Spatial Software Pipelining on Distributed Architectures for Sparse Matrix Codes

by

Michelle Duvall

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

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2004
LIBRARIES

© Massachusetts Institute of Technology 2004. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by ....
Anant Agarwal
Professor
Thesis Supervisor

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

BARKER

Spatial Software Pipelining on Distributed Architectures for Sparse Matrix Codes

by

Michelle Duvall

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Wire delays and communication time are forcing processors to become decentralized modules communicating through a fast, scalable interconnect. For scalability, every portion of the processor must be decentralized, including the memory system. Compilers that can take a sequential program as input and parallelize it (including the memory) across the new processors are necessary. Much research has gone towards the ensuing problem of optimal data layout in memory and instruction placement, but the problem is so large that some aspects have yet to be addressed.

This thesis presents spatial software pipelining, a new mechanism for doing data layout and instruction placement for loops. Spatial software pipelining places instructions and memory to avoid communication cycles, decreases the dependencies of tiles on each other, allows the bodies of loops to be pipelined across tiles, allows branch conditions to be pipelined along with data, and reduces the execution time of loops across multiple iterations. This thesis additionally presents the algorithms used to effect spatial software pipelining. Results show that spatial software pipelining performs 2.14x better than traditional assignment and scheduling techniques for a sparse matrix benchmark, and that spatial software pipelining can improve the execution time of certain loops by over a factor of three.

Thesis Supervisor: Anant Agarwal
Title: Professor

Acknowledgments

I would like to thank several people who have been key in helping me to complete this thesis and in supporting me throughout my work. First, I would like to thank my advisor, Anant Agarwal. His support and direction have not only made this thesis possible, they have made my work here enjoyable. Second, I would like to thank Walter Lee. He has always been willing to implement additions to RawCC or read my papers on the slightest notice, and his help in developing this system has been invaluable. Third, I would like to thank my fiance, Chris Leger. He has been a bastion of support for me, always encouraging me and ever ready to listen to my plans of action, to evaluate new ideas with me, and to walk through crazy examples to help me figure out where my code is wrong. I have learned heaps about C++ and memory thanks to his help and debugging skills. Finally, I would like to thank my parents, Mary and Dan Duvall, and my siblings for their constant support and understanding. They have always been ready with encouragement when I needed it, and have accepted and understood all the times I have told them "I just don't have time - maybe later". Also, I would like to thank Walter and Chris for their help in editing this thesis.

Contents

1 Introduction **15**

2 Background **21**

2.1 Sparse Matrix Applications 21

2.2 Software Pipelining 22

2.3 Raw Architecture 23

2.4 RawCC 23

2.5 Maps and the Space-Time Scheduler 24

2.6 Asynchronous Global Branching 25

3 Spatial Software Pipelining **27**

3.1 Motivation 27

3.2 Example Motivating Tree-shaped Communication 30

3.3 The Basic Idea 32

4 Algorithms **35**

4.1 Input Information 36

4.2 Create Map 37

4.3 Merge 40

4.4 Tile Assignment 48

4.5 Branch Propagation Path 52

4.6 Reconciliation 54

5 Module Interface Specifications **67**

5.1 Interaction with RawCC 67

5.2 The Memory Map 69

5.3 Objects Files 70

5.4 Schedule Files 73

5.5 The Basic Block Map 75

6 Results and Analysis **79**

6.1 Implementation Assumptions 79

6.2 Performance 81

6.3 Unstructured Results 82

6.4 Applicability 83

6.5 Compatible Optimizations and Effects 84

7 Future Work **87**

7.1 Handling Communication Cycles 87

7.2 Loop Unrolling and Software Pipelining 88

7.3 Integrating with RawCC 89

7.4 Using Profiling Information 90

7.5 Using Influence Across Basic Blocks 91

7.6 Improving the Tile Assignment and Virtual Tile Merge Algorithms . 92

7.7 Determining and Increasing Applicability 92

8 Related Work **95**

8.1 Bottom-Up-Greedy 96

8.2 Partial Component Clustering 96

8.3 Convergent Scheduling 97

9 Conclusion **99**

List of Figures

4-1 The final tile assignment of the canonical case described in Chapter 1. Rectangles represent physical tiles on Raw and surround the instructions assigned to execute on the tiles; the switch for each tile is represented by the dotted oval labelled `comm`. Ovals represent instructions; directed solid arrows between ovals represent data-bearing dependencies. Each dependency is drawn in the color of its source instruction's tile; inter-tile dependencies are represented by a chain of arrows connecting the instructions through the `comm` switches along the route. The root tile is shown in black, as are all branch dependencies. If a node is memory mapped, the text of the node is in the color corresponding to the memory id. The dashed ovals share a common defId. 56

4-2 The intermediate stages of the virtual dependency graph when performing spatial software pipelining on the canonical case described in Chapter 1. Figure (a) shows the virtual dependency graph immediately after creation; figure (b) shows the result of merging virtual tiles with dependencies to the root tile into the root tile; figure (c) shows the result of removing cycles. The figure specifications are the same as in Figure 4-1, which shows the final layout found by spatial software pipelining. 57

4-3 Three examples demonstrating EST, path, and cost of virtual tiles, and color and palette of virtual dependencies. Figure (a) demonstrates how the palette is constructed for each virtual edge, figure (b) demonstrates how EST and path are calculated for each virtual tile, and figure (c) demonstrates the effects of dependency cycles on palettes. Note that the EST and path variables in (c) are in parentheses; these are not the EST and path used by the current module, but are possibilities for a more correct implementation. All other EST and path variables are shown as calculated in the current module. Recall that when considering the merge of two virtual tiles, the merge creates a new dependency cycle when the palette of some incoming dependency to one of the tiles is a strict superset of the palette of some outgoing dependency from the other tile. 58

4-4 An example demonstrating how the cost function is calculated in simulated annealing in the tile assignment phase. Each square represents a physical tile, each oval a virtual tile, and each arrow a virtual dependency. The dotted arrows all represent the same data word being sent to different destinations. The window size is assumed to be 2, and the number next to each edge on each link represents the amount added to the cost due to that edge on that link. The numbers within or just next to cycles represent the additional cost added because of the cycles. The total cost of this layout is 236. 59

4-5 C-like pseudocode exhibiting how to create the virtual dependency graph. 60

4-6 C-like pseudocode for loop to enforce memory id and defId mapping consistency across basic blocks. 61

4-7 C-like pseudocode for the function used to merge virtual tiles. This function is called after dependency cycles have been merged away (if such merges were possible). 62

4-8 C-like pseudocode for the cost function for simulated annealing. The function calcLayoutEntropy walks through each dependency and adds the appropriate cost due to that dependency; additionally, it calls two helper function to add in the cost due to conflicting data flow and too much data flow. 63
4-9 Additional C-like pseudocode for the cost function for simulated annealing. This function calculates the appropriate entropy due to each dependency. 64
4-10 C-like psuedocode for the algorithm to determine the branch condition source tile for each non-root tile. 65
4-11 C-like pseudocode for verifying appropriate construction of branch condition propagation route. 66

5-1 Model of the interaction between RawCC and the spatial software pipelining module. Phases of RawCC are shown with solid lines; items added by the module are shown with dotted lines. The phases of RawCC that are replaced by the spatial software pipelining module are grayed out. 76
5-2 A skeletal perl script to create the memory mapping file sched.map using RawCC. 77
5-3 A snippet of a sched file, used for communicating the instruction-level dependency graph and instruction assignments. 78

List of Tables

3.1 Sequence of events for one iteration of the canonical case loop with induced communication cycle as compared to one iteration of the spatial software pipelined layout. This shows the instruction executed on each of tiles 0 and 1 in each cycle, and also shows each data word as it moves through the network. Note that there are 9 additional cycles per iteration in the induced loop. 33

3.2 The effects of removing cache conflicts (due to arrays A and B) with extra synchronization between tiles and without such synchronization. Note that removing cache conflicts has less of a performance benefit when the tiles are decoupled. We show here bloodgraphs for one iteration of the loop in each of the layouts; each bloodgraph is taken from an iteration in which no cache conflicts occur on any tile, and each begins with the first instruction after the branch on the tile that houses `i`. Note that white represents useful work being done; all other colors represent stalls. Each line in each bloodgraph shows the time and cause of stalls on that tile; the first line shows Tile 0, the second Tile 1, etc. For the induced loop layout, the tile that houses `i` is Tile 1; for the others, it is Tile 0. A key is included to show which color represents which kind of stall. 34

6.1 Performance comparison of RawCC to RawCC with spatial software pipelining. 82

6.2 Breakdown of improvement to unstructured (not unrolled) for each loop of interest. Loops are presented in the order in which their basic blocks are evaluated by the spatial software pipelining module. Additionally, the actual execution time for each loop is compared with the execution time for that loop obtained by evaluating that loop first. . . 83

Chapter 1

Introduction

Current technological trends indicate that wire delay will become the primary bound on chip complexity and scalability in microprocessor design [1]. As such, scalable microprocessors must be built of decentralized components, including a decentralized memory system, in order to limit wire delay and allow scalability. Processors with replicated compute units connected together via a fixed interconnect are necessary for such scalability. Each compute unit must contain a processing element (dispatch unit, register file, and functional units), a tightly-coupled memory bank (cache), and a decentralized interconnect (interconnect control) with which it can access other portions of the memory. For efficiency, such an architecture exposes the distributed memory as well as distributed processors to the compiler and is called a *bank-exposed architecture* [5].

Each memory bank is directly addressable by its local processing element but must be accessed through the interconnect when addressed by other processing elements. Thus the latency of a memory access increases when a processing element addresses non-local memory. As the interconnect must be decentralized for scalability, a memory access may have to go through a number of "hops" to be satisfied. The memory system then is like that of a NUMA, a Non-Uniform Memory Access machine, and the latency for memory accesses differs depending on the source processing element and destination memory bank of the access. Furthermore, such a system may have two mechanisms for accessing memory. The first mechanism may be used when the

memory bank in which the data resides can be determined at runtime; call these accesses *static* accesses. Static accesses are much faster than the second mechanism, *dynamic* accesses, in which the data's location (memory bank) must be determined at runtime.

Let us consider a compiler that takes a sequential program written for a uniform memory system and distributes the data and intructions across a system as described above. The goal of the compiler is to minimize runtime while ensuring correctness. To do this, the compiler must utilize the parallelism (instruction level parallelism - ILP - or otherwise) inherent in the program, create and utilize as many static accesses as possible, utilize knowledge of latency differences between processing elements and memory banks, and maximize *data affinity* without sacrificing parallelism. Data affinity is maximal when every instruction accesses memory locally addressable by the instruction's assigned processing element. The trade-off, of course, is to balance processor load while maximizing data affinity.

Several techniques have been evaluated and implemented in compilers to aid appropriate layout of data in the distributed memory of such a system. *Static promotion* creates static memory accesses that may be scheduled using the fast interconnect [4]. Static promotion makes use of *equivalence class unification* (ECU) and *modulo unrolling* [5]. ECU uses pointer analysis to help determine data placement such that data that may be accessed by a group of instructions is placed in the same compute unit as those instructions. In ECU, however, arrays are treated as a single object, forcing an array to be mapped to a single memory bank. (In contrast, fields of `structs` are treated as individual objects, allowing a single `struct` to be mapped across multiple memory banks.) Using loop unrolling, modulo unrolling creates static accesses out of array accesses in which the index expressions are affine transformations of enclosing loop induction variables [3]. This allows the arrays to be low-order interleaved across N memory banks while preserving data affinity: the unrolled loop instructions are also low-order interleaved across N memory tiles. ECU and modulo unrolling are useful when compiling dense matrix applications, but are not as useful when compiling sparse matrix applications. *Software serial ordering* enforces memory dependence of

dynamic accesses through explicit synchronization using static communication, allowing the latencies of dynamic accesses to be largely overlapped; this is particularly useful in sparse matrix applications when dynamic accesses must be used to obtain the desired parallelism.

These transformations allow the compiler to appropriately lay out data that is accessed deterministically. For arrays, the compiler can efficiently distribute array accesses located inside nested loops *when the index expression is an affine transformation of the loop induction variables.* For example, a loop of the form

```
for(i = 0; i < bound; i++) {
  for(j = 1024; j > jbound; j--) {
    A[i] = B[j/4 + 3i];
  }
}
```

may be distributed across available compute units. However, consider array accesses of the following form, an example of a sparse matrix application:

```
for(i = 0; i < bound; i++) {
  A[X[i]] = B[Y[i]];
}
```

This canonical example is essentially a series of random accesses of A and B. While X and Y may be distributed across compute units using modulo unrolling, A and B may not unless other loops exist to govern their placement. Distributing A and B forces all accesses to A and B within this loop to be dynamic accesses; such distribution can utilize software serial ordering to decrease synchronization overhead, but the dynamic accesses themselves are much more expensive than static accesses. On the other hand, if A and B are placed onto single compute units, all accesses to them become static but all are mapped onto a single compute unit, allowing no memory parallelism for A and B. This placement of A and B puts a lower bound on the execution time of the loop. However, if A and B are the bottleneck, then mapping X and Y to single compute units as well may be wise: it would remove execution pressure of the loads of X and Y from A's and B's compute tiles while not

increasing the running time, and it would free other tiles for any other computations that may be necessary. Each layout then limits the way in which accesses to the arrays may be scheduled. For example, if all of A, B, X, and Y are on the same tile, then loop unrolling will provide all speedup possible. If A, B, X, and Y are each on a different tile and if the communication latencies between X's and A's tiles, Y's and B's tiles, and B's and A's tiles can be determined at compile time, then appropriate loop unrolling and software pipelining may be utilized to obtain more speedup (due to utilization of parallel functional units) than in the first case. The difficulty here is to determine which memory and instruction layout will provide the best runtime speed and utilization. Spatial software pipelining, described in this thesis, does this assignment by placing each of X, Y, A, and B onto separate tiles; Figure 4-1 shows the final layout generated by our spatial software pipelining module.

Other issues that might affect the execution are describes in the next paragraphs. The branch condition that determines whether all tiles will continue executing the loop needs to be propagated to all tiles involved in computation. If the data flow in the layout interferes with the propagation path (to all tiles) of the branch condition, cycles are lost due to the interference. Thus either the layout must take into account the branch condition propagation path, or the branch condition propagation path must be specified to correspond to the layout such that interference between data flow and the flow of the branch condition does not arise.

Additionally, data flow between the tiles may interfere with itself in a given layout. That is, the data flow between tiles provides a certain amount of synchronization between tiles, and if the data flow is not orchestrated in an appropriate manner, the synchronization between tiles may become much tighter than is desirable. Tighter synchronization between tiles implies more possibilities for wasted processing cycles due to unforseen stalls and mutual data dependencies that must be satisfied in certain orders. Thus any layout algorithm must attempt to reduce the amount of unnecessary synchronization induced by data flow patterns. Chapter 3 explains this concept in more detail.

This thesis presents the mechanism spatial software pipelining, which may be

used to help reduce unnecessary synchronization when performing data and instruction layout, and describes the module that we have implemented to perform it. We begin this thesis in Chapter 2 by presenting background relevant to understanding spatial software pipelining as well as a brief background of the systems on which the implemented module is built. Next, in Chapter 3, we explain the motivation and basic philosophy behind spatial software pipelining and give a motivating example. In Chapter 4 we present the algorithms used in our spatial software pipelining module as well as pseudocode for those algorithms. In Chapter 5 we give the specifications for the interfaces between the module and the compiler on which it was built. We then present an analysis of the efficacy and applicability of spatial software pipelining and give results (obtained via a cycle-accurate simulator) in Chapter 6. We go over future work in Chapter 7, and then we briefly discuss related work in Chapter 8 before concluding.

Chapter 2

Background

This chapter presents background information helpful in understanding spatial software pipelining and briefly explains the system on which we have implemented the spatial software pipelining module. We first explain sparse matrix applications and traditional software pipelining (Sections 2.1 and 2.2), which help to understand the motivation behind spatial software pipelining. The later sections explain Raw and RawCC, the architecture and compiler on which we have implemented the module.

2.1 Sparse Matrix Applications

Spatial software pipelining specifically targets sparse matrix applications. Sparse matrix applications use quite large data sets in array or matrix form, but frequently have nondeterministic access patterns. As described in Chapter 1, it is very difficult to determine how to place data objects (or memory objects) across the distributed memory in a NUMA. Distributing nondeterministically accessed arrays forces expensive dynamic accesses and often increases execution time. This may be due either to the increased synchronization necessitated by preserving the order of dynamic accesses or simply to the increased cost of each access. However, determining how to place such arrays appropriately is very difficult, and is a problem not addressed by RawCC. Spatial software pipelining has been developed to help find appropriate memory and instruction assignments in the presence of such arrays.

2.2 Software Pipelining

Software pipelining is the inspiration for spatial software pipelining. Traditional software pipelining is a mechanism used to lengthen data dependencies between instructions in a single iteration of a loop. The basic idea is to restructure the kernel of the loop such that short dependencies between instructions within an iteration of the loop are remade into longer dependencies that span multiple iterations of the loop. The resulting code has: a *prologue*, the instructions necessary to prime the resulting software pipeline so that the new kernel of the loop can execute appropriately; the new kernel, in which a single iteration contains instructions executing from each of the multiple iterations of the original loop kernel; and an *epilogue*, the instructions necessary to drain the software pipeline after the new kernel finishes executing. Traditional software pipelining is particularly useful in conjunction with loop unrolling and VLIWs.

Spatial software pipelining extends software pipelining to apply across distributed architectures. Where software pipelining restructures the kernel of the loop to lengthen short dependencies, spatial software pipelining assigns the instructions in the loop to functional units to try to minimize the number of communication dependencies (not words) with other functional units. That is, spatial software pipelining attempts to pipeline the loop across multiple functional units; rather than decreasing the *latency* of one iteration as in traditional software pipelining, spatial software pipelining increases the *throughput* of the loop by removing the causes of unnecessary synchronization. We will show that spatial software pipelining thus allows a certain amount of slack to exist between the functional units, decreasing the effects of stalls on other functional units. Additionally, because spatial software pipelining effects the pipelining across functional units simply through instruction assignment, the code generation is much simpler than in traditional software pipelining; no prologue or epilogue is required. However, spatial software pipelining may not produce an assignment of instructions to functional units that is as load balanced as an assignment produced with more conventional means; this is because spatial software pipelining gives the

highest priority to removing communication cycles between functional units.

2.3 Raw Architecture

This section describes the Raw architecture, the architecture targeted by our spatial software pipelining module. The Raw microprocessor consists of a 2-dimensional mesh of 16 identical, individually programmable tiles. Each tile contains an in-order single-issue MIPS-4000-style processing element, a cache memory bank with a 32 KB data cache and a 32 KB instruction cache, and a switch with a static router, a dynamic router, and a 64 KB instruction cache for the static router [13], [8]. Each tile is fully connected with two communication networks: a *static network* and a *dynamic network*. The static network is used for static accesses as discussed above; it is much faster than the dynamic network as it has only a single-cycle latency from processor to switch (or vice versa) and a single-cycle latency across each tile. However, as discussed previously, memory accesses completing over the static network must have destination memory banks (here, tiles) known at compile time. The slower dynamic network is used for dynamic accesses; messages over the dynamic network are routed at runtime. Because each tile has its own pair of instruction streams for its processor and switch, different tiles may execute independently, using the networks (primarily the static) for memory and control dependences. See [13] for more details on the Raw architecture.

2.4 RawCC

RawCC is the parallelizing compiler developed to target Raw and the compiler on which the spatial software pipelining module is built. RawCC is built on top of the SUIF compiler infrastructure [14]; it takes a sequential C or Fortran program and parallelizes it, assigning instructions and data across the available tiles of Raw. RawCC consists of two major components: Maps and the space-time scheduler. Maps, the memory front end, is responsible for memory bank disambiguation, i.e., determining

which memory references may be static accesses and to what data such accesses refer. The space-time scheduler, the back end of RawCC, is responsible for mapping instructions and data to the Raw tiles as well as for scheduling instructions and communication. Both portions of RawCC are discussed in more detail in the following section.

2.5 Maps and the Space-Time Scheduler

Maps consists of three main sub-components. The first sub-component performs pointer analysis and array analysis, which provide information used for memory bank disambiguation. The second sub-component performs static promotion, identifying and producing static references through equivalence class unification (ECU) and modulo unrolling. Finally, the third sub-component performs dynamic access transformations, including software serial ordering (described previously) [2]. More detail on Maps can be found in [4], [3], [5] or [2].

The space-time scheduler handles coordination, both in terms of control flow across basic blocks (the control orchestrator) and in terms of exploiting parallelism within basic blocks (basic block orchestrator). The control orchestrator implements *asynchronous global branching*, branching across all tiles using the static network, and *control localization*, an optimization that allows some branches to affect only a local tile [9]. The basic block orchestrator assigns instructions and data to the Raw tiles, schedules the instructions, and orchestrates communication between the tiles. First, assuming N available tiles, the basic block orchestrator's instruction partioner uses clustering and merging to partition the original instruction stream into N instruction streams, balancing parallelism against locality (data affinity). Second, the global data partioner partitions the data into N data sets and associates each data set with one instruction stream, attempting to optimize for data affinity. Third, the data instruction placer uses a swap-based greedy algorithm to minimize communication bandwidth and place each data set-instruction stream pair on a physical tile. Fourth, the communication code generator creates the static switch code for each tile. Finally,

the event scheduler schedules the computation and communication instructions for each tile, attempting to minimize runtime while ensuring the absence of deadlock on the static network. More detail on the space-time scheduler can be found in [9].

RawCC currently uses loop unrolling to help partition instructions in the loops of sparse matrix codes, placing the instructions on functional units to minimize the number of *inter*-iteration dependencies that cross functional units. In contrast, spatial software pipelining tries to minimize the number of pairs of functional units between which *intra*-iteration dependencies exist. The main difference between the two is in the partitioning: RawCC places different iterations of the loop onto different functional units, but spatial software pipelining pipelines a single iteration of the loop across multiple functional units. Additionally, when placing conditional basic blocks like loops, spatial software pipelining determines the route by which the branch condition should be propagated (see the next section); this route is created to facilitate pipelining a single iteration of the loop. On the other hand, RawCC simply uses a fixed propagation route to broadcast the branch condition.

2.6 Asynchronous Global Branching

As mentioned above, asynchronous global branching is the mechanism that Raw uses to orchestrate a branch across all tiles. On such a branch, the *root* tile computes the branch condition and then broadcasts the condition to all other physical tiles via the static network. Each tile then receives the branch condition and takes a local branch on it at the end of the tile's basic block execution; each corresponding switch branches appropriately as well. Note that the local branches are performed asynchronously with respect to one another, giving rise to the name.

Chapter 3

Spatial Software Pipelining

This chapter presents the motivation and basic idea behind spatial software pipelining, the mechanism that we have developed to help partition loops in sparse matrix codes appropriately. We explain how communication cycles and other communication dependencies enforce synchronization between tiles and and show that when such synchronization can be avoided, the execution time of loops decreases. We give an example demonstrating the detrimental effects of communication cycles and give numbers comparing them. We explain that the basic idea in spatial software pipelining is to decouple the tiles by removing unnecessary synchronization due to the communication dependency graph between tiles. Specifically, the goal is to remove cyclic communication dependencies and to cluster instructions and data such that the compiled code efficiently uses the processing power available. We explain that removing cycles and other unnecessary synchronization may be done by forcing the communication dependency graph to resemble a tree by pipelining portions of a loop across multiple functional units as well as across the interconnect between functional units.

3.1 Motivation

As mentioned in Chapter 2, loops in sparse matrix codes frequently access arrays in ways that cannot make use of distributed data, or rather, in ways in which a deterministic access pattern can not be found. Such accesses to arrays cannot be

statically scheduled, and if an array is placed across multiple tiles then such a loop must use unacceptably slow, dynamic accesses. When an array is placed on a single tile, however, each iteration of the loop has at least two instructions that must be executed on the tile: the access (due to data affinity) and the branch on the loop condition. If the loop does not simply consist of an array access, which is generally the case, it may still be possible to exploit the parallelism in a single loop iteration by appropriately placing surrounding instructions and their related memory. The problem remains to find a good assignment of instructions and data to tiles. A good assignment will make use of as many tiles as are necessary to exploit the parallelism of the loop and no more.

We have developed spatial software pipelining on Raw to help create good assignments for loops such as those described above. Good assignments for such loops will in turn help to speed the execution time of RawCC's generated code for sparse matrix codes. Specifically, spatial software pipelining helps loops that cannot gain from other parallelization techniques such as modulo unrolling. Spatial software pipelining places instructions such that loop iterations are efficiently pipelined across the physical Raw tiles while problematic accesses to specific arrays are scheduled on a single tile. Data flow in each iteration is pipelined along with control flow information using Raw's aynchronous global branching. Thus spatial software pipelining is useful in loops where iterations can be easily pipelined and interleaved with each other - where a write to some problematic array is not connected via a long chain of instructions in the dependency graph to another access to that array.

Let us now look at a single iteration of a loop containing accesses to problematic arrays. For the rest of this section, unless otherwise specified, a *tile* refers to the collection of instructions scheduled to execute on a single physical tile as well as the collection of data to be placed on that phsyical tile. Working under the assumption that an array that is accessed randomly is better off on a single tile than interleaved across tiles, if a tile contains an instruction with a reference to such an array then the tile must contain *every* instruction with a reference to that array in that loop. A tile `dest` has an incoming dependency from another tile `src` if `dest` needs to receive any

data from `src` within a single loop iteration. The converse is an outgoing dependency. Thus both data and control flow information create dependencies between tiles on the paths by which they travel. Each dependency is fulfilled when the source tile routes the necessary data to the destination tile. Though statically scheduled, these routes need not be consumed immediately by the destination; the destination's switch may stall until the incoming data is ready to be used.

If a tile `tile` has both outgoing and incoming dependencies with a single neighbor `nt`, `tile` and `nt` are in a *communication cycle* with each other and are forced to execute in approximate lockstep, i.e., `tile` and `nt` must work on the same iteration at approximately the same time. If `tile` does not have enough instructions to fill the delay caused by communicating with `nt`, then `tile` stalls on every iteration, the utilization of `tile`'s physical tile goes down, and the total loop execution time increases. However, if the two tiles do not have cyclic communciation dependencies, then `tile` may be able to inject data from one iteration into the network and continue on to the next iteration even before `nt` consumes the data. This allows `tile` to pipeline data to `nt` using the static network, thus effectively pipelining the loop across `tile`, `nt`, and the static network. To sum up, while each communication dependency acts as a loose form of synchronization between two tiles, the combination of dependencies and the structure of the dependency graph can tighten the synchronization between tiles. Tighter synchronization is not ideal: it reduces the beneficial effects of individual control that Raw gains using separate instruction streams.

Extending the above observation, if a tile is part of a dependency cycle of any size, all tiles in that cycle must execute in approximate lockstep. Similarly, if a tile has two (or more) incoming dependencies from neighboring source tiles, the source tiles must execute in approximate lockstep with each other, and the chain extends backwards through the dependency graph. It may happen that ancestor synchronization is not a problem; however, if a tile has a single ancestor with two paths in the dependency graph from the ancestor to the tile, and if the paths are of vastly different lengths, then this could be a major problem due to the fact that the interconnect between processing units has limited buffer space. This gives rise to the observation that the

more the communication dependency graph resembles a tree, the less synchronization between tiles exists. This is the key idea in spatial software pipelining.

3.2 Example Motivating Tree-shaped Communication

This example shows the motivation behind trying to avoid communcation cycles. To show the adverse effects of communication cycles and dependencies, we examine the assignment found by our spatial software pipelining module for the loop in the canonical case described in Chapter 1. We make small modifications and compare the runtime of the loop to see the adverse effects mentioned. Recall that the C code for the canonical case is as follows.

```
for(i = 0; i < bound; i++) {
  A[X[i]] = B[Y[i]];
}
```

The execution times for each layout are given in cycles and are generated using Raw's cycle accurate simulator btl. The tile assignment for this dependency graph is shown in Figure 4-1. Each rectangle represents a physical tile on Raw. Each oval represents an instruction. Data-bearing dependencies from one instruction to another are shown by arrows; non-data bearing dependencies are not shown. Each dependency is drawn in the color of the source instruction's tile. The dotted ovals labelled `commi` represent the switches of each tile `i`; each data-bearing dependency that must be routed through multiple tiles to reach its destination is represented by a chain of arrows through the `comm` nodes on the data's actual route.

Figure 4-1 shows that the module-generated layout for the canonical case has data flowing in only one direction between each pair of connected tiles. This shows the freedom from enforced synchronization that is the result of spatial software pipelining. The runtime for this data and instruction layout is 179,230 cycles.

To see the effect of communication cycles, let us move instructions 0, 1, and 22 from tile 0 to tile 1. This is equivalent to moving the home tile of the loop induction

variable. Note that no computation is being moved: only the variable's home is being shifted. Running the loop for the same number of iterations with this slightly different layout gives an execution time of 377,662 cycles. The change in execution time is due to the fact that tiles 0 and 1 have been forced to synchronize with each other more fully through their communication. Previously the only synchronization between tiles 0 and 1 was due to the order of the data sent by tile 0 and the restricted network bandwidth between the tiles (that is, the fact that the network could only hold so many data words at once). Now there is additional synchronization due to the order of the data sent by tile 0 and tile 1 in relation to each other.

The following sequence of events demonstrates how the added synchronization affects the execution time (each instruction has cost 1). Table 3.1 shows in tabular form exactly which instruction is executed in each cycle, and also shows which words are in transit over the network on each cycle. The bottom half of the table shows for comparison the same sequence without the induced cycle; note that the moves are not necessary in that layout. In each iteration, tile 1 must first send the current value of i to tile 0 via instruction 0 while tile 0 executes instruction 4. Tile 1 then executes instruction 3; as soon as tile 0 receives i via instruction 1 (a delay of three cycles) tile 0 executes instructions 2, 5 and 6. While waiting for the load (6) to complete, tile 0 executes instructions 17 and 19, then sends the result of 6 to tile 1 while executing 20. Tile 0 continues with instructions 21, 18, which sends the updated value of i to tile 1, and finally 23 (the branch); then tile 1 receives the data of 6 from tile 0 and executes instruction 7, receives the branch condition (result of 21) and notes it, and finally executes instructions 8 and 9. Immediately tile 1 receives the updated value of i, executing instruction 22. Tile 1 can finally branch (24) on the branch condition and repeat the loop as given. Note, however, that tile 0 was idle while tile 1 executed the branch receive, 8, 9, 22, and 24 in addition to the idle cycles at the beginning of the loop iteration. As tile 0 was the bottleneck in any case, this clearly increases the execution time of the loop dramatically.

This is a simple example, but the increased synchronization between the two tiles is clearly apparent, as is the detrimental value of increased synchronization. Initializing

arrays X and Y to remove cache conflict in arrays A and B (here, setting every X[i] to one constant and every Y[i] to a non-conflicting constant) and comparing exection times demonstrates the detrimental value of increased synchronization even more forcefully. Table 3.2 compares the execution times with and without cache conflict for each of RawCC's unmodified layout, the layout produced by the spatial software pipelining module, and the modified layout with an induced communication cycle; additionally, it shows the speedup gained by removing all cache conflicts due to A and B. Note that with spatial software pipelining the speedup gained by removing cache conflicts is much less than that gained in the original layout. This is due to the fact that spatial software pipelining decouples the tiles to a certain extent; because the synchronization between tiles is much less, cache misses have less chance of influencing adjacent tiles. Even in the case where one communication cycle is induced, the removal of cache conflicts does not decrease execution time by the same factor as that in the original layout. This is yet another indication that decoupling the tiles - decreasing the synchronization between tiles - is beneficial to the execution time of a program.

3.3 The Basic Idea

The goal of spatial software pipelining is thus to remove cyclic communication dependencies and to cluster instructions and data such that the compiled code efficiently uses the processing power available. (If cyclic dependencies cannot be removed without dramatically reducing possible parallelization, we can leave the cyclic communication and utilize software pipelining appropriately on involved tiles to get better results. See Chapter 7 for more details.) Additionally, though not implemented, any assignment that turns a dependency graph into more of a dependency tree will have a better execution time.

We have implemented a module to perform spatial software pipelining at the basic block level. The module takes in the dependency graphs of instructions for each basic block within a program and creates memory and instruction assignments consistent

Table 3.1: Sequence of events for one iteration of the canonical case loop with induced communication cycle as compared to one iteration of the spatial software pipelined layout. This shows the instruction executed on each of tiles 0 and 1 in each cycle, and also shows each data word as it moves through the network. Note that there are 9 additional cycles per iteration in the induced loop.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T1,cyc	0	3	3	-	-	-	-	-	-	-	-	-	-	7	21	8	9	22	24
net,	-	0	0	0	-	-	-	-	-	-	6	6	6	21	18	18	-	-	-
cyc	-	-	-	-	-	-	-	-	-	-	-	21	21	18	18	18	-	-	-
T0,cyc	4	4	-	-	1	2	5	6	17	19	20	21	18	23	-	-	-	-	-
T1	7	8	21	9	24	3	3	-	-	-									
net	21	21	-	-	-	-	-	6	6	6									
net	-	-	-	-	-	-	-	-	-	21									
T0	2	4	4	5	6	17	19	20	21	23									

across basic blocks.

In RawCC's clustering, a preset propagation pattern is used to propagate branch conditions, and control flow information propagation paths are not taken into account when doing placement. This is a large contributor to extraneous communication dependencies; clearly if the control flow information is propagated over a path already being used for data propagation, communication dependencies are reduced. To address RawCC's problem, the module we have implemented generates a branch propagation route for each basic block, and RawCC generates the switch code necessary to propagate control flow information by means of the generated route. This increases the likelihood that the communication dependency graphs can be made to resemble trees.

As mentioned previously, the key idea in spatial software pipelining is that when communication patterns resemble trees, the execution time in loops is better. While spatial software pipelining does try to minimize communication, the main difference between spatial software pipelining and traditional heuristics minimizing communication is that spatial software pipelining assumes that commincation bandwidth is not the primary limiting resource. That is, spatial software pipelining assumes that enough bandwidth exists between tiles that extra *words* of data may be easily sent over the interconnect, but that the number of *paths* by which the data is sent should be minimized.

Table 3.2: The effects of removing cache conflicts (due to arrays A and B) with extra synchronization between tiles and without such synchronization. Note that removing cache conflicts has less of a performance benefit when the tiles are decoupled. We show here bloodgraphs for one iteration of the loop in each of the layouts; each bloodgraph is taken from an iteration in which no cache conflicts occur on any tile, and each begins with the first instruction after the branch on the tile that houses `i`. Note that white represents useful work being done; all other colors represent stalls. Each line in each bloodgraph shows the time and cause of stalls on that tile; the first line shows Tile 0, the second Tile 1, etc. For the induced loop layout, the tile that houses `i` is Tile 1; for the others, it is Tile 0. A key is included to show which color represents which kind of stall.

Cache Conflict	Original RawCC Execution Time	With Spatial Software Pipelining	SSP Layout With Induced Loop
yes	259,141	179,230	377,662
bloodgraphs			
no	184,059	171,239	324,830
Speedup	1.408	1.047	1.163

white	non-floating point op
red	net receive stall
blue	net send stall
green	resource/bypass stall
black	cache stall

Chapter 4

Algorithms

This chapter describes first, the input required for the spatial software pipelining module and second, the algorithms used within the module to perform memory to (physical) tile and instruction to tile mappings. Recall that the hoal of spatial software pipelining is to remove cyclic dependencies and reduce the amount of inter-tile synchronization created by communication. As we describe the algorithms, we will keep a running example to help explain each. The example will walk through the steps taken by the spatial software pipelining module to produce a layout for the canonical case described in Chapter 1. Recall that the C code for the canonical case is as follows.

```
for(i = 0; i < bound; i++) {
  A[X[i]] = B[Y[i]];
}
```

In our example, the number of target tiles on Raw is 4 (a 2 by 2 grid), and the loop is not unrolled. The dependency graph (less non-data bearing dependencies, which are used but not shown) is contained in Figure 4-1, which also shows the final layout assigned. Instruction 6 is the access of array Y, 13 that of X, 9 that of B, and 16 is the store to array A. As we walk through this example, this graph will be changed in various ways; these changes are shown in Figure 4-2, and we will refer to this figure throughout our descriptions.

4.1 Input Information

The input required for the spatial software pipelining module is as follows. First, the module requires the order in which basic blocks should be processed. As the module currently works on basic blocks independently of each other, a simple ordering is all that is required; Chapter 7 discusses possible extensions that would require more information. In our example, we will consider only the one basic block.

Second, the module requires the dependency graph between instructions on a basic block level. That is, it needs a list of the instructions to be executed in each basic block including information about each instruction (e.g. cost, latency); lists of the dependencies between instructions within the same basic block; and a mapping from each instruction to a *defId*, where two instructions that have the same defId must be executed on the same tile. Note that the first two lists together specify a dependency graph (specifically, a directed acyclic graph, or DAG) on the instructions, and note that the last mapping is a mechanism for ensuring that certain instructions, possibly in different basic blocks, execute on the same tile. Again, the dependency graph for our example is contained in Figure 4-1; in this figure, instructions 0, 1, and 22 share a defId and are thus pictured with dashed ovals.

Third and finally, the module requires a list of the memory objects and a mapping from each memory instruction (e.g. a load) to the *memory id* of the object that the instruction accesses. This map provides another mechanism for determining which instructions must execute on the same tile; all loads and stores to and from a given memory object (e.g. an array - currently treated as an entire object - or a field in a structure) must be located on the same tile as the object to make use of data affinity. The memory map ensures that this occurs. In Figure 4-1, each memory id is assigned a color; if an instruction is mapped to a memory id, then the text of the instruction is shown in the color of the memory id.

The spatial software pipelining module then works in five phases. In the first, it creates a virtual dependency graph based on the previously specified dependency graph; the new graph encapsulates the fact that all instructions with the same defId

must be on the same tile and, similarly, that all instructions with the same memory id must be on the same tile. We define the nodes in this graph as *virtual tiles* and the communication and other dependencies as *virtual dependencies.* Second, the module merges the virtual tiles together in a bottom up greedy fashion until the number of virtual tiles is less than or equal to the number of physical tiles. Third, the module assigns each virtual tile to a physical tile using simulated annealing and a cost function, described below, that encapsulates the ideas of spatial software pipelining. Fourth, the module creates the branch propagation paths as needed based on the ultimate tile assignment. The first phase occurs for all basic blocks before any later phase occurs for any basic block; the later phases are all performed on one basic block before the final phase, reconciliation, is performed on the next basic block. The following sections describe these phases in more detail.

4.2 Create Map

This section describes the phase that creates the virtual dependency graph. To construct the graph, the following steps are taken on each basic block. First we construct the virtual tiles, and then we construct the virtual dependencies. Once we have created the graph, we merge all virtual tiles in the same equivalence class as determined by memory ids and defIds. The rest of this section describes these steps in more detail.

Each instruction is represented by a *node*, which contains all relevent information about the instruction, including lists of outgoing and incoming dependencies. Each dependency between two nodes is represented by an *edge*, which encapsulates the dependency type, data type, source node, destination node, and any other relevent dependency information. In Figure 4-1, nodes are represented by ovals and edges are represented by arrows. Edges that do not represent the transfer of data (i.e. edges that are not true dependencies) are not shown.

The module begins by walking through each node and assigning it to a virtual tile in the following fashion. If the node has a memory id `memId` that has been seen before,

the node is added to the virtual tile corresponding to `memId`. If the node has a defId `defId` that has been seen before, the node is added to the virtual tile corresponding `defId`. If both `memId` and `defId` have been seen before, the corresponding virtual tiles are merged and the node is added to the resultant virtual tile. (Note that a node can have at most one memory id and at most one defId.) If the node must be placed on a given physical tile, the node is placed on the virtual tile corresponding to the appropriate physical tile; if such a physical tile has not been required thus far, a new virtual tile is created and marked as being necessarily assigned to the physical tile. (Except for branches on loop conditions, which will be discussed later, and memory ids and defIds, which may not be mapped to a specific physical tile, this is very rare.) Pseudocode representing the creation of the virtual dependency graph is given in Figure 4-5.

In our example, nodes 3 and 9 are mapped to the same memory id, red, because node 3 loads the base address of B and node 9 accesses B (this example is with no optimizations, so there is no code hoisting to move node 3 out of this basic block). Since 3 and 9 are mapped to the same memory id, they are placed in the same virtual tile. Similarly, 4 and 6, 10 and 16, and 11 and 13 are each placed in the same virtual tile. Additionally, nodes 0, 1, and 22 all share the same defId and are thus placed in the same virtual tile. All other nodes have their own virtual tiles. The virtual tiles resulting from the create map phase are shown in Figure 4-2(a).

Each virtual tile is responsible for maintaining state about the nodes that it encapsulates. Items such as which memory ids are represented, which defIds are represented, the cost of executing the virtual tile as a whole (as determined by a simple list scheduler), and the latency are among the data maintained by virtual tiles.

Because branches on loop conditions have dependencies due to branch condition propagation, including the branches in the virtual dependency graph effectively forces physical tile assignment to occur at the same time as virtual tile merges. Since the branch condition propagation path can be set after virtual to physical tile assignment, the branch dependencies should not affect either the virtual tile merge algorithm or the tile assignment algorithm. Thus the branches are added to dummy virtual tiles

that are not included in the graph being created; all other virtual tiles are included and are maintained in a list for future phases. In our example, nodes 23, 24, 25, and 26 are each assigned to a dummy virtual tile and not included in the virtual dependency graph; these nodes are not shown in Figure 4-2.

Each edge is assigned a virtual dependency as follows (note the similarity to the assignment of nodes to virtual tiles). The virtual tile `src` is the virtual tile to which the edge's source node has been assigned. Similarly, the virtual tile `dest` is the virtual tile to which the edge's destination has been assigned. If a virtual dependency already exists from `src` to `dest`, the edge is added to that virtual dependency. Otherwise, a new virtual dependency is created. In our example, each edge shown in Figure 4-2 is assigned to a new virtual dependency because no two dependencies share the same source and destination virtual tiles. However, for the sake of illustration, we assume that node 1 also has outgoing edges to nodes 4 and 6. Then the edge from 1 to 4 would have a new virtual dependency `vdep` created to house it, but the edge from 1 to 6 would be added to `vdep`.

Each virtual dependency is responsible for maintaining state about the edges it encapsulates; examples are the source nodes of the edges and the number of words of data moving across the virtual dependency (i.e. the number of different source nodes from which data-bearing dependencies emanate). Additionally, each virtual dependency is responsible for identifying its source and destination virtual tiles; similarly, each virtual tile is responsible for maintaining a list of its incoming and outgoing virtual dependencies.

After each basic block has had its virtual dependency graph constructed, we create equivalence classes by partitioning the set of memory ids and defIds such that if two ids are in the same virtual tile, then they are in the same partition. All virtual tiles in a partition are merged together such that after this portion of the create map phase no virtual tile is in the same partition as another virtual tile in the same basic block. To do this, this phase creates three lists: `mem-mem`, consisting of (memory id, memory id) pairs; `def-def`, consisting of (defId, defId) pairs; and `mem-def`, consisting of (memory id, defId) pairs. This representation is more efficient to compute and store

than traditional representations used to compute equivalence classes. Again, these lists represent memory ids and defIds that must be placed on the same physical tiles as each other and thus must be on the same virtual tiles as each other.

The following loop is executed until none of the virtual dependency graphs changes, performing the merges of virtual tiles in the same partition. First, for each basic block id pairs are added to the lists as follows. If a virtual tile `vtile` has multiple memory ids represented in it (remember that all nodes of the same memory id must be on the same virtual tile), all new pairs are added to `mem-mem`. If `vtile` has multiple defIds represented, all new pairs are added to `def-def`. If `vtile` has both memory ids and defIds represented, each new (memory id, defId) pair is added to `mem-def`. Next, for each basic block, `mem-mem`, `def-def`, and `mem-def` are walked through and virtual tiles are merged appropriately. That is, for each pair (`id1`, `id2`), if a virtual tile represents `id1` and a different virtual tile represents `id2`, merge the two virtual tiles. Since each such merge could create more id pairs, the entire list-merge loop must be repeated until no basic block has a virtual tile merge (note that this is a simple fixed-point algorithm). Pseudocode for this loop is provided in Figure 4-6. In our example, no id pairs are created and no virtual tiles are merged due to this part of the create map phase.

The resulting virtual dependency graphs, one for each basic block, represent a consistent view of the virtual universe at the end of this phase. That is, all defIds or memory ids that must be on the same physical tile are contained within the same virtual tile and all tile constraints from a given basic block have been propagated to all other basic blocks. In the next phase, the virtual tiles are merged together to reduce their number to less than or equal to the number of physical tiles; additionally, merges deemed appropriate due to spatial software pipelining are performed.

4.3 Merge

This section describes the phase that merges the virtual tiles until their number is at most the number of physical tiles. This and the next two phases occur on a single basic

block before assignments are reconciled with the virtual tiles of the next basic block. The next basic block is then put through these three phases itself. A merge can not occur between two virtual tiles corresponding to two different physiscal tiles; for this section, we will use "merge" to mean "merge if possible" and will omit the "if possible" from the rest of the discussion. This phase occurs in six steps. First, all virtual tiles that would be in a communication cycle due to branch condition propagation are merged together. Second, all dependency cycles in the virtual dependency graph are removed if possible. Third, data is collected for each virtual tile for use in the later parts of this phase. Fourth, two quick optimizations force some merges to occur early in the merge process. Fifth, the virtual tiles are sorted by critical value for the last phase. Sixth, the virtual tiles are merged in a bottom up greedy fashion until the number of virtual tiles is at most the number of physical tiles. The rest of this section describes these six steps in more detail.

In the first step of the merge phase, any virtual tiles with virtual dependencies from themselves to the virtual root tile are merged into the root tile. In a loop, the *root tile* is the tile on which the branch condition is calculated. The root tile essentially has outgoing dependencies to all other virtual tiles within the basic block because all physical tiles must obtain the loop condition from the root tile to be able to execute the end of loop branch. Thus, any tile which has an outgoing dependency to the virtual root tile has a cyclic communication dependency with that tile; in the spirit of spatial software pipelining, all dependency cycles should be removed. Note that this step does not occur in a basic block that does not have a branch condition to propagate.

In our example, the branch condition is calculated by node 21, shown in black in Figure 4-2. Thus the virtual tile with node 21 in it is the root tile, and all ancestor virtual tiles of it are merged into it. Looking at Figure 4-2(a), we see that the virtual tiles of nodes 20, 19, 17, and the tile of nodes 0, 1, and 22 must be merged into the virtual tile of 21 in that order. Additionally, the tile of 18 is merged in because of its outgoing dependency to 22, and the tile of 2 is merged in because it has an antidependency (not shown) to node 22. This last merge is not necessary since an an-

tidependency does not represent communication, but we do not currently distinguish between types of virtual dependencies when determining whether to merge. (Antidependencies are helpful later in creating dependency cycles that should be merged away.) The virtual tiles resulting after these merges are shown in Figure 4-2(b).

In the second step of the merge phase, a depth first traversal of the virtual dependency graph with a visited list allows this phase to find and remove dependency cycles by merging all virtual tiles in each cycle together. It is important to note here that although a DAG may be assumed for the dependency graph on nodes, the virtual dependency graph is most likely *not* a DAG. This portion of the merge phase is responsible for attempting to make the dependency graph between virtual tiles into a DAG again given the previous assignments and the memory id constraints.

The merging of dependency cycles in our example proceeds as follows. The virtual tile of nodes 3 and 9 has an outgoing virtual dependency (from 3) to the virtual tile of node 8, which has an outgoing virtual dependency itself (to 9) back to the virtual tile of 3 and 9, creating a cycle. This cycle can be seen in Figure 4-2(b). To get rid of this cycle, the virtual tile of 8 is merged with that of 3 and 9. Similarly, that of 15 is merged with that of 10 and 16, that of 5 with that of 4 and 6, and that of 12 with that of 11 and 13. This results in seven virtual tiles: the root (nodes 0, 1, 2, and 17 through 22), the access of Y (4, 5, and 6), the access of X (11, 12, and 13), the logical shift of Y[i] (7), the logical shift of X[i] (14), the access of B (3, 8, and 9), and the store to A (10, 15, and 16). The seven virtual tiles, Root, Y, X, shift Y, shift X, A, and B are shown in Figure 4-2(c).

In the third step of the merge phase, various data are collected for the virtual tiles and virtual dependencies. (Currently, gathering this information for dependency cycles is not handled well, so the data for virtual tiles in dependency cycles is not helpful. The reason is that rare virtual dependency cycles at this point are formed because of assignments in previous phases, and there is no way to remove these cycles. However, the data gathering techniques could be carefully thought out and corrected to gather the correct information for virtual tiles in cycles.) A virtual tile is a *parent* of another virtual tile if the one has an outgoing virtual dependency to the other.

The following paragraphs detail the data collected in this step, and Figure 4-3 shows examples demonstrating each.

The *estimated start time* (EST) of each virtual tile is an integer set to the greatest number across the parents of the virtual tile, where the number corresponding to a given parent is the parent's EST plus the parent's cost. The EST of a virtual tile with no parents is 0.

The *resident path cost* of each virtual tile, hereafter referred to as *path*, is an integer set to the greatest number across the children of the virtual tile, where the number corresponding to a given child is the path of the child minus the EST of the child plus the EST of the tile plus the cost of the tile. The path of a virtual tile with no children is the EST of that tile plus the cost of that tile. Thus, the paths of all virtual tiles on the critical path are exactly equal to the length of the critical path, while the paths of any tiles not on the critical path but merging into it are less than the length of the critical path by the amount of leeway between the sub-path and the critical path at the point of the merge.

The *path id* of each virtual tile is a bit vector in which each bit identifies a leaf tile (i.e. a tile with no children). On a virtual tile, the bit corresponding to a leaf tile is set if and only if the leaf tile is a descendent of the virtual tile.

Each virtual dependency is uniquely identified by a *color*, and the *palette* of each virtual dependency is computed as follows. The palette of each virtual dependency `vdep` contains color `clr` if one of the following is true: if `clr` is the color of `vdep`; or if `clr` is in the palette of some incoming virtual dependency to the source virtual tile of `vdep`. It is easy to see that the palettes for every virtual dependency in a cycle will be identical. Additionally, when one considers merging two virtual tiles, if some incoming virtual dependency of one has a palette that is a superset of the palette in some outgoing virtual dependency of the other, then merging those two virtual tiles will create a dependency cycle. (If the palettes are identical, then the dependency goes from one tile to the other or the two tiles were already in a cycle.) The palettes are a mechanism used to quickly determine when a cycle will be created by a merge of two virtual tiles; this allows us to consider merging together all nodes in a cycle as

soon as the cycle is created, thus removing it immediately.

Of course, if we handled dependency cycles properly with respect to EST and path, the module could gather all this data as soon as the virtual dependency graph is created (at the end of the first step). The dependency coloring mechanism could be used to do cycle detection and resolution instead of a depth first search with visited list. Since we do not handle dependency cycles correctly in this respect, we wait to gather all this data and do dependency coloring until after we have merged the cycles.

The *initial window size* is either the maximum cost of the virtual tiles or the sum of the cost of the nodes divided by the number of physical tiles. The initial window size represents the first bound that we will place on the size of virtual tiles; all merges should result in tiles with cost less than or equal to the window size (unless other conditions are met).

The values cost, EST, and path for each virtual tile in our example are included in Figure 4-2(c), which shows the virtual dependency graph after the cycles have been merged away. Note that virtual tiles X and Shift X are not included on the critical path from Root to A, so the paths of X and Shift X are lower than the paths of the other virtual tiles (17 instead of 21). The initial window size in our example is set to 8; this is the cost of virtual tile Root, the largest virtual tile in our virtual dependency graph. Note that if Root had only one node with cost 2, then the initial window size would be 5, the average cost of the nodes per physical tile.

In the fourth step of the merge phase, this phase runs through the virtual tiles twice to make two quick optimizations. In the first optimization, small virtual tiles (tiles with a small cost) that have no incoming dependencies and only one outgoing dependency are merged directly into their descendents if the resulting cost is less than the initial window size. This allows instructions such as moves and load immediates to be placed on the same tiles as their children, assuming it would cost more to communicate their results than to force the instructions to reside with their children. In the second optimization, if a virtual dependency between two virtual tiles has a cost greater than one (if it is responsible for communicating more than one word of data), then the two tiles on either end of that virtual dependency are merged together

if the resulting cost is less than the initial window size. Both of these optimizations are ways to circumvent the virtual tile merging functionality and force instructions that might otherwise be split up to reside on the same tile. While it is probable that these merges would be made in any case to reduce the communication between virtual tiles, doing these merges now allows the cost to be taken into account *before* considering merges with other tiles rather than after. In our example, neither of these optimizations has any effect, and the virtual tiles remain as in Figure 4-2(c).

In the fifth step of the merge phase, the virtual tiles are sorted based on the following properties, in this order: path, path id, EST, and virtual tile id. The resulting order of virtual tiles determines which tiles will be considered first in the following merges. Note that all leaf tiles will occur before their ancestors in this ordering if the ancestors are not part of a path to another leaf tile. This is because all ancestors will have a path less than or equal to the path of their leaf descendants and will have earlier ESTs.

In our example, Root, Y, Shift Y, A and B all have the highest path, and all of the virtual tiles have the same path id. (A is the only leaf tile, so all path ids are '1'.) Ordering these five virtual tiles by EST, we obtain A, B, Shift Y, Y, Root. Ordering X and Shift X by EST because they have the same (lesser) path, we obtain Shift X, X. Thus the final order of the virtual tiles in our example is A, B, Shift Y, Y, Root, Shift X, X.

At this point, the virtual dependency graph is copied for reversion. If the next part of the merge phase does not produce the necessary number of virtual tiles, the virtual dependency graph is reset to this point.

Finally, in the sixth step of the merge phase, the virtual tiles are merged together in a bottom up greedy fashion in the following way. The algorithm walks through each virtual tile `vtile` and attempts to merge `vtile` with other related virtual tiles until no other merges are possible. Currently we only consider merging `vtile` with its parents or with spouses (parents of children) that share a path id with the original (pre-merge) `vtile`, but we are investigating when it is reasonable to add siblings (children of parents) or children as well. In order to encourage merges that reduce

communication along the critical paths, the algorithm keeps a priority queue `priQ` sorted as before, and considers merges with relatives in the order that they appear on `priQ`. `PriQ` is initialized with all parents and spouses of `vtile`, and whenever a merge with some virtual tile on `priQ` is successful (determined as described below), any new parents or spouses are added to `priQ`. Each virtual tile is removed from `priQ` after a merge between it and `vtile` is considered, and this continues until `priQ` is empty, at which point `vtile` is set to the next virtual tile in the ordered list. Pseudocode for this loop is given in Figure 4-7.

While the priority queue remains non-empty, we determine whether a merge between `priQhead`, the element at the head of `priQ`, and `vtile` succeeds or fails. The list `toMerge` is initialized with `priQhead` and `vtile`, and is then defined recursively: if merging the virtual tiles of `toMerge` would create a cycle (this is where the palettes are used), then all virtual tiles in that cycle are added to `toMerge`. Next, the following terms are determined:

cost := the execution time of all the nodes in `toMerge` as determined by a simple list scheduler

conf := the number of virtual tiles in `toMerge` that represent a memory id

confc := 20 (the cost of data conflict; this value has been used because it has empirically been shown to work well; additional possibilities for determining conflict cost are presented in Chapter 7)

commc := 5 (the cost of additional communication; again, this value has been used because it has empirically been shown to work well)

ncm := the number of virtual dependencies between virtual tiles in `toMerge` and virtual tiles not in `toMerge`

ocm := the number of virtual dependencies between `vtile` and other virtual tiles

phystiles := the number of physical tiles to which the virtual tiles in `toMerge` are mapped

If $phystiles < 2$ and

$cost + confc * (conf - 1) + commc * ncm <= windowsize + commc * ocm$,

the considered merge is deemed successful and all virtual tiles in `toMerge` are merged together. In this case, all nodes in `toMerge` are removed from `priQ` and new parents and spouses of `vtile` are added to `priQ`. If the merge fails, we note whether the merge failed because we tried to merge fixed virtual tiles ($phystiles >= 2$) or because the window size was not large enough. We remove `priQhead` from `priQ` whether the merge succeeds or fails.

After walking through all of the virtual tiles, some number of post-merge virtual tiles exist. If the number of virtual tiles is less than or equal to the number of physical tiles, then the basic block moves on to the next phase, tile assignment. If the number of virtual tiles is greater than the number of physical tiles, however, one of two things happens. If all failed merges in this phase were due to trying to merge fixed virtual tiles (as opposed to having too small of a window size), the walk through each virtual tile is repeated, ignoring any cycles. That is, in considering each merge, only `priQhead` and `vtile` are added to `toMerge`; any virtual tiles in created cycles are ignored. This allows us to do placement for basic blocks in the presence of cycles that can not be merged away. Otherwise, the window size was too small for some merge to succeed. In that case, the virtual tiles are reverted to a pre-merge state, the window size is increased, and the last (sixth) step of the merge phase is attempted again. This continues until a suitable number of virtual tiles is obtained.

In our example, the seven virtual tiles shown in Figure 4-2(c) are merged down into the four virtual tiles shown in Figure 4-1 in the following way. First, tile A is considered for merges first because it appears first in our ordering (constructed in step five). Recall that the window size is 8 and that this puts a limit on the final cost of any merges. Virtual tiles A and B conflict because they are mapped to different memory ids, so the conflict cost (20) plus the cost of the resulting virtual tile (10: cost 4 plus cost 4 plus the latency of the load of B, 2) is too high; tiles A and B are not merged. However, A is considered for a merge with Shift X; this merge succeeds, creating virtual tile A Shift X cost 5. B is then merged with Shift Y but not with

Y (Y and B conflict). Y is not merged with Root because the resulting virtual tile would be cost 12 (which is greater than the window size) and the communication of Y would not be reduced. X is not merged with Root for the same reason. There are thus five virtual tiles formed by this process, and since four are needed, the virtual tiles revert to the original seven, the window size is increased, and the process repeats. This occurs until the window size reaches 12, at which point Y is merged with Root and four virtual tiles result.

4.4 Tile Assignment

This section describes the tile assignment phase, the phase responsible for taking the final virtual tiles and assigning them to physical tiles. Note that, as mentioned previously, the tile assignment occurs independently from the merge phase. Certain restrictions are enforced due to the assignments of previous basic blocks, but otherwise we assume first, that we can find a decent placement given the final virtual tiles and second, that creating the appropriate number of virtual tiles without considering placement yields an easier, cleaner merge algorithm and better clusters.

The tile assignment phase uses simulating annealing to determine a final placement. Any virtual tile that already corresponds to a physical tile is immediately placed on the appropriate physical tile. The remaining unmapped virtual tiles are placed randomly across the remaining unoccupied physical tiles, with the constraint that each physical tile can have at most one virtual tile (excluding the branches that were removed from the virtual dependency graph at the first). This constraint is held throughout the mapping process, as is the constraint that fixed virtual tiles are mapped to their corresponding physical tiles. The simulated annealer then tries to minimize communication distance, communication size, and most importantly, the number and size of communication cycles. We determine number and size of cycles by doing essentially a depth first traversal with visited list of the directed graph formed by the physical tiles and the data sent across the interconnect between adjacent tiles.

After the initial assignment, simulated annealing is performed with the following

cost function, described below, until either an appropriate number of iterations has been executed or the cost function returns cost 0. (This is the optimal cost.) Currently the appropriate number of iterations is a hard-coded 5,000, but one could imagine having the number of iterations decrease as the number of non-fixed virtual tiles decreases. Clearly having a fixed number of iterations is not helpful given that the number of target tiles can change (specifically, can increase), but currently we are testing this module on at most 16 physical tiles; the bound for simulated annealing does need to be more carefully planned and implemented for this module to be more generally applicable. At each step, if the cost of the assignment at that step is less than the optimal cost, then the optimal cost and assignment are set to the cost and assignment evaluated at that step. When the simulated annealer exits, the assignment chosen is that of the lowest-cost assignment found during the annealing process.

Define a *link* as a connection between two neighboring tiles across which data may flow in a single direction. The cost function to evaluate each assignment takes into account the premises of spatial software pipelining using the following rules: one, that data travelling from a tile to a neighboring tile travels for free; two, that if the same piece of data travels along two routes and the second route shares some initial links with the first route, then the shared links are free for the second route; additionally, if only the last link of the second route is different, then it is similar to point one, and the last link of the second route is also free; three, that otherwise each piece of data adds cost one for each link it must cross; four, that data entering an unoccupied tile is doubled in cost (this represents the idea that unoccupied tiles may be used to run other things); five, that if there is no dependency between any one tile and its neighbor tile along a given route, then the data has "left the path", and data on every link in that route after the path has been left is doubled in cost (this represents the idea that dependencies between neighboring tiles forge channels by means of which it is easier to propagate data); six, that the number of data words over the window size (defined in the merge phase) that travel in either direction between two neighboring tiles affects all tiles, and so the product of the difference and the number of tiles is the additional cost due to too much traffic between two tiles; and seven, that

communication cycles affect the tiles by effectively multiplying the window size by the number of tiles in the cycle, and so the additional cost due to each cycle is the product of the window size, the number of tiles, and the number of tiles in the cycle. Pseudocode for the cost function is given in Figures 4-8 and 4-9.

In our example, none of the virtual tiles must be placed on specific physical tiles, so the initial assignment is random. However, the simulated annealer quickly finds an optimal assignment for our example in which the cost is zero. The final placement puts tile A on physical tile 3, B on 1, X on 2, and Y on 0 (see Figure 4-1). Recall that the possible communication channels for the given tiles look like this:

```
0 - 1
|   |
2 - 3
```

Thus physical tile 0 has interconnect channels to and from tiles 1 and 2, but not to 3. Similarly, 1 and 2 cannot communicate directly. The final placement is appropriate since tile 3 only receives data from tiles 2 and 1 (not counting the branch condition, which originates on tile 0), tiles 2 and 1 each only receive data from 0 and send data to 3, and tile 0 only sends data.

Since our example thus far is too simple to adequately show how the cost function of simulated annealing is calculated, let us walk through a calculation of the layout in Figure 4-4. In the figure, each square represents a physical tile, each oval represents a virtual tile, and each arrow represents a virtual dependency. We assume that each virtual dependency represented as a solid arrow represents one word of data, and that the dotted arrows represent dependencies that encapsulate the same data word; that is, the dotted arrows represent a single data word being sent to multiple destinations. Each physical tile is labelled with a letter that we will use to refer to both the physical tile and the corresponding virtual tile. Note that F and D do not have corresponding virtual tiles. We will refer to virtual dependencies as edges for the rest of this example. According to the first rule, all edges from a tile to a neighboring tile are free; thus

A-B, C-G, G-K, H-L, L-H, K-O, O-N, N-M, M-I, I-J, and I-E are all free. The second rule says that data already travelling along a given route is also free, including one additional free edge. The dotted edges represent the same piece of data, therefore C-K is completely free, and the C-K portion of C-P and C-O portion of and C-N are free. The third rule says that otherwise each piece of data on each link adds cost 1; we add 1 each for the B-F and F-G portions of B-G, 1 each for the G-K and K-O portions of G-O, 1 each for the K-O and O-P portions of C-P, and 1 for the O-N portion of C-N. The cost is now 7. The fourth rule says that data entering an empty tile should double in cost, so we add 1 additional for the B-F portion of B-G bringing the total cost to 8. The fifth rule says that neighbors should communicate, stipulating that data that has "left the path" should double in cost. We see that there is no edge directly from B to F nor from O to P, but that certain edges do cross these links. Thus the B-F portion of B-G is doubled, adding 2 to our cost, the F-G portion of B-G is doubled, adding 1, and likewise, the O-P portion of C-P is doubled, adding 1 to our cost. This brings our total cost to 12. The sixth rule says that each word of data over the window size increases the cost by the number of tiles. In our example here, we assume that the window size is 2 and that the number of tiles is 16. The link from G to K is shown carrying three words of data, adding 16 to our cost; similarly, the link from K to O adds 16 to our cost, bringing the new cost to 44. Finally, the seventh rule says that each cycle increases the cost by the product of the window size, the number of tiles, and the number of tiles in the cycle. We see in Figure 4-4 that H and L are in a two tile cycle, increasing the cost by $2 * 2 * 16 = 64$, and that I, J, M, and N are in a four tile cycle, increasing the cost by $2 * 4 * 16 = 128$. Thus our final cost in this example is 236. A better assignment would place P the virtual tile on P on tile D, that on B on F, and that on A on B, reducing the cost by 8 (2 for the O-P link, 4 for the B-F link, and 2 for the F-G link).

Recall that the final assignment is the assignment with the lowest cost found at

any point by the simulated annealer.

4.5 Branch Propagation Path

Once the tile assignment phase has completed, the branch propagation phase creates the branch propagation route for the basic block. This phase only executes if a branch condition needs to be propagated to all of the physical tiles. However, if a branch condition does need to be propagated, this phase is responsible for trying to reuse paths that already exist from the previous phases; if we use pre-existing paths between tiles, we can avoid creating new inter-tile dependencies that could result in unnecessary synchronization. This phase executes in two parts: in the first, each link between two neighboring tiles is assigned a cost representing whether a dependency exists between the two tiles; in the second, each tile is assigned a neighboring tile from which to receive the branch condition. Note that this algorithm assumes that some path connects the root tile to every other tile.

The first part of this phase is a simple test on each outgoing dependency from each occupied tile `ptile`. (The virtual tiles have by this point all been assigned to physical tiles.) If a dependency from `ptile` to its neighbor `nt` exists, then the link from `ptile` to `nt` is assigned a cost equal to the EST of `ptile`. If a dependency from `nt` to `ptile` does not exist, then the corresponding link is assigned a negative cost. If no dependency in either direction between `nt` and `ptile` exists, the links in both directions are assigned cost zero. In our example, the links from tile 0 to tiles 1 and 2 are assigned 0, the EST of tile 0, and the links from tiles 1 and 2 to tile 3 are assigned 12.

The second part of the branch propagation phase proceeds as follows. First, the root tile is assigned to itself as its branch condition source (since the root tile is the tile from which the branch condition propagates). While some tile still has an unassigned

source, the following steps are executed. For each physical tile (including those that are unoccupied), a branch condition source is assigned to any tiles that do not already have one if possible; the methodology for assigning such a source is described in the following paragraph. If all tiles have assigned sources, the loop exits. If no tiles were assigned sources in this iteration, the tile with the earliest EST that can accept a forced assignment is forced. Note that some unassigned tile must accept a forced assignment at this point. This loop is repeated until all tiles have been assigned a source, at which point the branch condition propagation path has been generated, and the path may be noted for RawCC.

The algorithm for determining the branch condition source tile for each non-root tile is as follows. (Pseudocode is given in Figure 4-10.) If some of the tile's parents have not been assigned and the assignment is not currently being forced, no source tile is assigned. If a tile has no incoming dependencies (i.e. all links to it were assigned a value less than or equal to zero in the first portion of this phase), the source tile is set to be the neighboring tile with the least EST that has already received the branch condition. If no neighboring tile has received the branch condition, no source tile is assigned - the assignment fails. If a tile does have incoming dependencies, the source tile is set to be the neighboring tile that has already received the branch condition and for which the link value is minimal but positive. If this fails and the assignment must be forced, the source tile is set as before to be the neighboring tile with the least EST that has already received the branch condition.

In our example, every tile is assigned a branch condition source tile on the first iteration. Tile 0 is assigned itself because it is the root tile. Tile 1 has one incoming dependency from tile 0; since tile 0 has already received the branch condition, tile 1's branch condition source tile becomes tile 0. Similarly, tile 2's branch condition source tile becomes tile 0. Tile 3 has two incoming dependencies, one each from tiles 1 and 2. Since both tiles 1 and 2 have received the branch condition, tile 3 receives

the branch condition via the length with the lowest value; since both links from tiles 1 and 2 to tile 3 have value 12, tile 3's branch condition source tile becomes 1.

Once all of the tiles have been assigned a branch condition source tile, it is easy to verify that the branch condition propagation route has been constructed appropriately (with no cycles, for example) while creating the text required for the compiler. First, initialize an array of length two times the number of tiles to -1, and ensure that each tile has a unique id between zero and the number of tiles. Then call a recursive function, defined as follows, on the root tile. The recursive function, called on tile `ptile`, first verifies that the value at index two times `ptile` is less than zero; if not, the route was constructed in error. Next, the function sets the value at index two times `ptile` to the assigned branch condition source tile, `srct`, and it then sets the value at the next index to `ptile`. Finally, for each neighboring tile `nt`, if `nt`'s branch condition source tile is `ptile`, the function calls itself with `nt` as the argument. Note that `srct` is a neighbor of `ptile`, but can not have `ptile` as its source tile. Pseudocode may be found in Figure 4-11. Once the function returns from the root tile, any indices in the array still less than zero indicate that the branch condition propagation route has not reached all tiles appropriately. (With the above algorithm for creating the route, though, the route is always created appropriately.) The final route for our example would be '0 0 0 1 0 2 1 3', indicating, as expected, that our route was correctly generated.

4.6 Reconciliation

After the merge, tile assignment, and branch propagation path phases complete for a given basic block, as discussed in Sections 4.3, 4.4, and 4.5, the next basic block to undergo these phases must be reconciled to previous assignments. This phase may actually be thought of as occuring before the merge phase for a given basic block, but

for clarity we have left it until now for discussion. Essentially, this phase is responsible for updating the virtual dependency graph based on the assignments of previous basic blocks. That is, any defId or memory id that appears in basic block `bblock` and that has been assigned to a physical tile `ptile` in a previous basic block must also be assigned to `ptile` in `bblock`. Additionally, multiple virtual tiles in `bblock` may have nodes with defIds or memory ids that must be mapped to `ptile`; all of these virtual nodes must be merged together in `bblock` before the merge phase of `bblock` initiates.

Once all virtual tiles in all basic blocks have been assigned physical tiles, the resulting clustered instructions may be scheduled and run. Note that the outcome is very dependent on the order in which basic blocks are assigned. As mentioned earlier, this is still done by hand.

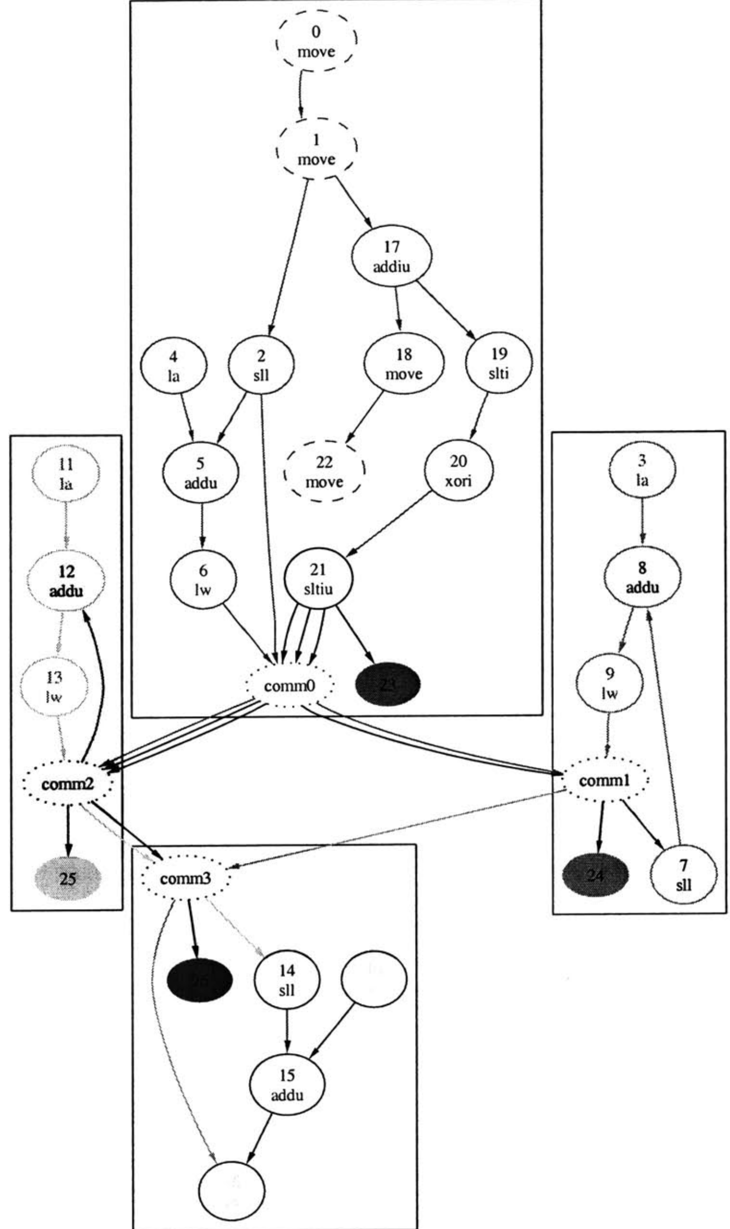
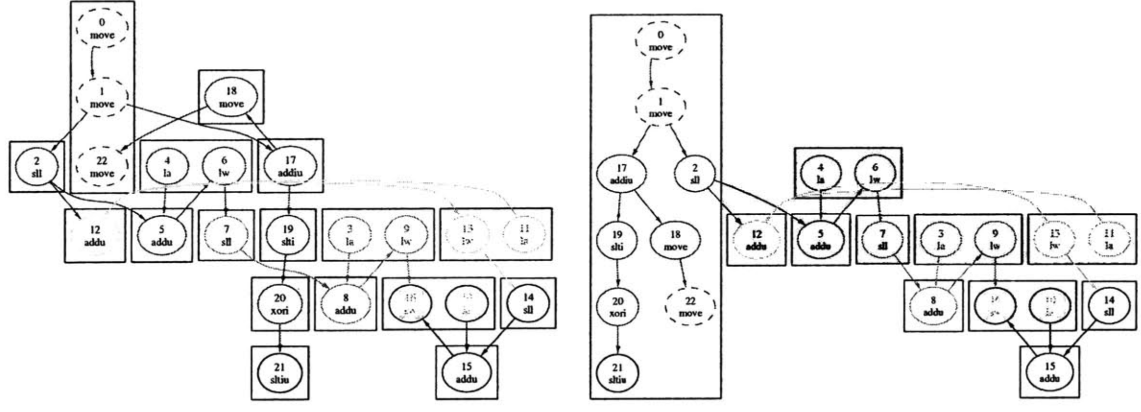
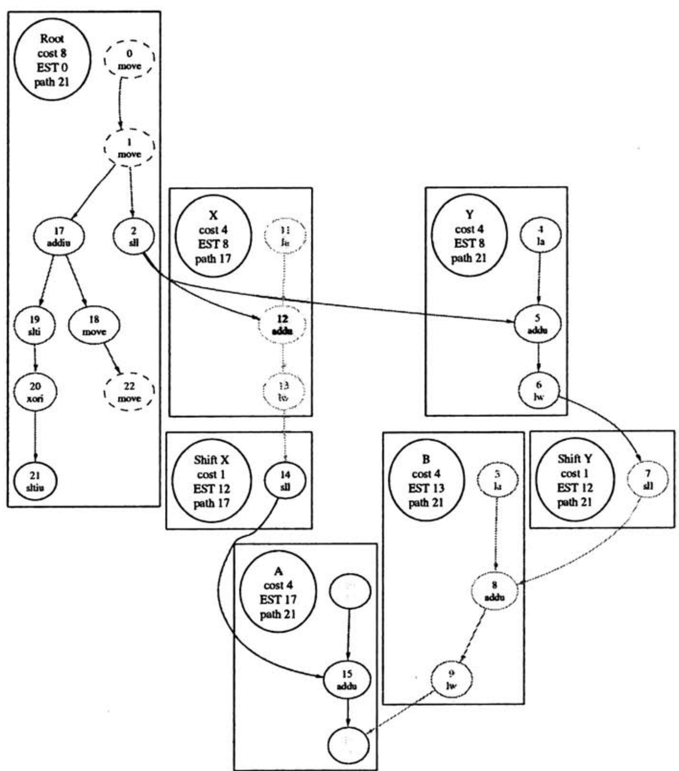


Figure 4-1: The final tile assignment of the canonical case described in Chapter 1. Rectangles represent physical tiles on Raw and surround the instructions assigned to execute on the tiles; the switch for each tile is represented by the dotted oval labelled `comm`. Ovals represent instructions; directed solid arrows between ovals represent data-bearing dependencies. Each dependency is drawn in the color of its source instruction's tile; inter-tile dependencies are represented by a chain of arrows connecting the instructions through the `comm` switches along the route. The root tile is shown in black, as are all branch dependencies. If a node is memory mapped, the text of the node is in the color corresponding to the memory id. The dashed ovals share a common defId.



(a) Virtual tiles after creating virtual map

(b) Virtual tiles after merging in to root tile



(c) Virtual tiles after merging cycles together

Figure 4-2: The intermediate stages of the virtual dependency graph when performing spatial software pipelining on the canonical case described in Chapter 1. Figure (a) shows the virtual dependency graph immediately after creation; figure (b) shows the result of merging virtual tiles with dependencies to the root tile into the root tile; figure (c) shows the result of removing cycles. The figure specifications are the same as in Figure 4-1, which shows the final layout found by spatial software pipelining.

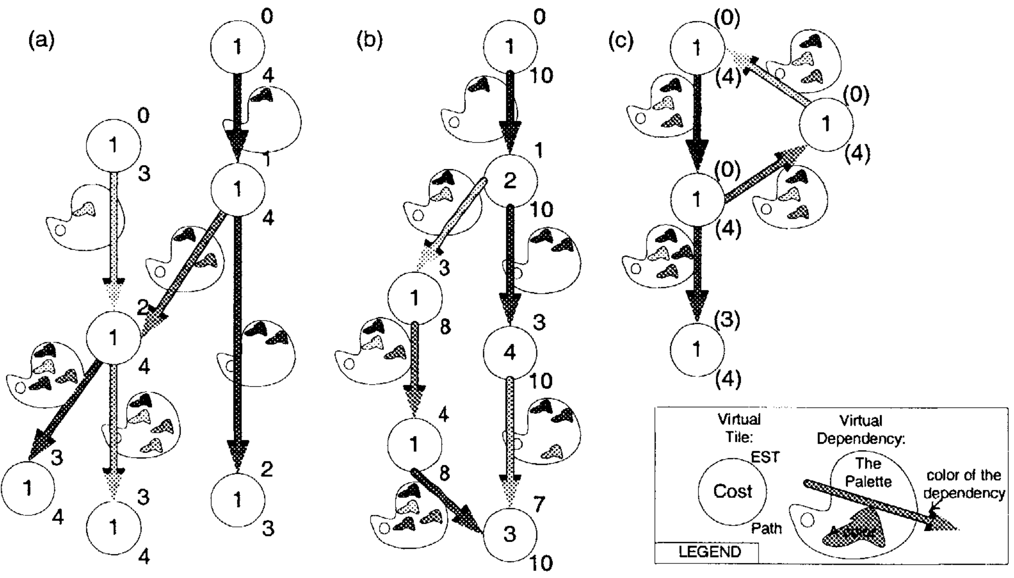


Figure 4-3: Three examples demonstrating EST, path, and cost of virtual tiles, and color and palette of virtual dependencies. Figure (a) demonstrates how the palette is constructed for each virtual edge, figure (b) demonstrates how EST and path are calculated for each virtual tile, and figure (c) demonstrates the effects of dependency cycles on palettes. Note that the EST and path variables in (c) are in parentheses; these are not the EST and path used by the current module, but are possibilities for a more correct implementation. All other EST and path variables are shown as calculated in the current module. Recall that when considering the merge of two virtual tiles, the merge creates a new dependency cycle when the palette of some incoming dependency to one of the tiles is a strict superset of the palette of some outgoing dependency from the other tile.

Figure 4-4: An example demonstrating how the cost function is calculated in simulated annealing in the tile assignment phase. Each square represents a physical tile, each oval a virtual tile, and each arrow a virtual dependency. The dotted arrows all represent the same data word being sent to different destinations. The window size is assumed to be 2, and the number next to each edge on each link represents the amount added to the cost due to that edge on that link. The numbers within or just next to cycles represent the additional cost added because of the cycles. The total cost of this layout is 236.

```
void createVirtualDepGraph(int num_tiles) {
  make physical tile representations to hold end−of−loop branches;
  for each node n (instruction) {
    if (n is an end−of−loop branch on physical tile pt) {
      add n to pt; continue; }
    Virtual Tile n_so_far = null;
    if (n has a memory id mi) {
      if (mi has been seen before on some virtual tile t_mi) {
        add n to t_mi; n_so_far = t_mi; } }
    if (n has a defId di) {                                               10
      if (di has been seen before on virtual tile t_di) {
        if (n_so_far) {
          merge(n_so_far, t_di); }
        else {
          add n to t_di; n_so_far = t_di; } } }
    if (n is fixed on physical tile pt, but has no memory id or defId) {
      if (pt has been seen before on virtual tile t_pt) {
        add n to t_pt; /* t_pt is a virtual, not a physical, tile */
        n_so_far = t_pt; } }
    if (n_so_far is still null) {                                         20
      create a new virtual node t to encapsulate n; } }

  for each virtual tile t in the set of virtual tiles {
    for each node n in t {
      for each incoming dependency d to n {
        virtual tile src = the virtual tile of the src node of d;
        if (src == t) { /* self-edge; nothing to create */ }
        if (there exists a virtual dependency vd from src to t) {
          add d to vd; }
        else {                                                            30
          create a virtual dependency from src to t encapsulating d; } } } } }
```

Figure 4-5: C-like pseudocode exhibiting how to create the virtual dependency graph.

```
forceIdsTogether() {
  for each basic block b { lsForcedMerges(b, mem_prs, def_prs, mem_def_prs); }
  do until no more change {
    changed = false;
    for each basic block b {
      if (doForcedMerges(b, mem_prs, def_prs, mem_def_prs)) {
        changed |= lsForcedMerges(b, mem_prs, def_prs, mem_def_prs); } } } }

bool doForcedMerges(BasicBlock b, list mem_prs, list def_prs, list mem_def_prs)
{                                                                                   10
  bool changed = false;
  for each pair (m1, m2) in mem_prs {
    if (a virtual tile t1 mapped to m1 exists in b &&
        a virtual tile t2 mapped to m2 exists in b && t1 != t2) {
      merge(t1, t2); changed = true; } }
  for each pair (d1, d2) in def_prs {
    if (a virtual tile t1 mapped to d1 exists in b &&
        a virtual tile t2 mapped to d2 exists in b && t1 != t2) {
      merge(t1, t2); changed = true; } }
  for each pair (m1, d2) in mem_def_prs {                                           20
    if (a virtual tile t1 mapped to m1 exists in b &&
        a virtual tile t2 mapped to d2 exists in b && t1 != t2) {
      merge(t1, t2); changed = true; } }
  return changed; }

bool lsForcedMerges(BasicBlock b, list mem_prs, list def_prs, list mem_def_prs)
{
  bool changed = false;
  for each virtual tile t {
    if (a node (instruction) n1 exists in t mapped to memory id m1 &&               30
        a node n2 exists in t mapped to a different memory id m2 &&
        the pair (m1, m2) is not in mem_prs && nor is (m2, m1)) {
      add (m1, m2) to mem_prs; changed = true; } }
    if (a node n1 exists in t mapped to defId d1 &&
        a node n2 exists in t mapped to a different defId d2 &&
        the pair (d1, d2) does not exist in def_prs && nor is (d2, d1)) {
      add (d1, d2) to def_prs; changed = true; } }
    if (a node n1 exists in t mapped to memory id m1 &&
        a node n2 exists in t mapped to defId d2 &&
        the pair (m1, d2) does not exist in mem_def_prs) {                          40
      add (m1, d2) to mem_def_prs; changed = true; } }
  return changed; }
```

Figure 4-6: C-like pseudocode for loop to enforce memory id and defId mapping consistency across basic blocks.

```
int mergeNodes() {
  /* returns the number of virtual tiles that result from this
     function; this function is called within a loop that first resets
     the virtual tiles to those of just before the first time this was
     called, calls this, and then repeats if the number returned by
     this is > the number of physical tiles */
  for each tile t in the list of virtual tiles {
    resetPriQ();
    path_o_interest = t->path_id;
    addParentsToPriQ(t);                                                    10
    /* adds the other relatives - spouses, siblings, children - to the
       priQ if their path_ids are the same as the path_o_interest*/
    addRelativesToPriQ(t, path_o_interest);
    while (!priQ.empty()) {
      Tile toCheck = pop the head off priQ;
      int new_comm_cnt = 0;
      /* checkMerge returns the estimated cost of merging t, toCheck
         and all virtual tiles in any dependency cycles */
      int tmp = checkMerge(t, toCheck, new_comm_cnt);
      int curr_comm_cnt = getNumComms(t);                                   20
      int COMM_CYCLES = 5; /* count each communication as 5 cycles */
      if ((tmp + (COMM_CYCLES * new_comm_cnt)
           <= winSize + (COMM_CYCLES * curr_comm_cnt))) {
        addParentsToPriQ(toCheck);
        addRelativesToPriQ(toCheck, path_o_interest);
        merge(t, toCheck); } } }
  return size of virtual tiles; }
```

Figure 4-7: C-like pseudocode for the function used to merge virtual tiles. This function is called after dependency cycles have been merged away (if such merges were possible).

```
int calcLayoutEntropy() {
  for each tile t in the list of virtual tiles {
    /* empty all incoming links to t of any source instruction ids */
    zeroSrcs(t);
    for each outgoing dependency d from tile t { d->flag = false; } }
  int ent = 0;
  for each tile t in the list of virtual tiles {
    for each outgoing dependency d from tile t {
      ent += calcEdgeEntropy(t, d, d->dst->phsyical tile id); } }
  for each tile t in the list of virtual tiles {                              10
    ent += NEEdgesConflictCost(t);
    ent += NEEdgesOversizeCommCost(t, nTiles); }
    ent += inCommCycle(t, winSize, tGoal, nTiles);
  return ent; }

int NEEdgesConflictCost(Tile t) {
  int ret = 0;
  /* conflict due to bidirectional movement */
  for each neighbor tile nt that t monitors { /* N and E neighbors */
    if (link(t, nt)->srcs->size != 0 && link(nt, t)->srcs->size != 0) {      20
      ret += nTiles * (link(t, nt)->srcs->size+link(nt, t)->src->size); } }
  return ret; }
int NEEdgesOversizeCommCost(Tile t, int nTiles) {
  int ret = 0;
  /* conflict due to too much movement */
  for each neighbor tile nt that t monitors { /* N and E neighbors */
    if (link(t, nt)->srcs->size + link(nt, t)->srcs->size > wSize) {
      ret += nTiles*(link(t,nt)->srcs->size+link(nt,t)->srcs->size-wSize);
  return ret; }
int inCommCycle(Tile t, int winSize, Tile goal, int nTiles) {                 30
  if (t->flag) {
    if (t == goal) { t->comms += winSize * nTiles; }
    return 0; }
  t->comms = 0; t->flag = true;
  for each neighbor tile nt { /* N, S, E, W */
    if (link(t, nt)->srcs->size() > 0) {
      inCommCycle(nt, winSize, goal, nTiles); } }
  t->flag = false; return t->comms; }
```

Figure 4-8: C-like pseudocode for the cost function for simulated annealing. The function calcLayoutEntropy walks through each dependency and adds the appropriate cost due to that dependency; additionally, it calls two helper function to add in the cost due to conflicting data flow and too much data flow.

```
int calcEdgeEntropy(Tile first_tile, Dependency me, int dst_id) {
  if (me->flag) { return 0; } /* saw already; cost included */
  Tile src_tile = first_tile;
  /* getRoute returns the next tile in routing from src to dst_id;
     in Raw, this is dimension ordered routing by assumption */
  Tile next_tile = getRoute(src_tile, dst_id, x);
  if (next_tile->id == dst_id) {
    /* add the srcs - the ids of source instructions - of this dependency
       to this link if they are not already there */
    addSrcs(src_tile, next_tile, me);                                          10
    /* don't add anything to the entropy cost; this is a free edge */
    me->flag = true; return 0; }

  int ent = 0;
  while (src_tile->id != dst_id) {
    /* look at all smaller dependencies that might use the same route
       path first so that their srcs are added to links first */
    if (next_tile->id != dst_id &&
        there is a dependency de from first_tile to next_tile) {
      ent += calcEdgeEntropy(first_tile, de, next_tile->id); }                  20
    src_tile = next_tile; next_tile = getRoute(next_tile, dst_id, x); }

  src_tile = first_tile; next_tile = getRoute(src_tile, dst_id, x);
  bool on_path = true; int old_ecnt = INT_MAX;
  while (src_tile->id != dst_id) {
    if (on_path && there is no dependency from src_tile to next_tile) {
      on_path = false; }
    /* if any srcs are already on this link from src to next, those srcs
       are free; addSrcs returns the number of new srcs on this link */
    int ecnt = addSrcs(src_tile, next_tile, me);                               30
    int toadd = ecnt;
    if (old_ecnt < ecnt && next_tile->id == dst_id && on_path) {
      /* free edges for those already heading to src_tile */
      toadd = old_ecnt; old_ecnt = ecnt; }
    if (next_tile is not occupied) { toadd *= 2; }
    if (!on_path) { toadd *= 2; }
    ent += toadd;
    src_tile = next_tile; next_tile = getRoute(next_tile, dst_id, x); }
  me->flag = true; return ent; }
```

Figure 4-9: Additional C-like pseudocode for the cost function for simulated annealing. This function calculates the appropriate entropy due to each dependency.

```
bool choosePropIn(Tile t, bool force) {
  /* returns true iff incoming changes */
  if (t->incoming >= 0) {
    return false; }
  if (t has root) {
    t->incoming = t->id;
    return true; }
  else if (t has virtual tile assigned && !force) {
    for each dependency d incoming to t {
      if (source of d is unassigned) {                                  10
        return false; } } }

  if (t has no incoming dependencies) {
    /* that is, every link(nt, t)->value is <= 0 for every neighbor
       tile nt */
    int min_est = INT_MAX;
    for each tile nt neighboring t {
      if (nt->incoming >= 0 && nt->EST < min_est) {
        min_est = nt->EST;
        incoming = nt; } } }                                            20
  else {
    int link_value = INT_MAX;
    for each tile nt neighboring t {
      if (nt->incoming >= 0 && link(nt, t)->value < link_value) {
        link_value = link(nt, t)->value;
        incoming = nt; } }
    if (incoming < 0 && force) {
      int min_est = INT_MAX;
      for each tile nt neighboring t {
        if (nt->incoming >= 0 && nt->EST < min_est) {                   30
          min_est = nt->EST;
          incoming = nt; } } }
  }

  if (t->incoming >= 0) {
    return true; }
  else {
    return false; } }
```

Figure 4-10: C-like psuedocode for the algorithm to determine the branch condition source tile for each non-root tile.

```
traverseBBroute(tile t, int[] visited) {
  if (visited[2*t->id] >= 0) {
    error; }
  visited[2*t->id] = t->incoming; /* src in route */
  visited[(2*t->id) + 1] = t->id; /* dst in route */

  for each tile nt neighboring t {
    if (nt->incoming == t->id) {
      traverseBBroute(nt, visited); } } }
```

Figure 4-11: C-like pseudocode for verifying appropriate construction of branch condition propagation route.

Chapter 5

Module Interface Specifications

This chapter gives the exact specifications for the spatial software pipeling module's interaction with RawCC. In addition to explaining how to produce and use the assignments of the module, this section delineates the format of all files used as interfaces between RawCC or the user and the module. The module uses a post-optimization dependency graph based on RawCC's low-level IR to determine appropriate memory object and instruction assignments. Figure 5-1 illustrates how the spatial software pipelining module interacts with RawCC and shows that the module essentially replaces the memory bank management and instruction assignment phases of RawCC. The figure may be referred to throughout the rest of this chapter; in the figure, solid lines indicate RawCC, dotted lines indicate the additions of the spatial software pipelining module, and the grayed boxes indicate the phases of RawCC replaced by the module. Each arrow represents communication of some form via the interfaces described in the rest of this section.

5.1 Interaction with RawCC

This section describes the interactions between RawCC and the spatial software pipelining module. RawCC has the ability to write memory object types and place-

ments to a file (objects.out) and to place memory objects based on the input file objects.in. Similarly, RawCC can write instruction dependendency graphs and intsruction placements to a file (sched.out) and can force instruction placement based on the input file sched.in. RawCC can do any of these four things in any combination. A perl script uses objects.in, objects.out, and sched.out along with multiple runs of RawCC to create the memory map file. An additional run creates an appropriate sched.out file for use by the spatial software pipelining module. The module produces objects.in and sched.in representing the final placement; these are used by another full run of RawCC to create the final executable.

The various input files required by the spatial software pipelining module are produced as follows (see Chapter 4 for a full description of the maps needed for the module). First, the user must provide the basic block ordering required by the module. This ordering is provided in the file block.map, described in Section 5.5. A shell script performs the remaining steps described. Second, a perl script described in Section 5.2 uses multiple runs of RawCC on the target program to create the required memory map; the script produces the file sched.map. Third, RawCC is used to produce objects.out for the target program, and another perl script modifies this file to produce the zero memory map, an objects.in that maps every memory object to Tile 0. The format of the objects.in/out files is described in Section 5.3. Fourth and finally, RawCC is used to produce the zero instruction map (via objects.in), a sched.in that maps every instruction to Tile 0. The format of the sched.in/out files is describe in Section 5.4.

Once all files required by the module have been produced (i.e. block.map, sched.map, the zero objects.in, and the zero sched.out), the module is run to produce the files objects.new and sched.new. These two files represent the final memory object to tile and instruction to tile mappings determined by the modules, and they are in the format of the objects.in/out and sched.in/out files respectively. Objects.new may be

used immediately as objects.in, and sched.new may be used as sched.in for a final tile assignment. Note that scheduling is done by RawCC after the final assignments objects.new and sched.new are produced. Again, Figure 5-1 illustrates this interaction.

5.2 The Memory Map

This section presents the perl script used to generate the instruction to memory map needed by the spatial software pipelining module. This perl script is a hack; it would be much faster, easier, and cleaner to have RawCC generate the memory map directly, and this would allow for more flexibility as well. However, this would also entail developing a new mechanism for specifying to RawCC which memory objects should be assigned to which tiles. Integrating the memory map generator into RawCC, or rather, integrating the spatial software pipelining module more fully with RawCC, is one item left for future work (see Chapter 7).

The memory map is the interface defining which instructions correspond to which memory objects. That is, every instruction in every basic block is mapped to at most one memory id, and each instruction mapped to a memory id must be mapped to the same tile as the corresponding memory object. This means that multiple instructions mapped to the same memory id must be executed on the same tile. Note that in general only loads, stores, and certain instructions involved in calculating the base address of a load or store are memory mapped (i.e. mapped to a memory id). Most instructions, such as floating point and alu operations, are not memory mapped.

The file specifying the memory map, sched.map, is formatted as follows. Each line specifies a basic block identifier, an instruction identifier, and a memory object identifier. The lines are sorted in order of basic block id and then instruction id. Thus a portion of sched.map might look like this:

```
Block 13 Node 168 MemId 1
```

```
Block 14 Node 3 MemId 78
Block 14 Node 4 MemId 78
Block 16 Node 2 MemId 17
Block 16 Node 5 MemId 15
Block 16 Node 8 MemId 16
```

The perl script used to create sched.map works as follows (skeletal perl is provided in Figure 5-2). First, it obtains the memory ids of all memory objects in the target program; that is, it obtains the memory id of each memory object that appears in objects.out. Next it places every object on tile 0 (not distributed). If the target number of tiles is $i + 1$, the script iterates through i memory objects at a time, assigns each memory object in that iteration to one each of tiles 1 through i, and leaves all other memory objects assigned to tile 0. The script then uses RawCC to produce a schedule for that iteration. (Note that it only makes sense to use the spatial software pipelining module if the target number of physical tiles is greater than one.) Then, if an instruction `instr` in basic block `bblock` in the resulting schedule is *fixed* on tile `tile` (i.e. the instruction may not be assigned to any tile besides `tile`), then (`bblock`, `instr`) is memory mapped to the memory object assigned to `tile` in that iteration. Once all of the memory mapped instructions in the target program have been discovered and noted, the perl script sorts the mappings by basic block id and instruction id and finally creates the file sched.map.

5.3 Objects Files

The objects.in/out/new files specify a mapping from memory objects to physical tiles. Objects.in specifies the tiles on which RawCC must place the memory objects, objects.out lists the tiles on which RawCC did place the memory objects, and objects.new is the objects.in file produced by the spatial software pipelining module. For each memory object in the target program, a line in the object file of interest specifies a unique memory id, an object type (e.g. whether it should be distributed),

the physical tile on which the object resides or to which the object is mapped (the two are equivalent), the number of tiles across which an object should be distributed, the grain size by which an object should be distributed (e.g. four words of an array per tile), and the source variable name to which the object corresponds. The lines are ordered by increasing memory id. An object file might look like this (this is the actual objects.out file used to distribute array B in the canonical case discussed throughout this thesis):

```
Id 0 Type 0 Tile 1 Ntiles 1 Grain 1 Vars i
Id 1 Type 0 Tile 2 Ntiles 1 Grain 1 Vars a
Id 2 Type 1 Tile 0 Ntiles 4 Grain 4 Vars b
Id 3 Type 0 Tile 3 Ntiles 1 Grain 1 Vars x
Id 4 Type 0 Tile 0 Ntiles 1 Grain 1 Vars y
Id 5 Type 0 Tile 1 Ntiles 1 Grain 1 Vars time
Id 6 Type 0 Tile 2 Ntiles 1 Grain 1 Vars suif_tmp
Id 7 Type 0 Tile 3 Ntiles 1 Grain 1 Vars
```

The rest of this section explains certain restrictions imposed on the current system by the method used to create the memory map.

The current version of the spatial software pipelining module assumes that it is best to place every distributable array on a single tile; that is, although an array may be distributed across multiple tiles, each array is considered for assignment to only a single tile. This assumption is merely a byproduct of the method for producing the sched.map file. If RawCC were modified to assign memory ids to instructions directly, it could easily assign different memory ids to the instructions in different iterations of an unrolled loop. Suppose, for example, that a loop simply accesses each element of an array with memory id arr. Then, unrolled four times, the body of the loop would contain at least these four instructions accessing arr, supposing that the address of the current index of arr is in r1:

```
Node Id 5 Tile 1 Fixed 1 lw     r2, 0(r1)
```

```
Node Id 6 Tile 1 Fixed 1 lw     r2, 4(r1)
Node Id 7 Tile 1 Fixed 1 lw     r2, 8(r1)
Node Id 8 Tile 1 Fixed 1 lw     r2, 12(r1)
```

With the current method of mapping instructions to memory ids, each of nodes 5, 6, 7, and 8 will be mapped to memory id `arr`. However, RawCC would already have the information that the array could be distributed across four tiles, one for each original iteration in the unrolled loop; as such, RawCC could create a memory id for each original iteration, `arr0`, `arr1`, `arr2`, `arr3`. Then the indices of the array in the first subset (such that index mod 4 is 0) would correspond to memory id `arr0`, those in the second to `arr1`, those in the third to `arr2`, and those in the fourth to `arr3`. RawCC would then map node 5 to memory id `arr0`, node 6 to `arr1`, node 7 to `arr2`, and node 8 to `arr3`. This would allow the array to be distributed across tiles if appropriate; there would not exist the current artificial restriction that all of nodes 5, 6, 7, and 8 execute on the same tile. However, a new interface for passing the final memory object assignments to RawCC would have to be developed, since a legal memory object to tile mapping would be, for example, `arr0` and `arr2` on Tile 0, `arr1` on Tile 1, `arr3` on Tile 3. Such an assignment is currently not communicable to RawCC as the array is still treated as a single memory object; a single object may be either on a single tile or distributed evenly across a given number of tiles. Thus this extension remains to future work. Currently the only way to enforce that an array be distributed is to remove from the memory map all entries mapping instructions to the array's memory id and to make sure in the sched.out file that all array accesses (e.g. Nodes 4-7 in the above example) are fixed on their corresponding tiles and do not share defIds with any other instructions.

Additionally, this method of determining instruction to memory id mappings and passing the final memory assignments back to RawCC does not allow for migrating objects. That is, the method enforces the restriction that a memory object must remain on a given physical tile throughout the life of the program. However, it may

make sense for a memory object to be able to migrate from tile to tile, for example if the cost of placing all accesses from every basic block onto the same tile is much more than the cost of moving the memory object from one tile to another between basic blocks. Again, RawCC may have the cost information available; if it could determine that migrating a memory object beween two basic blocks would be cost effective, it could simply give the same memory object different memory ids depending on which basic block was accessing it. Then accesses to the object in one basic block would be memory mapped to a different id than accesses to the same memory object in another basic block. Again, this would require a different mechanism for creating the memory map file sched.map and a different mechanism for handing the resulting memory assignments back to RawCC.

5.4 Schedule Files

The sched.in/out/new files specify a mapping from each instruction in each basic block to a physical tile. As with to the objects files, sched.in specifies the tiles on which RawCC must place instructions, sched.out specifies the tiles on which RawCC did place instructions, and sched.new is the sched.in file produced by the spatial software pipelining module. Each basic block is listed in the sched files in the following manner. Each basic block begins with a block header, followed by a list of all instructions in the basic block (in identifier order) and information about each instruction. After the list of instructions comes a list of intra-basic block dependencies and information about each dependency. Each dependency is assumed to go from a source instruction of lower id to a destination instruction of higher id; a minor change to the module code could address this assumption.

Each basic block header is formatted as follows. First is a line listing the block identifier. Following that is the total number of tiles, the number of tiles in the x

dimension, and the number in the y dimension on which the basic block may execute. (The module currently assumes that the total, x, and y number of tiles is the same for all basic blocks; when doing placement, it makes no checks to ensure that a memory object or defId used in one basic block is placed onto a tile that exists in a later basic block.) If the basic block ends with a conditional jump, the id of the instruction that creates the branch condition, the *root node*, appears on the next line. The result of the root node must be propagated to all physical tiles, and the path by which this is done appears on the line after. The propagation path is a list specified as follows: for each tile `tile`, if the tile is not assigned the root node and should receive the branch condition from a neighbor tile `nt`, then the list includes '`tile nt`'.

The lines after the basic block header represent instructions and are formatted as follows. First is the instruction identifier, followed by the tile on which the instruction is scheduled to execute. Next is a time, a cost representing the number of functional unit cycles needed to execute the instruction, and a latency representing the number of cycles after the cost before the result of the instruction becomes available. For example, a lw (load word) instruction, cost 1, latency 2, takes one cycle of ALU time for memory address calculation after which two cycles are needed for the memory access to complete before other instructions can use the value loaded. Next is the *fixed* variable indicating whether the instruction must be executed on the specified tile or whether the instruction may be reassigned to a different tile. Immediately following is a defId followed by the defCount, the number of instructions to which that defId has been assigned. (Recall that instructions that share a defId must be assigned to the same physical tile; the exception is stores and branches, which do not have valid defIds.) Finally, each line includes the assembly representation of the instruction.

The lines after instruction lines represent dependencies (edges) and are formatted as follows. Each line representing a dependency contains the source instruction iden-

tifier, the destination instruction identifier, a variable representing a data type, and a variable representing a dependency type. Dependency types may be true dependencies (write-read), output dependencies (write-write), anti-dependencies (read-write), or serial dependencies. True dependencies are those that actually represent data being transmitted between instructions and thus are the only dependencies that can generate sources in the simulated annealing cost function described in Section 4.4. All other dependencies are used to enforce instruction ordering; they must be included in the dependency graph to generate cycles during the merge phase of the module (Section 4.3) so that no dependencies are violated, but they do not represent actual data movement and so need not be considered in the tile assignment phase.

A snippet of a sched file is shown in Figure 5-3.

5.5 The Basic Block Map

The file block.map simply specifies the order in which basic blocks should be evaluated by the spatial software pipelining module. Recall that the merge and tile assignment phases occur on each basic block independently of any basic blocks later in the list; this makes it very important for the order of basic blocks given in block.map to be appropriately determined. A bad ordering will yield bad clustering results. Currently block.map is generated by hand by sorting the basic blocks by the product of the number of times the block is executed and the initial window size as determined by the module; any ties go to the basic block with higher id. Clearly the number of times each basic block is executed could be generated by a profiler and included in block.map, and the module could easily produce the order of the basic blocks itself based on the appropriate product. However, this is left to future work.

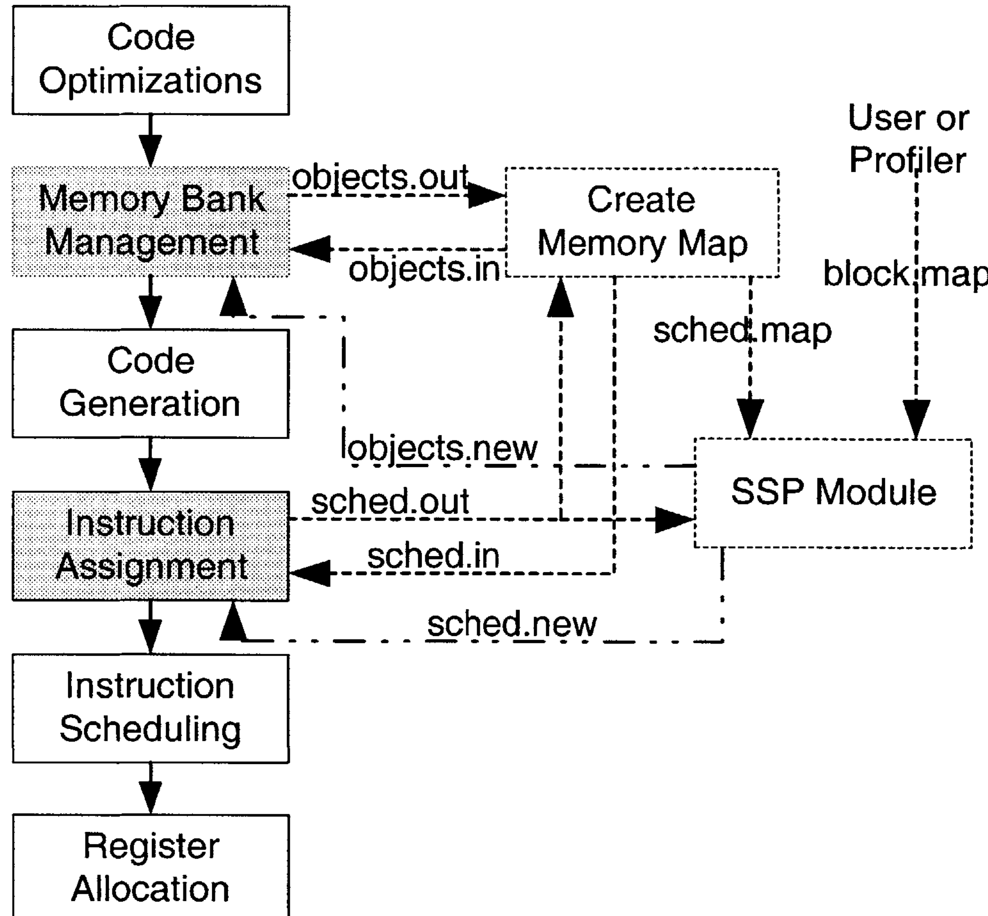


Figure 5-1: Model of the interaction between RawCC and the spatial software pipelining module. Phases of RawCC are shown with solid lines; items added by the module are shown with dotted lines. The phases of RawCC that are replaced by the spatial software pipelining module are grayed out.

```
#!/usr/bin/perl -w
$UNROLL = shift() || "";
# call RawCC on the target program to create sched.out
system "make clean"; system "make MEMSOUT=1 $UNROLL";
# get the original lines.
open(MEMSOUT,"<objects.out"); @lines = <MEMSOUT>; close(MEMSOUT);
$max = 0;
foreach $var (@lines) {
# initialize every memory object to be on tile 0, not distributed
    $var =~ s/Type \d+/Type 0/; $var =~ s/Ntiles \d+/Ntiles 1/;
    $var =~ s/Grain \d+/Grain 1/; $var =~ s/Tile ([^\d]?\d+)/Tile 0/;
    if ($1 > $max) { $max = $1; } }
$length = @lines;
%map_arr = ();
for ($id = 0; $id < $length; $id += $max) {
    if ($length < $id + $max) { $max = $length - $id; }
    for ($i = 0; $i < $max; $i++) {
        # assign the next mem objs to the tiles from 1 to max_tile
        $num = $i + 1; $lines[$id + $i] =~ s/Tile 0/Tile $num/; }
    open(MEMSIN,">objects.in"); print MEMSIN @lines; close(MEMSIN);
    # call RawCC to create an instruction assignment using objects.in as base
    system "make clean"; system "make MEMSIN=1 SCHEDOUT=1 $UNROLL";
    open(SCHEDOUT,"<sched.out");
    $block = 0;
    while ($sched = <SCHEDOUT>) {
        if ($sched =~ /Block (\d+)/) { $block = $1; }
        elsif (($sched =~ /Tile (\d+)/) && ($1 > 0)) {
            # (HACK) ignore fixed branches; otherwise, map it to the mem id
            if (($sched =~ /Fixed 1/) && !($sched =~ /bnez/)) {
                $sched =~ /Node Id (\d+) Tile (\d+)/; $node = $1;
                $mem = $id + $2 - 1;
                $map_arr{$block}->{$node} = $mem; } } }
    close(SCHEDOUT);
    for ($i = 0; $i < $max; $i++) {
        $lines[$id + $i] =~ s/Tile (\d+)/Tile 0/; } }
open(SCHEDMAP,">sched.map");
# sort this before printing
foreach $block (sort {$a <=> $b} keys %map_arr) {
    foreach $node (sort {$a <=> $b} keys %{$map_arr{$block}}) {
        print SCHEDMAP "Block $block Node $node MemId "
            ."$map_arr{$block}->{$node}\n"; } }
close(SCHEDMAP);
```

Figure 5-2: A skeletal perl script to create the memory mapping file sched.map using RawCC.

```
Block 26
Ntiles 16 X 4 Y 4
BBroot 28
BBroute 1 0 2 1 6 2 7 3 5 4 6 5 10 6 6 7 9 8 10 9 10 11 8 12 9 13 10 14 14 15
Node Id 0 Tile 10 Time −1 Cost 1 Latency 0 Fixed 0 DefId 2.147 DefCount 4
        move   $vr502.s32,$vr502.s32  #
...
Node Id 4 Tile 10 Time −1 Cost 1 Latency 0 Fixed 0 DefId 2.147 DefCount 4
        move   $vr502.s32,$vr502.s32  #
...                                                                          10
Node Id 20 Tile 12 Time −1 Cost 1 Latency 2 Fixed 0 DefId 2.161 DefCount 1
        lw     $vr516.f32,0($vr515.p32)       #
Node Id 21 Tile 12 Time −1 Cost 2 Latency 0 Fixed 0 DefId 2.162 DefCount 1
        li     $vr517.f32,0x3727c5ac #
Node Id 22 Tile 12 Time −1 Cost 1 Latency 3 Fixed 0 DefId 2.163 DefCount 1
        mul.s  $vr518.f32,$vr516.f32,$vr517.f32       #
Node Id 23 Tile 12 Time −1 Cost 1 Latency 0 Fixed 0 DefId −1.−1 DefCount 0
        sw     $vr518.f32,0($vr515.p32)       #
Node Id 24 Tile 10 Time −1 Cost 1 Latency 0 Fixed 0 DefId 2.164 DefCount 1
        addiu  $vr519.s32,$vr502.s32,1 #                                    20
...
Node Id 28 Tile 10 Time −1 Cost 1 Latency 0 Fixed 0 DefId 2.168 DefCount 1
        sltiu  $vr523.s32,$vr522.s32,1 #
Node Id 29 Tile 10 Time −1 Cost 1 Latency 0 Fixed 0 DefId 2.147 DefCount 4
        move   $vr502.s32,$vr520.s32  #
...
Edge Src 4 Dst 24 DataType 1 DepType 0
Edge Src 4 Dst 29 DataType 1 DepType 2
Edge Src 4 Dst 29 DataType 1 DepType 1
...                                                                          30
Edge Src 20 Dst 22 DataType 1 DepType 0
Edge Src 20 Dst 23 DataType 2 DepType 2
Edge Src 21 Dst 22 DataType 1 DepType 0
Edge Src 22 Dst 23 DataType 1 DepType 0
...
Edge Src 24 Dst 29 DataType 1 DepType 2
...
Edge Src 0 Dst 4 DataType 1 DepType 1
Edge Src 0 Dst 4 DataType 1 DepType 0
Edge Src 0 Dst 4 DataType 1 DepType 2                                        40
...
```

Figure 5-3: A snippet of a sched file, used for communicating the instruction-level dependency graph and instruction assignments.

Chapter 6

Results and Analysis

This chapter briefly lists the assumptions made (Section 6.1) in the spatial software pipelining module. In Section 6.2 this chapter presents the performance results the spatial software pipelining module as compared to RawCC. Additionally, in Section 6.4 this chapter goes over the conditions under which spatial software pipelining is applicable. Finally, it covers which other optimizations may be useful when used in conjunction with spatial softare pipelining (Section 6.5).

6.1 Implementation Assumptions

This section briefly lists the assumptions made by the implemented spatial software pipelining module. These assumptions are either related to the module-RawCC interface, removable by future work, related to the input, or easily fixed.

The assumptions related to the interface between the module and RawCC are as follows. First, in Chapter 4, all branches ending a conditional basic block were removed from the virtual tile and virtual dependency graph; to do this, the module assumed that all such branches are fixed and of the form `bnez`, that such instructions have no outgoing dependencies and only one incoming dependency from the root instruction, that there is exactly one such instruction per tile in corresponding basic

blocks, and that no other fixed `bnez` instructions exist. Clearly this is an erroneous assumption (it precludes some of the benefits of asynchronous global branching), and should be fixed. Second, to allow instructions with defIds and memory ids to be mapped to alternative tiles, the module assumes that the fixed variable has no meaning on instructions that are memory mapped or share defIds with other instructions has no meaning. However, other fixed instructions are forced to remain on the physical tiles to which they were assigned. Additionally, the module assumes that all negative defIds are unique.

These next assumptions may be removed by future work, Chapter 7. As discussed in Chapter 4, the module assumes that a decent assignment can be found even when the layout of each basic block is done without taking into account the effects on later basic blocks. Additionally, the algorithms assume that separate merge and then assign phases are better. Chapter 5 touched on the assumption that an array must be assigned a single tile and cannot be considered for distribution; recall, however, that this was solely due to the mechanism for mapping instructions to memory ids, and that this assumption may be bypassed. That chapter also discussed the assumption that memory cannot migrate between basic blocks, and again discussed the means to bypass that assumption.

There are a few assumptions about the input dependency graphs that are due to deficiencies in the current implementation. The first assumption is that the number of totally unconnected subgraphs in a basic block's dependency graph is less than the number of tiles. This assumption is due to the fact that there is currently no mechanism for determining whether a merge phase failed because there were too many unconnected subgraphs or whether it failed because the window size was too small. In the first case, the module would need an additional bit of code to consider merges between virtual tiles in different subgraphs. However, without the guarantee that the tiles *were* in different subgraphs, such code could wreak havoc with the

current merge system by allowing non-cycle creating merges between tiles that were very distantly related in the dependency graph. Such merges would violate spatial software pipelining. The second assumption is simply that enforced communication cycles are uncommon; this assumption is primarily due to the fact that the current module does not handle cycles well (see Chapter 7).

These last few assumptions can be easily fixed if it turns out that they are in error. First, the module assumes that RawCC's Space-Time scheduler [9] generates switch code that routes data in dimension order (i.e. in the y dimension and then the x dimension). This assumption is made when calculating the energy in a layout for the simulated annealing phase. Second, the module assumes that two instructions with different memory ids will have cache conflicts if placed on the same tile, and the estimated cost of such a merge is increased correspondingly. Finally, the module assumes that a true dependency is the only data-bearing dependency.

6.2 Performance

This section presents the performance numbers comparing the original execution times of programs when RawCC alone performs the assignment to the new execution time when the implemented spatial software pipelining module performs the assignment. Speedup factors as well as execution times are shown in Table 6.1. Note that the speedup for the canonical case with no optimizations looks fairly low. This is because the code produced by RawCC does not distribute any of the arrays as it used to; the runtime when distributing all of the arrays as before is 569,643, compared to which the spatial software pipelining speedup is 3.178x.

The improvement for unstructured, not loop unrolled, is 2.142x. We believe, however, that this number could improve significantly with the module modifications discussed in future work.

Table 6.1: Performance comparison of RawCC to RawCC with spatial software pipelining.

Source Program	Number Tiles	Optimizations	Original Execution	Execution With SSP	Speedup Factor
unstructured	16	none	7,465,525	3,484,997	2.142
canonical	4	none	259,141	179,230	1.446
canonical	16	unroll 4x	232,310	147,329	1.577

6.3 Unstructured Results

The spatial software pipelined version of the sparse matrix benchmark unstructured gains a 2.142x speedup over the original layout produced by RawCC when no loop unrolling is performed. Table 6.2 shows how this gain is apportioned to each interesting loop in unstructured, and additionally shows how spatial software pipelining performs on each loop when that loop is placed first. Note that some loops (3, 10, 77, 73, and 26) perform better when placed after other basic blocks than when place first in spatial software pipelining; these loops benefit from being forced into certain assignments. 77, 73 and 26 are all traditional loops that regularly access certain arrays; as mentioned, spatial software pipelining may not be able to help such traditional loops. However, for comparison, other traditional loops (both large and small) are 17, 1, 22, 26, 70, 73, 80, 31, and 34; spatial software pipelining was able to help in most of these loops. In each of the cases of 3 and 10, the undeveloped state of the merge function is apparent; in both cases, the merge function made some bad local merges that adversely affected the entire loop. In both cases, the dependency graph is very unbalanced with many values used multiple times each throughout a single calculation; this increases the number of outgoing dependencies from the virtual tiles calculating (or loading) the values, and in general makes it harder for the merge function to work effectively. As we mentioned, this is a result of the undeveloped state of the merge function; future work should investigate how to guide the merge function

Table 6.2: Breakdown of improvement to unstructured (not unrolled) for each loop of interest. Loops are presented in the order in which their basic blocks are evaluated by the spatial software pipelining module. Additionally, the actual execution time for each loop is compared with the execution time for that loop obtained by evaluating that loop first.

Source Loop	Times Executed	Original Execution	Execution With SSP	Speedup Factor	Execution If First	Optimal Speedup
75	10	194,409	59,530	3.266	59,530	3.266
8	5	373,154	153,675	2.428	122,347	3.050
5	5	225,789	163,057	1.385	119,148	1.895
3	5	203,858	65,827	3.097	90,915	2.242
10	5	97,309	68,551	1.420	70,183	1.387
77	10	24,011	13,702	1.752	20,654	1.163
13	5	49,007	27,189	1.802	22,207	2.207
73	10	3,821	3,862	0.989	3,871	0.987
17	1	31,026	15,145	2.049	14,994	2.069
34	5	20,220	15,304	1.321	6,578	3.074
26	5	6,517	5,454	1.195	5,528	1.179
1	5	14,306	13,070	1.095	10,894	1.313
38	5	4,084	1,238	3.299	1,227	3.328
80	5	13,628	6,073	2.244	4,371	3.118
70	5	12,543	7,623	1.645	4,348	2.885
22	5	8,227	7,123	1.155	5,632	1.461
31	5	11,832	5,708	2.073	3,630	3.260
average	-	-	-	1.895	-	2.228

appropriately in such cases to produce better assignments and should investigate the merge algorithm in more depth in any case.

6.4 Applicability

This section briefly describes the applicability of spatial software pipelining. Spatial software pipelining is useful when compiling sparse matrix applications or applications with pipelinable loops with irregular accesses to memory. Clearly it is only useful when targetting a multiprocessor, and it is most useful when compiling a program in which most of the execution time is spent in loops. While spatial software

pipelining is also applicable when compiling traditional pipelinable loops with regular accesses, there is generally less pipeline parallelism than parallelism from loop unrolling and modulo unrolling; in such cases, spatial software pipelining would not do as well. The following features may affect the amount of parallelism that spatial software pipelining can exploit in a given application. When the data flow graphs cannot be easily pipelined because merging memory accesses creates large cycles, or when memory objects within a program can not be partitioned to a fine enough degree, spatial software pipelining may not be able to merge virtual tiles intelligently. When the number of available processing units is high and necessary synchronization for dynamic accesses is low, the cost of dynamic accesses may be amortized by the distribution of compute cycles; in this case, spatial software pipelining may not be able to find a similar amount of parallelism. Note that if spatial software pipelining produces a layout without iterating in the merge phase, then the presence of additional tiles will only allow a better layout if there was conflict due to the tile assignment phase. With respect to these constraints, there are two items of future work: determining how important each of these factors is in general, and developing spatial software pipelining mechanisms to relax these constraints.

6.5 Compatible Optimizations and Effects

This section mentions other compiler optimizations that are compatible with spatial software pipelining. Most traditional compiler optimizations are compatible *if they are performed before the spatial software pipeliner.* For example, code hoisting, peephole optimizations, dead code elimination, and partial redundancy elimination are all useful in conjunction with spatial software pipelining if they are performed first. Performing these optimizations before spatial software pipelining allows the module to work with accurate cost and instruction information. Loop unrolling and

modulo unrolling are also quite compatible with spatial software pipelining if done before pipelining. Additionally, it would be very interesting to give the pipeliner the power to replicate an instruction across some tiles if appropriate, to determine how many times to unroll a loop, and, together with the rest of the compiler, to determine whether or not memory should migrate from one tile to another between basic blocks (or even if the migration could be done while other useful work was taking place). The first would allow the module to remove communication dependencies by replicating certain instructions; the second would allow the module to fill idle cycles (created, for example, by communication cycles) with useful work; the benefit of the third has been discussed before.

Chapter 7

Future Work

This presents future work stemming from this thesis. Although future work exists both on my specific implementation and on developments on the idea of spatial software pipelining in general, this chapter primarily relates future work on my specific implementation.

7.1 Handling Communication Cycles

As mentioned in Chapter 4, the current implementation of the spatial software pipelining module performs badly with communication cycles that can not be removed. Primarily this deficiency is due to the fact that we have not thought extensively about how to define the EST and the path for virtual tiles that form part of a communication cycle. Not having the EST and path variables appropriately defined for such virtual tiles affects the EST and path variables of all ancestors and descendants of such tiles and adversely affects the merge phase of my spatial software pipelining module. This in turn could lead to a bad memory and instruction assignment for the affected basic block. Unfortunately, a bad assignment for one basic block could adversely affect any subsequent basic blocks if subsequent basic blocks contain instructions using either the same memory objects or the same defIds as the badly

assigned basic block. Clearly, then, careful thought as to the appropriate assignment of EST and path to virtual tiles involved in communication cycles would yield more appropriate merges, assignments, and schedules than the current module. Luckily, it appears that unavoidable cycles are rare in the most computation intensive loops studied thus far.

7.2 Loop Unrolling and Software Pipelining

Once communication cycles are handled appropriately with respect to EST and path, it should be a simple extension to add software pipelining to tiles involved in data communication cycles. After the final tile assignments have been made, analysis may be able to determine how to do traditional software pipelining on the tiles to effectively utilize the parallel processing power available.

Any tile in a communication cycle may determine whether it needs to be unrolled and software pipelined to make use of otherwise idle cycles. If a tile in a communication cycles produces data in one instruction that is connected through true dependencies and instructions on other tiles to an instruction on the original tile, then the communication cycles *terminates* on that tile and the tile should be unrolled and software pipelined. Traditional software pipelining in addition to spatial software pipelining is aided by the fact that control flow information can be pipelined along with data flow information; since the branch conditions can be pipelined right along with the data, intermediate tiles may not need to be unrolled at all. This would be the case, when, for example, all destination instructions of incoming dependencies must be executed before any source instructions of outgoing dependencies.

Additionally, a tile in a communication cycle that should be unrolled and software pipelined may determine the number of times it should be unrolled. Given a window size, the number of intervening cycles may be calculated assuming that each

tile through which the data must propagate must be executed in full; the number of intervening cycles is then the number of intervening tiles times the window size plus a constant number of cycles for each interconnect. This number may then be used to determine how many times to unroll the tile, and the prefix and cleanup code are mutually exclusive exhaustive subsets of the instructions on the tile, and may be determined by the software pipelining module. Of course, it becomes much more difficult when multiple communication loops terminate at the same point, or a separate communication loop terminates at one of the virtual tiles in a communication loop, or a dependency chain extends more than once around the communication loop.

7.3 Integrating with RawCC

As mentioned in Chapter 5, there is much room for improvement if the interface between RawCC and the spatial software pipelining module is redefined. Having RawCC generate the memory map directly rather than using the current perl script hack would be much faster, easier, and cleaner and would allow for more flexibility. RawCC, or an extension of RawCC, could then assign different memory ids to the memory instructions corresponding to different iterations of an unrolled loop. This would provide a nice mechanism for allowing distributed arrays, but would require that RawCC treat different portions of an array as different objects (which it does not currently do). Additionally, this may be more effective if different conflict costs for id pairs were used instead of a single static conflict cost (see Section 7.5). Having RawCC generate the memory map would also allow RawCC to determine when migrating a memory object beween two basic blocks would be cost effective and to encapsulate this in producing the memory map. (However, this does give rise to the thought that other heuristics, such as "Place this memory object here if it is not too much trouble," would be useful in the simulated annealing portion of the spatial software pipelining

module.)

Additionally, it would be interesting to work more closely with the compiler to help determine whether or not to use spatial software pipelining on given loops. That is, as each basic block is processed by the spatial software pipelining module, the resulting placement could be compared with that generated by RawCC alone (given the defId and memory id restrictions thus far) to determine whether the module's assignment is highly beneficial or highly detrimental to the estimated execution time of that basic block.

7.4 Using Profiling Information

Currently the module user must generate block.map by hand for every program and each unroll factor. A profiler used in conjunction with RawCC could generate iteration counts and store these to block.map instead. A small addition to the current module could easily take the iteration counts and generate the basic block orderings that we are now generating by hand; this would decouple the ultimate effects of the module from the clever machinations of the module user. Additionally, the profiler would help determine when memory can afford to migrate, and the profiler results also could be used to generate weights for each of the basic blocks in comparison to the others. These weights would be useful in allowing cross basic block influence, described in Section 7.5.

Additionally, the profiler could be used to generate conflict costs for each pair of memory ids, where each cost represents the amount of conflict the two memory objects are likely to have if they are placed on the same tile. However, a means of communicating this information to the spatial sofware pipelining module would have to be developed.

7.5 Using Influence Across Basic Blocks

There are two interesting ways in which the current module could be improved to take into account information from basic blocks not yet processed. First, basic block weights (mentioned above) could be used to help create conflict costs that differ depending on which ids are being considered in a merge of two virtual tiles. Before the merge function of each basic block begins, each id pair that occurs in later basic blocks could be analyzed in each later basic block. For each pair a data structure could store a number corresponding to the effect of assigning the pair to a single tile. Such a number might be determined as follows. For each remaining basic block, take the virtual tiles representing the two ids and simulate a merge between them. Additionally, simulate the merge of any communication cycles created by their merge; this will result in a cost, which may be divided by the initial window size of the basic block. This number may be multiplied by the weight assigned to that basic block. The final number assigned to a pair could be the maximum produced across all the remaining unassigned basic blocks. Since any pairs already forced together by earlier basic blocks will be assigned tiles and forced together during reconciliation, these pairs need not be considered. Possible additional forced merges may be made incidentally by the assignment if an id pair is assigned to the same tile in unrelated basic blocks; the id pair costs could thus similarly influence the tile assignment phase.

This brings us to the second way in which the current module could be improved to take into account information from unassigned basic blocks. In a neat alternative, all of the merges could be done as now, but all of the tile assignment phases could wait until after all merge phases had completed. Then, the simulated annealing cost function could try to take into account the effects of any one tile assignment on later basic blocks. The reconciliation phase would have to happen before each merge phase, of course, but the implementation would not be much different. The difficulty would be in getting the tile assignment phases to cooperate with each other, and it is not

clear how best to take into account later blocks, except that in all likelihood the basic block weights would be quite convenient.

7.6 Improving the Tile Assignment and Virtual Tile Merge Algorithms

The algorithms used for the tile assignment and virtual tile merge algorithms were implemented without studying alternatives in depth. It would be interesting to see how other algorithms affected the efficacy of the implemented module. Both the tile assignment cost function and the merge algorithms most likely have a lot of room for improvement, and should be revisited. Additionally, it would be interesting to implement a check after the simulated annealing tile assignment phase to see if the entropy for the optimal assignment was horrible. If so, increasing the window size, redoing the merges, and retrying the simulated annealing step could help.

Also, currently the simulated annealer is not well written in terms of determining how long the annealer should run. Currently this is hard-coded, which is clearly bad when the algorithm is being run for an increased number of target tiles. This should be fixed.

7.7 Determining and Increasing Applicability

In addition to providing appropriate data and instruction partitioning within loops, it seems reasonable that spatial software pipelining could provide beneficial placement for other types of code. Specifically, if the data flow in adjacent basic blocks can be forced to follow the same channels, spatial software pipelining might provide the same effect as in loops without actually having the basic blocks be loops.

As mentioned in Chapter 6, much work remains to be done in determining exactly

when spatial software pipelining is applicable to increase the applicability of the spatial software pipelining module.

Chapter 8

Related Work

This chapter will briefly touch on aspects of other work related to spatial software pipelining. Because spatial software pipelining is ultimately a clustering algorithm, most of the work bearing on this thesis investigates alternative clustering algorithms. Keep in mind, however, that the problem of optimally assigning instructions and data to clusters is NP-complete; as such, each reasonably tractable and effective clustering algorithm targets some subset of application types to try to schedule well, and simply tries to avoid creating bad cluster assignments for the other applications. If a given application type can be classified, then the appropriate clustering algorithm can be used to generate clusters for that application. Spatial software pipelining targets sparse matrix codes; specifically, it targets data heavy applications whose memory access patterns to each memory object are nondeterministic.

Sections 8.1 and 8.2 will briefly discuss two clustering algorithms, Bottom-Up-Greedy (BUG), and Partial Component Clustering (PCC). While these algorithms are relevent in their approach to instruction placement, neither takes into account memory placement when doing instruction placement. Dealing with the restrictions imposed by memory placement is one of the main problems that spatial software pipelining addresses, and as such spatial software pipelining must perform clustering

on a dependency graph that is *not guaranteeably acyclic.* Since BUG and PCC both assume that the graph on which they are performing clustering is a DAG, neither is directly applicable to the problem at hand. Section 8.3 describes convergent scheduling, a recent development in instruction scheduling with which spatial software pipelining may be able to be integrated.

8.1 Bottom-Up-Greedy

Bottom-Up-Greedy, or BUG, uses two phases to do instruction placement, first walking a DAG from the leaves to the root nodes to create sets of possible functional unit assignments for each node, and second walking back down the DAG from the roots to the leaves to create the final instruction assignment. The final assignment of each node is based on the estimated time the operands become available and available resources on each possible functional unit. Like BUG, spatial software pipelining ignores register pressure and assigns nodes based on local information in the dependency graph. However, spatial software pipelining additionally takes moves between functional units into account and explicitly schedules such moves. BUG was originally described in [6].

8.2 Partial Component Clustering

The first phase of Partial Component Clustering (PCC) closely resembles the merge phase of spatial software pipelining. Like the merge phase in spatial software pipelining, the partial component growth phase of PCC works from the leaves up to the root nodes, following the longest path backwards and merging nodes into a partial component (like a virtual tile) until either the number of nodes in the partial component is equal to a threshold or no more merges are possible. This is quite similar to the creation of virtual tiles, in that the threshold resembles the window size. However, here

the similarity ends. While spatial software pipelining iteratively repeats the merge phase, PCC takes the partial components and assigns each one to some processing unit, then tries to improve the assignment through iterative descent. Spatial software pipelining does not do any assignment of instructions to functional units until the number of virtual tiles has been reduced to at most the number of functional units. PCC, however, does take into account register pressure and uses a reasonably accurate model to predict the actual generated schedule. It would be interesting to see if the model used here could be adapted and integrated with the cost function used to generate the cost of a merge. This could be quite beneficial to spatial software pipelining. More on PCC can be found in [7].

8.3 Convergent Scheduling

Convergent scheduling is another clustering mechanism recently developed to help satisfy different constraints in a relatively independent manner when doing instruction scheduling. Each type of constraint (e.g. register pressure, limiting communication between processing units, scheduling the critical path without delays) is represented by a pass that uses a defined interface to communicate its preferences for specific assignments of instructions to functional units. Each instruction is assigned a weight for each processing unit, and the passes use the defined interface to alter these weights to reflect preferences for final instruction assignment. Thus each set of heuristics for the different constraints can independently affect the final instruction placement, allowing new heuristics to be developed and added easily to the instruction space-time scheduler. More information can be found in [10]. It would be very interesting to develop the premises behind spatial software pipelining into another pass to be used in convergent scheduling. The method for limiting unnecessary synchronization between functional units due to communication would have to be quite different, but

it may be possible to get the better results both for sparse matrix codes and for other applications that might benefit slightly from the application of spatial software pipelining.

Chapter 9

Conclusion

In this thesis, we have presented spatial software pipelining, a mechanism for determining memory and instruction placement for pipelineable loops in sparse matrix codes. Because such loops frequently do not have deterministic access patterns, an alternative to simply interleaving the memory across available processing units (resulting in expensive dynamic accesses) needed to be found. Spatial software pipelining attempts to pipeline iterations of the loops across both processing units and interconnect, avoiding dependency cycles between processing units and placing memory to make use of data affinity. The primary concern in spatial software pipelining is to reduce unnecessary synchronization between processing units, allowing dynamic events such as cache misses to affect surrounding processing units only through overflow of the interconnect buffer space. That is, the goal of spatial software pipelining is to attempt to force the communication dependency graph between tiles to resemble a tree in order to decouple the processing units as much as possible.

We have presented the motivation behind spatial software pipelining, both in theory and in practical application Chapter 3). We have presented the algorithms used to implement spatial softare pipelining in a module on top of RawCC, and have presented the speedup gained by spatial software pipelining as compared to the

layout generated by RawCC with the same optimizations. We have shown that spatial software pipelining can reduce the amount of time spent in execution of certain loops by more than a factor of three, and have shown that decoupling processing units while pipelining data across the interconnect as well as the processing units can drastically reduce the effects of dynamic events (such as cache misses) on processing units other than the receiving unit.

We have listed the assumptions that the implemented module makes, and we have touched on some interesting aspects of the module and spatial software pipelining that could be explored in future work. Many items left to future work involve increasing the applicability and efficacy of the module, though also interesting would be studies on the effectiveness of different merge and assignment algorithms in obtaining loops that are appropriately pipelined. Finally, we have presented other work related to this thesis.

Bibliography

[1] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 327nd International Symposium on Computer Architecture (ISCA 2000)*, June 2000.

[2] Rajeev Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures.* PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, January 2000.

[3] Rajeev Barua, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Memory Bank Disambiguation using Modulo Unrolling for Raw Machines. In *Proceedings of the ACM/IEEE Fifth Int'l Conference on High Performance Computing (HIPC)*, December 1998.

[4] Rajeev Barua, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Maps: A compiler-managed memory system for raw machines. Atlanta, GA, June 1999. In *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA-26).*

[5] Rajeev Barua, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Compiler Support for Scalable and Efficient Memory Systems. In *Proceedings of IEEE Transactions on Computers*, November 2001.

[6] John. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD dissertation, 1985.

[7] Paolo Faraboschi, Giuseppe Desoli, and Joseph A. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett Packard, HP Laboratories Cambridge, December 1998. This is a full TECHREPORT entry.

[8] Walter Lee and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. March 2004.

[9] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. San Jose, CA, October 1998. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-8).*

[10] Walter Lee, Diego Puppin, Shane Swenson, and Saman P. Amarasinghe. Convergent Scheduling. Istanbul, Turkey, November 2002. In MICRO-35.

[11] Jesus Sanchez and Antonio Gonzalez. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. Monterey, CA, December 2000. In *Proceedings of the 33rd Int. Symp. on Microarchitecture (MICRO-33).*

[12] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* PhD thesis, 1989.

[13] Michael B. Taylor. *The Raw Processor Specification.* MIT Laboratory for Computer Science, October 2003.

[14] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen

Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), December 1996.