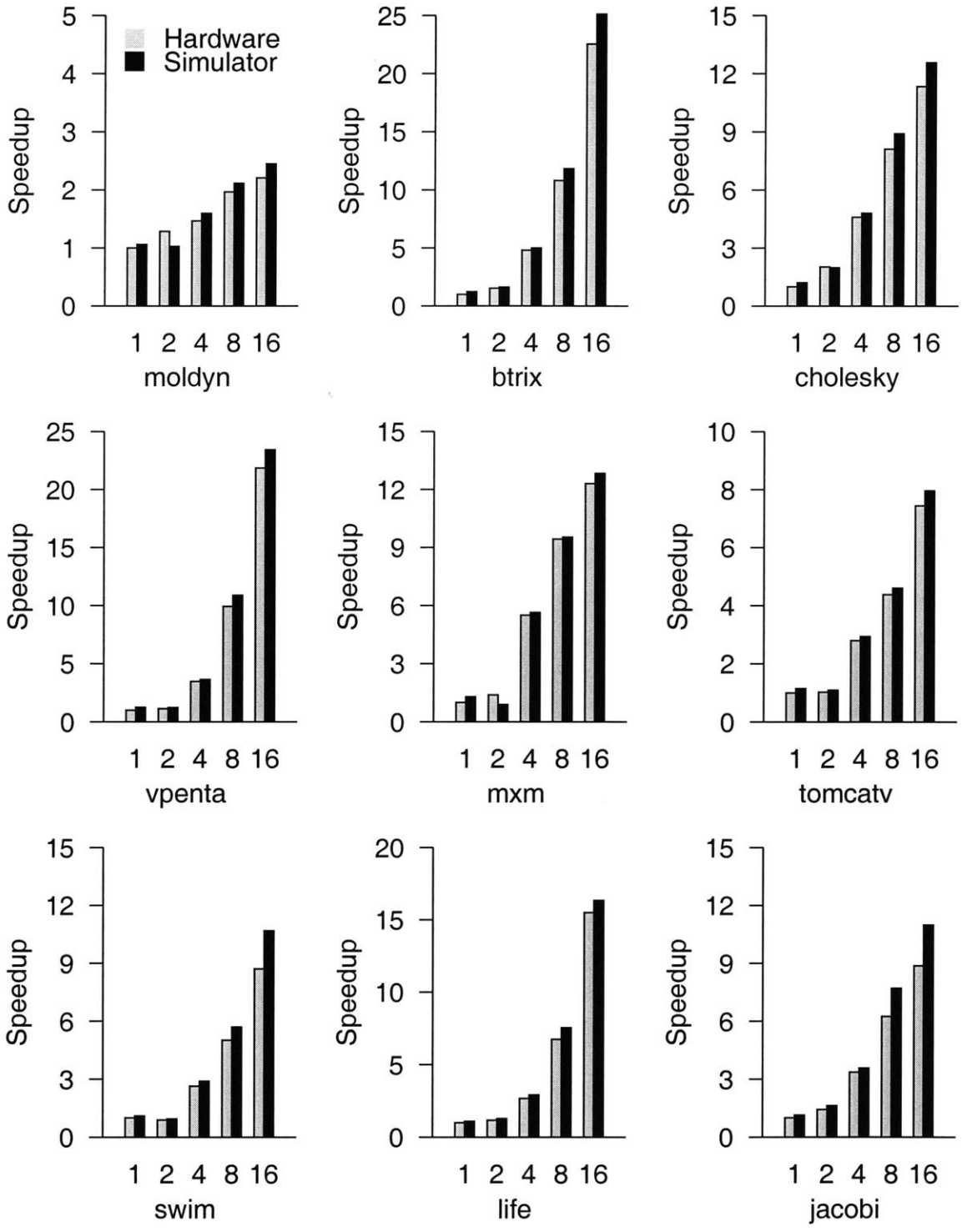


Figure A-2. Performance of Rawcc parallelized applications on the hardware and on the simulator, part II. This is a magnification of Figure 7-1.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32	N=64
sha	1.00	1.38	1.14	1.63	2.16	1.89	1.97
aes	1.00	1.48	2.05	3.52	4.23	3.73	3.74
fpppp	1.00	0.95	1.96	3.46	7.09	7.50	11.75
unstruct	1.00	1.67	3.11	2.97	3.43	3.98	4.01
moldyn	1.00	0.97	1.54	1.74	3.19	3.46	3.50
btrix	1.00	1.34	3.21	9.79	22.01	34.60	45.13
cholesky	1.00	1.31	2.99	5.29	8.03	10.21	10.87
vpenta	1.00	0.84	3.10	8.51	17.15	26.53	35.90
mxm	1.00	0.86	2.85	3.85	6.71	8.10	8.84
tomcatv	1.00	0.89	1.96	3.75	5.33	6.13	6.89
swim	1.00	0.92	1.69	3.22	5.22	11.44	16.19
jacobi	1.00	1.12	1.98	3.65	5.11	10.67	10.89
life	1.00	0.66	1.43	3.05	6.12	10.78	18.41

Table A.1. Benchmark speedup versus performance on a single-tile: pSpats.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32	N=64
sha	1.00	1.50	1.81	2.17	2.52	1.93	1.86
aes	1.00	1.45	2.46	3.49	4.66	4.36	4.22
fpppp	1.00	1.09	2.13	3.89	6.07	7.92	10.03
unstruct	1.00	1.65	2.97	3.32	3.57	3.44	3.32
moldyn	1.00	0.98	1.52	2.01	2.31	2.28	2.43
btrix	1.00	1.32	4.06	9.57	20.30	40.34	89.51
cholesky	1.00	1.63	3.93	7.29	10.28	10.14	9.80
vpenta	1.00	0.97	2.86	8.56	18.40	37.19	73.03
mxm	1.00	0.70	4.36	7.37	9.91	11.06	12.35
tomcatv	1.00	0.95	2.55	4.01	6.92	8.21	10.24
swim	1.00	0.86	2.62	5.16	9.67	18.68	28.58
jacobi	1.00	1.42	3.10	6.70	9.54	22.86	21.99
life	1.00	1.16	2.64	6.82	14.78	27.78	55.67

Table A.2. Benchmark speedup versus performance on a single-tile: Spats.

Benchmark	N=1	N=2	N=4	N=8	N=16	N=32	N=64
sha	1.00	1.38	1.14	1.63	2.16	1.89	1.97
aes	1.00	1.48	2.05	3.52	4.23	3.73	3.74
fpppp	1.00	0.95	1.96	3.46	7.09	7.50	11.75
unstruct	1.00	1.85	3.18	3.60	3.23	2.86	3.00
moldyn	1.00	1.26	1.88	1.78	1.57	0.93	0.81
btrix	1.00	1.72	5.55	15.67	34.65	67.60	0.00
cholesky	1.00	1.82	4.75	8.97	10.50	10.45	9.92
vpenta	1.00	1.21	5.16	13.69	30.93	61.92	123.06
mxm	1.00	1.16	4.54	7.39	9.32	10.70	11.59
tomcatv	1.00	1.29	2.92	5.20	7.95	9.56	10.41
swim	1.00	1.19	2.59	4.83	9.57	19.64	31.20
jacobi	1.00	1.66	3.24	7.32	9.08	22.62	22.96
life	1.00	0.95	2.36	5.87	12.75	22.26	49.49

Table A.3. Benchmark speedup versus performance on a single-tile: Convergent.

Benchmark	N=1	N=2	N=4	N=8	N=16
sha	1.00	1.51	1.86	2.24	2.58
aes	1.00	1.43	2.50	3.54	4.79
fpppp	1.00	1.06	2.05	3.73	5.81
moldyn	1.00	1.29	1.47	1.97	2.21
btrix	1.00	1.54	4.83	10.82	22.55
cholesky	1.00	2.03	4.61	8.13	11.36
vpenta	1.00	1.15	3.48	9.93	21.86
mxm	1.00	1.40	5.50	9.43	12.31
tomcatv	1.00	1.03	2.80	4.38	7.45
swim	1.00	0.89	2.64	5.02	8.72
jacobi	1.00	1.44	3.37	6.25	8.88
life	1.00	1.17	2.68	6.75	15.50

Table A.4. Benchmark speedup versus performance on a single-tile: Spats on chip

Benchmark	Ntiles	Hotuse	Oneuse	OneuseOOO	Muse	MuseOOO
sha	02	21.09%	40.24%	84.14%	38.65%	72.47%
sha	04	25.55%	55.75%	73.55%	18.68%	70.27%
sha	08	34.34%	51.16%	71.84%	14.49%	68.21%
sha	16	39.94%	50.10%	71.05%	9.94%	56.49%
sha	32	46.16%	46.39%	67.20%	7.43%	42.04%
sha	64	48.63%	45.65%	63.87%	5.70%	48.51%
aes	02	37.22%	44.99%	82.71%	17.77%	56.25%
aes	04	47.98%	45.78%	92.51%	6.23%	54.99%
aes	08	43.53%	50.50%	88.20%	5.95%	39.12%
aes	16	56.56%	42.51%	84.85%	0.91%	0.00%
aes	32	56.79%	42.76%	82.12%	0.43%	0.00%
aes	64	56.46%	43.53%	83.85%	0.00%	0.00%
fpppp	02	78.15%	14.84%	88.38%	6.99%	87.67%
fpppp	04	58.53%	29.00%	85.20%	12.46%	92.21%
fpppp	08	55.56%	32.26%	75.26%	12.16%	86.11%
fpppp	16	43.95%	45.71%	74.72%	10.33%	76.70%
fpppp	32	39.32%	53.75%	71.63%	6.92%	71.94%
fpppp	64	25.49%	71.70%	73.28%	2.80%	63.40%
unstruct	02	37.15%	24.14%	75.06%	38.69%	82.78%
unstruct	04	18.50%	45.46%	77.70%	36.03%	80.05%
unstruct	08	21.67%	51.04%	74.58%	27.28%	87.00%
unstruct	16	23.23%	52.67%	78.65%	24.08%	89.90%
unstruct	32	30.29%	50.54%	76.17%	19.16%	89.39%
unstruct	64	43.07%	41.09%	77.18%	15.82%	88.02%
moldyn	02	13.00%	47.43%	57.22%	28.15%	89.28%
moldyn	04	13.38%	49.36%	71.62%	24.09%	88.06%
moldyn	08	16.66%	50.23%	69.67%	20.98%	88.96%
moldyn	16	23.37%	47.82%	68.28%	17.63%	95.30%
moldyn	32	23.03%	51.40%	70.80%	13.76%	95.54%
moldyn	64	23.99%	54.38%	71.79%	10.04%	93.37%
btrix	02	7.71%	91.88%	68.04%	0.39%	75.85%
btrix	04	21.68%	77.04%	61.78%	1.27%	75.36%
btrix	08	28.96%	68.80%	39.34%	2.22%	68.07%
btrix	16	49.75%	47.17%	40.02%	3.06%	63.05%
btrix	32	70.67%	26.87%	54.90%	2.44%	51.58%
btrix	64	77.91%	20.09%	73.89%	1.98%	62.57%
cholesky	02	26.54%	70.06%	50.76%	3.37%	34.57%
cholesky	04	36.06%	55.74%	46.56%	8.18%	74.31%
cholesky	08	40.38%	59.22%	44.56%	0.37%	65.69%
cholesky	16	58.45%	41.22%	34.54%	0.31%	49.48%
cholesky	32	67.33%	32.39%	44.90%	0.26%	47.76%
cholesky	64	77.38%	21.74%	46.63%	0.86%	86.39%

Table A.5. Operand analysis data, part I, as shown in Figure 7-5.

Benchmark	Ntiles	Hotuse	Oneuse	OneuseOOO	Muse	MuseOOO
vpenta	02	15.45%	83.79%	70.68%	0.75%	76.55%
vpenta	04	26.01%	69.59%	66.83%	4.39%	83.78%
vpenta	08	30.83%	61.15%	67.06%	8.00%	72.07%
vpenta	16	43.97%	53.00%	52.11%	3.01%	66.02%
vpenta	32	53.80%	43.18%	59.68%	3.01%	81.34%
vpenta	64	71.39%	26.31%	53.18%	2.28%	57.35%
mxm	02	12.13%	87.18%	86.62%	0.54%	83.07%
mxm	04	39.00%	58.99%	80.55%	1.63%	33.33%
mxm	08	52.17%	46.39%	78.21%	1.11%	1.32%
mxm	16	72.46%	26.05%	90.36%	1.47%	40.18%
mxm	32	86.53%	12.22%	92.54%	1.01%	0.00%
mxm	64	95.14%	4.11%	90.01%	0.74%	22.14%
tomcatv	02	12.57%	80.40%	72.54%	3.22%	98.12%
tomcatv	04	22.94%	66.34%	61.85%	4.81%	90.94%
tomcatv	08	20.53%	68.29%	65.25%	4.01%	85.52%
tomcatv	16	29.21%	59.04%	65.72%	3.94%	71.92%
tomcatv	32	48.47%	41.42%	66.81%	3.26%	70.26%
tomcatv	64	51.19%	41.43%	77.72%	2.52%	79.04%
swim	02	19.41%	78.57%	75.26%	2.00%	36.89%
swim	04	54.09%	40.02%	68.32%	5.80%	80.26%
swim	08	53.61%	39.28%	67.55%	6.96%	95.43%
swim	16	64.68%	31.67%	66.27%	3.50%	89.90%
swim	32	70.60%	23.53%	67.05%	5.73%	93.52%
swim	64	78.78%	19.66%	56.30%	1.44%	80.65%
jacobi	02	49.77%	49.93%	60.13%	0.29%	86.66%
jacobi	04	66.68%	32.95%	87.56%	0.35%	9.09%
jacobi	08	87.17%	12.45%	52.22%	0.36%	25.00%
jacobi	16	86.39%	13.30%	85.23%	0.29%	20.00%
jacobi	32	91.77%	7.91%	87.90%	0.30%	18.18%
jacobi	64	95.05%	4.67%	90.32%	0.27%	18.18%
life	02	99.82%	0.09%	92.30%	0.07%	40.00%
life	04	72.59%	27.26%	99.94%	0.14%	50.00%
life	08	54.72%	45.15%	98.00%	0.12%	16.66%
life	16	53.05%	46.78%	99.04%	0.15%	21.73%
life	32	52.03%	47.81%	99.98%	0.15%	26.08%
life	64	64.76%	35.10%	99.99%	0.13%	9.09%

Table A.6. Operand analysis data, part II, as shown in Figure 7-5.

Bibliography

[1] Ishfap Ahmad, Yu Kwong Kwok, and Min You Wu. Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors. In *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 207–213, June 1996.

[2] S. Amarasinghe, J. Anderson, C. Wilson, S. Liao, B. Murphy, R. French, and M. Lam. Multiprocessors from a Software Perspective. *IEEE Micro*, pages 52–61, June 1996.

[3] David August, Wen mei Hwu, and Scott Mahlke. A Framework for Balancing Control Flow and Predication. In *1997 MICRO*, December 1997.

[4] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[5] Sanjeev Banerjia. *Instruction Scheduling and Fetch Mechanism for Clustered VLIW Processors.* PhD thesis, North Carolina State University, 1998.

[6] Rajeev Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures.* PhD thesis, M.I.T., Department of Electrical Engineering and Computer Science, January 2000.

[7] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *1998 HiPC*, December 1998.

[8] Ping Chao and Lavi Lev. Down to the Wire – Requirements for Nanometer Design Implementation. http://www.eet.com/news/design/features/showArticle.jhtml?articleId=16505500&kc=4235.

[9] Michael Chu, Kevin Fan, and Scott Mahlke. Region-based Hierarchical Operation Partitioning for Multicluster Processors. In *2003 PLDI*, pages 300–311, 2003.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth 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.

[11] Giuseppe Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: a New Approach. Technical Report HPL-98-13, Hewlett Packard Laboratories, 1998.

[12] Marios Dikaiakos, Anne Rogers, and Kenneth Steiglitz. A Comparison of Techniques Used for Mapping Parallel Algorithms to Message-Passing Multiprocessors. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, October 1994.

[13] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1986.

[14] Linley Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, October 1996.

[15] Ron Ho, Ken Mai, and Mark Horowitz. Managing wire scaling: A circuit perspective. In *IEEE Interconnect Technology Conference*, June 2003.

[16] Wen-mei Hwu, Scott Mahlke, William Chen, Pohua Chang, Nancy Warter, Roger Bringmann, Roland Ouellette, Richard Hank, Tokuzo Kiyohara, Grant Haab, John Holm, and Daniel Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1):229–248, January 1993.

[17] Krishnan Kailas, Kemal Ebcioglu, and Ashok K. Agrawala. CARS: A New Code Generation Framework for Clustered ILP Processors. In *2001 HPCA*, pages 133–143, 2001.

[18] Michael Kanellos. Sun kills UltraSparc V, Gemini chips, April 2004. http://news.com.com/2100-1006_3-5189458.html?tag=nefd.top.

[19] Vinod Kathail, Michael Schlansker, and B. Ramakrishna Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical report, HP HPL-93-80, February 1994.

[20] Kevin Krewell. 2004 in Review, December 2004. http://www.mdronline.com/mpr_public/editorials/edit18_52.html.

[21] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *1993 ICS*, pages 195–206, 1993.

[22] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *1988 PLDI*, pages 318–328, June 1988.

[23] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *1992 ISCA*, pages 46–57, May 1992.

[24] Sam Larsen and Saman Amarasinghe. Increasing and Detecting Memory Address Congruence. In *2002 PACT*, pages 18–29, 2002.

[25] Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *1998 ASPLOS*, pages 46–57, 1998.

[26] Walter Lee, Diego Puppin, Shane Swenson, and Saman Amarasinghe. Convergent Scheduling. In *2002 MICRO*, 2002.

[27] Rainer Leupers. Instruction Scheduling for Clustered VLIW DSPs. In *2000 PACT*, pages 291–300, 2000.

[28] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1):51–142, January 1993.

[29] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *1992 MICRO*, pages 45–54, 1992.

[30] Scott Mahlke. Distributed VLIW. http://cccp.eecs.umich.edu/research/dvliw.php.

[31] Jason E Miller. Software based instruction caching for the raw architecture. Master's thesis, Massachusetts Institute of Technology, May 1999.

[32] Samuel D. Naffziger and Gary Hammond. The implementation of the next-generation 64b Itanium microprocessor. In *2002 ISSCC*, pages 344–345,472, February 2002.

[33] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *2001 MICRO*, pages 40–51, 2001.

[34] Emre Ozer, Sanjeev Banerjia, and Thomas M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *1998 MICRO*, pages 308–315, 1998.

[35] Diego Puppin, Mark Stephenson, Saman Amarasinghe, Una-May O'Reilly, and Martin C. Martin. Adapting Convergent Scheduling Using Machine Learning. In *16th International Workshop on Languanges and Compilers for Parallel Computing*, October 2003.

[36] Rodric Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Memo TM-646, MIT LCS, June 2004.

[37] Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.

[38] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[39] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *1995 ISCA*, 1995.

[40] Steve Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *2003 MICRO*, pages 291–302, 2003.

[41] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *2004 ISCA*, pages 2–13, 2004.

[42] Elliot Waingold, Micheal Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *Computer*, pages 86–93, September 1997.

[43] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, MA, 1993.

[44] T. Yang and A. Gerasoulis. List Scheduling With and Without Communication. *Parallel Computing Journal*, 19:1321–1344, 1993.

[45] 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, 1994.