Spatial Instruction Scheduling for Raw Machines

by

Shane Michael Swenson

B.S., Computer Science and Engineering,
Massachusetts Institute of Technology (2001)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2002

Author ....
Department of Electrical Engineering and Computer Science
February 12, 2002

Certified by ....
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Spatial Instruction Scheduling for Raw Machines

by

Shane Michael Swenson

Submitted to the Department of Electrical Engineering and Computer Science
on February 12, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Instruction scheduling on software exposed architectures, such as Raw, must be performed in both time and space. The complexity and variance of application scheduling regions dictates that the space-time scheduling task be divided into phases. Unfortunately, the interaction of phases presents a phase ordering problem.

In this thesis, the structure of program scheduling regions is studied. The scheduling regions are shown to have varying characteristics that are too diverse for a single simple algorithm to cover. A new scheduling technique is proposed to cope with this diversity and minimize the phase ordering problem. First, rather than maintaining exact mappings of instructions to time and space, the internal state of the scheduler maintains probabilities for different assignments of instructions to time and space resources. Second, a set of small scheduling heuristics cooperatively iterate over the probabilistic assignments many times in order to minimize the effects of phase ordering.

A simple spatial instruction scheduler for Raw machines based on this technique is implemented and shown to outperform existing spatial scheduling systems on average.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

Acknowledgements go here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In order to maintain a high rate of performance improvement, modern microprocessors must continue to increase the number of instructions they execute per cycle by exploiting parallelism in applications. Unfortunately, this task is becoming increasingly difficult to perform at run time due to wire delays and the complexity of discovering and scheduling parallelism. Therefore, scalable microprocessor architectures of the future must give compilers control over not only when each instruction is executed, but on which processing element that execution takes place. The task of assigning instructions to resources and ordering them temporally, called space-time instruction scheduling, is critical to the performance of modern and future scalable architectures [12] [14]. This thesis explores the space-time instruction scheduling problem and offers a new approach based on weighted decision making and phase-unification. A spatial scheduling system based on this approach is implemented and targeted for the MIT Raw architecture. This system yields favorable results when compared with the existing spatial scheduling algorithms.

1.1 Software-Exposed Architectures

In order to maintain an increasing rate of performance improvement, microprocessor designers have developed a number of features designed to let modern processors exploit instruction level parallelism (ILP) in sequential programs. Currently, the majority of these modern architectures attempt to maintain the external interface of a single serial processor. Unfortunately the hardware resources required to exploit ILP while maintaining this interface tend to have quadratic complexity and cannot scale without a sacrifice in cycle time.

One solution to this scaling problem is to remove the restriction that the processor maintain the single serial processor interface and shift the responsibility for discovery and exploitation of ILP from hardware to software. Architectures of this kind are referred to as *software-exposed architectures* [5]. The burden of achieving high performance on a software-exposed architecture is placed almost entirely on the compiler, which determines how the hardware resources will be used through space-time instruction scheduling and register allocation.

1.2 Space-Time Instruction Scheduling

On a conventional monolithic processor, the instruction scheduling problem addressed by the compiler is purely temporal; the instructions must be ordered in time to form a schedule that minimizes completion time while satisfying data dependencies. The distributed nature of the computation and storage elements on a software exposed architecture and their exposure to the compiler, however, generalizes the static instruction scheduling problem to one that is both temporal and spatial. Thus the compiler must explicitly schedule ILP across the available computation resources while considering the trade-off between communication latency and completion time for instructions executed in parallel or in sequence. This is called *space-time instruction scheduling* [12].

To simplify the space-time instruction scheduling task, many compilers divide it into phases. For example, the general purpose compiler for MIT Raw machines [12] divides this task into the phases shown in Figure 1-1. First the instructions are partitioned into sets that can be executed in parallel. Then the sets of instructions are placed on specific computation units. Finally, the instructions assigned to each computation unit are scheduled temporally. After the schedule is finalized, the compiler

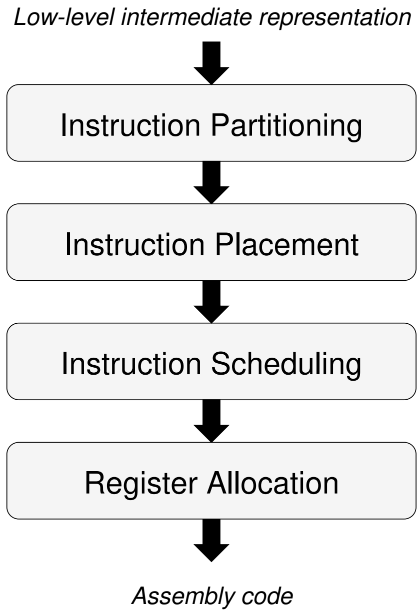

Figure 1-1: Back-end phase diagram for the MIT Raw compiler

assigns registers to program variables and temporary values and outputs the assembly code.

1.3 Phase Ordering Problem

This division of the back-end code generation tasks of space-time instruction scheduling and register allocation into multiple phases reduces the engineering and computational complexity of the tasks to a tractable level. Unfortunately decisions made in each phase often affect the outcomes of other phases in hard to predict ways. Furthermore, most decompositions of these tasks into phases suffer from the *phase ordering problem*, meaning that different program instances will benefit from different orders of the phases. For example, in some instances register pressure provides the main performance bottleneck and early register allocation will produce the best code. In others, instructions with long delays, such as I/O accesses, hamper performance, meaning that priority should be given to temporal scheduling in order to minimize the effects of those delays.

The phase ordering problem has two principle causes. First, the decisions made by each phase tend to be irreversible. For example, when the scheduler chooses a particular ordering of instructions, the register allocator is usually unable to make changes to the ordering in order to improve register allocation. When such violations of the phase interfaces are allowed, they must be very limited, with each phase only given a small number of allowable modifications to the work of other phases. This is necessary to keep the complexity of the individual phases under control since the task of building a particular code generation phase that can effectively do the work of the other phases as well is identical to the problem of building a single-phase code generator.

The second cause of the phase ordering problem is the lack of information sharing between phases. Generally each compilation phase passes only immediately relevant information to the next phase in line. The instruction partitioner outputs only a list of sets of partitioned instructions, the instruction placer outputs only a mapping of sets of instructions to function units, the instruction scheduler outputs only an ordering of the instructions or a mapping of instructions to time slots, and the register allocator outputs only a mapping variables to registers. The problem with these simple interfaces is that the phases often make arbitrary or uninformed decisions. For example, the partitioner may have to choose between two partitions with similar cost function values though one of them will perform much better due to the way it is affected by scheduling or register allocation. The partitioner cannot always choose correctly because it does not know how the decision will affect the future phases. Making the decision by completing the code generation on both paths and comparing the results quickly becomes intractable if the number of decisions made in this manner is not kept extremely small. Similarly, passing both options to the future phases and allowing them to make the decision also quickly becomes intractable as the number of equivalent options to be passes will very likely be combinatorial in the size of the input.

1.4 Combined-Phase Probabilistic Scheduling

This thesis offers a new approach to space-time instruction scheduling, that can overcome both principle causes of the phase ordering problem given in Section 1.3. To alleviate the irreversible decision problem, this approach uses a map of probabilities to represent the assignment of instructions to processing resources and time slots in place of a strict boolean mapping. Thus when a phase makes a decision that it has low confidence in, it gives low weight to the results of that decision, leaving the door open for future phases to reverse the decision. In the case where a phase makes a bad decision and assigns a high weight to it, future phases can still reverse the decision by decreasing its weight. The unshared information problem is mitigated by allowing all phases to examine the entire probability map and see information resulting from decisions that did not lead to the final output of the previous phases. When a phase is unsure about a decision, it also gives some weight to the alternatives, which can then be examined by future phases.

In the development of this approach, four steps were taken. First, the structure of certain representative programs was studied empirically. This analysis demonstrated that the variance in program structure was too great for a single spatial scheduling algorithm to effectively cover all cases, resulting in the phase-ordering problem. The combined-phase probabilistic scheduling approach was then developed to solve the space-time scheduling problem while minimizing the effects of the phase-ordering problem. Using this approach, a simple spatial scheduling phase was implemented, tested, and evaluated on the MIT Raw Machine [19].

1.5 Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces the Raw processor and RAWCC, the sequential compiler into which the work of this thesis is integrated. In Chapter 3 the characteristics of spatial instruction scheduling problem instances are examined in detail with the assistance of graph visualization tools.

The new combined-phase probabilistic scheduling approach developed for this thesis is presented in Chapter 4. A spatial scheduling algorithm based on this approach is detailed in Chapter 5. The results of an implementation of this algorithm on the Raw benchmark suite are provided in Chapter 6. Chapter 7 analyzes some of the advantages and disadvantages of the combined-phase probabilistic scheduling approach. Finally, Chapter 8 describes work related to the spatial instruction scheduling problem for Raw machines and Chapter 9 concludes.

Chapter 2

Compiling for the Raw Architecture

The Raw architecture is a distributed microprocessor architecture designed to scale beyond the limits of current high performance processor architectures [1]. A Raw machine is composed of a two dimensional mesh of simple replicated tiles. Each tile contains a processing unit, a memory bank, and a switch. The tiles execute independent instruction streams, though they are loosely synchronized by control flow and instruction dependencies.

RAWCC is the parallelizing C and FORTRAN compiler for Raw machines. It is implemented on top of the SUIF compiler infrastructure [20] and contains two main components. First, memory disambiguation [5] is performed on the memory usage of the program in order to discover memory accesses that are statically resolvable to a particular bank. Then the space-time scheduler [12] parallelizes the computation by assigning instructions to tiles, temporally scheduling the instructions assigned to each tile, and orchestrating any communication between tiles required for the correctness of the computation.

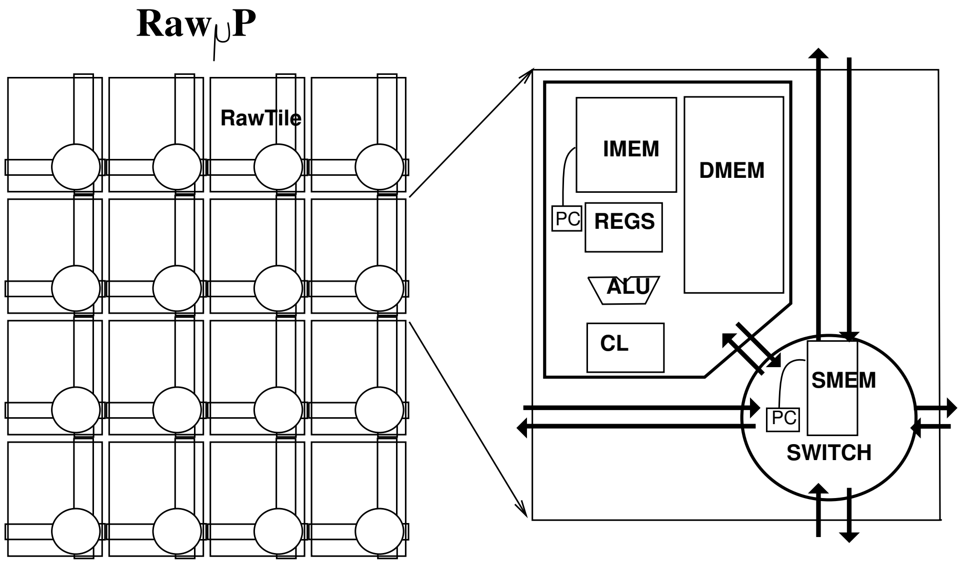

Figure 2-1: A Raw Microprocessor

2.1 Raw Architecture

The MIT Raw architecture [19] is a simple, highly scalable software-exposed architecture. A Raw machine, shown in Figure 2-1, is composed of an interconnected set of identical tiles that execute independent instruction streams. Each tile contains a simple five-stage RISC pipeline along with a portion of the machine's cache memory and is interconnected with the other tiles over a pipelined, point-to-point network. The tiles are kept simple and small in order to maximize the number of tiles that can fit on a chip and to facilitate a high clock rate. The interconnect network is integrated tightly with the processor on each tile to provide fast, register level communication. Unlike modern superscalar architectures, this communication network is fully exposed to the software.

2.1.1 Processing Unit

The main processing unit on each tile is a simple RISC pipeline that implements the MIPS R4000 instruction set and a small number of additional network access instructions. Table 2.1 lists latencies for the basic instructions supported by the

Table 2.1: Tile processor latencies

Instruction	Latency
Load	2 cycles
Store	1 cycle
Integer add or subtract	1 cycle
Integer multiply	2 cycles
Integer divide	36 cycles
Floating point add, subtract, or multiply	4 cycles
Floating point divide	10 cycles

processing unit. All operations, except for floating point divides, are pipelined.

The network access instructions control register-level communication between the main processing unit and the switch. The input ports on the processing unit are mapped into the register file name space so values from the network can be referenced as ordinary registers in any instruction that takes input from a register. The output port uses dedicated data pathways that are activated by the "S" bit, which is encoded into the opcode. These data pathways are integrated directly into the bypass network on the processing unit, allowing the processor to communicate values to the switch as soon as they are available, rather than waiting for them to reach the writeback stage of the pipeline. Using this feature, a word of data can be communicated between neighboring tiles in only 3 cycles.

2.1.2 Communication Network

The point-to-point communication network on a Raw machine is implemented through a dedicated switch processor on each tile. Each switch is connected to its processor and its four neighboring switches via an input and output port. Ports on the perimeter of the machine are used for communication with external devices and memory banks.

The network supports both static and dynamic routing. Static routing eliminates the need to compose and route dynamic message headers and allows messages as small as a single word to be efficiently communicated between tiles. This enables the Raw compiler to take advantage of very fine grained ILP in sequential programs. The dynamic network is a slower, more traditional, runtime-routed network that can handle unpredictable communication more easily than the static network.

Static Network

The static network is comprised of the static switch on each tile, the ports and wires that connect them to their respective processors, and the ports and wires that connect them to the switches on their neighboring tiles. Each switch processor on the static network can be programmed, giving the software full control over communication on the static network. A word of data in this network can travel across one tile in one clock cycle so the total time to communicate a single word between two tiles is the Manhattan distance between them plus one cycle to write the word into the network and one cycle to read it.

The compiler uses this network to communicate data when both the source and destination tiles are known at compile time. This includes all communication between instructions due to data dependences since the compiler chooses which tile each instruction is assigned to. Thus most of the communication occurs on this network.

For the static network to route data correctly, the instruction stream of each switch along the path must contain instructions to correctly pass the data to the next switch on the route. This low overhead approach decreases communication latency and increases available bandwidth by eliminating the need to create, send, and later decode a message header for each message. Furthermore, the static scheduling of routes means that routing channels can be set up before the arrival of the data words destined for them.

In order to assist the static network in tolerating events of unknown or variable latency, and to allow the compiler some slack in scheduling communication, the static network implements near-neighbor flow control through blocking semantics on all read and write ports. A write instruction will always block until the output buffer has space available, and a read instruction will always block until a value is waiting to be read on the requested port. This eliminates the need for the precise timing that would otherwise be required to prevent switches from overwriting values on output ports or reading garbage on input ports. Blocking semantics allow the tiles to execute instruction streams independently and loosely synchronized, rather than operating in lock-step as is the case with VLIW machines. This flexibility increases Raw's tolerance of dynamic events with unpredictable delays, such as cache misses and I/O events, over that of an architecture that requires lock-step synchronization of program counters.

One important property of communication in the static network is that it is *statically ordered*. This means that the relative order of arrival of messages at each port on each tile is specified at compile time and guaranteed at run time. This property holds by induction because the compiler fixes the order in which each switch processes the messages it receives.

Dynamic Network

The dynamic network uses a dynamic switch on each tile, which performs traditional worm-hole routing by making decisions based on the header of each message. This network is used when either the source or destination of a message is unknown at compile-time.

The dynamic network is much slower than the static network for a number of reasons. First, messages on the dynamic network must carry a header which contains additional information, such as the destination tile and a message ID. This header must be constructed, inserted into the network, routed, and handled by the destination tile, using processing resources in every step. Second, since the arrival time and order of messages on the dynamic network is unpredictable, the receiving tiles are required to use expensive mechanisms, such as polling or interrupts, to receive them. Third, due to the lack of flow control on the dynamic network, messages may need to be spilled to memory if more arrive at a tile than can fit in the incoming message queue.

Since RAWCC chooses the assignment of instructions to tiles, all communication between instructions can be performed on the static network. Section 2.2.2 describes the alignment analysis which enables RAWCC to statically infer the resident tile for many memory references, allowing on-chip memory references to be performed with reduced use of the dynamic network. Through this extensive use of the static network, Raw machines can exploit very fine grained ILP in sequential programs.

2.1.3 Distributed Memory

The cache memory for Raw processors is distributed across the tiles. This enhances the scalability of the Raw architecture in many ways. First, by dividing the memory into multiple banks, the wires are kept short and higher clock speeds can be achieved. Second, since the memory banks are distributed among the processing resources, the software may be able to take advantage of locality and keep computations, memory references, and the banks they refer to close to each other in order to decrease memory latency. Third, each tile that is added to the machine increases the on-chip memory capacity and bandwidth without sacrificing clock speed. Fourth, if the memory and instructions can be parallelized while preserving locality, the full available processing power and memory bandwidth can be used without suffering from the high memory latency due to long wires and complex arbitration logic usually found on architectures with monolithic memory structures.

Raw is a *bank-exposed architecture* [5], meaning that it exposes its memory banks to the software, and that memory references can be explicitly directed to different banks at compile-time. Bank-exposed architectures have two primary scaling advantages over hardware-based unified systems: they can avoid the non-scalable delays associated with hardware arbitration logic and poor on-chip locality. However, these advantages are only gained when most of the memory instructions are *bank disambiguated*, meaning that they reference known banks at compile-time. Since the backup arbitration logic, the dynamic network in the case of Raw, is often much slower than dedicated hardware arbitration logic, the compile-time task of discovering which bank each memory instruction refers to is critical to the performance of these systems. RAWCC uses loop unrolling and alignment analysis [10] to perform this task, known as *bank disambiguation*.

2.2 Memory Bank Disambiguation

A bank-exposed architecture, such as Raw, can only achieve a significant performance advantage over a traditional architecture if most of the memory references are bank disambiguated. The memory accesses must also be distributed among the banks so that maximum memory bandwidth can be used. RAWCC uses two techniques to satisfy these criteria: *equivalence class unification* and *loop unrolling*.

2.2.1 Equivalence Class Unification

Equivalence class unification [5] begins with a pass by the pointer analysis package, SPAN [15]. SPAN assigns a unique *location set number* to each abstract object in the program. An abstract object is either a variable declaration allocated on the stack or a group of dynamic objects created at a single heap-memory allocation call site. All elements of a single array belong to the same abstract object, but the individual fields of a struct are each considered separate objects. After assigning location set numbers to abstract objects, SPAN assigns to each memory reference instruction a *location set list* containing the location set numbers corresponding to the abstract objects that the memory reference instruction may refer to.

After the pointer analysis is completed, RAWCC generates the program's *alias equivalence classes*. Alias equivalence classes form the finest partition of the location set numbers such that each memory reference instruction refers to location set numbers in only one equivalence class. To find the alias equivalence classes, RAWCC generates a bipartite graph with one node for each abstract object and each memory reference. Edges are added to the graph between each memory reference and all abstract objects whose location set numbers are in that reference's location set list. RAWCC then finds the connected components of the bipartite graph and for each component constructs one equivalence class containing the memory references in that component.

Finally, RAWCC assigns each equivalence class to a single tile. Since references in different equivalence classes may never refer to the same object, equivalence class unification disambiguates all memory references while maintaining program correctness.

2.2.2 Loop Unrolling and Alignment Analysis

Equivalence class unification correctly disambiguates memory references; however, it is forced to map each array to a single tile, reducing opportunities for ILP exploitation in loops. RAWCC uses loop unrolling and alignment analysis [10] to achieve bank disambiguation and memory bank load balancing for array references with index expressions that are affine functions of inner loop induction variables.

When such an array is discovered, RAWCC assigns is elements to the Raw tile memory banks in a round-robin fashion. The enclosing loop is then unrolled by the minimum number of iterations required to guarantee that each array reference in the unrolled loop can be bank disambiguated.

On a software-exposed architecture, such as Raw, this provides opportunities for ILP exploitation in two ways. First, the low-order interleaving of array elements across memory banks means that multiple elements can be accessed simultaneously. Second, the unrolled loop produces a larger basic block than the original version which gives the compiler a greater opportunity to schedule instructions in parallel. However, to achieve maximal performance benefit from modulo unrolling, the space-time scheduler must generate code that effectively exploits ILP and takes advantage of the increased opportunity for locality and memory bandwidth usage offered by memory bank disambiguation.

2.3 Space-Time Scheduler

The space-time instruction scheduler in RAWCC performs the task of assigning instructions in each forward control flow region, or *scheduling region*, to processor resources and ordering them in time so that the computation executes correctly and completes as quickly as possible. Since this module is responsible for the discovery and exploitation of ILP, its effectiveness is critical to achieving the goal of high performance generated code. The RAWCC space-time scheduler divides its tasks into two phases: spatial scheduling and temporal scheduling.

RAWCC further decomposes the spatial instruction scheduling problem into two components: partitioning and placement. Instruction partitioning is the task of identifying ILP in a sequential instruction stream and dividing that stream into parallel streams, while placement is the related task of assigning the parallel streams to individual tiles. To partition the instructions, RAWCC first forms clusters of instructions that have no exploitable parallelism given the communication cost of the target processor. Then clusters with high levels of communication between them are merged until the number of clusters equals the number of available tiles. To perform placement, RAWCC then creates a bijection between tiles and sets of partitioned instructions using a greedy algorithm that attempts to minimize the incurred cost of communication over the entire schedule.

After the instructions have been spatially scheduled, RAWCC uses a greedy list scheduling algorithm to construct the temporal schedule for each tile. This scheduler is also responsible for scheduling the communication instructions required to satisfy instruction precedence and data dependence constraints in a way that is guaranteed to produce a computationally correct, deadlock-free schedule.

This approach to spatial instruction scheduling produces promising results though it has one significant shortcoming. Many of the instructions to be scheduled are disambiguated memory references that must be executed on a specific tile, and using this system, RAWCC is unable to efficiently consider information about pre-placed instructions when generating a schedule. Due to the non-uniformity of the Raw communication network, long communication delays are often introduced when the results of these memory references are transmitted to the tiles that need them.

This system also suffers from the phase ordering problem. One particular problem is a result of the division between spatial and temporal scheduling. When the partitioner seeks to minimize communication between partitions, it sometimes creates partitions that cannot be executed in parallel and must be executed sequentially instead. This defeats the purpose of partitioning.

The spatial scheduling system described in Chapter 5 addresses the pre-placed instruction and phase ordering problems faced by RAWCC. The pre-placed instructions are used as hints for determining the structure of the instruction stream to be scheduled and often point to areas of exploitable ILP. The phase ordering problem is addressed using a combined-phase probabilistic scheduling approach that takes temporal scheduling information into account when spatial scheduling is being performed.

Chapter 3

Empirical Analysis of Program Characteristics

The problem of performing optimal space-time instruction scheduling for software-exposed architectures is NP-hard. Since a tractable algorithm for exact solutions is unlikely, a heuristic approach leading to approximate solutions is required. In this chapter the structure of the individual scheduling regions taken from benchmark applications is examined and analyzed. This analysis yields an understanding of the program characteristics critical to effective space-time scheduling. The conflicting needs presented by these characteristics are then used to derive the structure of the phase ordering problem.

3.1 Benchmark Suite

Table 3.1 describes the set of benchmark applications used in this thesis. With the exception of SHA, they were extracted from the SPEC [6], Rawbench [3], and Mediabench [11] benchmark suites. The version of SHA used in this thesis was provided by Matt Frank. The benchmark set includes dense matrix applications, multimedia applications, and an application without regular memory usage.

With the exception of Life-static and Mxm, no ILP-enhancing modifications were made to the benchmarks, though a small number of other modifications were per-

Table 3.1: Benchmark characteristics. Sequential time is the run-time for uniprocessor code generated by the Machsuif MIPS compiler [18]

Benchmark	Type	Source	Language	Lines of code	Seq. time (cycles)	Primary Array size	Description
Life-perf	Dense Matrix	Rawbench	C	136	258K	32×32	Conway's Game of Life
Cholesky	Dense Matrix	NASA7 (Spec92)	FORTRAN	175	113K	$4 \times 4 \times 16$	Cholesky Decomposition & Substitution
Tomcatv	Dense Matrix	Spec92	FORTRAN	267	261K	18×18	Mesh Generation with Thompson's Solver
Vpenta	Dense Matrix	NASA7 (Spec92)	FORTRAN	192	175K	16×16	Inverts 3 Pentadiagnols Simultaneously
Mxm	Dense Matrix	NASA7 (Spec92)	FORTRAN	58	1.61M	32×64, 64×8	Matrix Multiplication
Adpcm	Multimedia	Mediabench	C	302	340K	1024	Speech Compression
SHA	Multimedia	Matt Frank	C	615	1.21M	512×16	Secure Hash Algorithm
Fpppp-kernel	Irregular	Spec92	FORTRAN	1013	1.92K	–	Electron Interval Derivatives
Jacobi-small	Dense Matrix	Rawbench	C	69	39.5K	16×32	Jacobi Relaxation

formed. Currently the Raw architecture does not support double-precision floating point arithmetic, so all floating point operations were converted to single-precision. In addition, data set sizes were reduced to improve simulation time. Table 3.1 lists the data set sizes used for each benchmark in this thesis.

Since the outer loop of Mxm had already been unrolled four times for increased performance on VLIW machines, this unroll factor was increased to 16 to improve performance on 8 and 16 tile Raw machines. Due to the small data set size, Life contained a disproportionately large number of dynamic memory references. These dynamic references were made static through array padding.

3.2 Visualizing Scheduling Regions

A direct approach to understanding the structure of program scheduling regions is to look at them. Through visual analysis, the characteristics of the space-time scheduling problem can be examined in detail. Heuristics that effectively exploit these characteristics can then be developed to approximately solve the problem.

For visualization purposes it is convenient to represent a scheduling region by

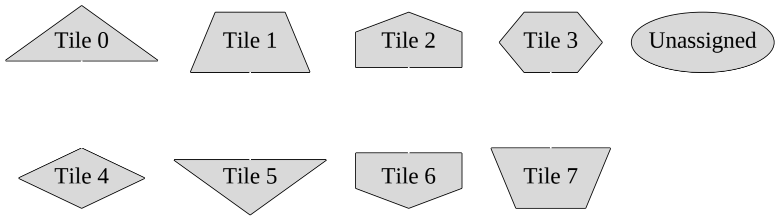

Figure 3-1: Shapes of nodes assigned to specific tiles

a directed acyclic graph with nodes representing instructions and edges representing dependencies. To simplify the layout, only true dependencies, edges from instructions that produce values to instructions that use them, are included. Output, and anti-dependences are omitted from the visual representation as they clutter the image and often obscure the underlying structure of the graph.

Once the scheduling region is in the proper form, the dot [9] directed graph layout system is used to generate an image. When the image is used for screen viewing or color printing, nodes can be shaded with different colors to signify placement on different tiles. In this chapter, however, nodes assigned to distinct tiles are differentiated by their shape. Figure 3-1 enumerates the shapes used for instructions assigned to particular tiles in this thesis.

Many of the scheduling regions studied in this thesis contain a small number of instructions that copy a value for the use of a very large number of other instructions. Figure 3-2 shows a region with this behavior. It was taken from Cholesky compiled for eight tiles. Removal of the edges representing these copy operations simplifies the graph structure and allows the layout engine to more clearly display that structure, as seen in Figure 3-3. Whenever a graph for a scheduling region has been modified in this way, the caption will contain a string of the form, "(n)", where n is the number of copy instructions whose outgoing edges have been removed. The scheduling region graphs in this thesis contain directed edges that point from left to right or top to bottom depending on the aspect ratio of the graph. Note that appendix A contains full-page representations of all scheduling region graphs displayed in this chapter.

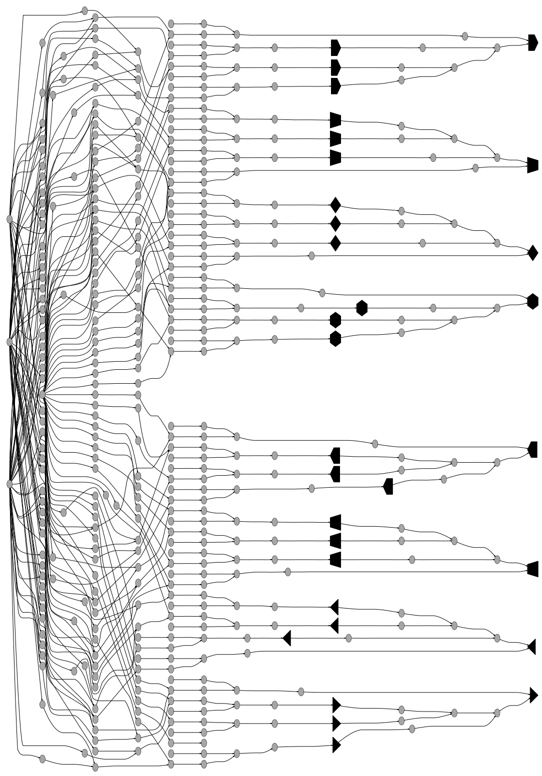

Figure 3-2: Cholesky 8 tiles

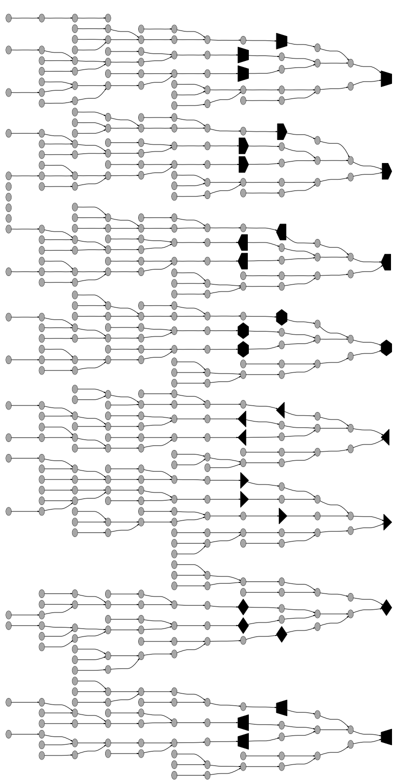

Figure 3-3: Cholesky 8 tiles (4)

3.2.1 Choice of Scheduling Regions

Most of the benchmark programs contain tens or hundreds of scheduling regions meaning that a subset must be chosen for detailed study. The scheduling regions chosen for presentation in this thesis are derived from the inner loops of the benchmark applications. Analysis of these regions is important for two reasons. First, applications tend to spend the majority of their execution time in the inner loops, meaning that optimizing the schedules for the bodies of these loops is essential for rapid program execution. Second, the innermost loops are often unrolled by the compiler. Thus, the scheduling regions derived from these inner loops are generally the largest regions for each application and provide the greatest source of parallelism available in the application.

3.2.2 Dense Matrix Applications

All of the dense matrix applications examined in this thesis contain scheduling regions similar to the region taken from Mxm, compiled for 8 tiles, shown in Figure 3-5. This type of scheduling region is the result of performing loop unrolling on a loop that indexes an array or set of arrays using affine transformations of induction variables. The eight unrolled loop iterations present in this scheduling region are disconnected from each other after the initial copy edges are removed. Thus there are no dependencies between unrolled iterations of the loop, allowing them to be executed in parallel as long as the memory usage pattern does not create bottlenecks.

The scheduling region shown in Figure 3-6, taken from Vpenta compiled for 8 tiles, exhibits similar structure to the previously examined regions from Cholesky and Mxm. Once again, the loop has been unrolled and there is no communication between iterations. Two multicast nodes with nearly 200 children each were removed from this graph to simplify the layout.[1] This region has the property that for each iteration of the original loop, all static memory references are confined to a single tile. Thus it is

[1] The communication code generator for RAWCC will allow at most one copy of a particular value to be sent to each tile, and since the children of these nodes are likely to be distributed across all tiles, the communications become simple broadcasts.

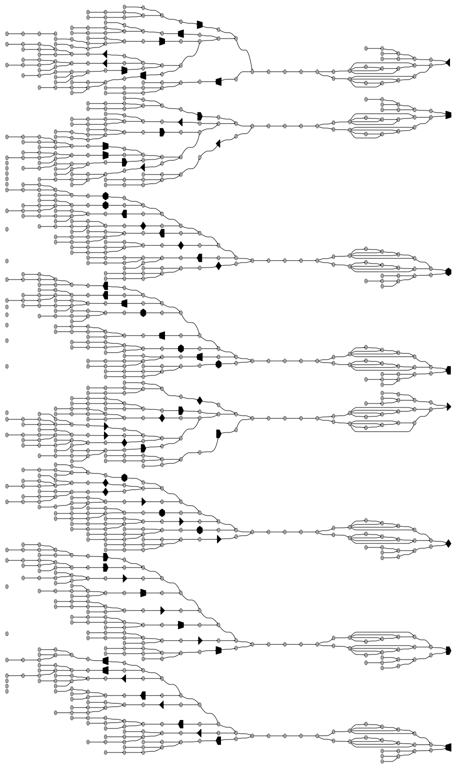

Figure 3-4: Life 8 tiles (2)

possible to execute this entire region without performing any communication on the static network after the initial multicasts have taken place. This property is shared by the Cholesky, Tomcatv, and Vpenta benchmark applications.

3.2.3 Multimedia Applications

Unlike the parallel structure exhibited by the dense matrix applications, the multimedia applications examined in this thesis exhibit highly serial structure. Representative scheduling regions taken from Adpcm and SHA are shown in Figure 3-7 and Figure 3-8 respectively. Any exploitable ILP in these regions has much finer grain and less regular structure than the available parallelism found in the dense matrix applications. These applications will therefore derive an extremely limited performance benefit from increases in the number of available tiles. After detailed inspection of the scheduling region in SHA, we estimate that it contains at most three or four-way parallelism.

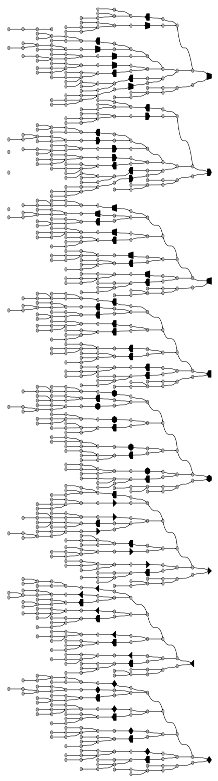

Figure 3-5: Mxm 8 tiles (3)

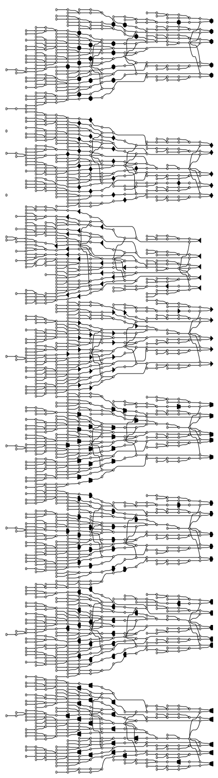

Figure 3-6: Vpenta 8 tiles (2)

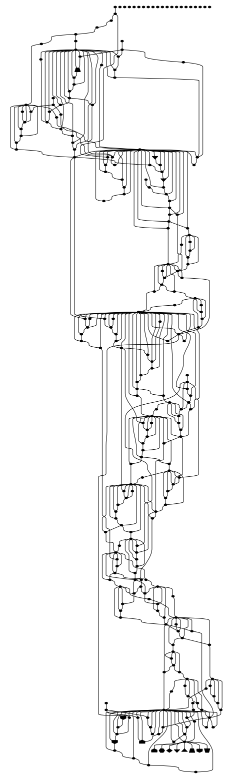

Figure 3-7: Adpcm 8 tiles

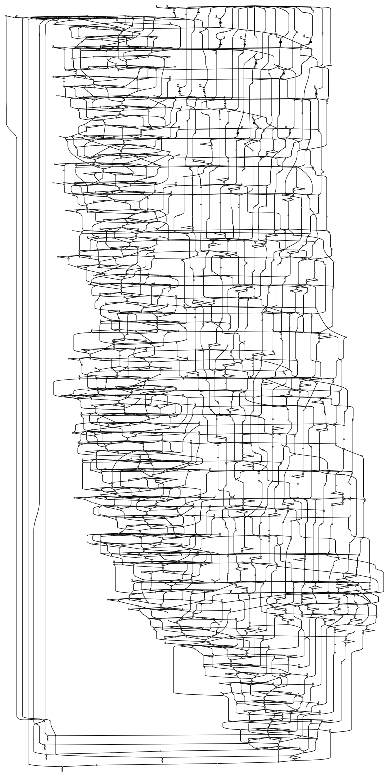

Figure 3-8: SHA 2 tiles

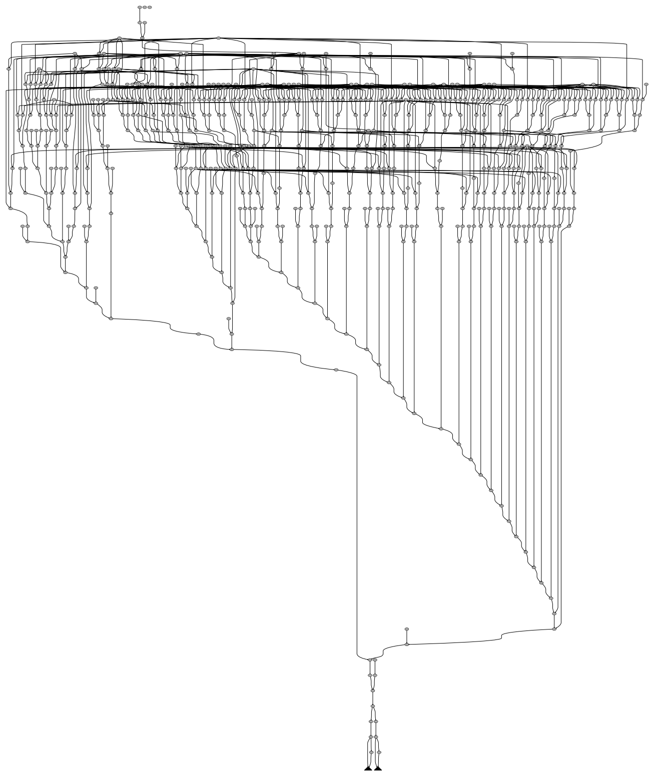

Figure 3-9: Fpppp 8 tiles

3.2.4 Irregular Applications

The Fpppp-kernel is the only application examined in this thesis without regular memory usage. Figure 3-9 displays the single scheduling region contained in this kernel. While the kernel contains a large amount of ILP, it is more fine-grained than the ILP found in dense matrix applications. Another point of interest in this region is the long reduction that dominates the lower half of the graph. A transformation of this reduction from its linear form to a tree topology could improve the performance of this region when it is compiled for a large number of tiles.

3.3 Discussion

Visual examination of scheduling region graphs offers many clues for effectively solving the space-time scheduling problem for software-exposed architectures.

3.3.1 Pre-Placed Instructions

The memory reference distribution and disambiguation efforts of RAWCC result in many static memory references. Since the memory banks accessed by these instructions are known at compile time, each static memory reference instruction is assigned to the tile containing the memory bank it refers to. These instructions are called *pre-placed instructions*. If instruction pre-placements are ignored, particularly when performing spatial scheduling, a large amount of unnecessary communication is often the result.

The instruction partitioning module in the initial implementation of RAWCC ignores instruction pre-placement information. Though the instruction placement module considers the locations of pre-placed instructions when mapping the partition to tiles, the pre-placed instructions are rarely mapped to the same tile as their neighbors due to uninformed decisions made by the partitioner. This increases the delay associated with memory references, making the temporal scheduling task more difficult and potentially eliminating some of the most efficient schedules from consideration. Also, since these memory references usually occur at the same point in each of the original loop iterations, network congestion delay may become a factor if all of the tiles attempt to perform the remote memory accesses at the same time. This suggests that dividing the instruction assignment task into partitioning and placement phases is inappropriate in the presence of pre-placed instructions.

Conveniently, the pre-placed instructions generated by loop unrolling and alignment analysis often provide useful information about the structure of a scheduling region. For unrolled loops without loop-carried dependencies, the disambiguated memory references are spread across the breadth of the region and thus point to good starting locations for the search for parallelism. In cases such as Cholesky, where each loop iteration contains disambiguated references destined for only a single tile, the pre-placed instructions immediately suggest an almost perfect partition. On an application like Mxm, the pre-placed instructions do not suggest as clean of a partition though they still make a good starting point.

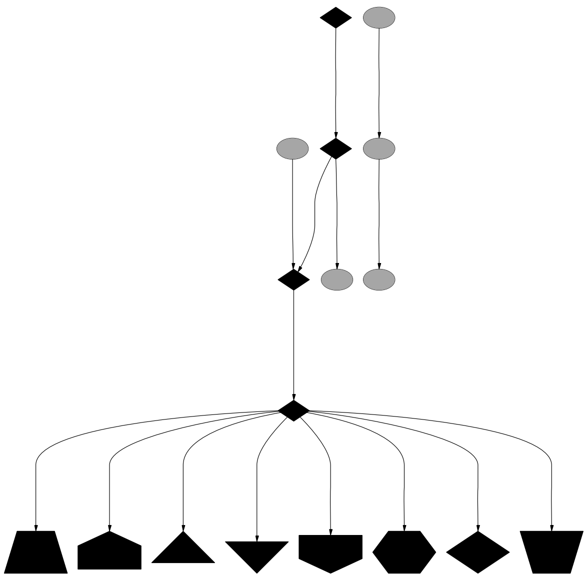

Figure 3-10: A scheduling region with a critical path marked by diamonds

The serial nature of the multimedia applications, and their less easily identified memory-access structure, means that the pre-placed instructions provide less assistance in the partitioning of their scheduling regions. The sparse distribution of pre-placed instructions throughout the regions does offer some guidance for the placement task, however.

As the only disambiguated memory references in the Fpppp kernel occur at the very end of the scheduling region, pre-placed instructions play very little role in the space-time instruction scheduling of this application.

3.3.2 Critical Path

A *critical path* in a scheduling region is a linear sequence of connected instructions whose total latency is the maximum over all such sequences in the region. Figure 3-10 shows a small scheduling region from the Cholesky benchmark with instructions on a critical path displayed as diamonds. On a machine with unlimited parallel computation resources and zero communication latency, the shortest possible execution time for a scheduling region will be exactly the latency of the critical path since, by definition, instructions on the path cannot be executed in parallel.

The critical path is an important consideration for temporal scheduling, as in reality, the principle task of a temporal scheduler is to hide the long latencies along the critical path by scheduling other useful work during those latencies.

Instruction partitioning systems must also be aware of the critical path. Instructions on a critical path generally ought to be assigned to a single tile since any inter-tile communication on the critical path immediately increases the total latency of the path. This suggests that an instruction partitioner for a highly serial application, such as SHA, may achieve good results if it first assigns the critical path to a single tile, and then searches for groups of instructions not on the critical path that can be assigned to a different tile without increasing the total length of the schedule.

The critical path plays a different role in applications with a high degree of parallelism, which includes all of the dense matrix applications described in this thesis. In these applications, each separate loop iteration has a critical path and the most important consideration is that the loop iterations be executed in parallel if possible. While the temporal scheduler must observe the critical path to ensure that the instructions assigned to each tile execute as efficiently as possible, the partitioner should instead focus on the high-level structure of the graph. If the partitioner chooses a critical path to partition and then concentrates on discovering ILP near that critical path, it may miss the opportunity to take advantage of the coarse-grained parallelism at the loop iteration level.

The Fpppp kernel demands a hybrid approach to space-time scheduling. The first part of the scheduling region contains a large amount of ILP, which a purely critical path-based system may not make efficient use of. Then the second part of the region is dominated by a reduction which presents a critical path. If the partitioner is solely concerned with discovering ILP, it may distribute the instructions along the reduction path among the tiles and introduce many unnecessary communication delays in the portion of the schedule that is dominated by the critical path. Thus an effective partitioner for this region must be aware of how to effectively exploit ILP, how to partition around the critical path, and how to tell when and where each technique is appropriate.

The scheduling region taken from the Life benchmark shown in Figure 3-4 also presents these mixed demands. Each loop iteration contains a number of parallel threads anchored by pre-placed instructions. These threads are combined by a reduction shortly after the pre-placed memory operations. A good schedule for this region requires that the threads be executed in parallel across the tiles with proper consideration given to the locations of pre-placed instructions. Then the reduction must be assigned to a single tile, preferably the tile of the pre-placed instruction nearest to the beginning of the reduction.

3.3.3 Register Pressure

The Fpppp scheduling region contains 28 nodes with five children or more for a total of 228 values. Most of these nodes have few, if any, ancestors and therefore tend to be scheduled early in the computation. Since the early portion of Fpppp exhibits a large amount of ILP, each of these nodes is likely to have children assigned to many different tiles. This has two implications. First, it is likely that the network is congested at the start of the schedule as these instructions communicate their values to their children. Second, each value is replicated by the number of tiles it is sent to which means that the schedule also suffers from high register pressure. This means that a partitioner that is too aggressive in seeking ILP may cause over-replication of values and suffer a performance loss, due to the spilling of these values to memory, when compared to a less aggressive approach.

This mistake can also be made by the temporal scheduler. Temporal schedulers often interleave instructions from unrelated tasks in order to maximize processor usage in the presence of long-latency instructions. If a scheduler interleaves tasks too aggressively when it is presented with a region containing a high degree of parallelism, such as those found in Fpppp or Vpenta, the processor may need to maintain more intermediate values than it can store in registers. This requires values to be stored in memory and drastically slows the execution of the schedule.

Chapter 4

Combined-Phase Probabilistic Scheduling

In this chapter, a combined-phase probabilistic approach to the space-time instruction scheduling and register allocation problems is presented. Rather than mapping each instruction to a single point in time, space, and register file, this approach maintains a probability distribution over time, space, and registers for each instruction. A series of simple heuristic algorithms then modify the probability distributions in order to iteratively improve the final schedule.

4.1 Motivation

The empirical analysis of benchmark application characteristics detailed in Chapter 3 clearly indicates that different classes of scheduling regions have different needs that must be met for efficient code generation. Furthermore, many applications contain regions requiring different techniques at different points within the region. Finally, even within very localized areas there are often multiple conflicting factors, such as communication cost, load balance, and register pressure, to be considered, and it is often the case that elimination of any of those factors from consideration will result in a poor schedule.

The task of performing optimal space-time scheduling and register allocation simultaneously is NP-hard so searching for an optimal solution is unreasonable. To make matters worse, both instruction scheduling and register allocation are also NP-hard, meaning that those phases must be performed heuristically. The engineering complexity of combining heuristics in order to build a computationally feasible algorithm that is directly capable of analyzing and considering all of these factors and their tradeoffs is extremely daunting. The classical solution to this problem has been to divide the space-time scheduling and register allocation tasks into separate phases and give each phase responsibility for ensuring that a subset of the constraints and tradeoffs are taken into account. Unfortunately this leads to the phase ordering problem.

The probabilistic scheduling framework proposed in this chapter is designed to minimize the effects of the phase ordering problem and facilitate the use of many simple space-time scheduling and register allocation heuristics to produce a scheduling system that is both flexible and effective. This is accomplished by giving the heuristic algorithms, or phases, two new abilities: they can apply their knowledge and analysis to the previous work of other phases and make modifications to that work, and they can express their level of uncertainty with a decision and suggest alternatives.

4.2 Detailed Description

The combined-phase probabilistic scheduling system, shown in Figure 4-1, consists of three main components: a probabilistic schedule representation that maintains the current schedule state, a set of small heuristic space-time scheduling and register allocation algorithms that modify the schedule, and a driver that chooses heuristic algorithms to apply to the schedule in order to iteratively improve it.

4.2.1 Probabilistic Schedule

A conventional schedule representation maps each instruction to a single processing resource, time slot, and register. The probabilistic schedule generalizes this representation by associating with each instruction a probability distribution over time, space,

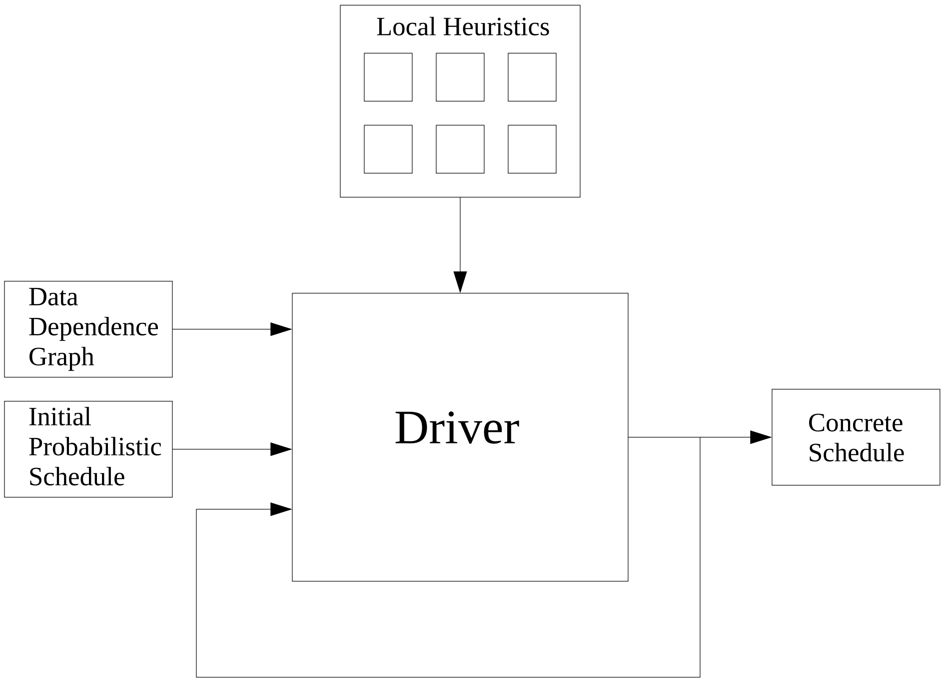

Figure 4-1: Combined-phase probabilistic scheduling system

and registers. This distribution describes the desirability of assigning that instruction to each (processing resource, time slot, register) triple. Note that while the schedule is discussed in terms of probabilities, decisions made in the probabilistic scheduling approach are rarely random, especially when a concrete schedule is extracted from the probabilistic schedule. If this step was performed randomly, the quality of the schedule would worsen due to low-probability events that would occasionally cause instructions to be scheduled at inappropriate times, on inconvenient tiles, or in conflicting registers based on the assignments of their predecessors and successors.

This generalized representation gives the heuristic scheduling phases much more flexibility than a conventional representation. Rather than simply accepting or reversing the decisions made by previous phases, algorithms can make gradual modifications to the schedule. Also, if one phase determines that a particular assignment is a bad idea, it can reduce the probability of that assignment without actually reversing it, thereby increasing the probabilities of any alternatives suggested by previous phases.

In addition to modifying the work of previous phases, the local heuristic algorithms are also given more control over how they express their own decisions. Conventionally, compiler phases do not have the opportunity to express uncertainty about decisions or provide alternatives. Using the probabilistic schedule, however, algorithms can express high confidence in some decisions while expressing low confidence in others and giving some probability to the alternatives.

4.2.2 Local Heuristics

Local heuristics are small simple algorithms that work cooperatively to iteratively improve the probabilistic schedule. Each heuristic is responsible for applying some specific piece of knowledge to the schedule. By applying many different local heuristics, knowledge about many different aspects of the space-time scheduling and register allocation problem is built up to produce an efficient schedule. The term, local heuristic, does not mean that the algorithms are prohibited from examining the entire scheduling region. Instead it means that each algorithm employs some specific piece of knowledge that is local to their area of the scheduling problem domain.

The heuristic algorithms used for combined-phase probabilistic scheduling generalize conventional compiler back-end phases in two ways. First, they may output a probability distribution over space and time instead of a single mapping for each instruction. Second, they may examine previously generated scheduling information directly related to their task. This differs from conventional interfaces where the phases perform non-overlapping tasks, such as temporal scheduling and register allocation.

There are three general types of algorithms used in the probabilistic scheduling approach. Improvers apply local scheduling heuristics in order to produce a more efficient schedule. Selectors make specific decisions and increase their probabilities in order to make the schedule more concrete. Finally, constraint enforcers apply constraints to the schedule in order to insure its validity.

Improvers

Improvers embody the heuristic scheduling techniques required to produce good schedules. They take the current schedule and consider individual factors such as pre-placed instructions, load balance, and communication cost in order to make improvements to it. It is their task to ensure that good schedules are produced.

For example, a critical path improver may find a critical path in the dependence graph and give all of the instructions on it higher weight on the most popular tile for those instructions. A communication improver may modify an instruction's probability map so that it is more likely to be assigned to the same tile as one or more of its neighbors. A register allocation improver may look for areas of the schedule with many live ranges and attempt to rearrange instructions in time and space to reduce register pressure.

Selectors

Selectors ensure that the probabilistic schedule eventually converges to a concrete schedule. Without the intervention of selectors, probability distributions for instructions may stay wide and flat indefinitely, with no clear concrete schedule ever forming. A selector alleviates this problem by iterating over the instructions, selecting entries in their probability maps based on selection criteria, and increasing the probabilities for those entries. For example, a single-entry selector may choose to increase the probability of the highest time, tile, and register triple for an instruction. A maximum-tile selector may choose to increase the probabilities for all time slots on the tile with maximum total probability for a particular instruction. A median-time selector may increase the probability of an instruction being assigned to any of the tiles in its median time slot.

Constraint Enforcers

Constraint enforcers are responsible for preventing the probabilistic scheduling system from outputting an invalid schedule or spending an inordinate amount of time working on one. For example, a constraint enforcer can ensure that pre-placed instructions are always mapped to their proper tile. A precedence constraint enforcer can make sure that an instruction is always scheduled before its children and that there is sufficient time between the instructions to allow for the proper communication delay due to their tile assignments. Constraint enforcers are necessary to guarantee the correctness of the schedule and ensure that the other heuristics are performing productively by not spending too much time improving impossible schedules.

4.2.3 Application of Local Heuristics

Because this scheduling approach employs a set of distinct local heuristics to produce space-time schedules and register allocations, it risks suffering from the phase ordering problem. If each local heuristic is applied to the schedule only once and in some fixed order, the phase ordering problem will negatively effect many schedules, as it does with conventional scheduling approaches. To minimize the effects of the phase ordering problem the local heuristics must each be applied multiple times. If the number of iterations is large and the order of heuristic application varies, the effects of going first or last or immediately before or after some other heuristic diminish. Thus the decision of which heuristic to apply at which time is of significant importance to the effectiveness of this approach.

The primary responsibility for choosing local heuristic algorithms to apply to the schedule is given to the *driver*. Through these decisions, the driver chooses which local knowledge should be applied to the schedule in the next iteration. If the driver performs well, the solution will improve over time as additional local knowledge builds with each iteration. In addition to this task, the driver must initialize the probabilistic schedule at the start of the procedure and output a concrete schedule when it determines that the schedule is complete.

The principle task of the driver is to choose algorithms to apply to the probabilistic schedule in order to ultimately produce an efficient schedule. This can be as simple as iterating over a fixed set of algorithms for a fixed number of iterations; however, more sophisticated approaches are likely to produce considerably better results. Another possibility is for the driver to examine the graph and favor algorithms that are more relevant to it. For example, algorithms based on critical path optimization could be favored on highly serial graphs, while algorithms that try to balance the load among tiles could be used for parallel graphs. An even more sophisticated driver may observe the effects of different local heuristics on the schedule and more frequently apply those that were able to improve the schedule in previous iterations.

The decision by the driver to terminate and output the concrete schedule can be based on a fixed list of algorithms to run or adapted to the task at hand. One possibility is to measure the quality of the schedule and terminate when the quality stagnates or worsens for a period of time.

When outputting the concrete schedule, the driver must finalize assignments based on the probability distributions in the probabilistic schedule. Though the driver is free to use any selection criteria to make these assignments, simply choosing the time, tile, register triple with maximum probability in each instruction's distribution is completely general. If different selection criteria are desired, such as choosing the median time slot, a selector that uses the proper criteria can be run to ensure that the preferred entry has maximum probability immediately before the concrete schedule is issued.

4.3 Example

Figure 4-2 gives a simple example of the operation of a combined-phase probabilistic scheduler. To simplify the example, only spatial scheduling is performed, though multiple heuristics interact to produce the schedule. Assume the target machine contains two tiles and communication of a word between tiles requires one clock cycle. The lower portion of Figure 4-2 displays the probabilistic schedule after each of four heuristics are applied to the schedule. Large circles represent high probability for the instruction to be assigned to that tile while small circles represent low probability.

When the schedule is initialized, each instruction is uniformly distributed across the tiles. Then the critical path heuristic identifies the critical path as containing

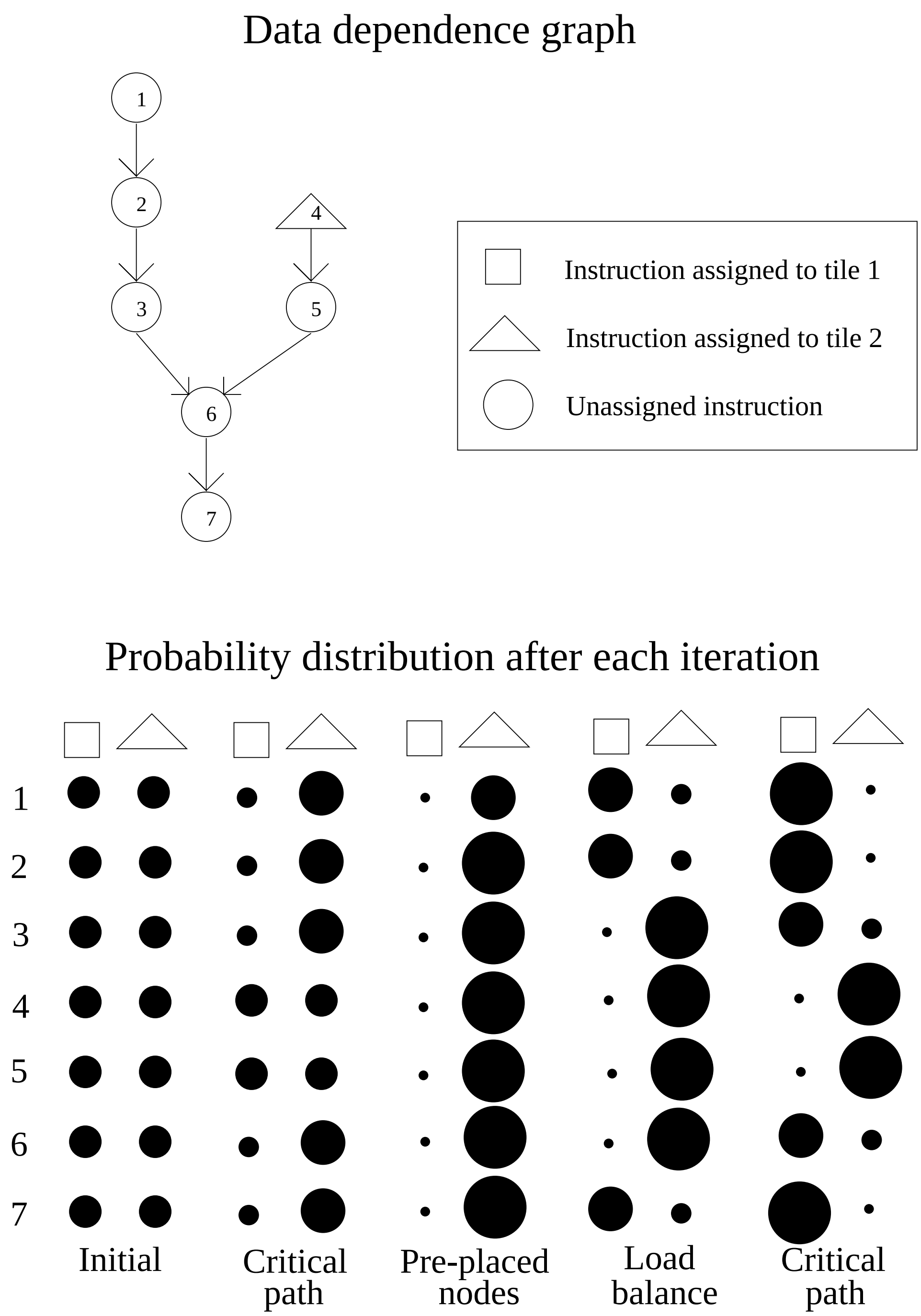

Figure 4-2: Simple example

instructions 1, 2, 3, 6, and 7 and probabilistically places them on tile 2. Next the pre-placed nodes heuristic increases the probability that instructions 2, 3, 4, 5, 6, and 7 are placed on tile 2. After this the driver notices that the schedule is imbalanced and invokes the load balance heuristic. This heuristic chooses to place instructions 1, 2, and 7 on tile 1 in order to improve load balance. Finally the critical path heuristic is executed again, except that this time tile 1 is chosen to hold the critical path. At this point, the driver decides that the schedule is complete and terminates execution.

Chapter 5

Algorithm

This chapter describes a spatial instruction scheduling system that uses the probabilistic scheduling approach. It has been implemented in C++ and integrated with RAWCC [12]. The system accepts a data dependence graph and a machine description as input and returns a mapping between instructions and tiles as output. The RAWCC temporal list scheduler and register allocator are then used to complete the code generation task.

5.1 Probabilistic Schedule Representation

An efficient probabilistic scheduling representation cannot represent arbitrary probability distributions for each instruction. Each instruction could potentially condition its probability distribution on the distributions of every other instruction in the region, giving the representation combinatorial space complexity. In order to achieve tractable space requirements while attempting to allow enough freedom and expressiveness for the scheduler to work well, the probabilistic schedule in this algorithm is represented by a three-dimensional array of probability values, p_{ijk}, indexed by instruction ID, time slot number, and tile number. This allows each instruction to maintain a joint probability distribution over time slots and tile numbers. While the quadratic space requirement of this representation becomes prohibitive on large scheduling regions, it is hoped that the additional expressiveness of the joint distribution over separate unconditional distributions for space and time will allow better interaction through local scheduling heuristics due to information sharing. The number of time slots is chosen to be the exactly the length of the critical path in order to save space. This means that temporal scheduling information is slightly coarse-grained; however, this is acceptable since it is only used within the probabilistic scheduler. For each instruction, the space-time probability map is initialized with a uniform distribution over all tiles and all time slots in which the instruction could in principle be scheduled. The set of possible time slots is based on the longest paths from the instruction to root and leaf instructions.

Each dimension of the probability map is incremented by one to create storage space for commonly accessed sums of probabilities. Using this additional storage on the perimeter of the matrix, sums over each node, tile, time slot, node and tile, node and time slot, and tile and time slot can be maintained along with the data associated with each node, tile, and time slot triple. This facilitates constant time lookup of sums over any combination of node, tile, and time slot. In order to ensure that single-value updates run in constant time, the old value to be updated is subtracted from the new value and this difference is added to each of the seven affected totals in addition to the entry being updated.

One important feature of the representation is its handling of probability normalization. For each instruction, the sum of the values over the entire joint probability distribution should equal 1. For performance reasons, rather than adjusting the entire probability map for an instruction every time a value is changed, the probability maps are only normalized when the driver or one of the scheduling heuristics requests the operation.

The probabilistic schedule representation provides a number of functions to simplify common tasks. These include finding the tiles with greatest and least probability for a given instruction, finding the median time slot in the time probability distribution for an instruction, and finding the tiles with greatest and least probability for a given time slot. Functions are also provided to multiply any of the totals, and their constituent values, by a given factor.

Let N equal the number of instructions in the region, T equal the number of time slots used, and S equal the number of tiles on the target architecture. Then let the probabilistic schedule

$$P = \{p_{nts} | 0 \leq n < N \text{ and } 0 \leq t < T \text{ and } 0 \leq s < S\}$$

such that

$$p_{nts} = Pr[\text{instruction } n \text{ is scheduled at time } t \text{ on tile } s].$$

Next define

$$p_{nTs} = \sum_{t=0}^{T-1} p_{nts},$$

$$p_{NTs} = \sum_{n=0}^{N-1} \sum_{t=0}^{T-1} p_{nts},$$

$$p_{NTS} = \sum_{n=0}^{N-1} \sum_{t=0}^{T-1} \sum_{s=0}^{S-1} p_{nts},$$

and so on. Note that $p_{nTS} = 1 \forall n$.

5.2 Heuristics Implemented

This section contains a list, and short descriptions, of the heuristic algorithms implemented for the probabilistic scheduling approach to the spatial scheduler.

5.2.1 do_preplaced_nodes

This improver heuristic gives instructions an increased probability of being assigned to the same tile as their nearest pre-assigned instruction. It executes in two phases. First the heuristic iterates over the tiles and uses breadth first search to calculate the distance in the dependence graph from each instruction to the nearest instruction pre-assigned to that tile. Then it iterates over the instructions. If an instruction is pre-assigned to a particular tile the heuristic sets the probability of that instruction being assigned to any other tile to zero. If the instruction is not pre-placed, then for each tile, the probability of the instruction being assigned to that tile is divided by the distance to the nearest pre-placed node assigned to that tile.

```
do_preplaced_nodes()
  For each instruction n do:
    If n is preplaced on tile s then
      Set P[n][T][s] = 1 and P[n][T][s'] = 0 for all s' != s
    Else
      For each tile s do:
        Set P[n][T][s] = 1 / min. dist. from n to preplaced instr. on tile s
      End
    End If
  End
```

5.2.2 balance_tile_load

This improver heuristic iterates through the time slots and attempts to equalize the total weight assigned to each tile in each time slot. This is accomplished by multiplicatively scaling the probabilities for lightly loaded tiles in each time slot.

```
balance_tile_load()
  For each time slot t do:
    For each tile s do:
      Set P[N][t][s] = P[N][t][S] / S
    End
  End
```

5.2.3 round_robin_tiles

This selector assigns instructions to tiles in a round-robin fashion by increasing the probability of assignment to the selected tile by fifty percent.

```
round_robin_tiles()
  Let a = 0
  For each instruction i do:
    Let a = a + 1
    Set P[i][T][a mod S] = 1.5 * P[i][T][a mod S]
  End
```

5.2.4 strengthen_max_tiles

This selector heuristic iterates through the instructions and increases each instruction's probability on the tile it already has highest probability on by fifty percent.

```
strengthen_max_tiles()
  For each instruction i do:
    Let s = maximum probability tile for i
    Set P[i][T][s] = 1.5 * P[i][T][s]
  End
```

5.2.5 find_parallelism

This improver heuristic iterates over the time slots and attempts to increase parallelism usage in time slots with high load imbalance. High load imbalance is considered to be cases where the tile with highest total weight has more than fifty percent more weight than the least heavily loaded tile. The heuristic creates a set of nodes that may be allocated to the heavily loaded tile in the current time slot and finds the pair of nodes that are the greatest distance apart in the graph, while memoizing all computed distances. If these nodes are sufficiently far apart, one of them, along with its near neighbors, has its probability for the heavily loaded tile in that time slot reassigned to the most lightly loaded tile in the time slot.

```
find_parallelism()
  For each time slot t do:
```

```
    Let b = most heavily loaded tile in time slot t
    Let l = most lightly loaded tile in time slot t
    If P[N][t][b] / P[N][t][b] > 1.5 Then
      Let I = {instructions i|P[i][t][b] > threshold1}
      Let instructions i and j be instructions in I with max distance between them
      If distance between i and j > threshold2 Then
        Let p = P[i][t][l]
        Set P[i][t][l] = P[i][t][b]
        Set P[i][t][b] = p
        For each neighbor n of i do:
          Let p = P[n][t][l]
          Set P[n][t][l] = P[n][t][b]
          Set P[n][t][b] = p
        End
      End If
    End If
  End
```

5.2.6 examine_neighbors

This improver heuristic iterates over the instructions and attempts to reduce communication by placing them on the same tile as their neighbors. It first counts the number of neighbors of an instruction that are likely to be assigned to each tile. Then for each tile the heuristic multiplies the probability for the instruction on that tile by the number of neighbors that were counted for that tile.

```
examine_neighbors()
  For each instruction i do:
    For each tile s do:
      Let C[s] = 0
    End
```

```
    For each neighbor n of i do:
      Let s = most likely tile for n
      Let C[s] = C[s] + 1
    End
    For each tile s do:
      Set P[i][T][s] = P[i][T][s] * C[s]
    End
    Let Total = P[i][T][S]
    For each tile s do:
      Set P[i][T][s] = P[i][T][s] / Total
    End
  End
```

5.2.7 highlight_critical_path

This selector heuristic begins by building a set of instructions that comprise a critical path in the data dependence graph. If any instructions on the critical path are pre-placed instructions, the critical path is assigned to the tile pre-placed instruction. Otherwise the critical path is assigned to the least weighted tile.

```
highlight_critical_path()
  Let C = a critical path in the dependence graph
  Let s = least weighted tile
  For each instruction i in C (in topological order) do:
    If i is pre-placed Then
      Let s = tile for i
      Break
    End If
  End
  For each instruction i in C (in topological order) do:
    If i is pre-placed Then
```

```
    Let s = tile for i
  Else
    Set P[i][T][s] = 1.5 * P[i][T][s]
  End If
End
```

5.3 Driver Implementation

The driver used for this implementation is very simple. It applies a fixed set of scheduling heuristics to the probabilistic schedule in a fixed order and assigns each instruction to the tile with maximum total probability for that instruction.

This primary design goal for this driver was simplicity. A more complex driver that iterates over the local heuristics many times could produce improved schedules if the heuristics were properly tuned for multiple iterations. To work well on a more varied set of scheduling regions, the driver needs to incorporate some kind of feedback during the scheduling process. At the minimum, the termination condition could be based on the amount of recent improvement made to the schedule. Also, using feedback to choose which heuristic to run based on their past performance could enhance the efficiency and effectiveness of the scheduling process.

Chapter 6

Implementation and Results

This chapter presents an implementation of the probabilistic spatial scheduler described in Chapter 5 targeted for the Raw architecture. Benchmark application performance results were obtained using the Raw simulator, a cycle-accurate simulator of the Raw architecture described in section 2.1.

6.1 Implementation

The probabilistic schedule in this implementation is represented by a three-dimensional array of double precision floating point values indexed by instruction ID, time slot number, and tile number.

Of all combinations and orders of scheduling heuristics studied for this thesis, the most efficient schedules were generated by the sequence: do_preplaced_nodes, balance_tile_load, and strengthen_max_tiles. All benchmark results presented in this thesis were obtained using this driver. Part of the reason for this simplicity is nearly all of the benchmarks had pre-placed nodes that were very effective at giving spatial scheduling hints. Any additional work after do_preplaced_nodes tended to worsen the schedules rather than improve them.

6.2 Methodology

The probabilistic spatial instruction scheduler described above was integrated with RAWCC, the sequential compiler for Raw. The back end phase ordering for this compiler is: alignment analysis, loop unrolling, pointer analysis, data and memory placement, spatial instruction scheduling, temporal list instruction scheduling, and graph-coloring register allocation.

Application performance results were obtained using the cycle-accurate Raw simulator. This simulator properly models the communication network, including congestion, as well as the execution activities of each tile. We obtained experimental results for the suite of benchmarks detailed in section 3.1. Each tile is modeled exactly with an extended MIPS R4000 core and switch processor. Instruction latencies for arithmetic instructions are listed in Table 2.1.

Speedup values are derived from comparison to the execution time of each benchmark compiled by the Machsuif [18] MIPS R4000 compiler and executed on the MIPS R4000 core located on a single Raw tile.

Section 3.1 describes the set of benchmark applications used to generate the results presented in this chapter.

6.3 Speedup

Figure 6-1 presents the speedup from compilation with RAWCC with conventional partitioning and RAWCC with probabilistic partitioning compared to single-processor performance. Cycle counts and speedup from conventional to probabilistic partitioning are given in Table 6.1. Probabilistic partitioning improves on conventional partitioning on most of the benchmarks.

Both versions of RAWCC demonstrate significant speedup on all dense matrix benchmark applications. Probabilistic partitioning generates more efficient schedules than conventional partitioning on all dense matrix applications and all tile quantities except for Vpenta on 16 tiles, where probabilistic partitioning performs two percent

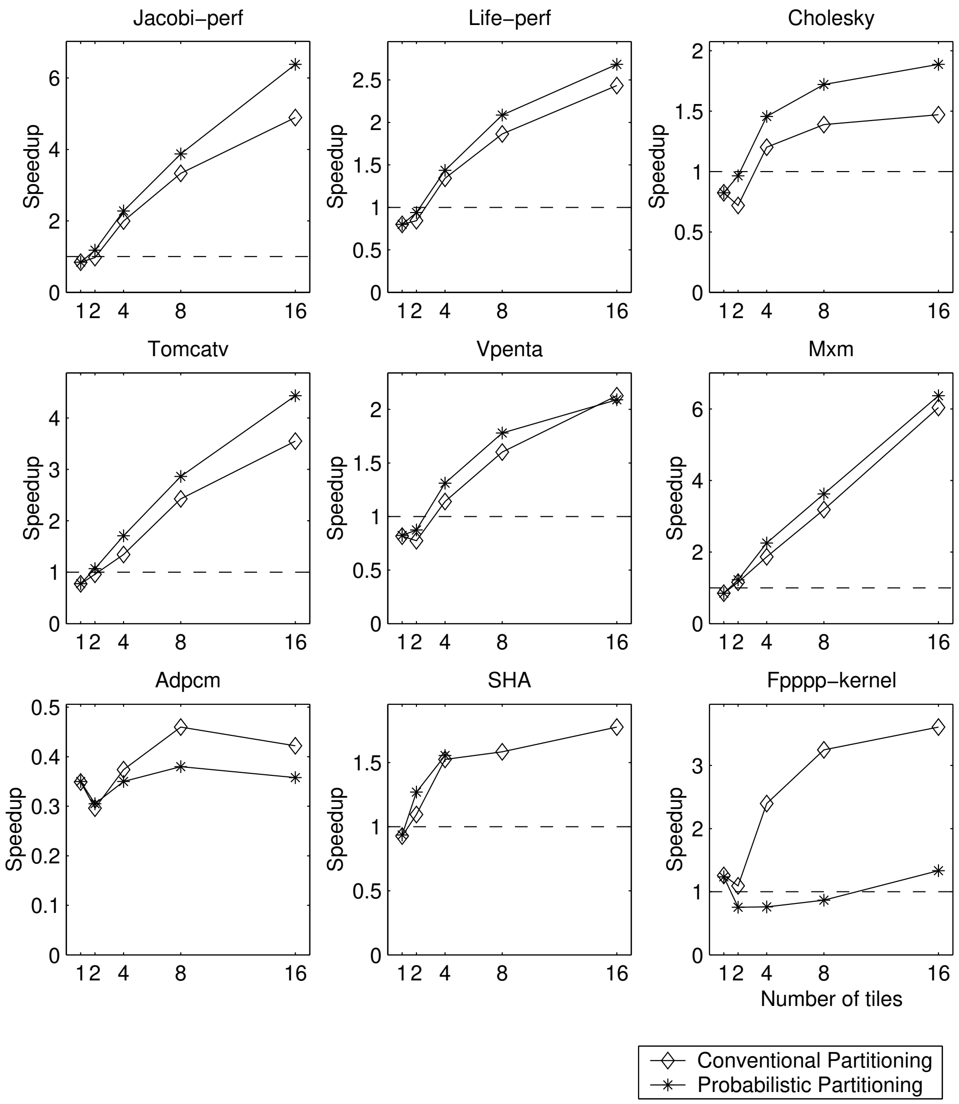

Figure 6-1: Speedup of benchmarks on Raw with 1, 2, 4, 8, and 16 tiles compared to benchmarks compiled with Machsuif for MIPS R4000 running on one tile

Table 6.1: Execution cycle counts for probabilistic partitioning and original partitioning of Raw benchmarks

Number of tiles		01	02	04	08	16
Life-perf	Conv. part.	346,568	328,748	206,348	148,757	113,857
	Prob. part.	347,011	294,993	193,113	132,802	103,225
	speedup	0.999	1.11	1.07	1.12	1.10
Cholesky	Conv. part.	136,735	156,801	93,818	81,154	76,614
	Prob. part.	136,737	116,993	77,409	65,546	59,716
	speedup	1.00	1.34	1.21	1.24	1.28
Tomcatv	Conv. part.	337,279	271,073	194,175	107,436	73,460
	Prob. part.	334,694	243,887	152,524	91,057	58,807
	speedup	1.01	1.11	1.27	1.18	1.25
Vpenta	Conv. part.	214,168	226,056	153,616	109,362	82,249
	Prob. part.	212,602	200,263	133,483	98,376	83,759
	speedup	1.01	1.13	1.15	1.11	0.982
Mxm	Conv. part.	1,893,783	1,388,108	860,460	505,527	266,442
	Prob. part.	1,893,783	1,314,319	714,924	444,181	252,611
	speedup	1.00	1.06	1.20	1.14	1.05
Adpcm	Conv. part.	973,051	1,149,439	908,910	739,167	805,888
	Prob. part.	972,028	1,115,093	970,617	894,493	948,292
	speedup	1.00	1.03	0.936	0.826	0.850
SHA	Conv. part.	1,300,544	1,100,046	790,878	760,884	678,299
	Prob. part.	1,290,816	949,533	775,043	–	–
	speedup	1.01	1.16	1.02	–	–
Fpppp-kernel	Conv. part.	1523	1752	800	590	531
	Prob. part.	1552	2545	2521	2210	1438
	speedup	0.981	0.688	0.317	0.267	0.369
Jacobi-small	Conv. part.	47,052	39,773	22,989	15,784	11,500
	Prob. part.	47,052	34,492	19,361	12,388	8838
	speedup	1.00	1.15	1.19	1.27	1.30

worse than conventional partitioning.

The results are mixed for the two multimedia benchmark applications. Probabilistic partitioning demonstrates improvement over conventional partitioning on SHA for two and four tile Raw machines; however, the algorithm runs out of memory when compiling SHA for 8 and 16 tiles due to unrolling of a very large loop and the quadratic space requirements of the chosen probabilistic schedule representation. Both partitioning algorithms perform poorly on Adpcm due to a large number of dynamic memory references.

The results on Fpppp-kernel illuminate the dependence of the current probabilistic partitioning algorithm on pre-placed instructions for information about scheduling region structure. Fpppp-kernel contains exploitable ILP; however, with no preplaced nodes to serve as guides, the probabilistic partitioner is unable to effectively use it.

Chapter 7

Analysis

This section presents an analysis of the positive and negative features of the probabilistic spatial scheduler implementation described in Chapter 6.

7.1 Potential Problems

This combined-phase probabilistic scheduling approach presents a number of challenges that must be addressed in a successful implementation. Some of these challenges include working with unconditional probabilities, ensuring that the schedule converges in a meaningful way, finding the right balance between overly aggressive and overly passive heuristics, and composing the heuristics so that the resulting schedule is positively affected by the running of multiple heuristics.

7.1.1 Unconditional Probabilities

One of the most challenging aspects of the probabilistic scheduling approach is the lack of conditioning between the probability maps of instructions. This inhibits simple operations on groups of instructions.

Consider the critical path heuristic. The primary goal of the critical path heuristic is to place all instructions along the critical path on the same tile. The actual identity of that tile is unimportant. What the critical path heuristic would like to specify is that all instructions on the critical path will be assigned to the same tile; however, the choice of which tile they should be assigned to is left to a future heuristic, perhaps a pre-placed instruction or load balancing heuristic, that is better equipped to make that decision. Unfortunately there is no way to represent this in the current design of the probabilistic scheduling system.

The critical path heuristic is left with two choices. One option is to attempt to express that the critical path can be placed on any tile but giving the nodes on the critical path uniform distributions across the tiles. This is unacceptable as it fails to form any sort of link between instructions on the critical path, which is the intended purpose of the critical path heuristic. The second option is to make the conditional probabilities unconditional by choosing a tile and assigning the critical path instructions to that tile. While the critical path heuristic is satisfied, this method of "deconditioning" conditional probabilities as one serious drawback. It brings back the irreversible decision problem that the probabilistic scheduling system had hoped to overcome. If the probability distributions of all instructions on the critical path were conditioned on a single instruction in in the path, changing the assignment of that instruction would change the assignment of the entire critical path and the irreversible decision problem would be alleviated. Without modification, however, the probabilistic scheduling system cannot represent conditional probabilities between instructions explicitly and care must be taken to work around this problem instead.

One possible solution is to represent the conditional probabilities implicitly in the heuristics. For example, the critical path heuristic may choose a small number of instructions to act as indicators, and ensure that the other critical path instructions are always placed consistently with those indicators. For this technique to succeed, the driver must cycle through the heuristic set many times to ensure that all important conditional relationships are represented.

7.1.2 Schedule Convergence

Another challenge that must be addressed when implementing a probabilistic scheduling system is the need for the schedule to converge to something meaningful. Due to the unconditional probability problem discussed in the previous section, it is easy to end up with a set of heuristics that are incapable of making a firm decision. When faced with seemingly interchangeable solutions, as can be the case with the critical path heuristic, they balk at the task of selecting one of the solutions and implementing it. The result is that noise effects, rather than the intentional efforts of the scheduling heuristics, dominate the scheduling process as selectors are forced to make arbitrary decisions in order to make progress. While some amount of this seems essential to stimulate the scheduling process, developing improvers that can also perform the role of selectors seems desirable.

7.1.3 Heuristic Cooperation

Tuning the heuristics so they work well with each other is a task that requires careful consideration. When heuristics are either too aggressive or too passive, the schedules they produce suffer. Heuristics that are overly aggressive continually reverse the decisions of other heuristics without regard for the fact that the overridden decisions may have been more appropriate for the current situation. Heuristics operating in this manner are not cooperating, and this can result in unpredictable, and even cyclic, behavior.

On the other hand, if the heuristics are tuned to be too passive and only make small changes to the schedule probabilities, little progress will be made. All decisions will be made by selectors, and while selectors are good at making decisions, they are ill-suited for the task of producing good schedules on their own.

Machine learning techniques may be able to effectively automate the task of properly tuning heuristics for efficient cooperation. This problem is essentially a search problem across the space of tunable parameters. Machine learning may be able to discover patterns in the space to aid solution, or at least cover a much greater area of the space than would be possible with manual experiments.

7.1.4 Heuristic Composition

Finally, when many heuristics with dissimilar goals are applied to the probabilistic schedule, it is hoped that the composition of their effects brings out the best that each has to offer with the bad decisions of each overridden by the good decisions of the others. If the reverse of this occurs, and the good decisions of each are overridden by the bad decisions of others, the resulting schedule is unlikely to perform well. This means that at some level there must be compatibility between heuristics. Perhaps they even need to be aware of each others' goals and actions.

Local heuristic algorithms developed for this system require slightly different structure than conventional back end compilation phases. The most significant of these is that they must be able to examine knowledge previously added to the schedule by other algorithms and effectively integrate their new knowledge with it. Success in this area is key to good heuristic composition and, as a consequence, good schedules.

A second area in which heuristic algorithms in this system differ from conventional compiler phases is that they must judge their decisions and report those judgments in terms of a confidence measure. Furthermore, along with this confidence measure they may express some confidence in alternatives to the decision they made. As many heuristics already employ cost functions in their work, giving confidence in decisions may come easily. Suggesting good alternatives, however, may require more work.

Chapter 8

Related Work

8.1 MIMD and VLIW Compilation

The space-time scheduling problem for Raw is similar to the scheduling problems faced by compilers for Multiple Instruction Multiple Data (MIMD) and Very Long Instruction Word (VLIW) architectures. Many of the components of the Raw compiler are borrowed from work targeted for these architectures.

The division between spatial and temporal scheduling in the Raw compiler is partially motivated by Sarkar's work with MIMD machines [16]. Yang and Gerasoulis also separate spatial and temporal scheduling though they do not address the placement problem because their targeted machine has a symmetric network [21]. The MIMD scheduling problem has been studied in great detail, and [2] provides a survey of some representative work. One major difference between the scheduling problems for Raw and MIMD machines is the presence of predetermined processor mappings in the Raw scheduling problem. The predetermined processor mapping problem, which is the primary focus of this project, does not occur in MIMD machines.

The Bulldog [8] compiler faces a scheduling problem similar to the Raw scheduling problem. It targets a VLIW machine with distributed processing units, register files, and memory banks. Thus Bulldog must also address the issue of predetermined processor mappings for memory accesses. Bulldog also solves the scheduling problem in two phases, assignment and scheduling. The assignment algorithm uses a greedy depth-first traversal that schedules each rooted subgraph in the precedence graph separately. This type of greedy approach is shown in [13] to be inappropriate for parallel precedence graphs such as those produced by the Raw compiler through parallel loop unrolling. To produce efficient schedules for parallel precedence graphs, the algorithm may need to intermingle instructions from different connected components of the dependence graph.

Chapter 9

Conclusions

9.1 Summary

In this thesis the structure of scheduling regions found in representative applications was studied empirically. This study revealed that significant variance exists in the structure of scheduling regions; however, most scheduling regions can be characterized by a small number of specific features. Thus, while enough structure exists to be exploited by simple scheduling heuristics, the range of possible structures is too great for a single heuristic or composition of heuristics to efficiently and effectively schedule.

Following the analysis of scheduling regions, this thesis offers an architecture that facilitates the use of local heuristics to iteratively improve a schedule. This architecture is based on a probabilistic schedule representation that allows local heuristics to share a rich knowledge base and an iterative improvement paradigm that allows each heuristic to make multiple contributions to the final schedule. Using this architecture, local knowledge is collected and combined to form an effective global solution.

A simple implementation of a spatial scheduler for Raw machines was built using this architecture. While much of the implementation remains to be completed, the system already shows promise on most benchmarks. This leads to the suggestion that pre-placed instructions are critical constraints in the spatial scheduling problem.

9.2 Future Work

This thesis lays the groundwork for a large amount of future work. There are many distinct directions in which the work of this thesis can be expanded or completed.

9.2.1 Scheduling and Register Allocation

First and foremost, the probabilistic scheduling system implemented in this thesis only addresses one phase of the code-generation task, that is spatial scheduling. One logical extension to this work is to implement temporal scheduling and register allocation in the probabilistic scheduler as well. One possible approach is to generate priorities for instructions and pass those priorities to the temporal list scheduler. A more thorough solution is to implement the list scheduler as a phase in the probabilistic scheduler and then develop a set of improvers that work with temporal scheduling information.

A more ambitious extension to this research would be to add register allocation. A probability distribution over the register set for each instruction could be added either independent of the space-time probability map or as a three-way joint probability distribution. Additional heuristics that perform register allocation could then be added to complete the phase-unification aspect of the combined-phase probabilistic scheduling system.

9.2.2 Better Heuristics and Tuning

One area in which this research should certainly be extended is in the set of scheduling heuristics. While the current set of heuristics performs well on many of the benchmarks there is much room for improvement.

Each heuristic is intended to add some piece of local knowledge about the problem structure to the solution. Heuristics that can effectively recognize parallelism in applications lacking pre-placed nodes, such as Fpppp, would greatly enhance the effectiveness of this system. Also, heuristics could be added that consider factors such as cache misses, alternative instruction selection, and unusual quirks of the target architecture.

The current set of heuristics could also benefit from tuning. Some seemed to be too passive while others proved themselves to be much too aggressive. More moderate approaches on both counts could yield improvement.

9.2.3 Driver Improvement

Finally, the driver is another source of interesting opportunities for future work. Currently the implemented driver does not examine the probabilistic schedule except for the sake of outputting it. The driver could certainly examine the schedule and choose heuristics based on their previous performance or based on the apparent needs of the scheduling region. If the schedule is spatially unbalanced, the driver may use a load-balancing heuristic. If the schedule is overly fragmented, it may use a heuristic that swaps blocks of instructions between tiles in order to reduce communication and streamline flow. If the schedule contains too many live values, it may use a heuristic that reduces register pressure. This type of adaptive driver could potentially perform well over a much wider range of benchmarks than the current driver. An immediate extension of the current driver in this area is to examine pre-placed instructions and if they exist and are distributed throughout the graph, use the pre-placed instruction heuristic. If pre-placement information is not available, or if it does not provide helpful information, the driver may use a different spatial scheduling heuristic that makes some seed decisions at the beginning.

Another interesting way in which the driver could be improved is to use some type of machine learning algorithm to discover patterns of scheduling heuristics that produce good schedules. Genetic programming is one approach that may yield good results when applied to this system.

Appendix A

Graphs

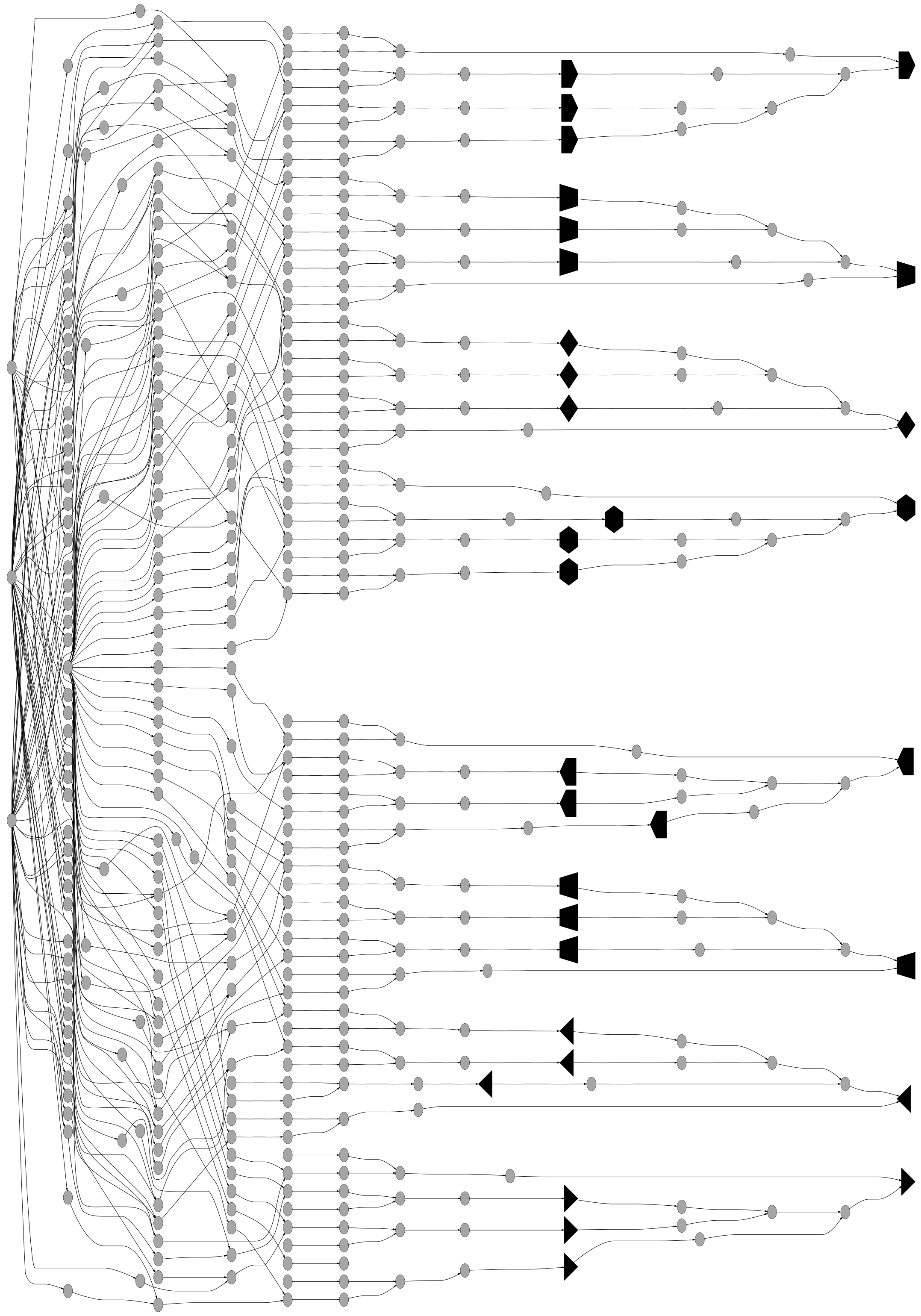

Figure A-1: Cholesky 8 tiles

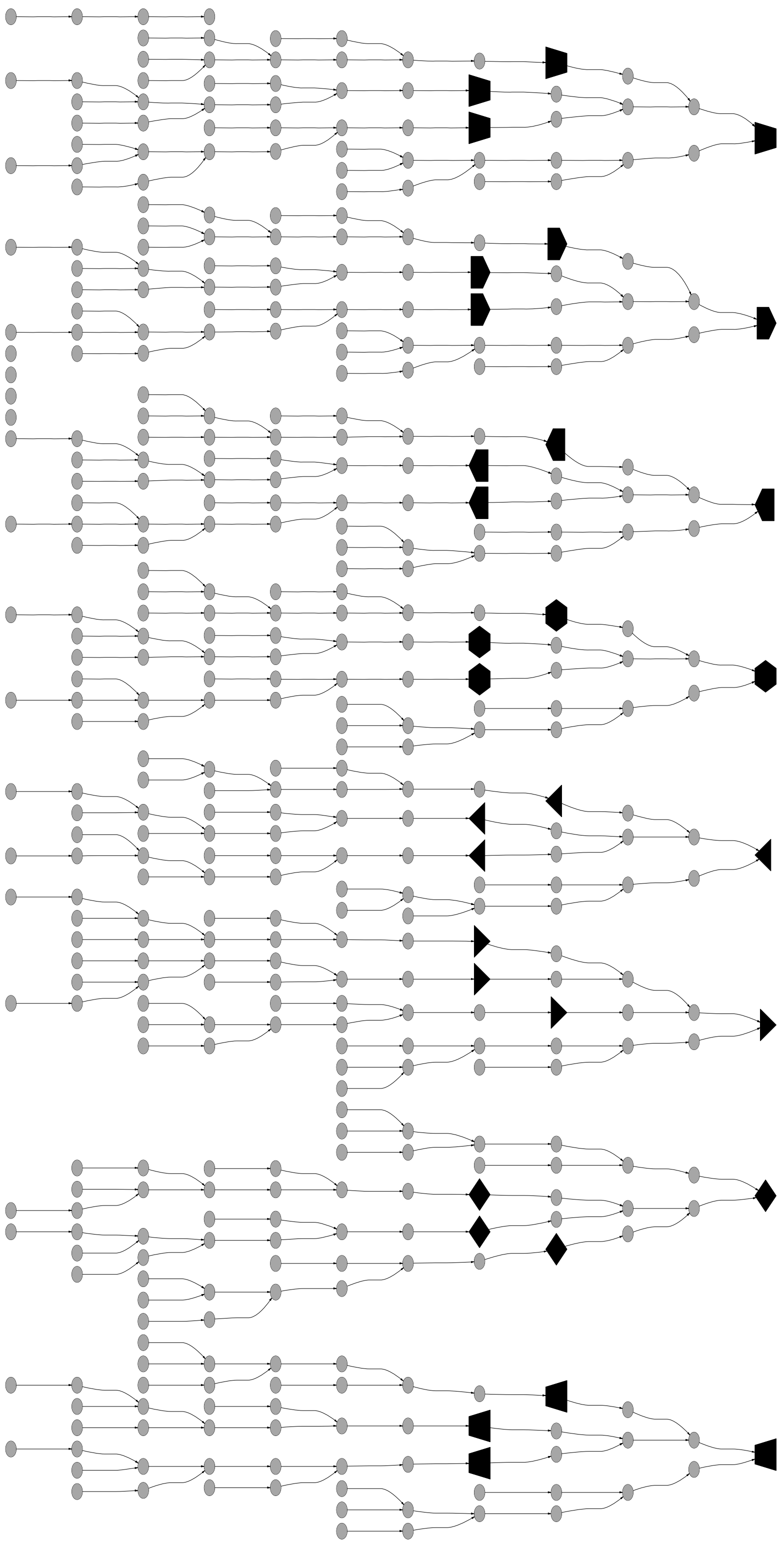

Figure A-2: Cholesky 8 tiles (4)

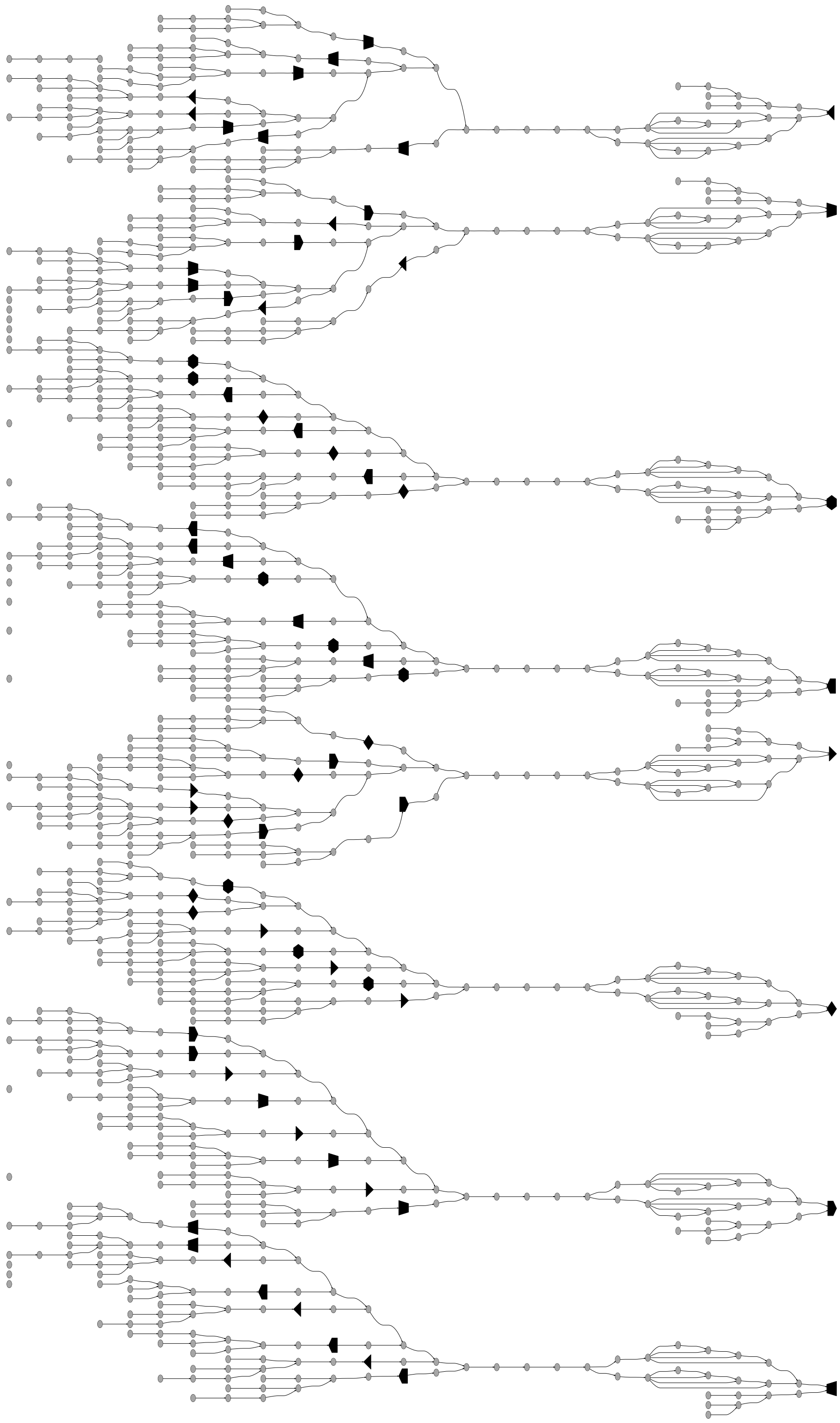

Figure A-3: Life 8 tiles (2)

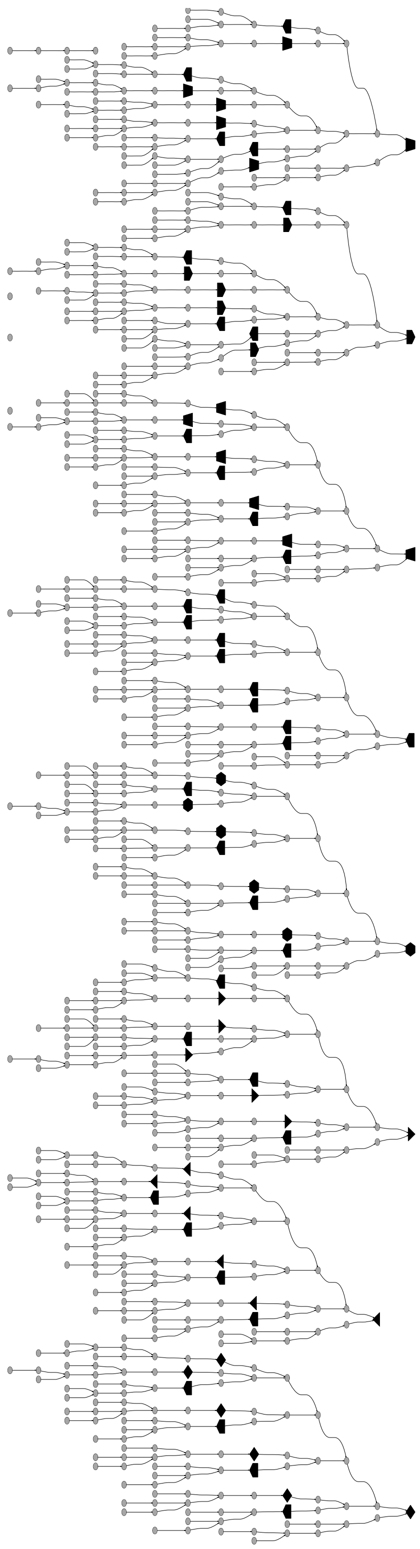

Figure A-4: Mxm 8 tiles (3)

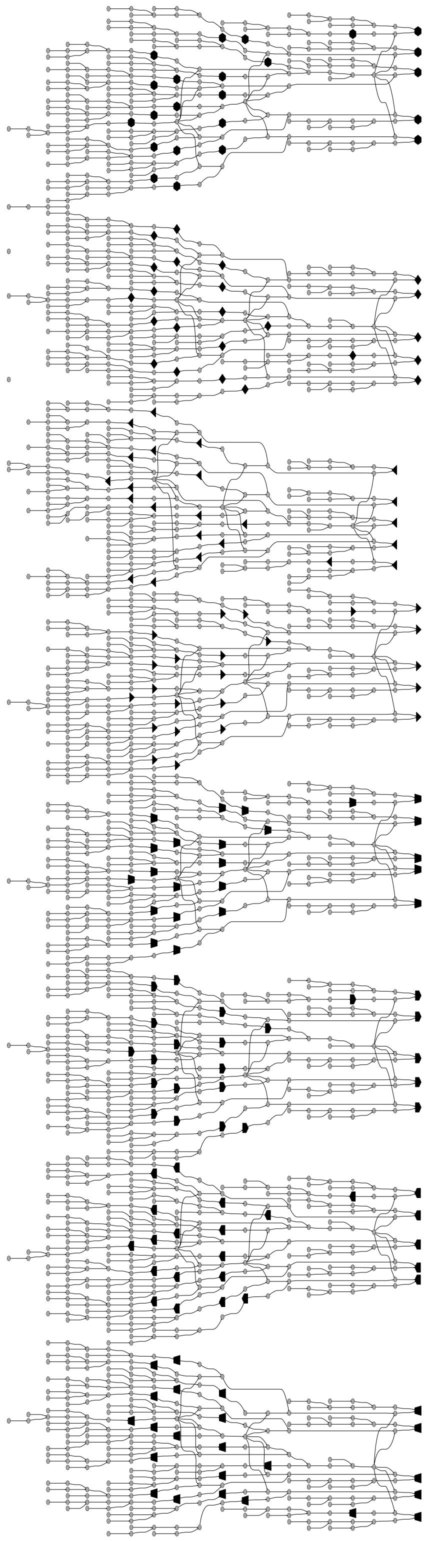

Figure A-5: Vpenta 8 tiles (2)

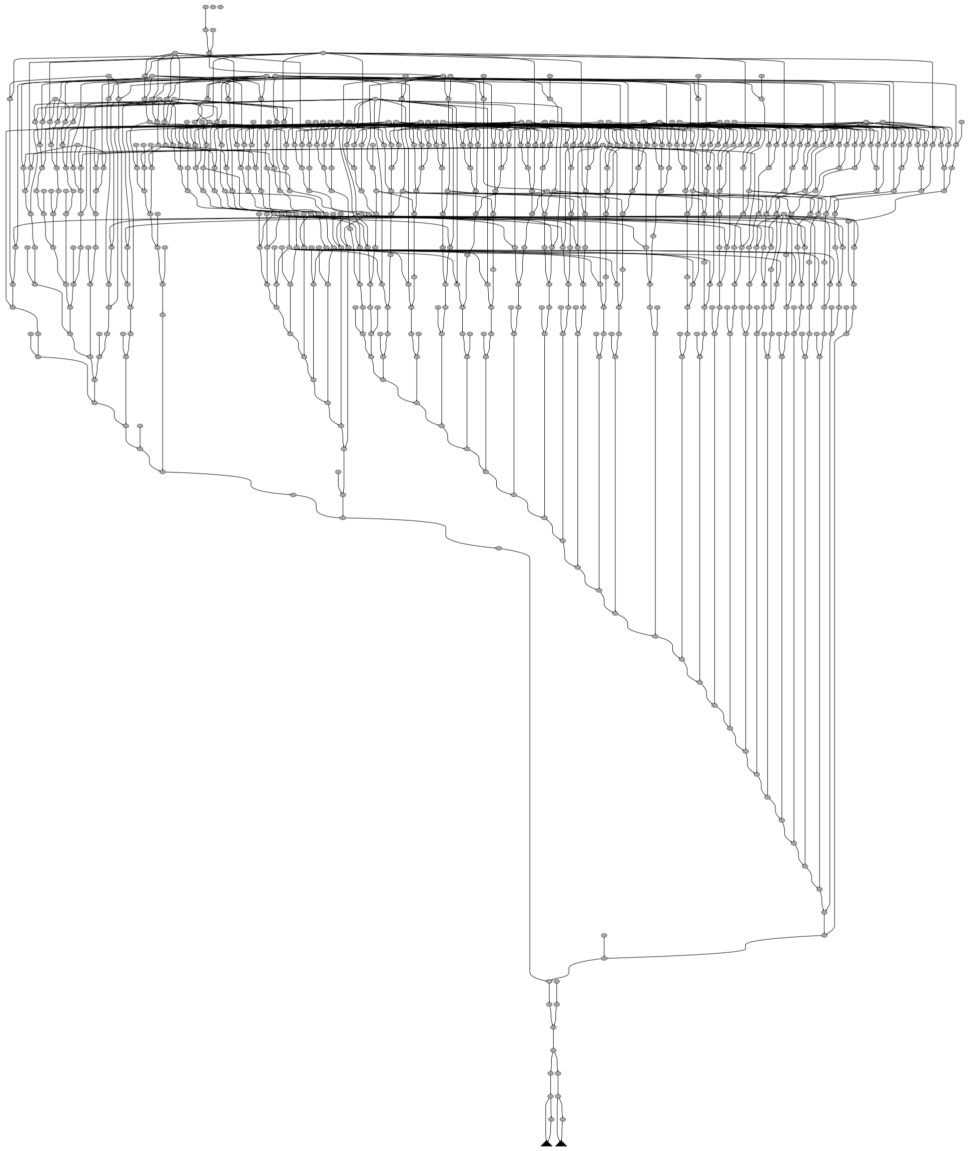

Figure A-6: Fpppp 8 tiles

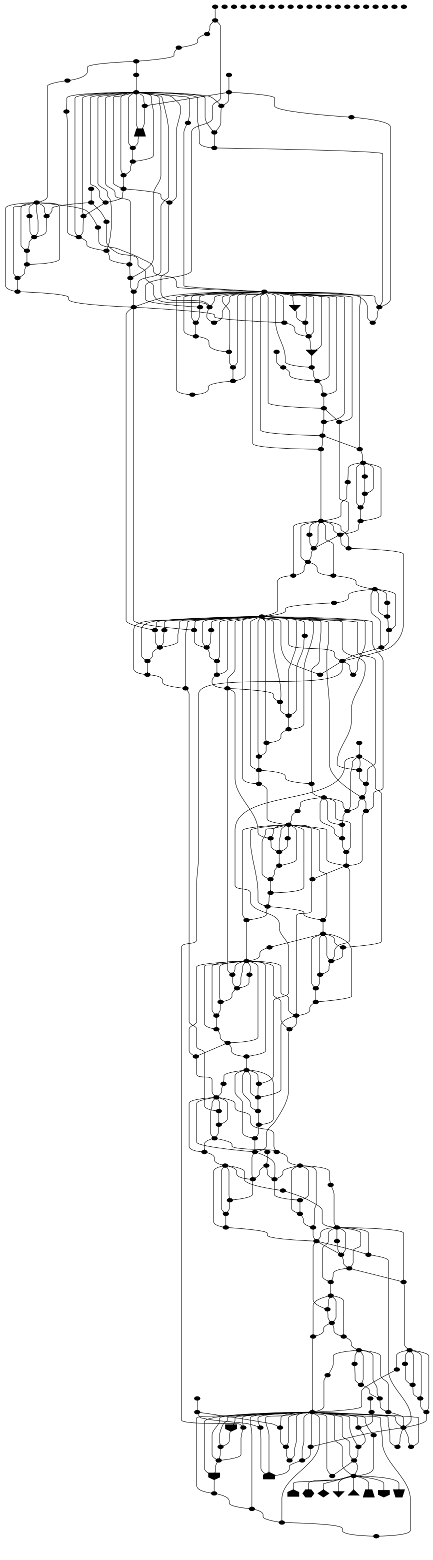

Figure A-7: Adpcm 8 tiles

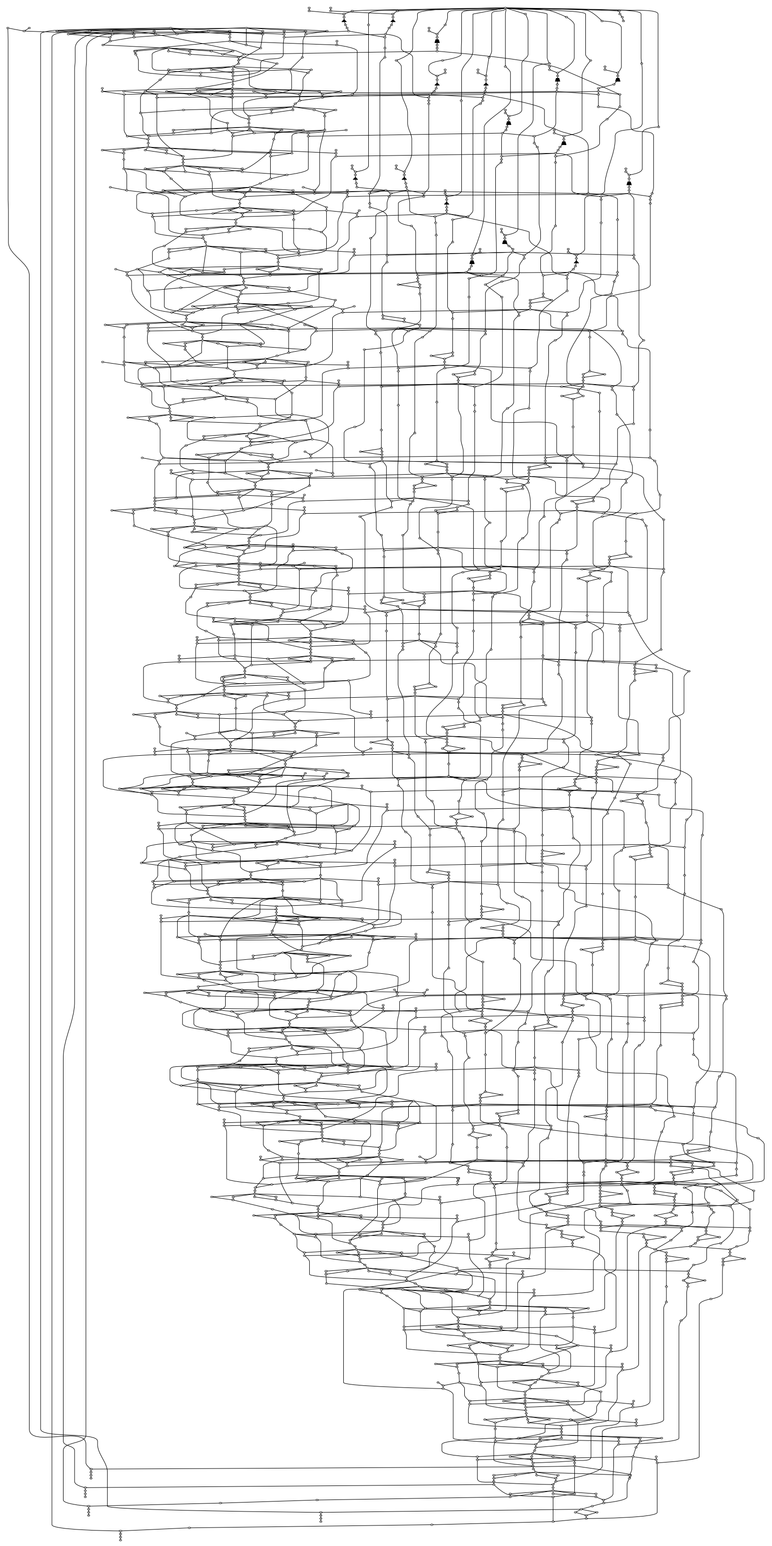

Figure A-8: SHA 2 tiles

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, pages 248–259, June 2000.

[2] I. Ahmad, Y. Kwok, and M. 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.

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The Raw Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA-26)*, Atlanta, GA, May 1999.

[5] Rajeev Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures*. PhD dissertation, Massachusetts Institute of Technology, January 2000.

[6] Standard Performance Evaluation Corporation. The SPEC Benchmark Suites. http://www.spec.org/.

[7] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical Report HPL-98-13, HP, February 1998.

[8] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985.

[9] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[10] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Techniques for Increasing and Detecting Memory Alignment. Technical Memo LCS-TM-621, MIT/LCS, November 2001.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997. IEEE Computer Society.

[12] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth International Conference on Architectural Support of Programming Language and Operating Systems*, San Jose, October 1998.

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

[14] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *34th Annual International Symposium on Microarchitecture (MICRO-34)*, pages 40–51, December 2001.

[15] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, May 1999.

[16] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, 1989.

[17] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley, New York, second edition, 1996.

[18] M. D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, January 1996.

[19] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All To Software: Raw Machines. *Computer*, pages 86–93, September 1997.

[20] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), December 1996.

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