RICE UNIVERSITY

Cluster Assignment and Instruction Scheduling for Partitioned Register-Set Machines

by

Jingsong He

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. Keith D. Cooper, Chairman
Professor
Computer Science

Dr. John Mellor-Crummey
Senior Faculty Fellow
Computer Science

Dr. Devika Subramanian
Associate Professor
Computer Science

Dr. Linda M. Torczon
Research Scientist
Computer Science

Houston, Texas

April, 2000

Cluster Assignment and Instruction Scheduling for Partitioned Register-Set Machines

Jingsong He

Abstract

For half a century, computer architects have been striving to improve uniprocessor computer performance. Many of their successful designs such as VLIW and superscalar machines use multiple functional units trying to exploit instruction level parallelism in computer programs. As the number of functional units rises, another hardware constraint enters the picture — the number of register-file ports needed grows directly with the number of functional units. At some point, the multiplexing logic on register ports can come to dominate the processor's cycle time. A reasonable solution is to partition the register file into independent sets and associate each functional unit with a specific register set. Such partitioned register sets have appeared in a number of commercial machines, such as Texas Instruments TMS320C6xxx DSP chips.

Partitioned register-set architectures present a new set of challenges to compiler designers — the compiler must assign each operation to a specific clusters and coordinate data movement between clusters. In this thesis, we investigate five instruction scheduling methods with different scopes to find a suitable one for partitioned register-set architectures. Next, we examine previous algorithms for the combined cluster assignment and scheduling problem and propose two new algorithms that improve upon the prior art. Then we study the difficulties introduced by limited number of registers and provide an approach to handle them. Finally we take several other measurements of partitioned register-set architectures that may shed light on some of the architectural decisions.

Acknowledgments

First, I wish to thank my advisor Keith Cooper most sincerely. This work is a direct result of his mentoring and numerous contributions.

I am also very grateful to members of my committee. John Mellor-Crummey carefully reviewed the thesis and provided many valuable suggestions. Devika Subramanian and Linda Torczon have been constant sources of help since my first day at Rice.

It was a great pleasure working with other members of the Massively Scalar Compiler group, Philip Schielke, Tim Harvey, Edmar Wienskoski, Alexander Grosul and Li Xu. I especially thank Philip Schielke and Tim Harvey, this work would not be possible without their endless help.

Fellow classmates Ming Zhang, Chen Ding, James Yang and Kai Zhang gave me a lot of help through my study at Rice.

Finally, I thank my wife and my parents for loving and supporting me.

Contents

Tables

Illustrations

Chapter 1

Introduction

For half a century, computer architects have been striving to improve uniprocessor computer performance. Adding functional units is a natural way to increase the potential peak performance of a processor. Many successful architecture designs such as VLIW and superscalar machines feature multiple functional units. For example, the Texas Instruments TMS320C6xxx has eight functional units [15].

To produce good code for these machines, the compiler must expose enough instruction level parallelism (ILP) to let the scheduler to keep the various functional units busy. The scheduler must order the operations in a way that lets them execute in parallel. Finally, the compiler must keep as many values in registers as possible, since the memory interface is rarely wide enough or versatile enough to meet the need for operands.

As the number of functional units rises, another constraint enters the picture — the number of ports available on the register file. Because a typical functional unit needs two register-based operands and produces one register-based result in each cycle, the number of register-file ports needed grows directly with the number of functional units. At some point, the ports require too much logic — the multiplexing logic on register ports can come to dominate the processor's cycle time.

This revelation is not new. In fact, the designers of the Multiflow Trace machines noted that "*. . . any reasonably large number of functional units requires an impossibly large number of ports to the register file . . . The only reasonable implementation compromise is to partition the register files . . .*" [7] To accomplish this, the architects create independent register sets and associate each functional unit with a specific

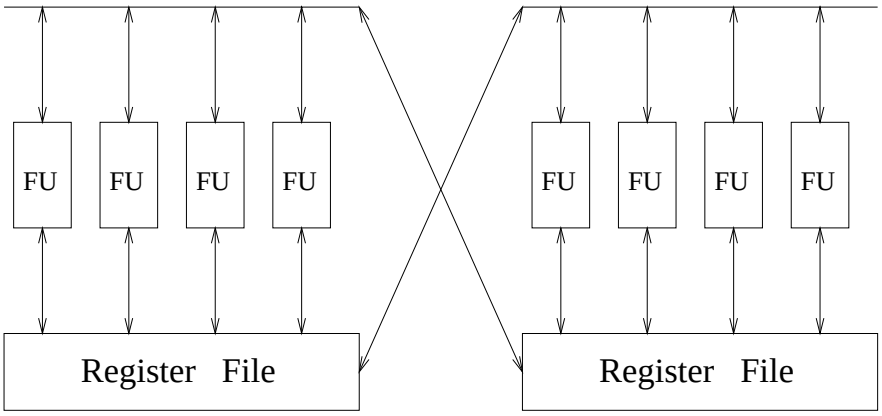


Figure 1.1 : A partitioned register-set machine

register file. Figure 1.1 depicts such a design, with two register files that each service four functional units. We call each set of functional units and its associated register file a **cluster**. To allow direct transfers between clusters (without going through memory), an inter-cluster transfer mechanism is usually added. The figure depicts this as an inter-cluster data path. Typically, the inter-cluster path can provide a limited number of operands in each direction on each cycle.

Partitioned register sets appeared in a number of digital signal processors (DSPs) such as Texas Instruments TMS320C6xxx series. With increasing use of DSPs in cars, microwaves, cellular phones and other consumer electronics, we expect to see more partitioned register-set architectures.

These partitioned register-set machines present a new set of challenges to compiler designers. The compiler must place each operation in a specific cluster and then work to ensure that its operands are either available in the local register file or available, in a timely fashion, over the inter-cluster data path. These problems are complicated by the latency and limited capacity of the inter-cluster data path.

In this thesis, we study the combined cluster assignment and instruction scheduling problem for partitioned register-set machines. Chapter 2 provides the theoret-

ical and experimental background. Chapter 3 compares five instruction scheduling methods to select a suitable scheduling scope for partitioned register-set machines. Chapter 4 examines previous algorithms for the cluster assignment and scheduling problem, and proposes two new algorithms which produce better scheduled code. Chapter 5 refines our algorithms to handle register sets of practical size. Chapter 6 takes several other measurements of partition register-set architectures that may shed light on some of the architectural decisions. Chapter 7 reviews our contributions.

Chapter 2

Background

In this chapter we describe the theoretical and experimental background for our research work. First, we introduce the instruction scheduling problem and the list scheduling algorithm. Second, we present the research compiler infrastructure that our instruction scheduler builds on. Then, we describe the partitioned register-set machine models and benchmark programs we tested. At last, we examine previous work on partitioned register-set machines.

2.1 Instruction Scheduling Problem

Instruction scheduling is the process by which a compiler reorders the instructions of a program in an attempt to decrease its running time, reduce its code size or improve other aspects of the program. Figure 2.1 shows an example. Assume that the processor has only one functional unit; memory access operations take three cycles; and all other operations take one cycle. The original code on the left takes 8 cycles while the carefully scheduled code on the right only takes 5 cycles. The NOP operations denote the cycles in which the machine has to wait for results of previous operations. The scheduled code effectively hides the latency of memory access operations. Notice that the scheduled code must use more registers.

VLIW and superscalar machines use multiple functional units to increase their peak performance. To produce good code for them, the compiler must expose enough instruction level parallelism (ILP) to keep the various functional units busy. Consider the code fragment in Figure 2.2(a). Assume that the machine has two identical functional units; multiply takes two cycles; addition and subtraction take one cycle.

```
LOAD @a => r0
NOP
NOP
ADD r0 r0 => r2
LOAD @b => r1
NOP
NOP
ADD r1 r2 => r3
```

(a) Before scheduling

```
LOAD @a => r0
LOAD @b => r1
NOP
ADD r0 r0 => r2
ADD r1 r2 => r3
```

(b) After scheduling

Figure 2.1 : Instruction scheduling example

Instruction level parallelism is exploited by the scheduled code in Figure 2.2(b) — when one functional unit is busy with a multiply, an addition and a subtraction are issued to the other functional unit. The amount of ILP available is subject to data dependence. For example, in Figure 2.2(b) we cannot move the last add into an earlier cycle because it has to wait for the results of previous operations.

Since general instruction scheduling problem is NP-complete [22], a number of heuristic methods have been developed that give approximate solutions. Among them, list scheduling [16, 14] is the dominant method. More advanced techniques, such as trace scheduling and software pipelining, typically use list scheduling to perform the actual assignment of operations into specific cycles.

List scheduling is a greedy algorithm driven by heuristics. To preserve program's correctness, the scheduler first builds a data precedence graph (DPG). Nodes in DPG represent operations, edges represent data dependences. An edge from node A to node B means operation B depends on operation A. The DPG for the code fragment in Figure 2.2 is shown in Figure 2.3. Each operation is also assigned a priority using some heuristics such as latency-weighted depth. After DPG is constructed the scheduler picks ready operations in order of priority and fills them into the schedule cycle by cycle.

```
ADD  r0  r1  =>  r0
MUL  r1  r2  =>  r2
SUB  r0  r4  =>  r0
SUB  r2  r5  =>  r2
ADD  r2  r0  =>  r3
```

(a) Before scheduling

```
MUL  r1  r2  =>  r2  |  ADD  r0  r1  =>  r0
NOP                  |  SUB  r0  r4  =>  r0
SUB  r2  r5  =>  r2  |  NOP
ADD  r2  r0  =>  r3  |  NOP
```

(b) After scheduling

Figure 2.2 : Instruction level parallelism

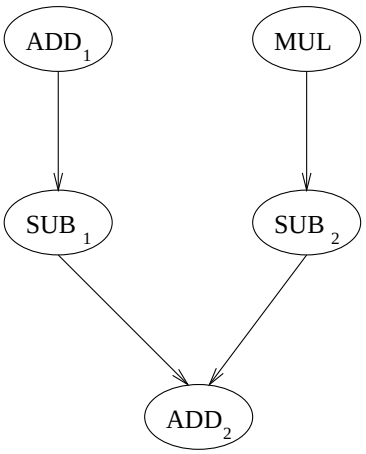


Figure 2.3 : A data dependence graph

2.2 Massively Scalar Compiler Project

Our instruction scheduler is part of the research compiler built and maintained by the Massively Scalar Compiler Project [3] at Rice University. Figure 2.4 shows the structure of the MSCP compiler. The front end of the compiler translates C or Fortran code into an intermediate representation called ILOC. ILOC is designed as the assembly language for an abstract RISC-like architecture. The ILOC code is then passed through various optimization phases. Finally the back end generates C code from ILOC.

The instruction scheduler is a component of the back end. It reads in the architecture description file for a target machine then schedules the code according to the specific architecture.

Different from commercial compilers, the back end of our research compiler produces C code instead of machine code. This C code can be instrumented to gather various statistics such as static operation count and dynamic instruction count. It is then compiled by standard C compilers and executed. By doing this, we are able to simulate the compiled code on abstract architectures with desired properties and evaluate quality of the code.

There are various optimization passes within the optimizer. All the benchmarks we tested were heavily optimized prior to scheduling. The optimizations we used include global value numbering, lazy code motion, algebraic reassociation, operator strength reduction, constant propagation, peephole optimization and dead code elimination.

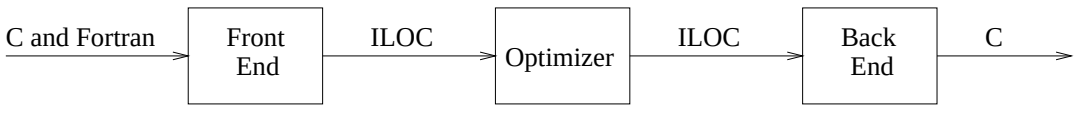


Figure 2.4 : MSCP research compiler

2.3 Architectural Models

Partitioned register sets appeared in a number of commercial machines from Multiflow TRACE in the 1980's to the TMS320C6xxx DSP chips developed by Texas Instruments recently. Our hypothetical two cluster machine model is based on the Texas Instruments TMS320C6xxx chips[15] which resemble the drawing in Figure 1.1. It consists of two identical clusters and is able to issue eight operations each cycle. Each cluster has four functional units, a register file, and an inter-cluster data bus to reach the other cluster. The functional units execute operations from the ILOC instruction set [4]. The four functional units are an integer unit, a floating-point unit, a control unit, and a memory access unit. To simplify the compiler, we assume that each ILOC operation is supported on exactly one of the four types of functional unit. Operation latencies are as follows: Integer operations take a single cycle, except for a two-cycle multiply. Floating point operations take three cycles, except for the six cycle multiply. Inter-cluster copies take one cycle. Branches require six cycles, while memory operations take five cycles. The odd ILOC operations that represent intrinsic functions (SIN, LOG, SQRT, *etc.*) take thirty cycles. We also assume that all functional units are fully pipelined, *i.e.* any functional unit can start a new operation in any cycle.

Our initial experiments in Chapter 4 assume an unlimited number of registers. Chapter 5 refines the results by restricting the register sets to practical size — each cluster has 32 integer and 32 floating-point registers.

As a VLIW machine, TMS320C6xxx requires NOP operations to be inserted into the scheduled code for cycles in which no operation starts. TMS320C6xxx has a multiple-cycle NOP operation which takes number of idle cycles as its only argument. Use of this special multiple-cycle NOP operation is reflected in our measurement of code size in Section 3.4.

In Chapter 3 and Chapter 4 we will also evaluate various algorithms using a four cluster architectural model. It is similar to the two cluster model but has four

identical clusters and can issue 16 operations each cycle. Each cluster has a distinct inter-cluster path to each other cluster.

2.4 Benchmarks

To evaluate various algorithms, we selected seven Fortran and C benchmark programs. Table 2.1 presents some basic statistics of these programs. `Doduc`, `fpppp`, and `tomcatv` are taken from the Spec '89 benchmark suite. `Rkf45` and `svd` come from a library of programs distributed with Forsythe, Malcolm, and Moler's numerical algorithms text [13]. `Nsieve` is the well-known Sieve of Eratosthenes benchmark written by Al Aburto. `Fft` is a 3-D fast Fourier transform written in C.

	doduc	fpppp	tomcatv	rkf45	svd	fft	nsieve
Number of Routines	45	16	2	6	2	–	–
Source Lines	5,354	2,620	210	690	382	1,037	326
ILOC Lines	36,728	12,821	1,359	1,527	2,427	2,754	733

Table 2.1 : Benchmark statistics

2.5 Previous Work

There are several previous algorithms for cluster assignment and instruction scheduling for partitioned register-set machines.

The Bulldog [11] compiler is the first study of this problem. It uses a two-phase approach — the first pass assigns each operation to a cluster, then the second pass uses list scheduling to construct a schedule that respects the cluster assignments. The Bottom-Up Greedy (BUG) algorithm is used in the assignment phase. The idea is to make a trial schedule focusing on the issue of minimizing the amount of data transfer latency. BUG does a depth first traversal of the DPG, starting at roots (representing

output values) working towards leaves (representing input values), at each node the best functional unit available at the time is chosen. The search is guided by the latency-weighted depth of the nodes, so that a critical path of the computation is always searched first.

The Multiflow Trace Scheduling Compiler [17] studied the ineffectiveness of BUG on highly parallel code and proposed a revised algorithm. The algorithm first partitions code into components, each of which contains relatively little parallelism and a relatively large amount of shared data. It then creates a partitioning of the components into equivalence classes. Two components are in the same equivalence class if one contains an operation whose result is read by other. During the assignment phase, each time a functional unit is chosen for an operation, the equivalence class becomes associated with the cluster containing the functional unit. When alternative functional units are considered for an operation, a penalty is imposed if the operation's equivalence class has an associated cluster different from the given cluster.

In their study of Limited Connectivity VLIW Architecture [5], Capitanio *et al.* gave a methodology that first schedules the code using Percolation Scheduling [19] then divides the code into substreams that minimize inter-cluster communication, and finally inserts necessary copy operations and recompacts the code. Using this methodology, they investigated different trade-offs in partitioned register set architecture design.

Özer *et al.* proposed an algorithm that brings together cluster assignment and scheduling into a single unified phase. They call their method "unified assign and schedule" (UAS) [20]. To integrate cluster assignment into list scheduling, UAS considers each possible cluster assignment for each ready operation. It checks the availability of a functional unit for the operation; if a functional unit is available, it checks the inter-cluster data paths to see if the necessary data-movement can be scheduled. It chooses the first cluster where the operation and its inter-cluster data-motion fits. They tested five priority functions for ordering the list of clusters and showed that UAS outperforms BUG.

Chapter 3

Scope of Instruction Scheduling

A good instruction scheduler for partitioned register-set machines must expose sufficient instruction level parallelism to effectively utilize the parallel hardware. As pointed out by the literature [11, 12, 17] only severely limited parallelism exists within basic blocks. To keep wide machines busy, we need to find more ILP by looking across basic block boundaries.

To select a suitable scheduling scope for our instruction scheduler, we investigated five scheduling methods with different scopes: basic block scheduling, trace scheduling, extended-basic-block scheduling without code duplication, extended-basic-block scheduling with code duplication and superblock scheduling. A **basic block** is a maximal length sequence of straight line code. Basic block scheduling simply applies list scheduling to basic blocks. Trace, extended-basic-block and superblock scheduling are described in the first three sections. Experimental results are presented and analyzed in the last two sections.

3.1 Trace Scheduling

Fisher [12] first described an algorithm called trace scheduling (TS) which has become a widely accepted instruction scheduling algorithm. TS divides a program into traces. A trace is a sequence of operations to be scheduled together. constructed.

When pickiol flow graph, the scheduler tries to pick a path that are most likely to be executed. To do this, it uses profile data or estimates of how often each block executes. The scheduler first finds the basic block with the highest estimated execution count, then grows the trace forward and backward from this block.

When growing the trace backward and forward, the same criteria are used for picking the next "good" block to add to the trace. In either case, the "best" flow-graph edge is chosen from a set of candidate edges: going forward, the candidate edges are those leaving the current end of the trace; going backward, the candidate edges are those entering the current beginning. An edge is **best** among all candidates if the scheduler considers flow most likely to proceed along it. The trace stops growing forward(backward) when there is no good successor(predecessor) or the chosen block has already been scheduled within an earlier trace.

Because trace scheduling attempts to schedule earlier traces as fast as possible at cost of making later traces longer, picking good traces is extremely important. Profile information or estimate of execution frequency is used to guide the selection of traces. Execution counts of each basic block and branch-taken frequencies are collected during profile-gathering runs over typical data. If no typical data is available, we assume a 10^l execution count for each basic block (l is the loop depth of the block) and a 50% probability for each branch to be taken.

List scheduling as described in Section 2.1 is used to schedule each trace. The scheduler reorders the trace, filling each cycle with operations from separate points on the trace. Using latency-weighted depth priority heuristic, time critical operations are usually scheduled early, while non-critical operations are often delayed.

Because of the movement of operations with respect to conditional jumps off the trace (splits) and jumps into the trace (joins), the scheduler has to insert new operations in front of off-trace successors or at the end of off-trace predecessors to preserve correctness of the program. This process of inserting correctness-preserving operations is called **bookkeeping**.

Figure 3.1 shows an example of bookkeeping at splits. Assume that the scheduler picks B_1, B_2 and B_3 to form a trace and schedules the trace as shown in (b). Notice that two operations $i = i + 1$ and $j = j - 1$ have been moved below the split. The program becomes incorrect. Copies of these two operations must be inserted on the

off-trace edge of the split (as shown in (c)) to preserve correctness of the program. Similarly we also need to do bookkeeping at joins. Figure 3.2 illustrates an example of this.

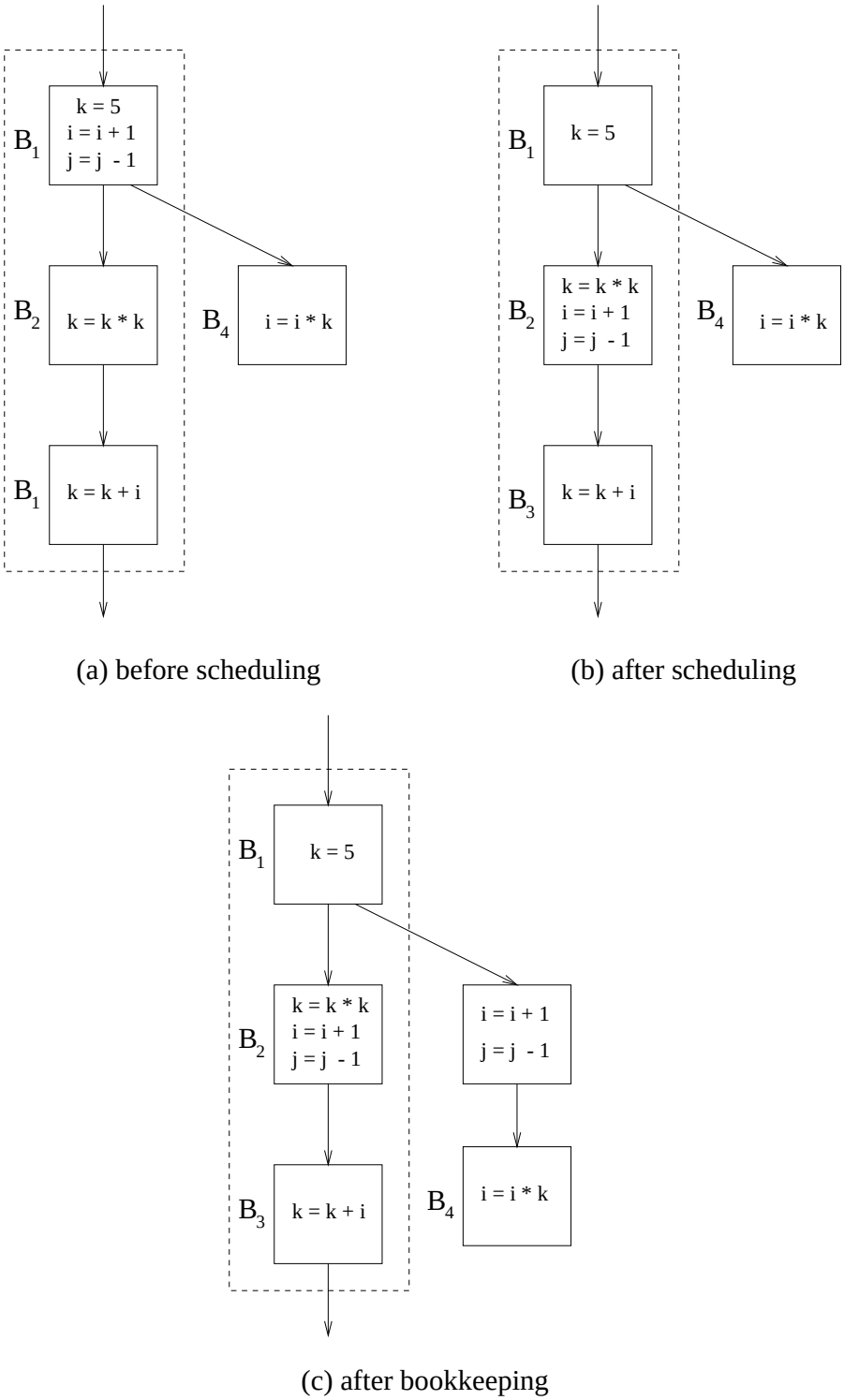
To preserve correctness, there are some restrictions on movement of operations respect to conditional jumps when scheduling a trace. Any operation that writes to some live-in variable(s) of any off-trace successor cannot be moved above the split. For example, the operation $k = k * k$ in Figure 3.1 is not allowed to move above the split since k is a live-in variable of the off-trace block B_4. Similarly operations that write on variables live out of off-trace blocks cannot be moved down a join. For example $j = j - 1$ in Figure 3.2 is not allowed to move down beyond the join. Such restrictions are encoded into the DPG by adding "pseudo-dependence" edges. For instance, a pseudo-dependence edge from the conditional branch in B_1 to $k = k * k$ must be added in the DPG for Figure 3.1(a) to prevent $k = k * k$ from moving above the split. A trace can be scheduled just like a basic block after all restrictions have been encoded into its DPG.

A more detailed description of trace scheduling algorithm can be found in Ellis's thesis [11].

3.2 Extended Basic Block Scheduling

An extended basic block (EBB) is a sequence of basic blocks, $B_1, B_2, \ldots, B_k$ such that for $1 \leq i < k$, B_i is the only predecessor of B_{i+1}, and B_1 does not have a unique predecessor [1] [10]. Figure 3.3 shows a control flow graph (DPG) of 5 blocks. There are several distinct EBBs: (B_1, B_2, B_4), (B_1, B_3) and (B_5). Since compiler cannot schedule within one instance a block such as B_1 in two conflicting ways, we must form EBBs as disjoint sets of basic blocks for scheduling. One possible partition is (B_1, B_2, B_4), (B_3) and (B_5). Another feasible partition is (B_1, B_3), (B_2, B_4) and (B_5).

Our EBB scheduler picks next EBB from the control flow graph in a similar way to the trace scheduler picking next trace. It first finds the basic block with the

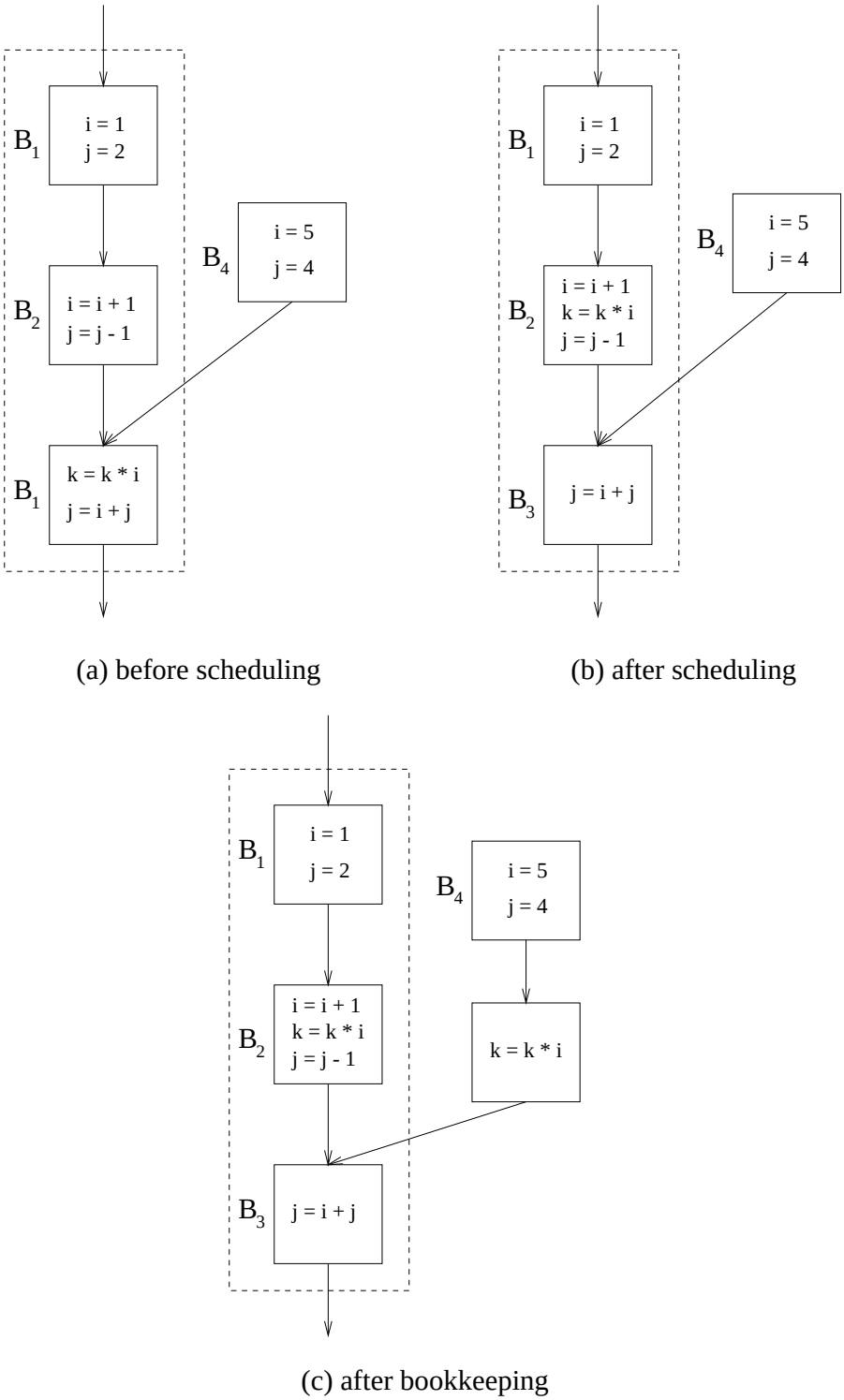


(a) before scheduling

(b) after scheduling

(c) after bookkeeping

Figure 3.1 : Example of bookkeeping at split



(a) before scheduling

(b) after scheduling

(c) after bookkeeping

Figure 3.2 : Example of bookkeeping at join

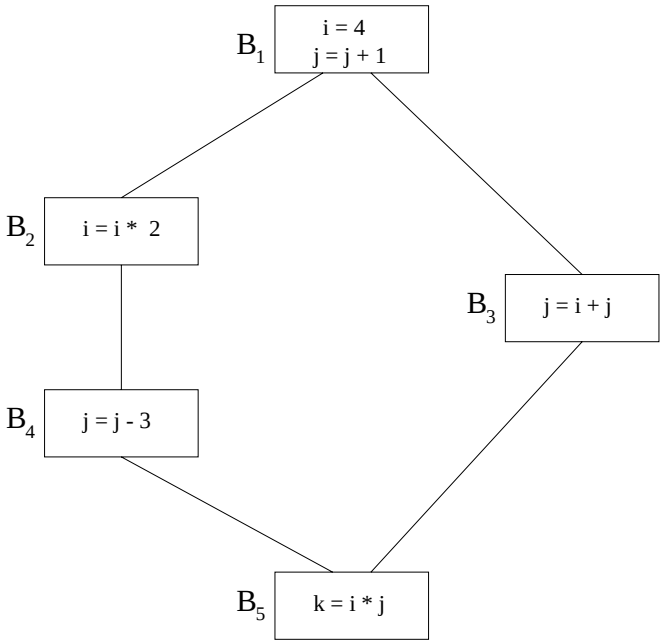


Figure 3.3 : EBB scheduling example

highest estimated execution count, then grows the EBB forward and backward from this block. The EBB stops growing backward when the current block has more than one predecessors or its only predecessor belongs to an already scheduled EBB. To grow forward, the scheduler picks a "good" successor with the additional condition that the chosen block has only one predecessor. The EBB stops growing forward when there is no good predecessor or the chosen predecessor has already been scheduled.

There are a number of EBB construction heuristics different from ours, but Philip Schielke's thesis [21] suggests that none of these heuristics has any clear advantage over others.

By definition, an EBB has no join inside it. We only need to worry about movement of operations respect to splits. To preserve correctness, any operation which writes on some live-in variable of any successor outside the EBB is not allowed to move up beyond a split.

We tested two versions of EBB scheduling: EBB scheduling without code duplica-

tion (EBB1) and EBB scheduling with code duplication (EBB2). EBB1 doesn't allow operations to move down a split, thus no bookkeeping is necessary.

As with trace scheduling, after the restrictions on code motion have been encoded into DPG, an EBB can be scheduled just like a basic block.

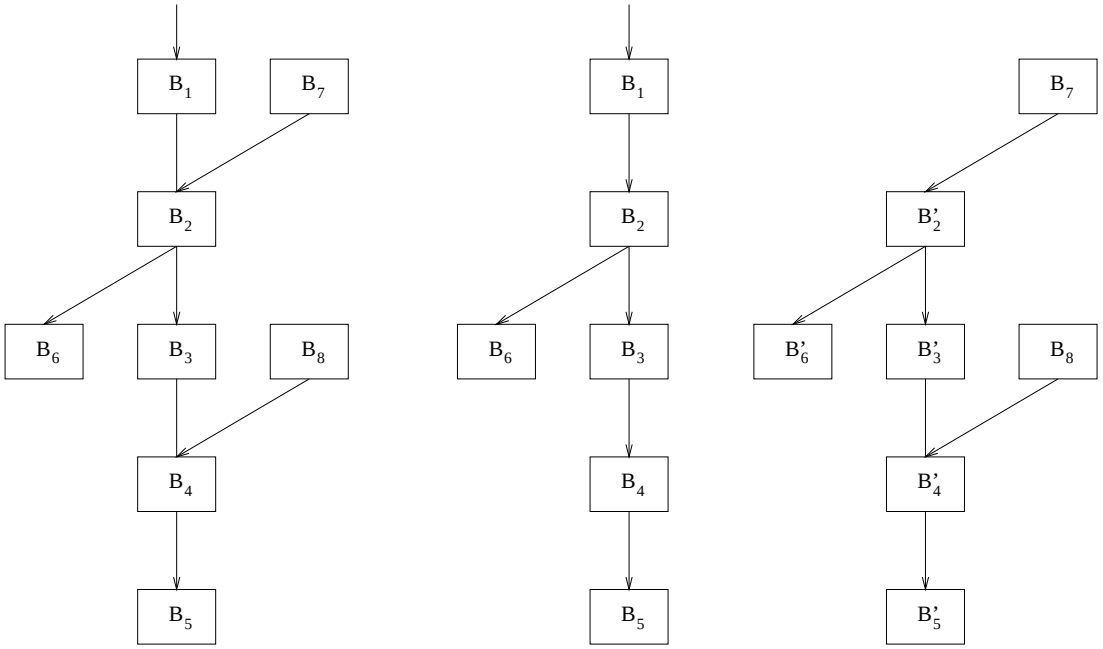
3.3 Superblock Scheduling

Hwu etc.[18] proposed an algorithm called superblock scheduling (SB). A superblock is a trace which has no side entrance. Control can only enter from the top but may leave in the middle of a superblock. Superblocks are formed in two steps: first traces are identified in exactly the same way as trace scheduling; second a copy is made of the tail portion of the trace from the first side entrance to the end (tail duplication) then all side entrances are moved to the corresponding duplicated basic blocks. Figure 3.4 shows an example.

Since there is no side entrance for a superblock, bookkeeping only need to be done at splits. The restrictions are identical to those for an EBB, and they are handled the same way. After restrictions of code movement have been encoded into DPG, a superblock can be scheduled like a basic block.

3.4 Experimental Results

Experimental results for five scheduling algorithms are listed in Table 3.1 and Table 3.2 in order of enlarging scope. The **dynamic instruction count** listed in Table 3.2 is the total number of instructions executed in a run of scheduled code. Here we make a distinction between **operation** and **instruction**: an **instruction** is defined as a set of operations that begin in the same cycle on different functional units. Therefore the dynamic instruction count represents the execution time of scheduled code. The **static operation count** listed in Table 3.2 is the number of operations in the scheduled code. It reflects the size of the scheduled code. Because of the use of



(a) before tail duplication

(b) after tail duplication

Figure 3.4 : Example of tail duplication in superblock scheduling

multiple-cycle NOP operation in our hypothetical machine models, consecutive NOP operations are counted as one operation.

As shown in Table 3.1, enlarging the scope of scheduling reduces the execution time of the scheduled code. Since EBB, TS and SB can schedule more than one basic blocks a time, they are able to exploit more instruction level parallelism and produce faster code than BB. EBB2 can reorder operations more freely than EBB1 therefore it has more chances to produce faster code than EBB1. Since branches are allowed out of and into the middle of a trace, a trace could be much longer than an extended basic block, therefore TS can find more ILP and produce faster code than EBB scheduling. Because tail duplication allows later traces to be more freely scheduled without restrictions from scheduling decisions of earlier traces, SB can produce faster code than TS. SB gives the fastest code overall, up to 36.6% faster than BB (for svd).

Table 3.2 shows the trend of increasing code size with scope of scheduling enlarged. Because the bookkeeping process duplicates some operations, EBB2, TS and SB produce larger code than BB and EBB1. Since SB also copies whole blocks during tail duplication, it produces much longer code than other scheduling algorithms, up to 283% larger than BB (for fpppp).

Although there is no code duplication by EBB1, Table 3.2 shows a slight increase of code size from BB to EBB1. After reordering operations in an extended basic block, sometimes the scheduler must insert some NOPs in front of the off-trace successor block to let all operations before the split finish, otherwise the code would not be correct. These NOPs make code produced by EBB1 slightly larger than that by BB.

3.5 Speed and Size Trade-off

For many years, compiler optimizations and scheduling techniques have been focused on making code faster. With digital signal processors (DSPs) being widely used in embedded systems and network computing gaining popularity, the size of compiled code becomes increasingly important. In embedded systems the compiled code is

	BB	EBB1		EBB2		TS		SB	
	# insts	# insts	% dec.	# insts	% dec.	# insts	% dec.	# insts	% dec.
doduc	13921893	11574599	16.9	11145406	19.9	9602784	31.0	9539988	31.5
fft	15482275	14736742	4.8	14601080	5.7	14995539	3.1	14753947	4.7
fpppp	144723232	135759328	6.2	134274304	7.2	130909008	9.5	130051760	10.1
nsieve	3810923264	3489669120	8.4	3489669120	8.4	3248780032	14.8	3248756480	14.8
rkf45	484847	423435	12.7	376374	22.4	358432	26.1	356804	26.4
svd	13195	10288	22.0	9211	30.2	8549	35.2	8363	36.6
tomcatv	403587456	379695200	5.9	349409184	13.4	331964192	17.7	331751456	17.8

Table 3.1 : Dynamic instruction counts after scheduling

	BB	EBB1		EBB2		TS		SB	
	# insts	# insts	% inc.	# insts	% inc.	# insts	% inc.	# insts	% inc.
doduc	38038	38201	0.4	40465	6.4	55428	45.7	100431	164
fft	1982	2026	2.2	2138	7.9	2229	12.5	3000	51.4
fpppp	13723	13751	0.2	16223	18.2	21428	56.1	52497	283
nsieve	295	298	1.0	308	4.4	328	11.2	365	23.7
rkf45	1664	1660	-0.2	1771	6.4	2106	26.6	3927	136
svd	2523	2537	0.6	2648	5.0	4148	64.4	8425	234
tomcatv	1359	1383	1.8	1612	18.6	2407	77.1	4551	235

Table 3.2 : Static operation counts after scheduling

often burnt into a read-only memory, or ROM. Smaller code size can make the ROM smaller hence reduce overall system's cost. In web applications, shorter code can reduce user's waiting time for code transmission. Some recent research work has focused on reducing code size[8, 9].

Since partitioned register-set architecture appears in a number of DSP chips such as Texas Instruments TMS320C6xxx series, we want to pick a base scheduling algorithm which gives a good trade-off between code size and speed.

From comparison of all 5 scheduling algorithms we can see TS gives the best result. It produces second fastest code, only 1% slower than SB in average, while the growth of code size is much more moderate than SB, 42% larger than BB in average compared to SB's 161%. We will use trace scheduling through our study on partitioned register-set machines, but similar results can be obtained with other scheduling algorithms.

It is worth noticing that EBB1 is the most attractive choice when code size is critical. It produces code 11% faster than BB in average with negligible growth in code size. Philip Schielke's thesis includes work on global scheduling techniques that do not replicate code[21]. Our numbers corroborate his results.

To address more about our concern of code size we also add an optional switch in our implementation of the scheduler — when code growth exceeds a limit set by the user, the switch will be turned on and SB, TS or EBB2 will degenerate to EBB1 for all remaining code.

Chapter 4

Assign and Schedule

In this chapter, we study cluster assignment and scheduling for partitioned register-set machines assuming each register set has unlimited number of registers. Section 4.1 introduces the combined cluster assignment and instruction scheduling problem. Section 4.2 and 4.3 describe two algorithms adapted from previous work on this problem. Section 4.4 proposes two new algorithms. Experimental results are presented in Section 4.5.

4.1 Introduction

The partitioned register-set machines present a new set of challenges to compiler designers. The compiler must place each operation in a specific cluster and then work to ensure that its operands are either available in the local register file or available, in a timely fashion, over the inter-cluster data path. These problems are complicated by the latency and limited capacity of the inter-cluster data path.

Figure 4.1 shows an example of cluster assignment and schedule. Assume that a two cluster machine has one functional unit in each cluster; multiply takes 2 cycles; memory access operations take three cycles; all other operations take one cycle; the inter-cluster data buses support one read in each direction every cycle — one read from Cluster 2 by Cluster 1 and one read from Cluster 1 by Cluster 2; the latency of an inter-cluster copy is one. The code segment in (a) is assigned and scheduled on one cluster only. It takes 10 cycles. These operations can be assigned to both clusters as shown in (b). The resulting scheduled code only takes 8 cycles. Here we assume that all live-in variables are initially stored in register file of Cluster 1 only

Cluster 1
LOAD @a => r3
LOAD @b => r4
SUB r1 r0 => r1
MUL r3 r0 => r3
ADD r1 r2 => r2
SUB r3 r1 => r3
ADD r4 r2 => r4
SUB r4 r1 => r4
ADD r4 r4 => r4
SUB r3 r4 => r3

(a) using one cluster

Cluster 1	Cluster 2
LOAD @a => r3	LOAD @b => r4’
ADD r1 r2 => r2	COPY r0 => r0’
SUB r1 r0 => r1	COPY r2 => r2’
MUL r3 r0 => r3	ADD r4’ r2’ => r4’
NOP	SUB r4’ r0’ => r4’
SUB r3 r1 => r1	ADD r4’ r4’ => r4’
COPY r4’ => r4	NOP
SUB r3 r4 => r3	NOP

(b) a good assignment on two clusters

Cluster 1	Cluster 2
LOAD @a => r3	COPY r0 => r0’
LOAD @b => r4	COPY r1 => r1’
NOP	SUB r1’ r0’ => r1’
COPY r1’ => r1	COPY r3 => r3’
ADD r1 r2 => r2	MUL r3’ r0’ => r3’
NOP	COPY r2 => r2’
NOP	SUB r3’ r1’ => r3’
COPY r3’ => r3	COPY r4 => r4’
COPY r2’ => r2	ADD r4’ r2’ => r4’
NOP	SUB r4’ r1’ => r4’
NOP	ADD r4’ r4’ => r4’
COPY r4’ => r4	
SUB r3 r4 => r3	

(c) a bad assignment on two clusters

Figure 4.1 : Example of cluster assignment

and all live-out variables need to be present in Cluster 1 at the end of execution. Any register name with a prime denotes a register in Cluster 2. Inter-cluster copies such as $COPY\ r2 => r2'$ are carefully inserted into the code to ensure all operands are available when an operation is executed.

Not all assignments over two clusters produce faster scheduled code than using only one cluster. Figure 4.1 shows a bad assignment that produces code which needs 3 more cycles than the single cluster code.

From the example we can see that a good assignment attempts to exploit instruction level parallelism by distributing operations to all clusters. At the same time it also places operations close to their operands to reduce the effect of inter-cluster copy latency. On the other hand, a bad assignment puts too many operations in one cluster and/or places operations far away from their operands. For example, in Figure 4.1(c), assigning $MUL\ r3\ r0 => r3$ to Cluster 2 requires the extra inter-cluster copy from $r3$ to $r3'$ which causes a delay of one cycle.

If each cluster has multiple functional units, more off-cluster operands will appear in the scheduled code than in our example. Ready operations have to compete for the use of limited capacity inter-cluster data buses and some of them must wait for the buses to be free. Again, closeness to operands, hence reduced inter-cluster data communication, is preferred.

These observations are reflected in design of our algorithms in later sections.

4.2 Bottom-Up Close Algorithm

The Bottom-Up Close algorithm (BUC) is adapted from Ellis' Bottom-Up Greedy algorithm (BUG)[11]. Like BUG, BUC is a two-phase algorithm. The first pass assigns each operation to a cluster. The second pass uses list scheduling to construct a schedule that respects the cluster assignments. BUC extends BUG by trying to balance the load across functional units and clusters.

Figure 4.2 shows a high-level sketch of the recursive BUC assignment algorithm.

```
buc_assign(dpg_node, sibling_locations)
{
    operand_locations = empty
    foreach operand node op_node of dpg_node
        buc_assign(op_node, operand_locations)
    dpg_node.cluster = best_cluster(operand_locations, siblings_locations, load)
    update sibling_locations
    update load
}
```

Figure 4.2 : Cluster assignment in BUC

It starts with a depth first traversal of the DPG, starting at the roots (representing output values) and working towards the leaves (representing input values). At each level, the search chooses the node with the largest latency-weighted depth in the the DPG. This focuses attention on the critical path through the computation. To process an operation, the algorithm first recurses on its operands. After each operand has a cluster assignment, the operation itself is assigned to the best cluster at the time. The *best_cluster* function picks the cluster closest to all the operands and any of its siblings that have already been assigned. We call two nodes **siblings** if they are both operands in the same operation. As the algorithm traverses the tree, it records the cluster assignment for each operation and passes that information along for any as-yet-unassigned siblings. The computation of closeness is a simple weighted computation that takes into account inter-cluster transfer costs. It breaks ties in favor of the cluster with fewer assigned operations and, thus, less load.

Because it uses depth-first search and a latency-weighted depth priority scheme, BUC tends to place operations on the critical path together. The load-guided tie-

breaking heuristic tries to distribute the off-critical-path computations onto other clusters in order to achieve a more balanced load.

4.3 Unified Assign and Schedule

Özer *et al.* proposed an algorithm that brings together cluster assignment and scheduling into a single unified phase. They call their method "unified assign and schedule" (UAS) [20]. A high-level sketch of the UAS algorithm is shown in figure 4.3.

The outer loop of the algorithm works in the same way as a list scheduler — ready operations are filled into the schedule cycle by cycle. Once a cycle is scheduled, it is never revisited. In the inner loop, to integrate cluster assignment into scheduling, UAS considers each possible cluster assignment for each operation it schedules. It first forms a prioritized list of clusters on which the current operation can possibly be scheduled. Then, in priority order, each cluster is examined to see if it has a free functional unit for the operation; if a functional unit is available, the inter-cluster data paths are checked to see if the necessary data-movement can be scheduled.

```
unified_assign_and_schedule()
{
    while(unscheduled operations exist)
        form a list of data-ready operations
        for each data-ready operation x in order of priority
            create priority list of clusters on which x can potentially be scheduled
            for each cluster c in the priority list
                if(x can be scheduled on c and all required copy operations can be scheduled)
                    schedule x and all required copy operations
        increment cycle counter
}
```

Figure 4.3 : UAS algorithm

Özer *et al.* tested five priority functions for ordering the search of clusters. Their experiments showed that UAS with all priority functions except for random ordering outperforms BUG.

4.4 Top-Down First and Top-Down Close Algorithm

As pointed out by Özer *et al.*, BUG does not perform well because cluster assignment is separated from scheduling; the assignment phase has trouble anticipating the actual use of functional units and inter-cluster data buses. Inevitably, a few decisions made by the assignment phase will keep functional units and inter-cluster buses unnecessarily idle in some cycles. UAS successfully solved this by integrating assignment and scheduling into the same phase. Unfortunately it causes a new problem: the act of assigning an operation to a specific cluster during scheduling can create the need for inter-cluster copies that did not exist in the code presented to scheduler; because UAS moves monotonically forward in the schedule, it cannot put these copies into earlier cycles where free slots are still available; unnecessary delays may be caused by late copies.

Consider the code fragment in Figure 4.4(a). Assume that a two cluster machine has one functional unit per cluster; multiply takes two cycles; memory access operations take three cycles; all other operations take one cycle; live-in variables are initially present in both clusters. The best schedule that UAS can produce is shown in Figure 4.4(b). Apparently if we move $COPY\ r1' => r1$ to cycle 3 as shown in Figure 4.4(c), a cycle can be saved. UAS cannot achieve this: the inter-cluster copy did not exist when it scheduled cycle 3; when the scheduler finds out that a copy operation is needed for the add operation at cycle 5, the best it can do is to schedule the copy in the current cycle and the add operation to the next cycle. This suggests that revisiting earlier cycles may be necessary. The better schedule in Figure 4.4(c) can be easily achieved by revisiting: when the scheduler finds out a copy is needed at cycle 5, it revisits all cycles after the latest write to $r1$ and finds out that the copy

```
LOAD @a => r2
MUL r1 r0 => r1
SUB r3 r0 => r3
SUB r2 r0 => r2
SUB r3 r1 => r3
ADD r2 r1 => r1
```

(a) unscheduled code

Cluster 1	Cluster 2
LOAD @a => r2	MUL r1 r0 => r1
NOP	NOP
NOP	SUB r3 r0 => r3
SUB r2 r0 => r2	ADD r3 r1 => r3
COPY r1' => r1	NOP
ADD r2 r1 => r2	NOP

(b) schedule produced by UAS

Cluster 1	Cluster 2
LOAD @a => r2	MUL r1 r0 => r1
NOP	NOP
COPY r1' => r1	SUB r3 r0 => r3
SUB r2 r0 => r2	ADD r3 r1 => r3
ADD r2 r1 => r2	NOP

(c) a better schedule by revisiting earlier cycles

Figure 4.4 : Example of revisit

can be put into the free slot at cycle 3.

In above discussion we have not considered the possibility that in cycle 5 the value $r1$ actually can be read directly over the inter-cluster data bus. The reason is: In practice, each cluster usually has multiple functional units; the inter-cluster data bus probably has already been taken by other operation(s). Even if the bus is available, we still prefer inserting a copy in earlier cycles because (1) it may give other operations the opportunity to be scheduled in the current cycle (2) the copied value could be used by later operations.

We developed two related algorithms that not only combine assign and schedule into the same phase but also revisit earlier cycles to insert inter-cluster copies. We call them Top-Down First (TDF) and Top-Down Close (TDC). Figure 4.4 gives an outline of TDC. It differs from BUC in that BUC traverses the DPG bottom-up (from outputs to inputs), while TDC traverses the DPG top-down (from inputs to outputs). At each cycle, the data-ready operations are considered in order of depth-weighted latency. For an operation o, if all its operands reside in cluster c and the appropriate functional unit in c is free, then o is assigned to c and scheduled into the current cycle. If the operands reside on multiple clusters, TDC looks for a free functional unit to execute o and then looks backward in the schedule for cycles where the needed inter-cluster transfers can be placed. If copies can be inserted to make all of o's operands available on cluster b in the current cycle, it schedules o onto b in the current cycle. If some of operands cannot be pre-copied, the algorithm checks to see if the inter-cluster data buses are available in the current cycle to read them directly. Copying is preferred over direct use because a copied value can be reused by other operations in the same cluster. After an operation and any necessary inter-cluster copies are scheduled, usage of resources such as functional units, inter-cluster data buses and registers is updated.

TDF is similar to TDC except for the order it uses to check the clusters. In TDC, the clusters are scored by the closeness of operands and considered in closest-first

```
tdc_assign_and_schedule()
{
    data_ready_set = all leaf nodes in dpg
    running_set = empty
    while(data_ready_set or running_set not empty)
        foreach op in data_ready_set, ordered by latency-weighted depth
            for each cluster c in order of closeness to operands of op
                if(!resource_conflict(op, c))
                    assign and schedule op to c
                    insert necessary inter-cluster copies
                    remove op from data_ready_set
                    add op to running_set
                    update resource usage
                    break
        increment cycle counter
        remove finished operations from running_set
        add data ready operations to data_ready_set
}

resource_conflict(op, assigned_cluster)
{
    if functional unit for op in assigned_cluster is busy
        return true
    foreach operand x of op not present in assigned_cluster
        if a copy can be inserted in previous cycles to move x to assigned_cluster
            continue
        else if x can be read directly from neighboring cluster in current cycle
            continue
        else
            return true
    return false
}
```

Figure 4.5 : TDC algorithm

order. In TDF, clusters are considered in canonical order, with no preference given for closeness.

Because they reconsider already scheduled cycles, both TDF and TDC can be more expensive than UAS. In practice, the size of scheduled blocks is often short, limiting this effect. We can limit the amount of extra work by limiting the algorithm to looking at the k previous cycles. Reasonably small values of k, such as 20 cycles, should avoid the asymptotic problems while discovering most of the opportunities.

4.5 Experimental Results

To compare the performance, we implemented these four algorithms in our research compiler and tested them on seven benchmarks against the two-cluster and four-cluster machine models which are described in Section 2.3. Trace scheduling is used in our experiments.

Our implementation of UAS limits itself to scheduling operations where all the off-cluster operands can be read directly from the remote register files. This excludes any solution with inter-cluster copies. The published description of UAS [20] states "... *If copies can be scheduled on their respective clusters, i.e. there are enough available inter-cluster buses in the current cycle, in Step 6, the current operation and associated copies are scheduled* (in the current cycle) ..." This is incorrect because the copied values cannot be used in the current cycle — there is a latency for inter-cluster copies. Without revisiting earlier cycles, the closest thing an UAS scheduler can do is to read these off-cluster directly without copying. Of course the scheduler can also schedule the copies in the current cycle and the current operation in the next cycle. But a delay of one cycle is caused. Although it is not clear which approach gives better performance, we believe our implementation (using the first approach) gives a good approximation of what UAS can achieve.

As suggested by Özer *et al.* [20], we use the Magnitude-Weighted Predecessor (CWP) priority function to order the clusters in our implementation of UAS.

	BUC	UAS	TDF	TDC	Fully conn.
doduc	10,396,703	10,413,748	9,982,047	9,973,867	9,602,784
fft	15,506,968	17,273,368	15,324,780	15,324,779	14,995,539
fpppp	145,485,840	147,601,248	137,898,320	137,651,264	130,909,008
nsieve	3,266,130,176	3,929,078,784	3,248,809,472	3,248,809,472	3,248,780,032
rkf45	379,635	397,467	368,083	367,819	358,432
svd	9,311	10,407	9,040	9,042	8,549
tomcatv	355,516,224	351,897,504	332,085,504	332,085,536	331,964,192

Table 4.1 : Cycle counts for the two cluster machine model

	BUC	UAS	TDF	TDC	Fully conn.
doduc	10,030,053	10,344,406	9,791,386	9,792,944	9,252,947
fft	15,290,297	18,225,168	15,239,198	15,240,086	14,744,885
fpppp	146,414,016	148,099,520	140,140,048	139,993,872	127,103,552
nsieve	3,266,147,840	4,277,894,144	3,248,827,136	3,248,827,136	3,248,780,032
rkf45	381,375	401,036	367,572	367,323	354,939
svd	9,320	10,646	9,247	9,255	8,448
tomcatv	355,438,848	353,349,088	329,423,936	329,424,000	329,315,808

Table 4.2 : Cycle counts for the four cluster machine model

The results for the two-cluster machine are shown in Figure 4.1, while Figure 4.2 shows the results on the four-cluster machine. The final column in each table shows the results for a fully-connected machine with the same number of functional units as the partitioned machine. In a fully-connected machine, each functional unit can access each register, thus no inter-cluster transfer is needed. Cluster assignment and scheduling devolves into simple scheduling. This should provide a lower bound on what can be achieved with a good assign and schedule algorithm.

As expected, BUC does not perform well. Because cluster assignment is separated from scheduling, the assignment has trouble anticipating the actual use of functional units and inter-cluster data buses. This causes underuse of functional units and inter-cluster data buses. UAS is also disappointing. Because it never revisits a cycle, a significant amount of inter-cluster bandwidth is wasted. This leads to unnecessary delay of inter-cluster data transfers and thus longer execution time. Both TDF and TDC produce better code. This is due to the extra compile time that they spend trying to insert data transfers into already scheduled cycles. Better utilization of functional units and inter-cluster data buses is achieved as a result. TDC and TDF have similar performance; neither has a clear advantage over the other.

Chapter 5

Handling Finite Register Sets

The results in last chapter assume that each cluster has an unlimited number of registers. This assumption is unrealistic — real machines have relatively small register sets. In this chapter, we first examine the difficulties introduced by finite register sets, then we look for an approach to handle them.

5.1 Introduction

For fully connected (*i.e.,* non-partitioned) machines, the problem of finite register sets is usually solved by doing register allocation before or after instruction scheduling. Unfortunately, instruction scheduling and register allocation often act against each other. When register allocation is done before instruction scheduling, spurious anti-dependences are introduced when the allocator maps multiple values to the same physical register. These anti-dependences lower the instruction level parallelism available for the scheduler to exploit. When instruction scheduling is carried out first, reordering of operations by the scheduler increases the demand for registers. This increased register pressure causes additional spill code.

On a partitioned machine, the introduction of inter-cluster transfers by cluster assignment complicates matters further. Performing allocation before TDC implicitly ignores those transfers. If the allocation is tight — that is, it has few unused registers — then TDC will be unable to duplicate values. Performing allocation after TDC deprives the cluster assignment algorithm of knowledge that might affect its decisions — the availability of registers on each cluster. If the allocation is tight, bad decisions by TDC lead to extra spills.

To find a good approach to handle finite register sets, we tested different schemes combining TDC with register allocation. The results are given in next three sections. In most of the experiments, we used a two cluster machine where each cluster has 32 integer and 32 floating-point registers.

For register allocation, we used a graph-coloring [6, 4] global register allocator that performs both clean spilling [2] and rematerialization [4].

5.2 An Initial Approach

As an initial attack on the problem, we have the compiler allocate registers for a single cluster's register set before running TDC. This has the effect of reserving a register for each enregistered value in each cluster. It provides TDC with maximal freedom, since a value in register i of one cluster has the same space allocated for it in each of the other clusters. On the other hand, the serious underallocation (only half of the registers are used for allocation) may cause a large amount of extra spill code.

To assess the impact of this underallocation, we did register allocation for different number of registers before trace scheduling on a fully connected machines. The results are shown in Table 5.1. The second column shows the result given by allocation for 64 registers. It provides the lower bound on what can be achieved by our initial strategy on a partitioned two cluster machine where each cluster has 64 registers. For comparison, the third column shows the result given by allocation for 128 registers. The final column shows the cycle counts for unlimited number of registers.

The impact of limiting the register set varies tremendously from example to example. On `fpppp`, where demand for registers is quite high, allocation to 128 registers more than doubles the cycle count. On `svd` and `fft`, the allocator does quite well. The impact of moving from 128 registers to 64 registers is less dramatic, but significant. With `fpppp`, the 64 register code takes 15% longer; on `doduc`, the loss is 9.5%. In two cases, the 64 register code is actually faster than the 128 register code by 1% or less. Experience with graph coloring allocators suggests that this arises from better

	64 registers	128 registers	∞ registers
doduc	19,625,586	17,759,936	9,602,784
fft	16,633,376	16,795,662	14,995,539
fpppp	309,889,312	263,669,072	130,909,008
nsieve	3,667,608,064	3,667,608,064	3,248,780,032
rkf45	421,147	424,060	358,432
svd	10,656	9,841	8,549
tomcatv	467,012,352	453,823,232	331,964,192

Table 5.1 : Cycle counts with different register allocation

spill code placement.

Apparently our initial approach is not suitable for programs with high register pressure such as `fpppp`. The huge amount of spill code caused by underallocation will overwhelm any gain from freedom for TDC to duplicate values. A method using more registers for allocation is needed.

5.3 Using Copy Registers

As an alternative, consider the strategy of reserving a small pool of registers in each cluster for use in inter-cluster copies. The graph coloring allocator handles the remaining registers. To assess the performance of this strategy, we ran the allocator followed by TDC for a two-cluster machine with a pool of 52 registers (26 integer and 26 floating-point) under the allocator's control and the remaining 12 registers reserved for inter-cluster copies. The final column of Table 5.2 shows the result of this strategy. On `doduc`, `fpppp`, `svd`, and `tomcatv`, it produces results between those of a fully-connected machine and those achieved by coloring for a single cluster. On `fft` and `nsieve`, reserving registers for copies leads to worse results — this suggests that the registers are better used for other purposes. `Rkf45` is perplexing; the code for 64 registers was faster than that for 128 registers due to spill placement. The code

	52/2	52/4	52/6	52/8	52/10	52/12
doduc	18,592,384	18,497,480	18,459,428	18,459,088	18,457,876	18,457,416
fft	17,055,576	16,983,284	16,880,086	16,910,808	16,937,688	16,930,008
fpppp	285,596,608	283,771,552	283,248,032	283,187,072	283,160,192	283,060,992
nsieve	3,918,767,360	3,747,920,128	3,747,920,128	3,747,920,128	3,747,920,128	3,747,920,128
rkf45	429,969	428,146	426,225	424,088	423,535	423,510
svd	10,122	10,058	10,039	10,033	10,027	10,027
tomcatv	461,932,000	455,338,784	455,228,576	451,342,336	451,337,248	451,337,248

Table 5.2 : Two cluster machine with varying number of copy registers, 52 registers under allocator's control

for the 52/12 scheme falls midway between them, but is still worse than the more restrictive 64 register allocation. Spill code placement in coloring allocators often produces unexpected results!

To show the sensitivity of the results to the number of copy registers, the earlier columns in the table show results for the same 52 register allocation with different numbers of copy registers. As expected, the general trend is toward slower code with fewer copy registers. The improvement arises from two key effects:

1. The availability of additional copy registers makes it easier to insert inter-cluster copies. This increases the likelihood that an operation with one or more off-cluster operands can be scheduled promptly (without a delay waiting on registers).

2. The additional copy registers let more off-cluster operands be reused. With fewer copy registers, these values get evicted because the register is needed as the target for other inter-cluster copies.

As with most scheduling and allocation problems, some NP-noise creeps into the problem. On `fft`, the 52/6 combination produces better results than either more or fewer copy registers.

	62/2	60/2	58/2	56/2	54/2	52/2
doduc	18,210,704	18,277,580	18,368,568	18,429,932	18,533,688	18,592,384
fft	17,067,856	17,075,532	17,062,216	17,063,258	17,055,578	17,055,576
fpppp	271,338,144	273,718,048	275,497,888	278,658,816	281,923,840	285,596,608
nsieve	3,918,767,360	3,918,767,360	3,918,767,360	3,918,767,360	3,918,767,360	3,918,767,360
rkf45	433,878	433,765	433,326	433,376	433,401	429,969
svd	10,077	10,081	10,188	10,180	10,157	10,122
tomcatv	467,111,776	467,127,040	460,634,688	461,901,536	461,911,008	461,932,000

Table 5.3 : Two cluster machine, two copy registers per cluster, varying number of registers under allocator's control

Of course, the 52/2 combination uses only 54 registers. The decrease in registers available to the allocator should have an impact on the amount of spill code that must be inserted. Table 5.3 shows the cycle counts for a configuration with 2 copy registers and a varying number of general purpose registers. As expected, increasing the pool of registers available to the allocator usually produces better code. Again, minor perturbations in that trend arise from the complex nature of the problem. For example, `rkf45` has an anomaly for 56/2 and 54/2, and `fft` shows a slowdown from 58/2 to 60/2.

5.4 A Refined Approach

To improve the approaches in the last section, we designed a scheme to use a fixed number of registers and vary the partition between copy registers and general purpose registers. The algorithm is slightly more complex. It reserves k registers for dedicated use as copy registers. It performs register allocation to a register set containing $(64-k)\times n$ registers, where n is the number of clusters. Finally, it uses TDC to perform cluster assignment and scheduling, but allows TDC to use any available register for a copy. (After allocation, some registers under the allocator's control are idle. This version of TDC scavenges those resources and uses them for inter-cluster copies.)

Table 5.4 shows the experimental data using this approach. The results vary from

	62/2	60/4	58/6	56/8	54/10	52/12
doduc	18,094,468	18,146,608	18,228,172	18,284,976	18,388,556	18,458,584
fft	16,948,956	16,960,472	16,920,020	16,922,072	16,937,432	16,945,368
fpppp	271,222,656	272,135,424	273,040,352	276,028,096	279,092,832	283,059,008
nsieve	3,747,920,128	3,747,920,128	3,747,920,128	3,747,920,128	3,747,920,128	3,747,920,128
rkf45	426,930	427,395	427,420	427,445	427,470	423,510
svd	10027	10,041	10,064	10,058	10,049	10,027
tomcatv	457,958,304	457,886,080	450,072,640	451,345,664	451,331,744	451,331,724

Table 5.4 : 64 registers per cluster, with varying number of copy registers

code to code, as might be expected. For example, `fft` achieves the best results for 58/6. The results also show the non-linear variation seen in the other experiments. Notice, for example, that `rkf45` does best with 52/12, but that its second best result is with 62/2. In between these two configurations, the code gets slower. Taken on average, the 62/2 configuration provides the best overall results. Since the configuration is enforced completely in the compiler, it can be varied to suit the situation. For performance critical applications, the compiler could try several different configurations and keep the best result.

Chapter 6

Other Measurements

In this chapter we take several other measurements of partitioned register-set machines. We hope they may provide some insights for architectural decisions. Section 6.1 examines the benefit of adding more clusters. Section 6.2 explores the use dedicated functional units for inter-cluster copies.

6.1 Machines with More Clusters

The scalability of partitioned register set machines will depend on many factors, including the amount of ILP that compilers can expose in real applications and the ability of the compiler to use all the additional resources provided by more clusters. The compiler issues will not dictate the answer to this question, but can certainly provide some input.

To assess the benefit of adding clusters, we repeated our experiment for a series of fully connected machines with one, two, four, and eight clusters. Each cluster had an unlimited register set. The compiler used TDC to perform assign and schedule. The resulting cycle counts are shown in Table 6.1. The improvement from four clusters to eight clusters is very small. This may be a fundamental property of the benchmarks. Alternatively, it may be a limitation of the ILP-exposing capabilities of our compiler. Adding simple techniques such as loop-unrolling might improve this situation.

Going beyond eight clusters may add significant complexity to the assign and schedule problem. Our work has assumed a complete set of inter-cluster data paths. The cost of providing these connections rises with the number of clusters. At some point, only schemes that provide partial connectivity will be cost-effective. This will

	1 Cluster	2 Cluster	4 Cluster	8 Cluster
doduc	10,802,426	9,602,784	9,252,947	9,180,027
fft	16,054,835	14,995,539	14,744,885	14,671,925
fpppp	144,909,760	130,909,008	127,103,552	126,994,624
nsieve	3,517,271,808	3,248,780,032	3,248,780,032	3,248,780,032
rkf45	370,306	358,432	354,939	354,939
svd	9,283	8,549	8,448	8,396
tomcatv	353,263,040	331,964,192	329,315,808	326,694,144

Table 6.1 : Cycle counts for machines with different number of clusters

require the coordination of multiple copies on multiple clusters to move off-cluster operands into position. The problem begins to resemble the more general problem of scheduling and assignment for large parallel processors; planning all data movement in the same level of detail that TDC does becomes impractical.

6.2 Dedicated Copy Units

In Chapter 5, the experimental data shows that TDC with 2 reserved copy registers can provide much of the performance of a fully-connected, two-cluster machine. There remains, however, an execution time penalty from the insertion of inter-cluster copies and from the need to wait for operands when transfers cannot be scheduled in a timely fashion. Adding a functional unit that is dedicated to inter-cluster copies into each cluster may improve overall performance. This should eliminate idle cycles on the inter-cluster data paths that arise from lack of a functional unit that can execute the copy operation.

To assess the viability of this idea, we tested three distinct configurations of a two cluster, 64 register per cluster machine. The first has no copy units and a partitioned register set. The second has one dedicated copy unit per cluster and a partitioned register set. The final configuration is the fully connected machine, which needs no

	no CUs	1 CU/cluster	Fully Conn.	Penalty Cycles	Saved Cycles
doduc	18,094,468	17,993,932	17,861,700	232,768	100,536
fft	16,948,956	16,852,372	16,795,664	153,292	96,584
fpppp	271,222,656	270,466,464	267,858,768	3,363,888	756,192
nsieve	3,747,920,128	3,747,920,128	3,667,608,064	80,312,064	0
rkf45	426,930	426,928	424,060	2,870	2
svd	10,027	9,930	9,846	181	97
tomcatv	457,958,304	457,813,920	453,823,232	4,135,072	144,402

Table 6.2 : Improved execution time by two cluster machine with dedicated copy units

copy units. Two registers in each cluster are reserved for copies. The scheme from Section 5.4 is used for allocation, cluster assignment and scheduling. Table 6.2 shows the results. Dedicated copy units help on most of the applications. The fifth column, labeled Penalty Cycles shows the difference between the configuration with no copy units and the fully connected machine. The last column shows the savings (in cycles) from using the dedicated copy units. The dedicated copy units reduce the penalty for partitioning on five of the seven benchmarks.

Chapter 7

Contributions

This thesis makes following contributions:

1. In light of recent interest in code size from both industry and compiler communities, we evaluated five instruction scheduling algorithms with different scopes by speed and size trade-off. Among them, trace scheduling gives the best result: It produces scheduled code significantly faster than that by basic block scheduling or extended basic block scheduling, while the growth of code size is moderate.

2. We developed two new algorithms which not only perform cluster assignment and scheduling together but also revisit already scheduled cycles to insert inter-cluster data transfer. They provide better results than previous algorithms in our experiments.

3. Unlike previous works which stopped at unrealistic machine models with unlimited number of registers, we studied the difficulties caused by relatively small register sets. One of our algorithms, TDC, is used to explore the interaction between register allocation and cluster assignment and scheduling. Our best results came from dedicating a small number of registers to inter-cluster copies and allowing the scheduler to scavenge other unused registers.

4. Finally, we made several other measurements on partitioned register-set machines, hoping to shed light on some of the architectural decisions. To asses the benefit of adding more clusters, we tested machine models with different number of clusters. We also explored the use of dedicated functional units for inter-cluster copies.

Bibliography

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

[2] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *ACM SIGPLAN Notices*, 24(7):258–263, July 1989. Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation.

[3] Preston Briggs. The massively scalar compiler project. Available on the web, 1994.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[5] Andrea Capitanio, Nikil Dutt, and Alexander Nicolau. Design considerations for limited connectivity vliw architectures. Technical Report TR59-92, Department of Information and Computer Science, University of California, Irvine, 1993.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[7] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papwoth, and Paul K. Rodman. A vliw architecture for a trace scheduling compiler. *IEEE Trans. on Computers*, 7:967, 1988.

[8] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded risc processors. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, 1999.

[9] Keith D. Cooper and Philip J. Schielke. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999.

[10] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. textbook to be published, 2000.

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

[12] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *Mathematics of Computation*, 49(180):427–444, 1987.

[13] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[14] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *ACM SIGPLAN Notices*, 21(7):11–16, July 1986. Proceedings of the ACM SIGPLAN 86 Conference on Programming Language Design and Implementation.

[15] Texas Instruments Inc. *TMS320C6700 CPU and Instruction Set, Reference Guide*. Texas Instruments, 1999.

[16] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, pages 261–294, September 1980.

[17] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7:51–142, 1993.

[18] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Laver. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993.

[19] Alexander Nicolau. A fine-grain parallelizing compiler. Technical Report TR-86-792, Department of Computer Science, Cornell University, 1986.

[20] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

[21] Philip J. Schielke. *Stochastic Instruction Scheduling.* PhD thesis, Rice, 1999.

[22] Jeffrey D. Ullman. *Complexity of Sequencing Problems.* John Wiley and Sons, New York, 1976.