Abstract

HAVANKI, WILLIAM ANDREW, JR. Treegion Scheduling for VLIW Processors. (Under the direction of Dr. Thomas M. Conte.)

The instruction scheduling phase of compilation is an important determinant of VLIW program performance. One scheduling framework divides a program into *regions* of code that tend to execute together, and then constructs schedules for each region. Several regions suggested in the past, such as traces and superblocks, are linear, and instruction scheduling for them works only on a single path of execution. A new tree-shaped region, called a *treegion*, encompasses a decision tree section of code. *Treegion scheduling* therefore produces schedules where several independent execution paths are processed together. Treegion formation is a profile-independent process, making treegions relevant regardless of varying program behavior. Various treegion scheduling heuristics, some of which utilize profile information, are examined and compared with scheduling for basic blocks and for "simple linear regions". Next, tail duplication is applied to the treegion formation process in order to expand the average treegion size and expose more instruction level parallelism. Treegion scheduling using tail duplication is shown to lead to better performance than superblock scheduling. By removing redundant ops from the schedule using the properties of dominator parallelism, this performance margin increases further. Experiments show

that for typical integer and floating-point code, treegion scheduling is an effective means of producing high-performance schedules for VLIW processors.

TREEGION SCHEDULING FOR VLIW PROCESSORS

by

WILLIAM ANDREW HAVANKI, JR.

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the requirements for
the Degree of Master of Science

COMPUTER ENGINEERING

Raleigh

1997

APPROVED BY:

Chair of Advisory Committee	

BIOGRAPHY

William Andrew Havanki, Jr., was born during a thunderstorm on the night of August 31, 1973, in Edison, New Jersey. He lived the first three years of his life in Colonia, New Jersey, until his family moved to the lovely seaside town of Point Pleasant on the Jersey Shore. He grew up there, happy to be a clam-digger and regular beach-goer. Finally, in 1991, he graduated from Point Pleasant Borough High School as valedictorian. He attended the College of Engineering at Rutgers University in Piscataway, New Jersey, as a Presidential Scholar, and received his B.S. degree in Computer Engineering in 1995, graduating *summa cum laude*[1]. He began attending North Carolina State University in Raleigh, North Carolina, in the fall of 1995 and has been there ever since. He joined the Tinker Microprocessor Design Laboratory in the spring of 1996.

[1] He missed a perfect 4.0 average by 0.003 points and is still sore about it.

ACKNOWLEDGEMENTS

Bill would like to thank the following people for their contribution, however indirect, to this thesis and the rest of my life.

My parents, for having me (and keeping me) and raising me as the wonderful son that I am; Kathy Bauerdorf, the love of my life, for sticking by me, putting up with me, and molding me into the great guy that I am; Brian, my younger brother, for bestowing upon me more coolness than I would otherwise have; All of my teachers from elementary, middle, and high school, who built my brain into the precision thinking machine that it is; All of my professors and TAs (well, most of the TAs anyway), who filled my brain with the copious knowledge that it has; My friends, both from N.J. and from N.C., who like me despite my obvious geekitude that I possess; Tom Conte, my advisor, for taking me in and continuing to craft me into the acute researcher I am yet to become; Sanjeev Banerjia, Sergei Larin, Kishore Menezes, Sumedh Sathaye, Matt Jennings, and the rest of the Tinker group who helped an idiot like me become the knowledgeable researcher that I am.

And the following, in no particular order:

Mrs. Calabrese, Mrs. Miller (Calabrese in first, Miller in fourth), Ms. Sharpe, Mrs. Schell, Mr. Brown, Mrs. Brennan, Mrs. Churchill, Mrs. Millar, Ms. Youngster, Mr. Kelly, Mr. Berardi, Mrs. Fedak, Mr. Billas, Mr. Ellicks, Jay, Paul, Putz, Mike

Ritchey, Russ, Arnold, Maynard Ferguson, Primus, John Williams, George Lucas, J. Michael Straczynski, Kenner, Hasbro, Mattel, Simtex, Sid Meier, Brue Video, McAfee, Jenkinson's, Master Lee, and 7-11. If I missed anyone, I'll get you next time.

Lastly, my work has been sponsored primarily by the National Science Foundation through a Graduate Research Traineeship in Computational Science and Engineering, grant GER-9454175. Since that fellowship feeds and houses me, they deserve particular acknowledgement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Instruction scheduling is an important step in determining a program's execution performance. The goal of many compiler phases is to allow the scheduler to produce schedules that execute faster; many common optimization passes, for example, are geared to help the scheduler produce efficient and fast schedules.

The opportunity for producing these high-quality schedules is very great in parallel processors such as very long instruction word (VLIW) and superscalar machines. These processors provide several separate computational units, or *functional units*, so that two or more independent instructions have the chance to execute simultaneously. The execution of several instructions per cycle is termed instruction-level parallelism, or ILP. The process of deciding which instructions are executed when, and to which functional units they are sent, is the complex job of the scheduler.

A significant portion of past work has centered on the concept of *regions* [1] as an overall framework for producing schedules. The process of *region formation* divides

a program into several regions, and each region is then scheduled individually. This framework has the beneficial qualities of a divide-and-conquer solution, and allows for many variations that can provide different focuses to the scheduling problem.

Several regions have been presented in previous work which relate to the new region described herein. Traces [2] are regions that encompass linear paths of execution in a program. Trace scheduling has been shown to be an effective way to produce efficient microcode and VLIW schedules. Superblocks [3] are regions which also encompass linear paths, but do not include entrances into them except at the beginning[1]; this constraint simplifies the work the scheduler must do. Hyperblocks [4] encompass judiciously chosen sections of a program, including several paths of execution; hyperblock scheduling utilizes *if-conversion*, the process of converting branching code into predicated straight-line code, to merge these paths into a linear schedule.

The topic of this thesis is a new kind of region. It is a section of a program that encompasses a decision tree [5], [6], and the region's name is a *treegion*. By its definition it encompasses multiple paths like a hyperblock. *Treegion scheduling* uses the concept of *speculative execution* to produce highly parallel schedules; in this way it resembles both hyperblock and superblock scheduling.

Treegions have several advantages. Profile information is not needed in order to form treegions, so the choices made during the formation process are relevant for any program behavior. Profile information can be used instead during treegion scheduling

[1] Entrances into the middle of a linear path are commonly called *side entrances*.

to prioritize some paths over others. Treegion scheduling allows the needs of multiple paths to be compared so that an efficient yet fair schedule can be produced easily.

The remainder of this thesis describes treegion formation, treegion scheduling, and various heuristics and enhancements that can be applied to these processes. Chapter 2 provides detailed motivation for treegions, an in-depth description of treegion formation, and an introduction to treegion scheduling. Chapter 3 compares various heuristics that can be applied to treegion scheduling. Chapter 4 presents an enhancement to the treegion formation process and shows its effects on schedule quality. Chapter 5 applies treegion scheduling in all its forms to floating-point benchmarks (see Section 1.2) and provides the results. Finally, Chapter 6 reviews previous work related to treegions and how they compare, and describes opportunities for further research in treegion scheduling.

1.2 Experimental Framework

All experiments presented in this thesis were performed using the LEGO compiler, an experimental compiler built by the TINKER microprocessor group at North Carolina State University. The compiler uses the Rebel textual code specification as its basis, and works with the TINKER instruction set, a superset of the instruction set used in the PlayDoh architecture [7].

All performance results are given in useful instructions completed per cycle (IPC), and are derived through static estimation. In brief, the lengths of schedules produced

by LEGO were multiplied with the profile weights of the code scheduled to yield estimates for the total cycles spent in each schedule. These totals were summed to produce the execution time for each benchmark. The effects of cache performance and branch prediction were ignored when calculating estimates.

All experiments were performed on the programs of the 1995 SPEC benchmark suite (SPEC95) after traditional optimizations and profiling were performed. The experiments in Chapters 2, 3, and 4 were performed on the eight integer benchmarks of the suite, known as SPECint95, and the experiments in Chapter 5 used the six floating-point benchmarks, SPECfp95.

Two VLIW machine models were used for each experiment. The first, called "4U", is a four-issue VLIW machine model with universal functional units. The second, called "8U", is an eight-issue version of the first. All instructions have a latency of one cycle except for loads, floating-point multiplication, and floating-point division, which have latencies of two, three, and nine, respectively. Both machines also have full bypass.

Chapter 2

Treegion Basics

This chapter first motivates the study of treegions. Then treegions are described using several examples, followed by an in-depth description of the treegion formation process. Treegion scheduling using the exit count heuristic is introduced, and finally, the results of applying treegion formation and scheduling to various SPECint95 benchmarks are presented.

2.1 Motivation for Treegions

A *region* has been defined in varying ways by the literature [8], [1]. Here I will define a region as a collection of operations (*ops*) or other regions which are related in their execution. This relationship may be that they execute in some order, such as traces [2], or that they encompass several execution paths, such as hyperblocks [4].

Regions are usually formed based on a program's *control flow graph*, or CFG. A CFG is a graph that represents the permitted execution patterns of a function or procedure in a program, or even a section of one.

The nodes in a CFG used in this study are *basic blocks* [8]. A basic block is a

group of operations that execute in order from the top of the block to the bottom, where there are no execution paths into the basic block except to the first operation, and no execution paths out of the basic block except from the last operation. A basic block can be considered a fundamental region.

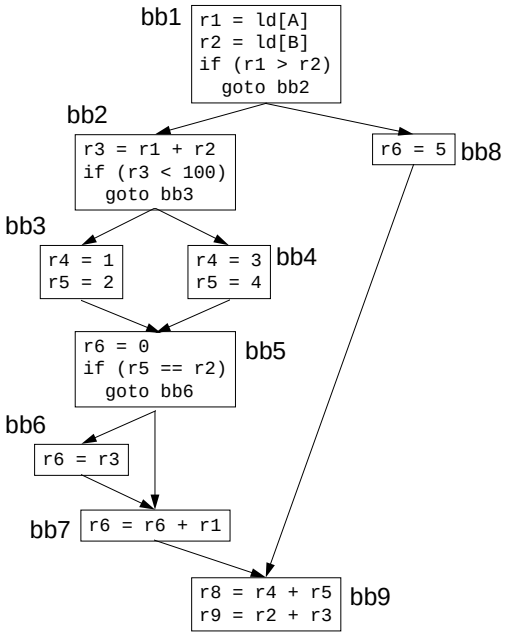


Figure 2.1: An example CFG. Nodes are basic blocks, and are labeled bb1 through bb9.

The edges in a CFG represent the directions that execution can flow between nodes. Together with the nodes the edges complete the picture of the execution behavior of a program, showing all possible threads through it. Figure 2.1 shows an

example of a CFG whose nodes are basic blocks.

An instruction scheduler will often create regions within a CFG as its first step. If the nodes in the CFG are basic blocks, these basic blocks are grouped together in ways that are amenable to producing high-performance schedules. This is done because basic block scheduling typically yields poor-quality schedules [2].

Often, region formation uses *profile* information [9] as a guide. Profile information is data that describes the "typical" or "normal" pattern of execution of a program. Often it is generated by running a program with "typical" inputs, observing and recording which sections of the program are run, and then attaching that information to the nodes and edges of the program's CFGs [9]. Superblock formation [3] and heuristics on trace formation [2], [10] are examples of region formation strategies that utilize profile information.

However, this may not be a foolproof strategy. During the lifetime of a program, the execution patterns may vary from what was considered normal when regions were originally formed (a *profile drift*). Although there are ways to adapt to profile drift [11], without them the schedules that are used may actually hurt program performance [12], [13].

Another pitfall of depending on profile information for region formation is manifested when the choice of which execution paths to include in a region is not clear, that is, when each path is followed about the same number of times. Studies in hardware-based profiling [14], [13] revealed that in such a situation, accidental selec-

tion of the less-executed path sometimes leads to higher performance overall since the chosen path contains more ILP. Selection of both paths for inclusion in a region would allow any ILP present in these cases to be exploited.

Other problems may arise that relate to *merge points* in the CFG. A merge point is defined as a node in the CFG which has more than one incoming edge. In Figure 2.1, nodes 5, 7, and 9 are merge points.

One problem due to merge points arises when the scheduler attempts to move operations upward in the schedule from their original location, often by using *speculation* [15]. If the scheduler desires to move an operation above a merge point, the operation needs to be duplicated so that one copy is moved up each incoming path. For example, trace scheduling [2] generates bookkeeping code to handle these contingencies. This adds to the complexity of the compiler.

Another problem due to merge points occurs with dynamic compilation and recompilation techniques [11], [16]. These techniques work with a restricted portion of a program at any given time. Since they have only local knowledge of the program CFG, they cannot identify merge points outside the current scope. The complexity of these techniques can be reduced if the regions scheduled in the program did not contain any merge points.

So, it is desirable to define a region that does not require reliable profile information during its formation, contains several execution paths, and also contains no merge points. Such a region is the topic of this thesis, and is described in the next

section.

2.2 Treegions

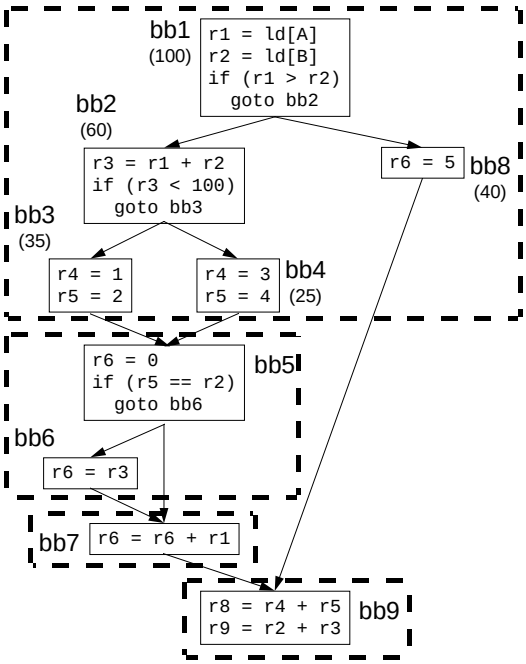


Figure 2.2: An example CFG with treegions. Treegions are bounded with dotted lines. The basic blocks in the topmost treegion are noted with possible profile weights.

A *treegion* is a tree-shaped subgraph of a CFG. Because a treegion is a tree, treegion formation, described later in this section, can be performed without relying on profile information. A treegion cannot contain merge points except at the treegion's root; so, the treegion scheduler is free to move ops upward in the schedule without coping

with the complexity of merge points. Figure 2.2 shows the CFG of Figure 2.1 after treegions have been formed.

2.2.1 Treegion Formation

```
 1 treeform (CFG)
 2 {
 3   Add top node(s) of CFG to unprocessed queue
 4   while unprocessed queue is not empty {
 5     Get first node in unprocessed list
 6     if node is already in a treegion continue
 7
 8     Make a new empty treegion
 9     absorb-into-tree (treegion, node)
10
11     for each sapling of current treegion
12       if sapling is not in a treegion
13         add sapling node to unprocessed queue
14   }
15 }

16 absorb-into-tree (treegion, node)
17 {
18   Add node to candidate queue
19   while candidate queue is not empty {
20     Get first node in candidate queue
21     if node is already in treegion continue
22     if node is a merge point and not the root continue
23
24     Move node into treegion
25     Add each successor of node to (front of) candidate queue
26   }
27 }
```

Figure 2.3: Treegion formation algorithm. The first algorithm, `treeform`, builds treegions over an entire CFG. The second, `absorb-into-tree`, adds nodes starting from the one given into the treegion.

Treegion formation is based solely on the CFG topology, and can be performed

so that every node in a CFG is contained in a treegion. Figure 2.3 gives pseudo-code for the algorithm used in treegion formation.

The algorithm keeps a queue of "unprocessed" nodes which do not belong to any treegions. The queue is initialized to contain all the entry nodes of the CFG (line 3). Treegion formation terminates when there are no remaining nodes to consider.

Each node taken off the unprocessed queue becomes the root of a new treegion unless it has been placed into a treegion already. This root node initializes a second queue of "candidate" nodes, which holds nodes that could possibly be placed into the new treegion. The new treegion is completed when there are no remaining nodes in the candidate queue.

Each node in the candidate queue is considered in turn for inclusion in the new treegion (lines 20 – 24). If the node is already in a treegion, it is not included. If the node is a merge point, and not the root node, it is also not included. Otherwise, the node is added to the treegion. All the successor nodes of the candidate node are then added to the candidate queue. In this way, all nodes up to merge points and nodes in other treegions are absorbed into the new treegion.

Candidate nodes which fail to be integrated into a treegion are called *saplings* of the treegion. After a treegion is formed, all its saplings which are not already in treegions are added to the unprocessed queue (lines 11 – 13). These saplings may then become the roots of new treegions, hence the name.

As an example, the treegion formation process for the CFG in Figure 2.2 is pre-

Table 2.1: An example treegion formation. UQ and CQ show the contents of the unprocessed queue and candidate queue, respectively, with the heads of the queues to the left. Node shows the node currently being considered, and Result shows what happens to the node.

UQ	CQ	Node	Result
1			
	1		spawn new treegion
	2 8	1	include
	3 4 8	2	include
	5 4 8	3	include
	4 8	5	skip (merge)
5	5 8	4	include
5	8	5	skip (merge)
5	9	8	include
5		9	skip (merge)
9	5		spawn new treegion
9	6 7	5	include
9	7	6	include
9		7	skip (merge)
7	9		spawn new treegion
7		9	include
	7		spawn new treegion
	9	7	include
		9	skip (in tree)

sented in tabular form in Table 2.1. Note that each node is placed into exactly one treegion.

2.2.2 Treegions in SPECint95

Experiments were run to characterize the SPECint95 benchmarks with respect to treegions. The statistics resulting from these experiments are presented in Table 2.2.

Treegions typically contain two to four basic blocks in this group of benchmarks.

Table 2.2: Treegion statistics. The data described below, from left to right, are total treegion count, average basic block count per treegion, maximum basic block count in a treegion, and average number of ops per treegion.

benchmark	**# treegions**	**avg # bb**	**max # bb**	**avg # ops**
compress	130	2.430769	8	17.638462
gcc	22965	2.854235	384	21.540126
go	4552	2.754394	89	20.959798
ijpeg	2190	2.391324	69	20.879909
li	1394	2.568149	44	18.292683
m88ksim	2071	3.382424	146	25.681796
perl	5053	3.140823	774	23.459133
vortex	1815	3.30303	39	33.539394

Some of the benchmarks contain extremely large treegions, most notably **perl** (774 basic blocks) and **gcc** (384 basic blocks); these benchmarks also contain additional treegions with hundreds of basic blocks each. Overall, the last three benchmarks in the table (**m88ksim**, **perl**, and **vortex**) tend to have larger treegions.

The process of forming treegions is not difficult to perform, and has the benefits of not altering the CFG and encompassing multiple paths of execution. It is a natural way to divide a CFG. The next section describes the advantages treegions have to offer to the scheduling process.

2.3 Treegion Scheduling

Many types of regions contain a single *maximal execution path*; that is, one path extends from the top of the region to the bottom, and all ops in the region are in that

path. Traces [2] and superblocks [3] fall into this category[1]. The schedule formed from regions of this kind contain the maximal path and no parts of other paths that may diverge from it. It is likely that the schedule derived from such a region does not fully use available resources in some cycle, which could be used for other paths, thus obtaining higher levels of ILP.

In contrast, a treegion includes multiple *independent* paths. The treegion scheduler has a bigger picture of program behavior and can therefore make greater use of available resources by scheduling instructions from several independent paths in the same cycle(s). The schedule can then be improved for the less-executed paths, and overall execution time will be reduced. An example of treegion scheduling appears in Section 2.3.2.

2.3.1 Treegion Scheduling Algorithm

```
1 scheduleTreegion (treegion)
2 {
3   Form DDG for treegion
4   sortDDGNodesBy*** (DDG)
5   listSchedule (DDG)
6 }
```

Figure 2.4: Treegion scheduling algorithm. The sort function can be replaced with various sorting schemes, each representing a different heuristic.

[1]Both regions may contain multiple paths, but they all lie along a single maximal path.

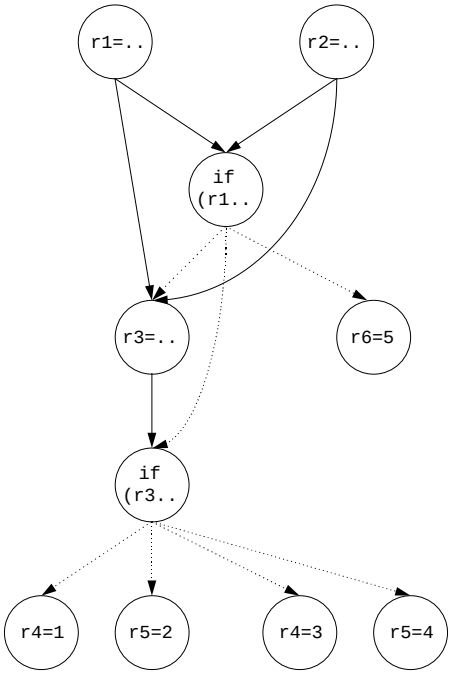


Figure 2.5: An example DDG. Solid arrows denote flow dependences, while dotted arrows denote branch dependences.

The basic treegion scheduling algorithm is shown in Figure 2.4. Fundamentally, treegion scheduling is a three-step process. In the first step, a *data dependence graph* (DDG) for the operations in the treegion is formed; the DDG shows dependences between all operations in the treegion, not just between operations in the same basic block. In the DDG, each node represents an op, and edges represent data dependences (flow, anti-, and output dependences) and branch dependences, which connect ops to the branches before and after them in control flow. Figure 2.5 shows the DDG for the topmost treegion in Figure 2.2.

The second step is the ordering of the DDG nodes. Here a heuristic is applied

that dictates by what characteristics the nodes should be sorted. It is in this step that the quality of the final treegion schedule is determined.

In the final step the resulting list of nodes is given to a list scheduler which constructs the schedule based on the priority of each node. Nodes toward the front of the list have highest priority and the scheduler tries to handle them first.

A cycle of a VLIW schedule consists of several *slots*, each of which corresponds to a functional unit in the machine. Each cycle of the schedule, starting with the first, is filled with ops that are ready to execute, or *data-ready*. An op is not *data-ready* if an op upon which it is dependent has not completed. (This term will also be used to describe an op that is waiting on a branch dependence.) A cycle is finished when no more ops are data-ready or machine resources are used up for the cycle, at which time the scheduler moves to the next cycle.

In this step, ops that lie in separate paths may be scheduled in the same cycle or in several successive cycles. It may happen that these ops interfere with each other by defining and/or using the same data; this would often be true for if-then-else structures which set some register to different values depending on the condition of the branch involved. If this situation occurs, *register renaming* [17] can be used to avoid this "colliding" of paths by renaming each path's data uses to some available register. If no registers are available, additional copy instructions can be inserted into available schedule slots in order to free up registers.

The treegion scheduler uses *speculation* to fill as many slots in each cycle as pos-

sible. Ordinarily some slots in a cycle may be left empty because no ops in the DDG node list are data-ready. Speculative execution allows ops that are waiting on branch dependences only to be placed into these empty slots. If a speculated op's branch is actually taken, that is, if execution actually would lead to the op being executed, then the work has already been done, and the schedule is more efficient. If not, then the results of the op are discarded or ignored. The treegion scheduler uses the *general speculation* model, which allows ops that may cause exceptions, such as loads and divisions, to be speculated as well [18].

It is possible that speculating an op above a branch may cause what will be termed a *live-out violation*. An op usually has a destination, which it is said to *define*. Later, this destination may be *used* as a source for another op. A destination value is *live* from its definition until its last use. This live range may extend in any direction that a branch takes; if it does, the destination is *live-out* along each path from the branch that leads to a use. If an op that defines some destination is moved above a branch where that destination is live-out on another path, then the wrong value will be in the destination if the live-out path is taken.

There are various solutions to this problem. The scheduler can refuse to speculate such an op at all. This solution is less complex but is likely to limit speculation. An alternative is to speculate the op and place appropriate copy operations along the live-out paths. Instead, the treegion scheduler once again uses register renaming. Here, to avoid a live-out violation, the destination (register) of the speculated op is

renamed to some other register, and its use(s) on the path from which it is speculated is (are) also renamed. In this way, the true value is preserved on the live-out path as well.

2.3.2 An Example of Treegion Scheduling

Table 2.3: An example superblock schedule. Ops in italics are speculated above branches upon which they are dependent.

Unit 0	Unit 1	Unit 2	Unit 3
r1 = LD (A)	r2 = LD (B)	b8 = PBR (bb8)	*b4 = PBR (bb4)*
p1 = CMPP (r1>r2)	*r3 = r1 + r2*	*b5 = PBR (bb5)*	*r4 = 1*
BRCF (b8, p1)	*p3 = CMPP (r3<100)*		
BRCF (b4, p3)	*r5 = 2*		
BRU (b5)			
bb4: r4 = 3	r5 = 4	b5 = PBR (bb5)	
BRU (b5)			
bb8: r6 = 5	b9 = PBR (bb9)		
BRU (b9)			
execution time = 35(5) + 25(6) + 40(5) = 525			

An example shows how treegion scheduling can create a better schedule than superblock scheduling. Both schedules use facilities present in the PlayDoh and TINKER architectures and often in other VLIW architectures, namely speculation, predication, and a full-featured compare-to-predicate (CMPP) operation. Registers beginning with "r" are general-purpose integer registers, those beginning with "p" are

Table 2.4: An example treegion schedule. Ops in italics are speculated above branches upon which they are dependent. Ops in shaded blocks show the effect of register renaming.

Unit 0	Unit 1	Unit 2	Unit 3
r1 = LD (A)	r2 = LD (B)	b2 = PBR (bb2)	*b3 = PBR (bb3)*
p1,p2=CMPP(r1>r2)	*r3 = r1 + r2*	*b5 = PBR (bb5)*	*b6 = PBR (bb5)*
BRCT (b2, p1)	*p3, p4 = CMPP (r3<100) ? p1*	*b9 = PBR (bb9)*	*r4 = 1*
bb2: BRCT (b3, p3)	*r5 = 2*	*r4a = 3*	*r5a = 4*
bb3: r6 = 5	BRCT (b5, p3)	BRCT (b6, p4)	BRCT (b9, p2)
execution time = 35(5) + 25(5) + 40(5) = 500			

predicate registers, and those beginning with "b" are branch target registers (BTRs). The CMPP instruction performs a comparison and places the Boolean result into a predicate register; the CMPP may have a second destination predicate, which is set to the complement of the result. The PBR instruction initializes a BTR, which is used to specify the target of a branch. The branch instructions used here are BRCT for branch if predicate is true, BRCF for branch if predicate is false, and BRU for unconditional branch. Ops may be predicated, denoted by a question mark followed by the predicate controlling an op's writeback.

Table 2.3 shows a possible schedule using superblocks for the topmost treegion of Figure 2.2. The schedule is for a four-issue universal functional unit machine with perfect caches, perfect branch prediction and unit instruction latency. The superblock

schedule is the top section of the schedule. The middle section is the schedule for bb4, and the bottom section is the schedule for bb8. The execution time for this code, given the profile counts in Figure 2.2, is estimated at 525 cycles. Note the unused slots in the superblock schedule.

Table 2.4 shows a possible treegion schedule for the same section of code. Several details of the schedule are worth noting. First, all available resources are being used in this schedule by the aggressive speculation of data-ready ops that were not visible to the superblock scheduler. Next, the last two ops in the fourth cycle, which belong to the weight-25 path through the treegion, have used register renaming to avoid conflicting with ops along the weight-35 path; this is necessary since both paths are being executed at once.

Another detail is that predication has been used so that the schedule executes correctly for any treegion path. The CMPP operation in the third cycle, for instance, sets its destinations to false if the BRCT before it is not taken; this deactivates the other branches later in the schedule as desired. In the final cycle, only one of the three branches is taken, depending on the predicate values calculated in the rest of the schedule. This is very similar to techniques used for critical path reduction [19].

Finally, the first operation scheduled in the fifth cycle (which defines r6) is unconditionally executed, although it could be predicated as well. This is possible because other paths in the treegion do not use r6, so executing it will not lead to incorrect

results for any path in the treegion[2].

The estimated execution time for this schedule is 500 cycles, 25 less than the superblock schedule. This is because the schedule length for the weight-25 path was reduced by one cycle due to speculation of its data-ready ops.

2.4 Experimental Results for Ordinary Treegion Scheduling

Treegion scheduling was performed on the benchmarks in the SPECint95 benchmark suite. For each benchmark, treegions were formed using the algorithm from Figure 2.3, and then the treegions were scheduled as described in Section 2.3. The sorting heuristic used was the exit count heuristic, which is described in Section 3.1. The term *ordinary* treegion scheduling shall refer to treegion scheduling using this heuristic.

The same benchmarks were also scheduled using basic blocks only, and also using *simple linear regions* or SLRs. An SLR is similar to a superblock, except that no tail duplication is performed during its formation. It is more fair to compare treegion scheduling to SLR scheduling, since superblocks can be extended through tail duplication. Treegions are compared to superblocks in Chapter 4 after tail duplication is applied to treegion formation.

The algorithm for SLR formation is given in Figure 2.6; it is an altered form of the treegion formation algorithm which creates a region encompassing a single execution path based on profile weight.

[2]It is assumed that r6 is not live-out along the other paths in the treegion. If it is, register renaming can be used here as in speculation to prevent conflicts.

```
 1 SLRform (CFG)
 2 {
 3   Add top node(s) of CFG to unprocessed queue
 4   while unprocessed queue is not empty {
 5     Get first node in unprocessed list
 6     if node is already in a treegion continue
 7
 8     Make a new empty SLR
 9     absorb-into-SLR (SLR, node)
10
11     for each sapling of current SLR
12       if sapling is not in an SLR
13         add sapling node to unprocessed queue
14   }
15 }

16 absorb-into-SLR (SLR, node)
17 {
18   current node =  node
19   repeat {
20     if current node is already in SLR break
21     if current node is a merge point and not the root break
22
23     Move current node into SLR
24     if current node has no successors break
25     current node = current node successor with highest
26       profile weight
27   }
28 }
```

Figure 2.6: SLR formation algorithm. The first algorithm, `SLRform`, builds SLRs over an entire CFG. The second, `absorb-into-SLR`, adds nodes starting from the one given into the SLR.

The results of these experiments are summarized in Figures 2.7 and 2.8. Figure 2.7 shows the IPC results for ordinary treegion scheduling using the 4U machine model, which on average outperformed basic block scheduling by 32% and SLR scheduling by 10%. For six of the eight benchmarks treegion schedules outperformed SLR schedules. However, **ijpeg** and **perl** both had slightly reduced performance with treegion

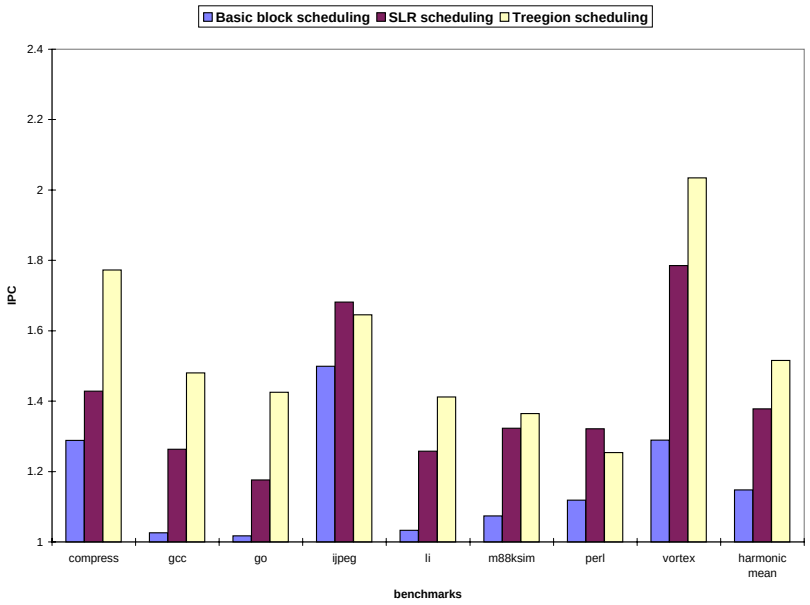


Figure 2.7: Results for ordinary treegion scheduling, 4U.

schedules.

The reason for this is the presence of treegions of significant size where one execution path through a treegion is taken 100% of the time. These kinds of treegions, called *biased treegions*, are characteristic of loop unrolling, and so are not uncommon in optimized code. In the area of a biased treegion, SLRs follow the single executed path alone and schedule it well, while treegion scheduling will intersperse ops from the other paths and extend the schedule for the executed path significantly. Figure 2.9 shows a biased treegion taken from **ijpeg**. SLR scheduling followed the leftmost path and produced a schedule eleven cycles long. Treegion scheduling included ops from the zero-weight blocks, which contain eighteen ops each, and produced a sched-

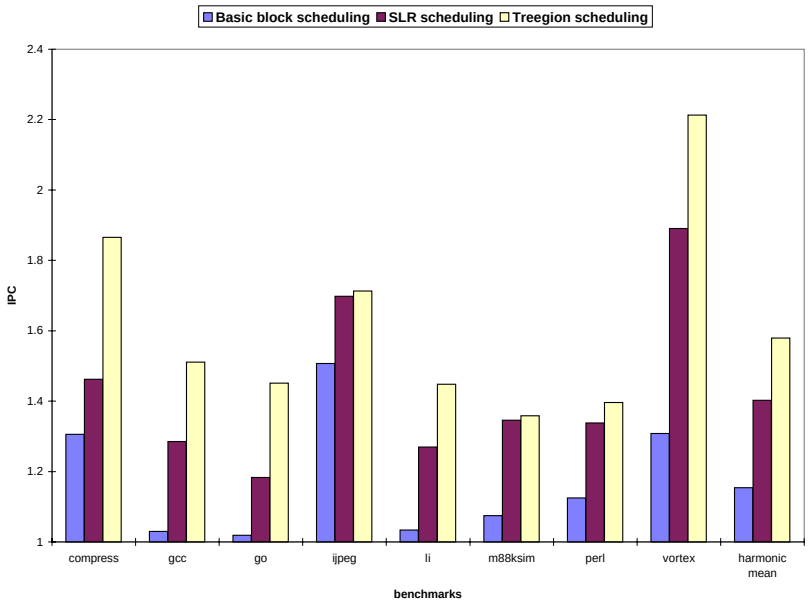


Figure 2.8: Results for ordinary treegion scheduling, 8U.

ule which, for the leftmost path, required 34 cycles of execution. The global weight treegion scheduling heuristic, presented in Section 3.3, begins to address this problem.

Figure 2.8 shows how with a wider machine model (8U) treegion schedules can produce even better results. Here, treegion schedules outperformed SLR schedules for all eight benchmarks, including **ijpeg** and **perl**. On average, treegion scheduling improved 37% over basic block scheduling and 13% over SLR scheduling. The improvement is due to increased opportunity for speculation due to the increased resources of the eight-issue model.

Having defined treegions and described their formation and scheduling, variations in these processes can be investigated. The next chapter covers some possible varia-

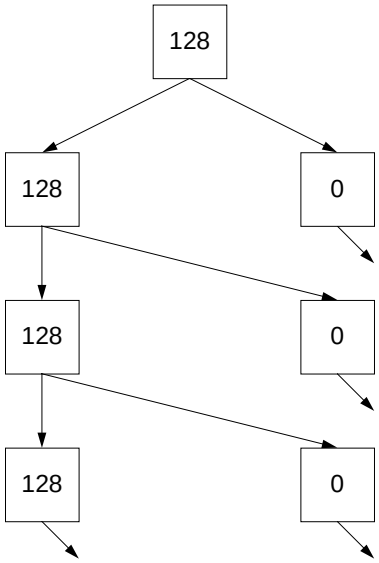


Figure 2.9: A biased treegion. The block numbers indicate profile weight. The leftmost path is the only path executed in the treegion.

tions in treegion scheduling and their impact on performance.

Chapter 3

Treegion Scheduling Variations

The experiments in Chapter 2 used a single heuristic, exit count, as the priority scheme for treegion scheduling. The results indicate that this heuristic does produce higher performance than basic block and SLR scheduling. This chapter gives a formal explanation of the exit count heuristic, and then develops three more heuristics that can be applied to treegion scheduling. The results for these new heuristics are compared with those from the previous chapter.

3.1 Exit Count Treegion Scheduling

The first heuristic to be covered here is the exit count heuristic. This heuristic was used in the experiments in Section 2.4.

The exit count heuristic was inspired by the helped count heuristic used in *speculative hedge* [12], which defines various ways to prioritize ops for scheduling. One method works by figuring how many exits from the encompassing region would have a critical need[1] satisfied if the op were scheduled in the cycle being considered. The

[1] An exit has a critical need for an op to be scheduled in the current cycle if the retirement goal

op is then said to "help" those exits, and the number of exits an op helps is the op's *helped count*. Applied to superblocks, this heuristic favors ops in the top portion of a superblock over those in the bottom portion since the former group helps many or all exits, while the latter group helps less due to side exits above it.

One salient property of this heuristic is that it is independent of profile information. The characteristics of the region DDG are the sole factor determining helped counts. The heuristic gives each exit in the region an equal weighting, and so does not unduly penalize exits taken less often.

The helped count heuristic can be easily modified to be used for scheduling treegions. Here, instead of using the DDG to determine the exits an op helps, as in speculative hedge, the CFG is used, and so the heuristic is renamed the *exit count* heuristic to highlight this distinction. Each op's priority is its *exit count* which, because of a treegion's shape, is equal to the number of exits in the subtree beneath the op. If two ops have the same exit count, the op with the higher dependence height[2] is given priority. If two ops have the same exit count and dependence height, they have equal priority.

Since the treegion scheduler is a list scheduler, the ops in a DDG are sorted into one large list prior to scheduling. The algorithms in Figure 3.1 show how this is done for the exit count heuristic. The first algorithm determines the exit count for a DDG node by scanning each execution path of the treegion and counting how many contain

for the exit cannot be met otherwise.

[2]Dependence height is described in Section 3.2.

```
 1 calculateExitCount (node)
 2 {
 3   node.exit count = 0
 4   for each exit op of treegion that contains node
 5   {
 6     follow path from exit op toward root of treegion
 7     if node op was found
 8       node.exit count = node.exit count + 1
 9   }
10 }

11 sortDDGNodesByExitCount (DDG)
12 {
13   Form DDG nodes into list
14   for each node in list
15     calculateExitCount (node)
16   Sort list by exit count
17   for each sublist of nodes with the same exit count
18     Sort sublist by dependence height
19 }
```

Figure 3.1: Exit count heuristic algorithm. The algorithm `calculateExitCount` determines the exit count for a single DDG node. The second algorithm, `sortDDG-NodesByExitCount`, sorts the nodes in the DDG for the treegion scheduler (second line of the algorithm in Figure 2.4).

the node op. The second algorithm prepares the list for the list scheduler.

The results of treegion scheduling using the exit count heuristic are shown in Figures 2.7 and 2.8. A description of the results was given in Section 2.4. The heuristic does produce better results than basic block or SLR scheduling.

3.2 Dependence Height Treegion Scheduling

The exit count heuristic is a profile-independent way to prioritize ops in a treegion, but it does not allow for as much speculation as may be possible. Ops with lower exit counts that are data-ready do not have as much opportunity to be speculated because the scheduler tends to fill up each cycle with ops with higher exit counts instead. These lower-priority ops exist farther down in the treegion, and so one would want to be able to speculate them to give them a fairer chance at execution.

The dependence height heuristic is one way to give all data-ready ops an equal chance to execute. The priority value given to each op is that op's height in the DDG. The height of an op with no successors in the DDG is one, and the height of a op a with a successor op b is the height of b plus the latency of op a. If an op has several successors, the maximum of the sums of the successors' heights and the op's latency is chosen as its height.

So, all data-ready ops at the same height in the DDG have an equal chance of being scheduled in a given cycle. Note that the exit count heuristic orders ops with the same priority value by dependence height; the heuristic described here takes that idea to its extreme.

The algorithm for sorting the ops in the DDG by dependence height is shown in Figure 3.2.

The results of treegion scheduling using the dependence height heuristic is compared with SLR scheduling and treegion scheduling using the exit count heuristic in

```
1 sortDDGNodesByDependenceHeight (DDG)
2 {
3   Form DDG nodes into list
4   Sort list by dependence height
5 }
```

Figure 3.2: Dependence height heuristic algorithm. This is the simplest sort examined in this study.

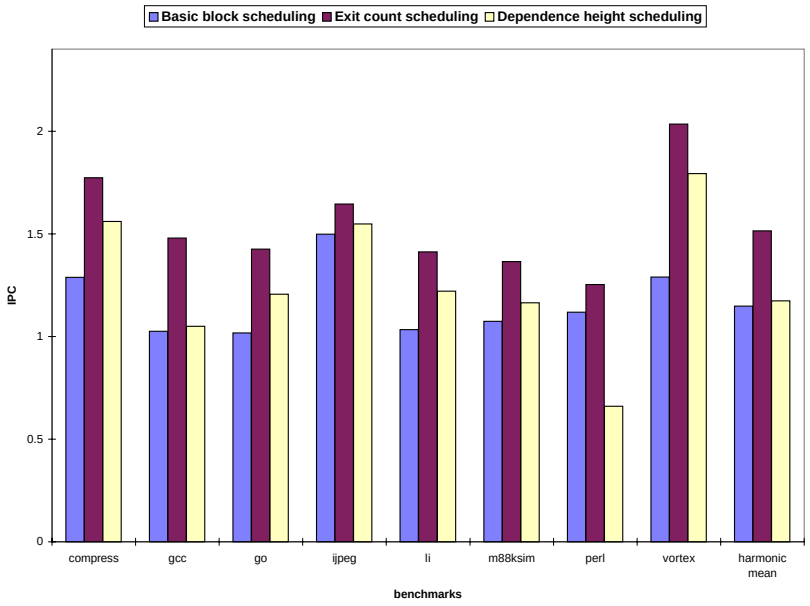


Figure 3.3: Results for dependence height treegion scheduling, 4U.

Figures 3.3 and Figures 3.4[3]. The results clearly show that the dependence height heuristic is a poor choice, despite the justification given previously. Of particular note is that the schedules for **perl** provide worse performance than even basic block scheduling.

[3]Note that the y-axis begins at zero, not one, for these graphs.

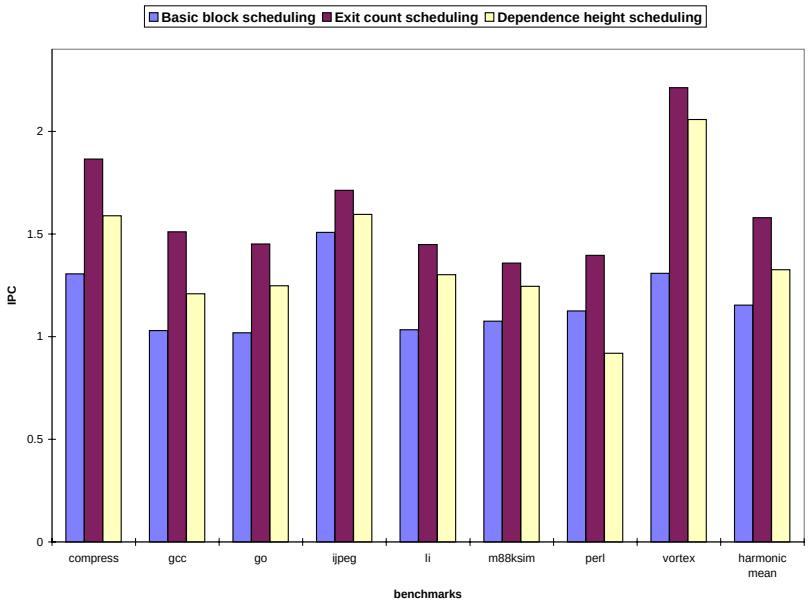


Figure 3.4: Results for dependence height treegion scheduling, 8U.

Some investigation revealed the cause of the poor performance. Many component files of the benchmarks contain large treegions, with tens to hundreds of basic blocks each, which also have rather large profile weights (Table 2.2 mentions a treegion in **perl** with 774 basic blocks.). In these treegions, many ops in the lower blocks of the treegions are data-ready at the outset of treegion scheduling, so many of them are speculated right away. This causes the section of the schedule before the branch out of the root block of the treegion to expand, delaying the first branch. This causes the cycle count for all paths through the treegion to increase, including those paths which are most frequently executed. As a result, the execution time estimate increases significantly.

The exit count heuristic, on the other hand, prevents this catastrophe by giving the ops before the first branch the highest scheduling priority, rather than an equal priority with many other treegion ops. As a result, these ops are scheduled in less time, which benefits all paths.

In conclusion, the dependence height heuristic should not be used for treegion scheduling, since it leads to overaggressive speculation and unnecessary delay in executing even the shortest paths.

3.3 Global Weight Treegion Scheduling

The heuristics described so far for treegion scheduling do not use profile information in any way. The dependence height heuristic only uses the DDG height to assign priorities to ops. The exit count heuristic primarily uses the treegion topology to decide which ops benefit more exits and thus should have higher priority. Both heuristics may give too high a priority to rarely executed ops. An op that is data-ready at the beginning of the scheduling process but is never executed is still given a high priority using the dependence heuristic. An op in a treegion block that is never executed is still given a high priority using the exit count heuristic if it benefits several exits, even though these exits are never taken. The problem with biased treegions described in Section 2.4 is a form of this problem.

The global weight heuristic uses both profile weight and dependence height to assign priorities to ops. The priority value assigned to an op is its profile weight. Ops

with the same profile weight are sorted by dependence height, so ops of a given weight on which more ops depend have priority over those on which less depend. Because a treegion is a tree, the weight of any op in a treegion is equal to the sum of the weights of the exits it benefits. Therefore, this heuristic uses the same information as the *helped weight* heuristic of speculative hedge [12], which was the inspiration for this and for the exit count heuristic.

```
1 sortDDGNodesByGlobalWeight (DDG)
2 {
3   Form DDG nodes into list
4   Sort list by weight
5   for each sublist of nodes with the same weight
6     Sort sublist by dependence height
7 }
```

Figure 3.5: Global weight heuristic algorithm. Rather than sorting the nodes primarily by exit count, this algorithm sorts them by their profile weight.

The algorithm for this heuristic is shown in Figure 3.5. It is the same as the exit count heuristic sorting except the list is sorted by profile weight first instead of exit count.

One would expect this heuristic to perform slightly better than the exit count heuristic because of its utilization of profile information, which implicitly reflects the treegion topology[4]. The results of experiments using this heuristic are compared to

[4]For example, ops near the root of the treegion tend to have higher weights, which implies that

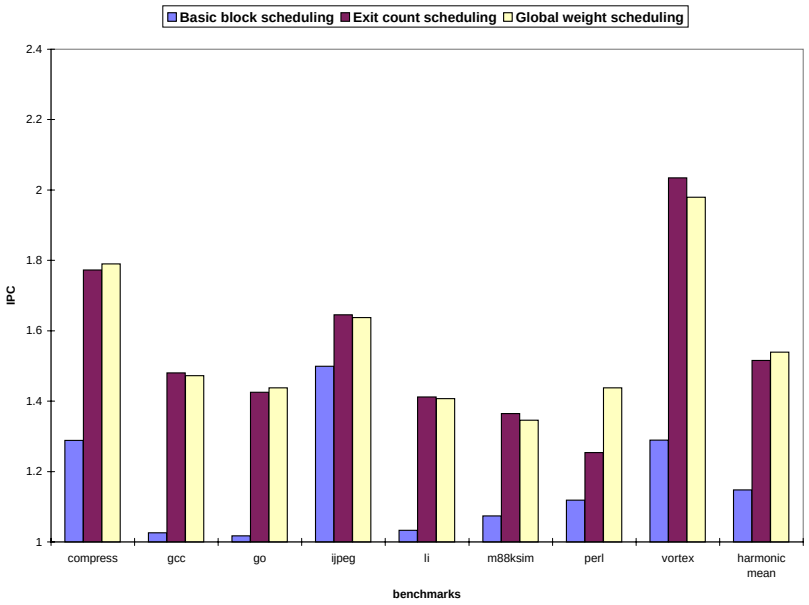


Figure 3.6: Results for global weight treegion scheduling, 4U.

SLR scheduling and exit count treegion scheduling in Figures 3.6 and 3.7.

The results show that the global weight heuristic produces performance about as good as the exit count heuristic. The 4U machine model benefits slightly from using profile weights since that allows the scheduler to focus on the more frequently executed sections of the treegion and use empty slots, which are more scarce in this model, for ops in those sections over those in others.

Surprisingly, this heuristic performs slightly worse for the 8U machine model. The reason for this was found after some detailed analysis of the schedules produced by the two heuristics, and once again it relates to biased treegions. The global

they help more exits.

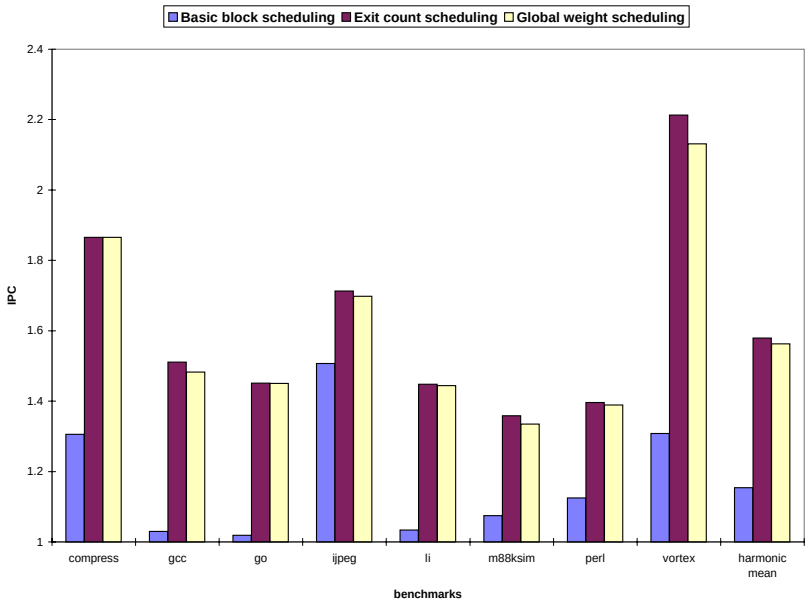


Figure 3.7: Results for global weight treegion scheduling, 8U.

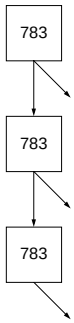


Figure 3.8: A linearized biased treegion. The block numbers indicate profile weight. Since the weights are all equal, the global weight heuristic degrades to the dependence height heuristic.

weight heuristic fails in these cases because the single executed path remains sorted by dependence height only, since the weight of each block in the path is the same. So, just as described in Section 3.2, the scheduler speculates ops from farther down in the path too high and delays the ops toward the top of the path, thus extending the

overall execution time. Figure 3.8 shows an example of a biased treegion taken from the **vortex** benchmark which also happens to be linear, thus showing a case where focusing on profile weight alone is useless. Since the weight information contributes nothing to prioritizing the ops in the treegion, the global weight heuristic does not work. As stated in Section 2.4, biased treegions such as this are often the result of loop unrolling, and so this failing is a significant problem in optimized code.

The next section describes a final heuristic which combines the exit count and global weight heuristics in order to avoid the failings shown above.

3.4 Weighted Count Treegion Scheduling

The weighted count heuristic combines the exit count and global weight heuristics. The motivation for the heuristic is that the good qualities of these heuristics can be used together, so that ops in often-executed paths get a higher priority, while ops which help many exits are also given an edge during scheduling.

This heuristic prioritizes ops primarily by their profile weight. If ops have the same weight, they are prioritized by exit count; this prevents the failing of the global weight heuristic. The algorithm for the weighted count heuristic is shown in Figure 3.9.

Figures 3.10 and 3.11 show the results for this final treegion scheduling heuristic compared to exit count treegion scheduling, the best heuristic so far. Overall, the weighted count heuristic slightly outperforms the exit count heuristic, producing

```
 1 sortDDGNodesByWeightedCount (DDG)
 2 {
 3   Form DDG nodes into list
 4   Sort list by weight
 5   for each node in list
 6     calculateExitCount (node)
 7   for each sublist of nodes with the same weight
 8     Sort sublist by exit count
 9   for each sublist of nodes with the same weight and exit count
10     Sort sublist by dependence height
11 }
```

Figure 3.9: Weighted count heuristic algorithm. This heuristic primarily sorts nodes by weight, then by exit count to prevent overaggressive speculation.

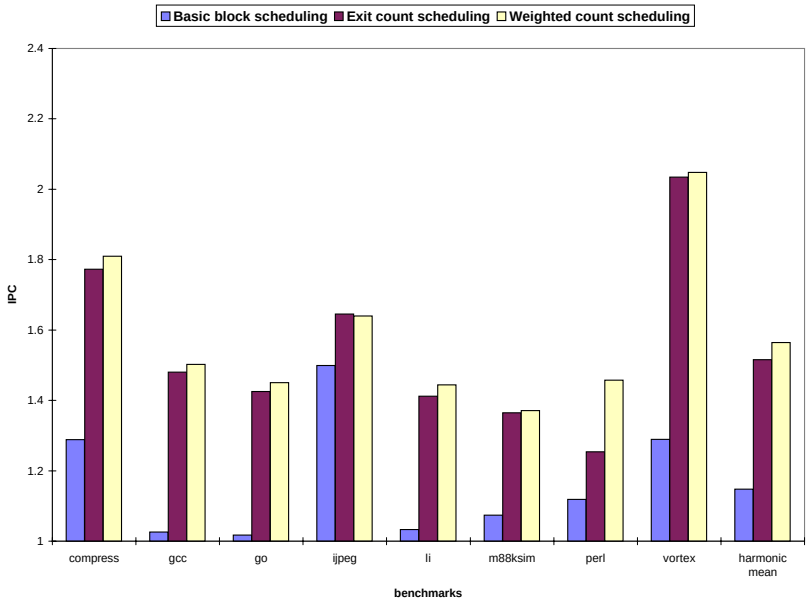


Figure 3.10: Results for weighted count treegion scheduling, 4U.

about 37% better performance than basic block scheduling and 14% better performance than SLR scheduling. The most notable feature of this heuristic is how well

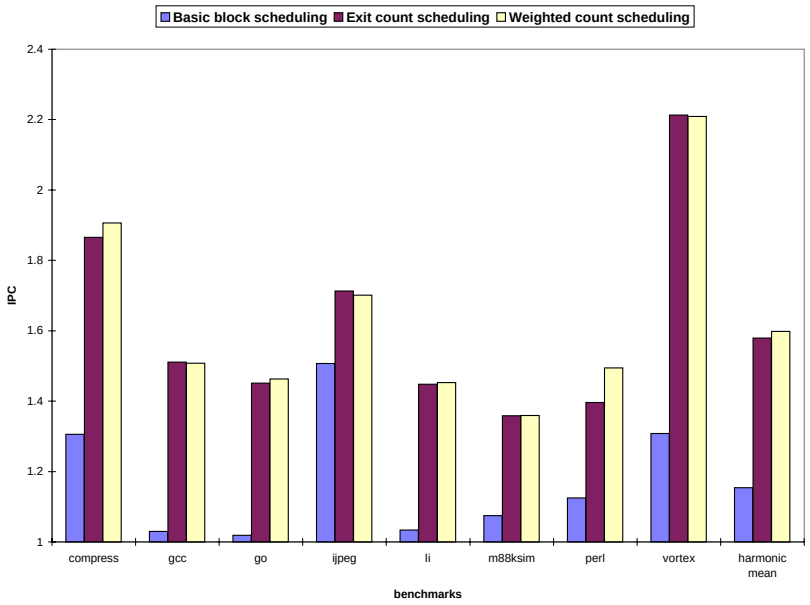


Figure 3.11: Results for weighted count treegion scheduling, 8U.

perl is scheduled compared to the exit count heuristic, particularly for the narrow machine model.

The results for the experiments conducted in this chapter show that treegion scheduling can be performed using either a profile-dependent heuristic, weighted count, or a profile-independent heuristic, exit count. Of these two, the weighted count heuristic produces the best performance overall, and is an effective application of profile information to scheduling. The exit count heuristic is nearly as good, and is useful for cases where profile information is lacking or is known to be inaccurate.

Chapter 4

Treegion Formation with Tail Duplication

Larger treegions would give more opportunity for the treegion scheduler to speculate and more flexibility in the decisions it can make, which in turn would lead to better schedules. Unfortunately the ordinary treegion formation process presented in Section 2.2 does not provide any way to produce larger trees than are naturally present in the CFG; the CFG topology dictates the treegion number and size. The algorithm needs to be modified to meet the goal of producing larger treegions.

The limiting factor that determines treegion size is the presence of merge points. Techniques that try to eliminate merge points will allow treegions to absorb more parts of the CFG during their formation.

This chapter describes one method for expanding treegions: tail duplication, which duplicates basic blocks in order to remove merge points. Results of treegion formation using tail duplication are presented as well as results from scheduling the expanded treegions.

4.1 Tail Duplication

4.1.1 Description

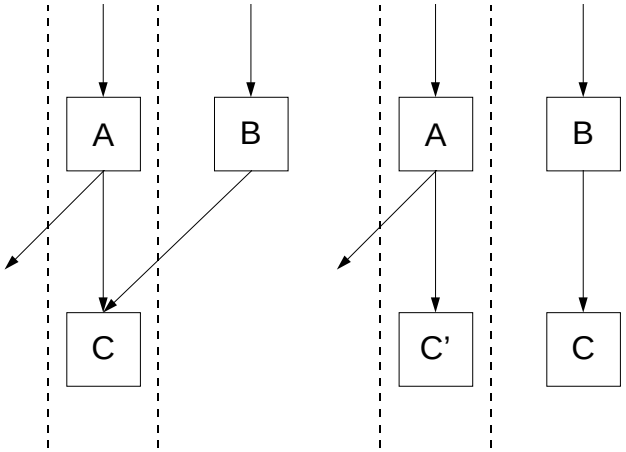


Figure 4.1: Tail duplication. The trace segment on the left, delimited with dotted lines, has a side entrance at block C. After C is tail duplicated, the duplicate C' is attached to A, while C is moved outside the trace. The side entrance has been removed.

Tail duplication [9] is a process used during superblock formation [3]. To summarize, superblock formation is a two-step process. The first step is ordinary trace formation [10], [2] using profile and other information as guides. In the second step, each trace is converted into a superblock by removing side entrances into the trace using tail duplication. An example of the second step of this process is shown in Figure 4.1.

At first, when a merge point is targeted for tail duplication, a new node identical to the merge point internally is created. One of the incoming edges of the merge point

is changed to lead to the duplicate, while the rest still lead to the original merge point. All outgoing edges from the merge point are copied to lead from the duplicate as well.

In the second step of superblock formation, the trace's nodes are traversed from top to bottom. If a merge point is encountered, it is tail duplicated; the duplicate replaces the original in the trace, while the original is moved outside the superblock. In this way, merge points (side entrances) are eliminated in superblocks.

This process can be easily applied to treegions. Any merge point that is a sapling of a treegion can be targeted for tail duplication. The duplicate of the sapling can then be absorbed into the treegion since it is not a merge point. In many cases, the original merge point will be left with only one incoming edge, allowing it to also be absorbed into a treegion, instead of spawning a new treegion.

The cost of using tail duplication is code expansion, which may affect instruction cache design and performance [20]. However, its use can expose more parallelism in execution paths and provide more opportunities for moving ops upward in the schedule.

Figure 4.2 shows the topmost treegion of Figure 2.2 after one tail duplication. The basic block bb5, originally a sapling of the treegion, has been tail duplicated into two basic blocks, bb5 and bb5a. Both blocks have then been absorbed into the treegion, allowing more ops to be incorporated into the treegion schedule.

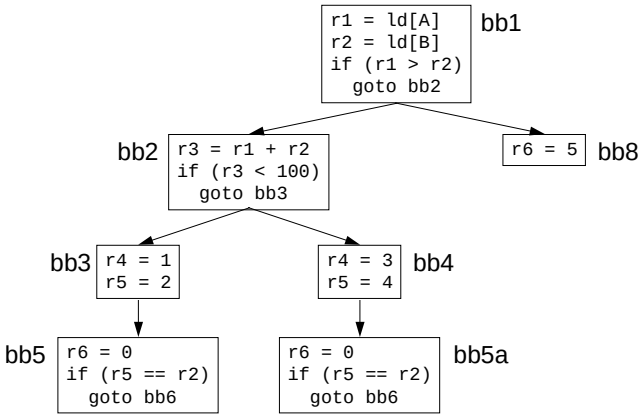


Figure 4.2: A treegion after tail duplication. The topmost treegion in Figure 2.2 is shown above after bb5, originally a merge point, has been tail duplicated into bb5a and both blocks have been absorbed into the treegion.

4.1.2 Tail Duplication Heuristics

Several heuristics are used to control the amount of tail duplication that can occur during treegion formation.

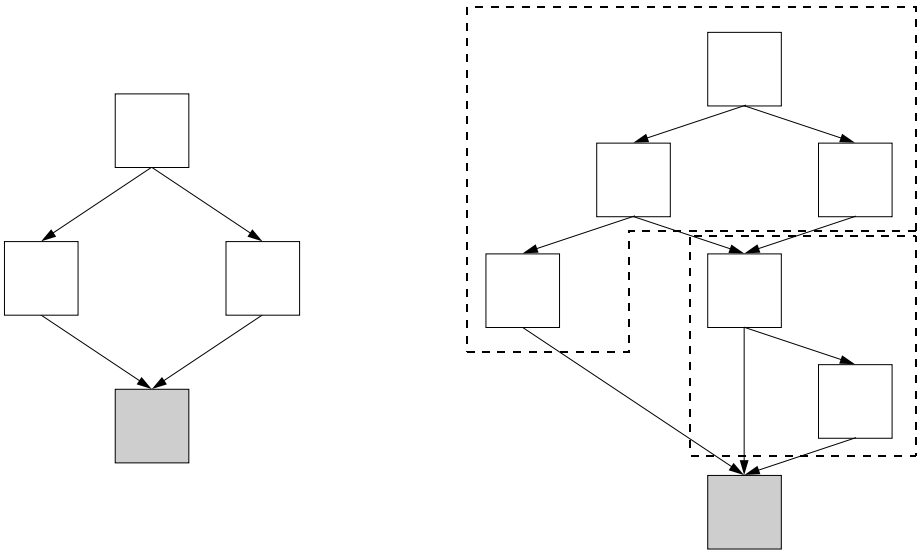
The first tail duplication heuristic is a code expansion limit. Excessive tail duplication can create bloated treegions, potentially at the expense of treegions further on in the CFG. As stated earlier, overall code size is also a concern. By limiting the code expansion of a treegion, the distribution of treegions can be more evenly balanced and, as a bonus, compilation time will not be overly long. The parameterization of code expansion is explained in Section 4.1.5.

Another heuristic places a limit on the number of execution paths in a treegion[1]. A large number of paths in a treegion will compete with each other for slots in the schedule, and some paths will be delayed more than others. Limiting the path count of a treegion can create a fairer schedule and also keeps code expansion down. For the experiments in this section, the path count limit for a treegion is set to 20, meaning that no tail duplication will be performed if the number of paths in a treegion exceeds 20. This number was chosen to allow the formation of treegions of significant size; the effects of varying this limit have not been examined at this time.

The next tail duplication heuristic pertains to terminal merges. A *terminal merge* is a merge point from which there are no outgoing edges; the CFG terminates at a terminal merge. If left alone, a terminal merge will form a treegion containing only itself. Tail duplication of a terminal merge can remove the single-block treegion completely, and code duplication is limited since the terminal merge has no successors in control flow that may be tail duplicated as well. In these experiments, terminal merges are always tail duplicated as long as the treegion has not exceeded its code expansion or path count limits. Figure 4.3(a) shows a terminal merge in a small CFG.

Another heuristic pertains to outside merges. An *outside merge* is a sapling of a treegion which has incoming edges from outside the treegion. Here, the merge point seems to be relatively important to the control flow from paths outside the treegion in question, and may be the start of a logically separate area of code, so

[1]Equivalently, a limit is placed on the number of exits from the treegion.



(a) Terminal merge (b) Outside merge

Figure 4.3: Tail duplication rules of thumb. The merge point on the left, a *terminal merge*, is always tail duplicated. The merge point on the right, an *outside merge*, is never tail duplicated unless it is a terminal merge.

these merge points are not tail duplicated. However, outside merges which are also terminal merges *are* tail duplicated. Figure 4.3(b) shows an example of an outside merge.

The final heuristic is a sapling's merge count. The *merge count* of a merge point is the number of incoming edges; this is always at least two, and often more. If a merge point has a high merge count, it is again likely to signify a new logical decision process in the CFG, or at least an important checkpoint in control flow. For

this reason, and because tail duplication would be needed several times to finally eliminate them, merge points with high merge counts are not tail duplicated (except for terminal merges). Possible values for the threshold merge count are considered in Section 4.1.5.

4.1.3 Modified Treegion Formation Algorithm

The treegion formation algorithm, modified to perform tail duplication, is shown in Figure 4.4. After a treegion is initially formed, the saplings are scanned until one is found that satisfies the heuristics for tail duplication. The sapling chosen is then tail duplicated, and the duplicate is attached to some predecessor contained in the current treegion. Treegion formation is then resumed starting with the duplicated node, which will be added to the treegion.

It is likely that once a merge point is tail duplicated, the merge point is left with only one incoming edge left. However, this sapling can be selected for tail duplication in a subsequent iteration, and the sapling itself will be added to the tree instead of a duplicate (lines 20, 24).

Tail duplication ceases if one of several conditions is met. It will often stop if the path count limit has been reached. Otherwise, it may stop early if no saplings remain, or if no remaining saplings qualify for tail duplication because of high merge counts, being outside merges, or being too large to limit code expansion.

It should be noted that the choice of which sapling to tail duplicate is arbitrary

```
 1 treeform-td (CFG)
 2 {
 3   Add top node(s) of CFG to unprocessed queue
 4   while unprocessed queue is not empty {
 5     Get first node in unprocessed list
 6     if node is already in a treegion continue
 7     Make a new empty treegion
 8     absorb-into-tree (treegion, node)
 9     do {
10       if treegion path count exceeds limit break
11       for each sapling of current treegion {
12         if sapling is in another treegion continue
13         if code expansion limit might be exceeded continue
14         if terminal merge use this sapling (break out of loop)
15         if sapling is outside merge continue
16         if sapling merge count exceeds allowed limit continue
17         use this sapling
18       }
19       if sapling selected for tail duplication {
20         if sapling is merge point
21           tail-duplicate (sapling onto node in tree)
22           absorb-into-tree (treegion, duplicate)
23         else
24           absorb-into-tree (treegion, sapling)
25       }
26     } while some sapling is selected for tail duplication
27     for each sapling of current treegion
28       if sapling is not in a treegion
29         add sapling node to unprocessed queue
30   }
31 }
```

Figure 4.4: Treegion formation algorithm with tail duplication. After the initial formation, tail duplication is performed and duplicates are added to the treegion until no saplings can be tail duplicated.

here, and hence profile-independent. Experiments were run to determine whether the saplings should be selected based on profile weight, i.e., whether the sapling with the greatest weight should be duplicated first. The assumption here is that the most heavily executed blocks ought to benefit the most from tail duplication.

The results shown in Figures 4.5 and 4.6 reveal that sorting by profile weight

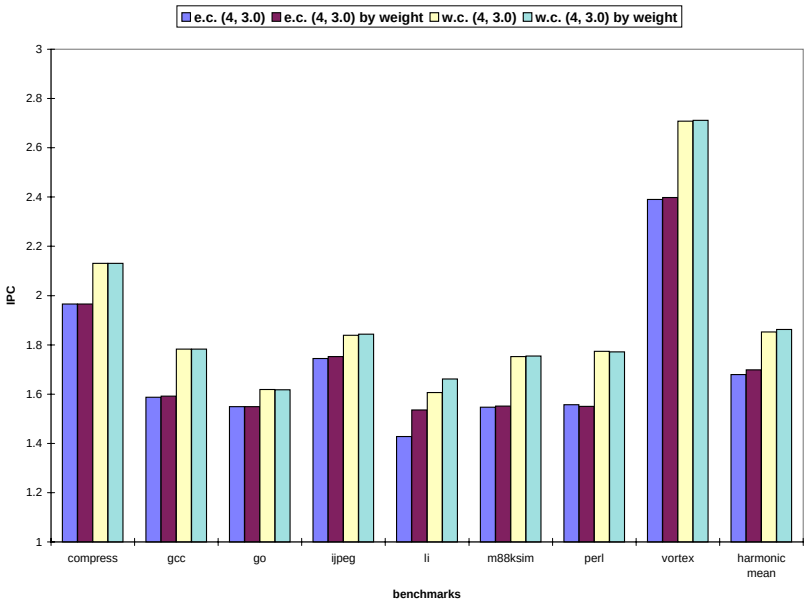


Figure 4.5: Results for tail duplication by weight, 4U. The abbreviation e.c. stands for exit count, and w.c. stands for weighted count.

on average yielded less than 1% better performance than not sorting at all; only **li** showed any significant improvement in performance, and some benchmarks performed slightly worse. This is due to the fact that high profile weight does not imply high parallelism. These findings help support the decision to completely divorce profile information from treegion formation, even with tail duplication.

4.1.4 Treegions with Tail Duplication in SPECint95

Treegions were reformed in the SPECint95 benchmarks using the modified algorithm shown in Figure 4.4. The maximum path count was limited to twenty execution paths, code expansion was limited to twice a treegion's original size, and the threshold

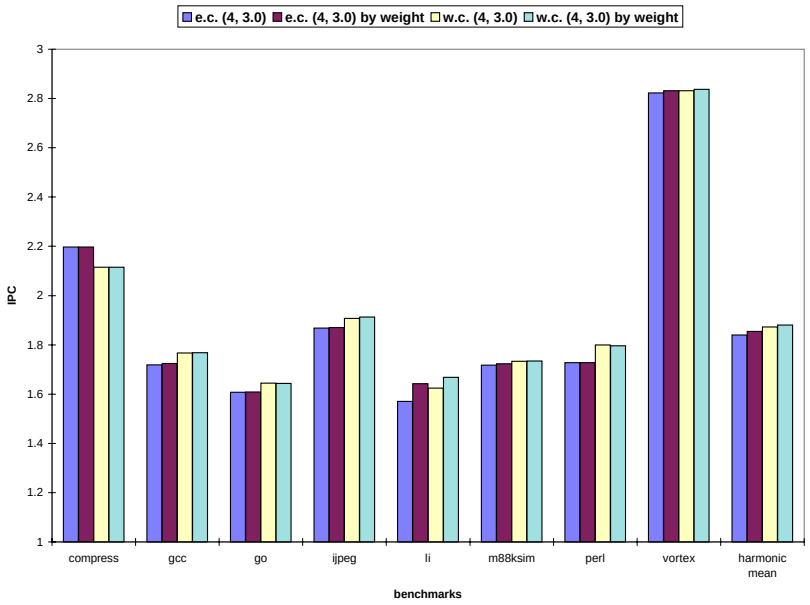


Figure 4.6: Results for tail duplication by weight, 8U. The abbreviation e.c. stands for exit count, and w.c. stands for weighted count.

merge count was set at four (so merge points with merge counts greater than four were not tail duplicated). The results of these experiments are summarized in Table 4.1. The total treegion count decreased for all of the benchmarks while the average treegion size increased from 3–4 basic blocks to 5–6 basic blocks. The size of each benchmark's largest treegion either stayed the same or increased slightly.

4.1.5 Treegion Scheduling with Tail Duplication

The expanded treegions formed using tail duplication were scheduled using the exit count and weighted count heuristics. The results are compared here to superblock

Table 4.1: Treegion statistics after tail duplication. Treegion execution path count was limited to twenty, code duplication limited to two times, and the threshold merge count was four.

benchmark	**# treegions**	**avg # bb**	**max # bb**	**avg # ops**
compress	87	5.195402	28	35.632184
gcc	15186	6.152575	384	41.123864
go	3280	5.608232	89	39.246646
ijpeg	1575	4.795556	55	37.392381
li	1053	4.584995	44	30.914530
m88ksim	1483	6.923803	146	48.937964
perl	3527	6.198469	774	42.955486
vortex	1175	7.724255	40	72.055319

schedules[2] rather than SLR schedules, since superblocks use tail duplication in their formation.

The first set of figures holds the threshold merge count at four and varies the code expansion limit, measured as a factor expansion of the number of ops in a treegion, between 1.5 times, two times, and three times. (The threshold merge count is varied later in this section.) The ordered pairs in the legend represent, in order, the threshold merge count and the code expansion limit.

Figures 4.7 and 4.8 show the results for exit count treegion scheduling for the 4U and 8U machine models. Overall, treegion scheduling manages to produce slightly better performance than superblock scheduling, by about 1% for 4U and about 7% for 8U. Some benchmarks, such as **compress** and **vortex**, perform significantly better

[2]The superblock schedules were generated by the LEGO compiler using processes, heuristics, and techniques as close as possible to those described in the literature [3], [21], [22].

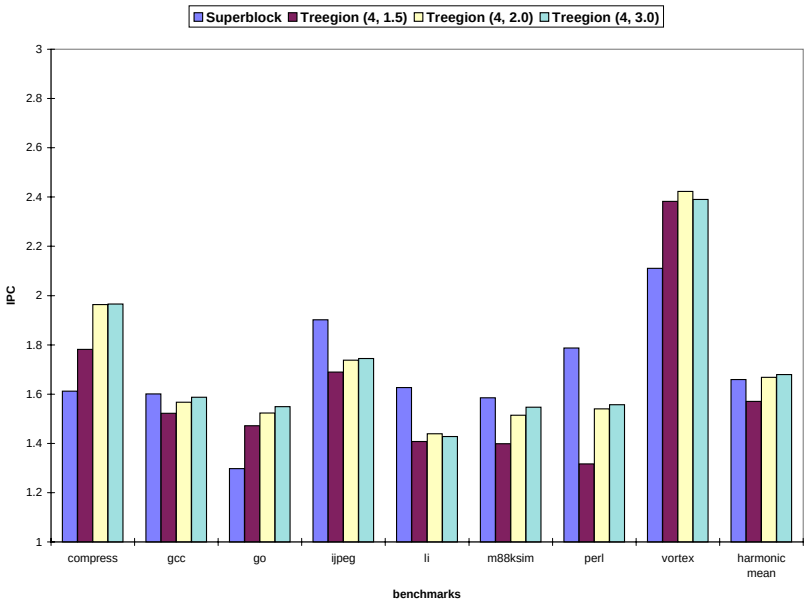


Figure 4.7: Varying code expansion for exit count tail duplicated treegion scheduling, 4U.

under treegion scheduling, while others such as **ijpeg** and **perl** perform worse. This implies that the first group does not contain many heavily-biased treegions, so the arbitrating effects of treegion scheduling benefit them, while the latter group do contain such branches, for which superblock scheduling is more attuned. This shortcoming of exit count treegion scheduling is addressed below.

Figures 4.9 and 4.10 show the results for weighted count treegion scheduling for the 4U and 8U machine models. Here, treegion scheduling outperforms superblock scheduling by about 12% for 4U and about 9% for 8U. This is a significant improvement over the exit count heuristic, especially for the 4U model. This model has less resources available each cycle, and the weighted count heuristic is able to focus on

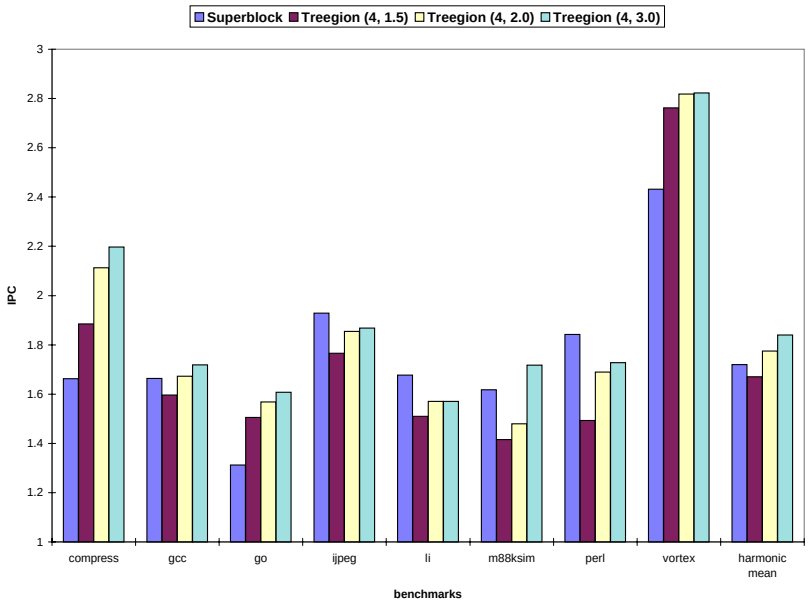


Figure 4.8: Varying code expansion for exit count tail duplicated treegion scheduling, 8U.

the more important heavily executed ops.

The second set of figures holds the code expansion limit to two times and varies the threshold merge count between four, five, and six. Again, the results are compared to superblock scheduling. Figures 4.11 and 4.12 show the results for exit count treegion scheduling, and Figures 4.13 and 4.14 show the results for weighted count treegion scheduling. The results show that varying this heuristic does not affect performance significantly for the benchmarks (except for **vortex**, where the average merge count is higher). This is due to the fact that merge points with very high merge counts are rather rare (except in **vortex**), and those that are present are not usually completely eliminated after tail duplication due to the code expansion limit.

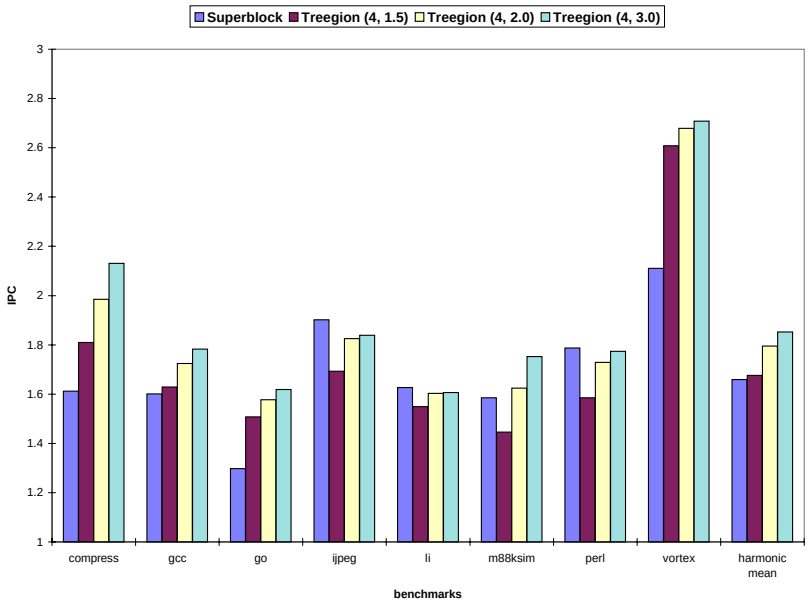


Figure 4.9: Varying code expansion for weighted count tail duplicated treegion scheduling, 4U.

Overall the results show that treegion scheduling is competitive with superblocks when tail duplication is applied to treegion formation. A further enhancement to treegion scheduling, described in the next section, will improve treegion scheduling performance further beyond superblock scheduling.

4.2 Dominator Parallelism in Tail Duplication

4.2.1 Description

Tail duplication is an effective means of expanding treegions in order to decrease program execution time. However, it introduces some problems. Besides the issues

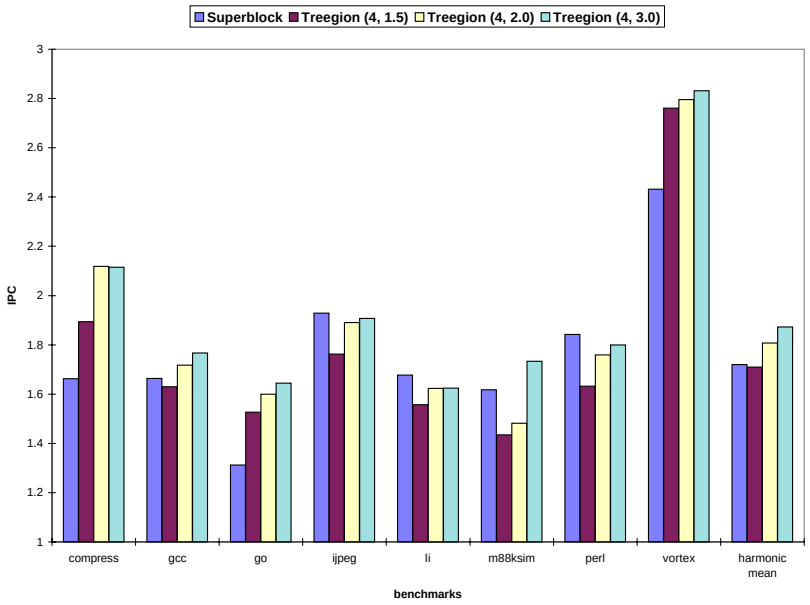


Figure 4.10: Varying code expansion for weighted count tail duplicated treegion scheduling, 8U.

associated with code expansion, tail duplication introduces redundant computation into treegion schedules. If some sapling is duplicated several times, several copies of its code will be integrated into the schedule, even though the duplicated ops in most cases will do the same work.

These problems can be solved by taking advantage of *dominator parallelism* in treegions, which arises when identical ops are speculated into the same block. This block *dominates* these ops since all execution paths leading to the ops must run through the block; this is true since there are no side entrances into treegions. As a result, the redundant ops can be replaced with a single speculative op which can do the work for all paths.

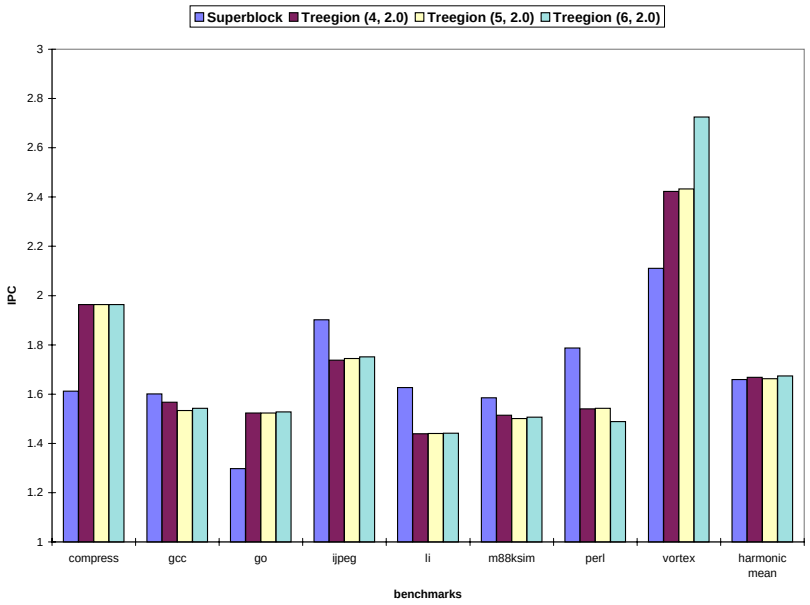


Figure 4.11: Varying merge count for exit count tail duplicated treegion scheduling, 4U.

This is the same process as the unification transform of percolation scheduling [23], which unifies several identical ops spread across several blocks into one op in a unique predecessor block while placing duplicates at side entrances into the code area being transformed. Since treegions are single-entry regions, this duplication at side entrances is not necessary. Exploiting dominator parallelism produces more efficient schedules. There are less ops to schedule and therefore shorter schedules can be generated with more *useful* speculative execution.

The algorithm for treegion scheduling requires only a minor change to take advantage of dominator parallelism. The list scheduler, as it schedules ops, tests each one to determine whether or not a duplicate op (i.e., an op identical to it that was created

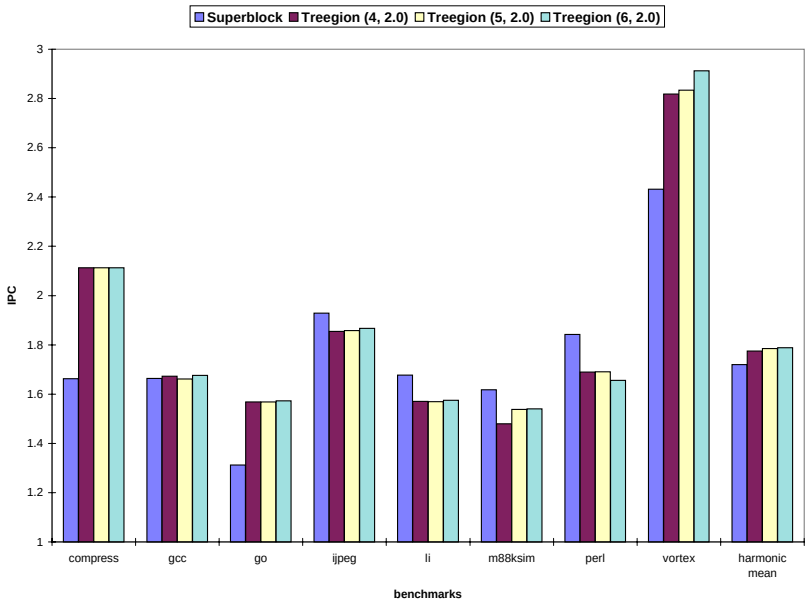


Figure 4.12: Varying merge count for exit count tail duplicated treegion scheduling, 8U.

during tail duplication) has already been scheduled in the current scheduling region. A *scheduling region* here means a section of a schedule between consecutive branches; it corresponds roughly to a basic block, but for treegion scheduling it may contain ops from blocks along independent execution paths. If a duplicate of the current op has indeed been scheduled in the current scheduling region, the op being tested is not scheduled at all. Note that this also removes the need to perform register renaming for the duplicate ops, which simplifies the job of register allocation.

It is possible due to varying dependencies along treegion paths that not all duplicate ops can be speculated into the same scheduling region. This may also happen depending on the priority list of the list scheduler and available resources. If so, then

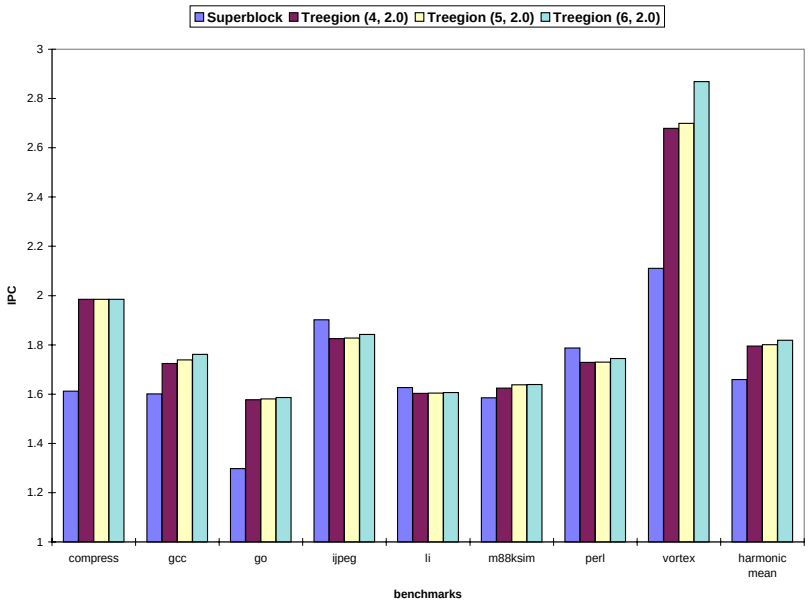


Figure 4.13: Varying merge count for weighted count tail duplicated treegion scheduling, 4U.

the first op in a new scheduling region is scheduled as usual, and subsequent ops are unified with this op if possible. So even under more limiting conditions some dominator parallelism can be extracted.

As an example of dominator parallelism, refer to Figure 4.2. The instruction `r6 = 0` in block bb5 and its duplicate in bb5a can be speculated as far as the scheduling region for block bb1, that is, scheduled before the exit branch of bb1. If these ops are scheduled in the scheduling region for either bb1 or bb3, one of them can be removed, which frees up one schedule slot for another useful operation. Note that the ops that determine the condition of the instruction `if (r5 == r2)` exhibit no dominator parallelism since `r5` has differing values depending on the execution path.

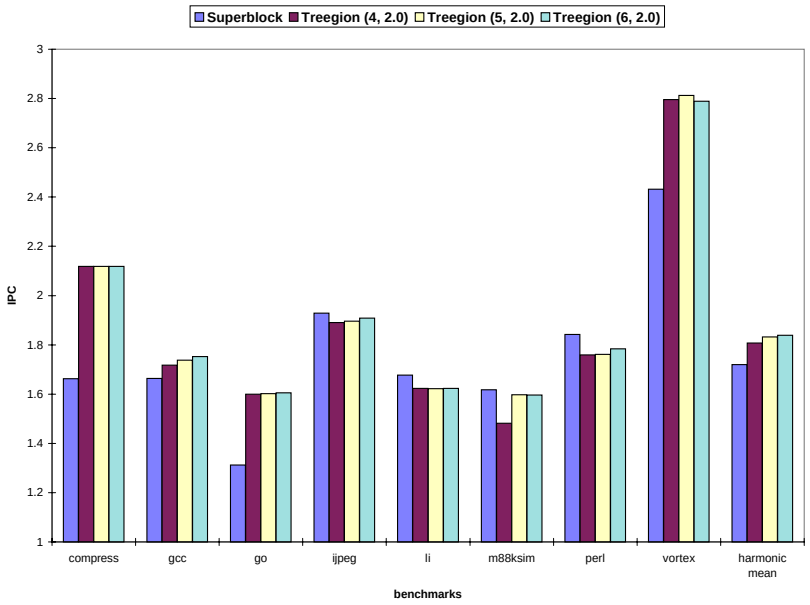


Figure 4.14: Varying code merge count for weighted count tail duplicated treegion scheduling, 8U.

Register renaming would change one of the conditions to `r5a == r2`, so the ops would no longer be identical.

4.2.2 Treegion Scheduling with Dominator Parallelism

The experiments of Section 4.1 which varied the code expansion limit on tail duplication were repeated using dominator parallelism to obtain more efficient schedules. The code expansion limit of 1.5 is not considered due to its poor performance compared to superblocks. The results are compared to superblock scheduling once again.

Figures 4.15 and 4.16 summarize the results using the exit count heuristic, and Figures 4.17 and 4.18 show the same results using the weighted count heuristic.

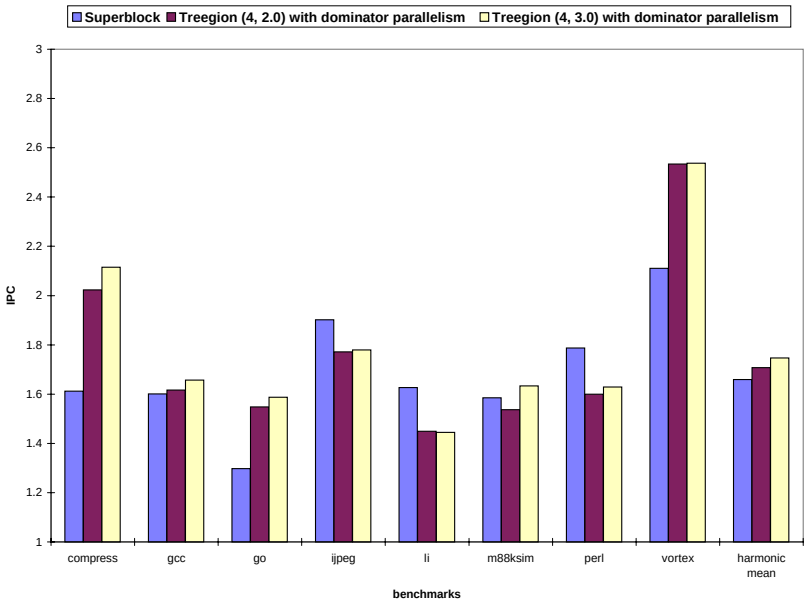


Figure 4.15: Varying code expansion for exit count tail duplicated treegion scheduling with dominator parallelism, 4U.

Overall, dominator parallelism increased performance by only a few percentage points (no more than 4%) over scheduling without it. However, the increase in performance relative to superblocks has been extended to 5 – 9% for the exit count heuristic, and to about 12% for the weighted count heuristic[3].

Tail duplication and dominator parallelism have been shown to be effective means of expanding treegions and producing efficient and fast schedules. These techniques help treegion scheduling do a better job of meeting the needs of multiple execution paths and produce schedules that outperform superblock scheduling.

[3]The performance for the weighted count heuristic under the 4U model without tail duplication was about 12% without dominator parallelism, so the improvement for this case is small.

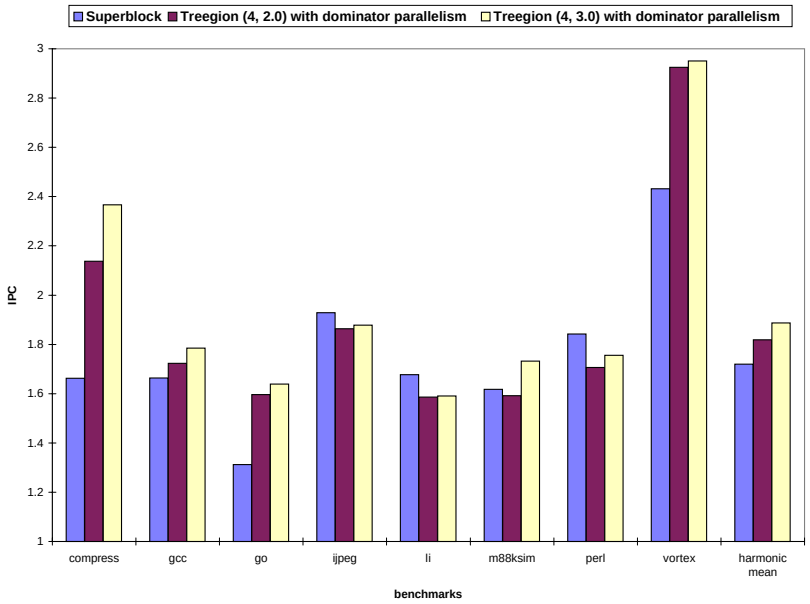


Figure 4.16: Varying code expansion for exit count tail duplicated treegion scheduling with dominator parallelism, 8U.

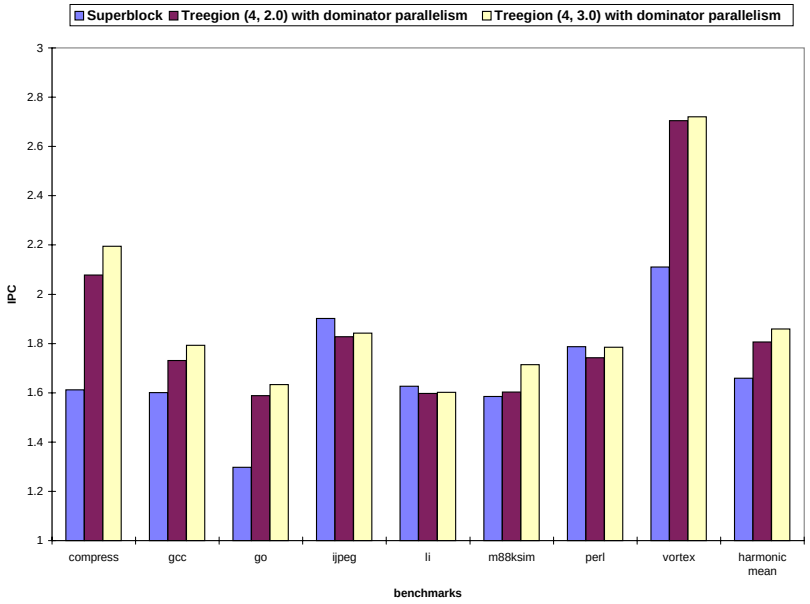


Figure 4.17: Varying code expansion for weighted count tail duplicated treegion scheduling with dominator parallelism, 4U.

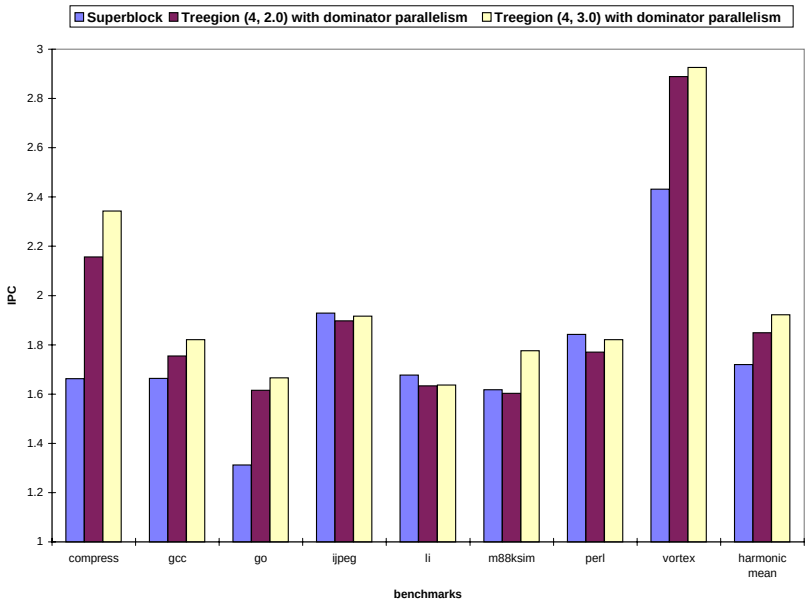


Figure 4.18: Varying code expansion for weighted count tail duplicated treegion scheduling with dominator parallelism, 8U.

Chapter 5

Treegion Scheduling for SPECfp95

This chapter contains the results of treegion scheduling for the six floating point benchmarks of the SPEC benchmark suite, collectively known as SPECfp95.

5.1 Ordinary Treegion Scheduling

Figure 5.1 presents the results from treegion scheduling without tail duplication. The first two graphs compare basic block scheduling and SLR scheduling to exit count or ordinary treegion scheduling. The next two compare basic block scheduling and exit count scheduling to dependence height scheduling, the next two to global weight scheduling, and the last two to weighted count scheduling.

The results show that for this set of benchmarks, the exit count, global weight, and weighted count heuristics all work about as well for treegion scheduling, while the dependence height heuristic is still not as good. Treegion scheduling increases performance by about 7% over basic block scheduling and by about 3% over SLR scheduling. High parallelism already present in these benchmarks leaves less ILP for treegions to reveal than for the integer benchmarks.

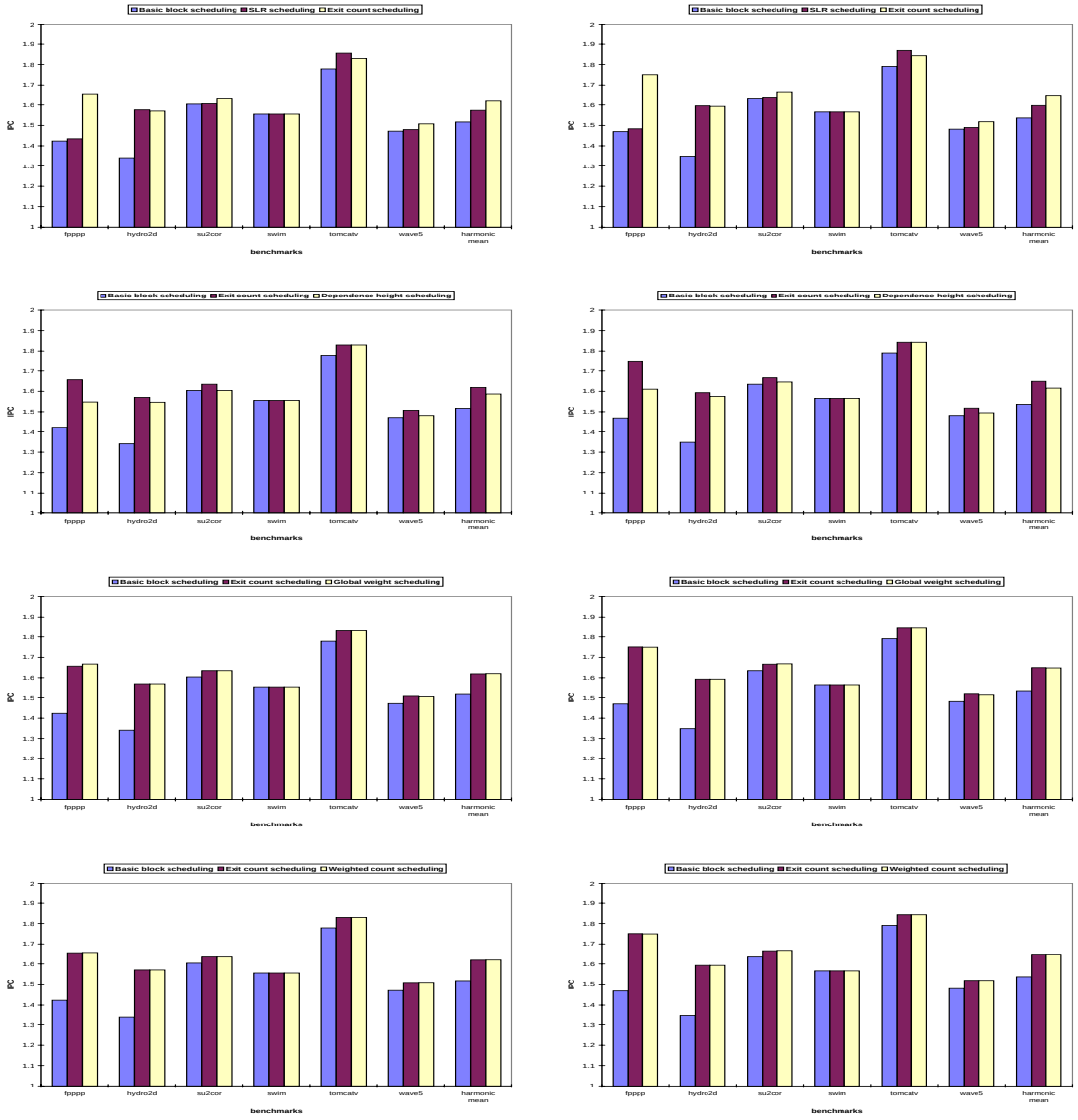


Figure 5.1: Ordinary treegion scheduling, SPECfp95. The four rows show results for the exit count, dependence height, global weight, and weighted count heuristics. The left column used the 4U machine model, the right the 8U model.

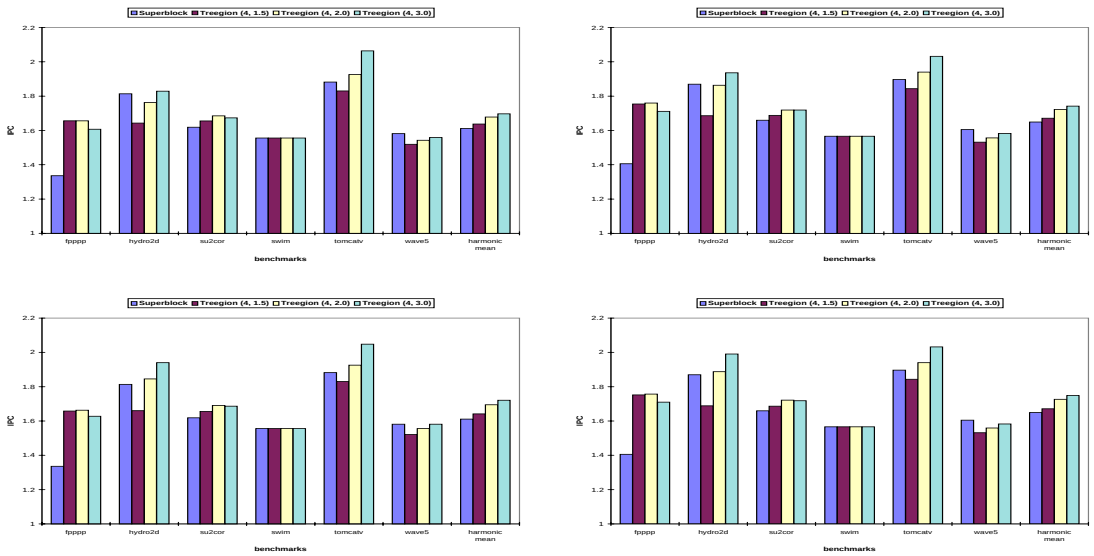


Figure 5.2: Varying code expansion for tail duplicated treegion scheduling, SPECfp95. The two rows show results for the exit count and weighted count heuristics. The left column used the 4U machine model, the right the 8U model.

5.2 Tail Duplicated Treegion Scheduling

Figure 5.2 shows the results for scheduling tail duplicated treegions under varying code expansion limits. The first row used the exit count heuristic and the second row used the weighted count heuristic. The exit count heuristic produces performance about 5.5% better than superblock scheduling, while the weighted count heuristic performs 6 – 7% better.

Two of the six benchmarks exhibit interesting behavior here. The first, **fpppp**, tends to do worse with more tail duplication. The reason is that a significant portion of the benchmark is composed of large nested loop arrangements, where the loops are very heavily executed. The tail duplication phase of treegion formation, which ignores

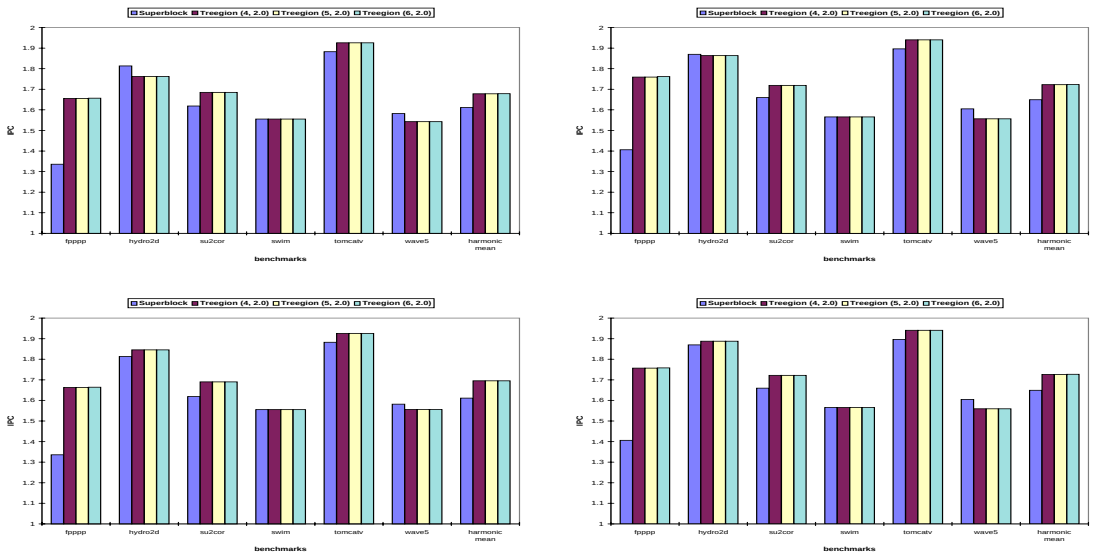


Figure 5.3: Varying merge counts for tail duplicated treegion scheduling, SPECfp95. The two rows show results for the exit count and weighted count heuristics. The left column used the 4U machine model, the right the 8U model.

the presence of backedges, begins to unroll these loops, when it should confine itself to within loop bodies. As a result, the loop schedules are split apart and performance declines.

Secondly, the **swim** benchmark is insensitive to tail duplication. Most of the execution time for this benchmark is spent in several very large basic blocks which loop back to themselves. The code expansion limit on tail duplication prevents the treegions containing these blocks from expanding much, and even when they do, all the ILP is contained in those large blocks, so execution time is barely changed.

These findings motivate a study on the interaction between treegions and loops in the future.

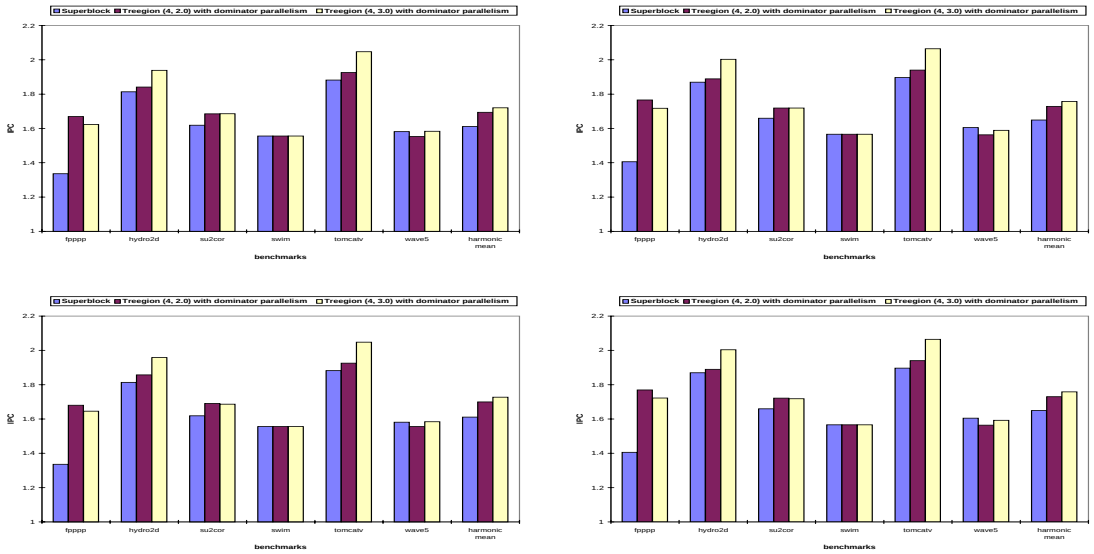


Figure 5.4: Dominator parallelism treegion scheduling, SPECfp95. The two rows show results for the exit count and weighted count heuristics. The left column used the 4U machine model, the right the 8U model.

Figure 5.3 shows the results of varying the merge count limit between four and six while holding the code expansion limit to 2.0. Changing the merge count limit has almost no effect on the performance of treegion scheduling for these benchmarks as well as the integer benchmarks.

5.3 Dominator Parallelism Treegion Scheduling

Figure 5.4 shows the results of applying dominator parallelism to treegion scheduling for the SPECfp95 benchmarks. Overall, treegion scheduling here outperformed superblock scheduling by about 6.5% for the exit count heuristic and by about 7% for the weighted count heuristic. The results shown in this chapter demonstrate that

treegion scheduling is an effective technique for scheduling scientific, floating-point code as well as ordinary integer code.

Chapter 6

Related Work

The first section of this chapter describes existing work on non-linear regions; much of this work inspired treegion scheduling in some way. The second section discusses more aspects of treegion scheduling which merit further investigation.

6.1 Previous Work

The most direct ancestor of work on treegions is decision tree scheduling [5]. Decision tree scheduling recurses down the paths of a decision tree (similar to a treegion) producing schedules iteratively between branches in the tree. Guarded instructions, the predecessors of predicated instructions [24], are used to occupy branch delay slots and perform stores early. Ops are prioritized by the weight of the execution paths running through them and by their position along the critical path, a similar heuristic to global weight treegion scheduling. Treegion scheduling has the advantage of using speculation to move ops above branches.

The IBM VLIW architecture is based on a multiple-instruction multi-branch *tree-instruction* [25]. This instruction contains a small decision-tree structure and the

conditionals that control its execution. The VLIW machine is able to evaluate the various conditionals, execute all necessary instructions, and jump to the next tree-instruction in a single cycle. Treegion scheduling, in contrast, constructs trees at the block level and does not assume the use of tree-instructions. Tree-instructions are formed from *RISC regions* which are in turn created selectively using the finite-resource global scheduling technique [6], a modified version of percolation scheduling [23] involving selective code motion. Treegions, on the other hand, are formed and scheduled in separate phases, with code motion and resource constraints considered during scheduling.

Hyperblocks [4] are an extension of superblocks that contain predicated code. Multiple paths are scheduled together, and their conditional execution is controlled by their predicates, so only instructions on the taken path through the hyperblock actually write back their results. Hyperblock formation is derived from if-conversion [24], the process of converting branching code into straight-line predicated code.

Trace scheduling [2] was the first attempt to schedule instructions across basic blocks. Its successor, Trace scheduling-2 or TRACE-2 [26], performs scheduling across a *cluster* of basic blocks, not just a single linear path. This allows the technique to detect dominator parallelism, eliminating some speculation, and to generate compensation code as scheduling proceeds. A speculative yield function, which determines the probability of speculating an op being useful, contributes to prioritizing the ops. TRACE-2 would allow merge points, but requires some scheduling complications tree-

gion scheduling avoids.

The speculative hedge heuristic [12] is the most significant contributor to the various heuristics of treegion scheduling. The goal of speculative hedge is to limit overspeculation and allow all exits from a region (superblock) to be retired in a timely fashion, without unnecessarily delaying any of them. Treegion scheduling heuristics are based on the same premise and work analogously to some components of speculative hedge, and treegion formation dictates the set of execution paths to schedule together, which speculative hedge by choice does not address. However, since treegions are larger than superblocks and encompass multiple paths, they yield higher levels of ILP.

6.2 Future Work

This study is the beginning of work on treegion scheduling. Several new paths of research extend from the work presented here.

An important feature of treegions is that they can be made and scheduled without profile information, and that their performance should not be compromised when program execution differs from the profile information. This difference can be termed a *profile shift*. Just how well treegions perform after a profile shift should be determined.

Classical optimizations have been defined for and applied to superblocks [3], [21] which help produce better performance for these regions. Treegions may also provide

opportunities for these optimizations as well, especially in the reduction of branch instructions and the removal of redundant operations beyond dominator parallelism.

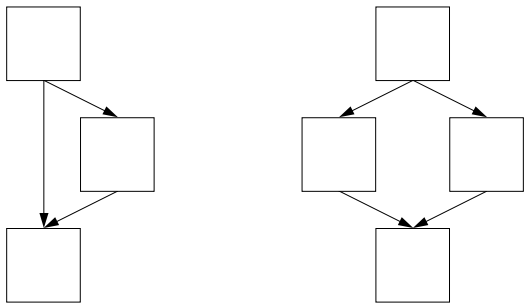


Figure 6.1: Hammocks. The left hammock results from an if-then construct; the right hammock results from an if-then-else construct.

A second method of eliminating merge points and expanding treegions is *hammock conversion*. A hammock is a simple CFG structure which results from an if-then or an if-then-else construct. The top block in a hammock contains a conditional branch, which leads to one (if-then) or two (if-then-else) succeeding blocks. A final block on the bottom consolidates control flow once again. Figure 6.1 shows examples of hammocks. If-conversion of hammocks into hyperblocks will eliminate merge points without the code expansion drawback of tail duplication. Use of hammock conversion alone and combined with tail duplication may improve treegion scheduling further.

The exit count and weighted count scheduling heuristics apparently perform well.

Their suitability would be further verified if their performance were compared to optimal treegion scheduling, which could be produced by an application of integer-linear programming. Optimal treegion scheduling is an important next step in the study of treegion scheduling heuristics.

The scheduling results for SPECfp95 suggest that a study of the interaction between treegions and loops may be worthwhile, or at the least modification of the treegion formation algorithm to consider loop structures in the CFG.

Finally, more detailed scheduling heuristics may be worth investigating. The exit count heuristic performs poorly if a large subtree of a treegion is never executed since it prioritizes those ops over the more important ones. The global weight heuristic can fail when scheduling very biased treegions. The weighted count heuristic partially eliminates these failings, but several other choices in heuristics should be developed and tested to determine just how well heuristics can do.

Chapter 7

Conclusion

This thesis has described treegion scheduling, a method of instruction scheduling for VLIW processors. The treegion is a set of basic blocks that comprise a tree-shaped subgraph of a program CFG. Thus, treegions can contain multiple disjoint paths, and their formation does not require the presence of nor high reliablility of profile information. The treegion formation process is a modified traversal of a CFG.

Treegion scheduling provides techniques for scheduling the multiple paths of a treegion to execute together. Register renaming can be used to remove conflicts between mutually exclusive paths and to allow speculation of instructions that define live-out variables of a given branch. Predication can be used to control the branching within a treegion and to other treegions. Treegion scheduling was shown to outperform basic block scheduling and SLR scheduling for the SPECint95 benchmarks.

Tail duplication, a process used in superblock formation, can be applied to the treegion formation process. Merge points below treegions can be eliminated through code duplication, allowing treegions to grow larger and expose more ILP during scheduling. These tail duplicated treegions, when scheduled, produce performance that rivals and

often exceeds superblock scheduling.

Dominator parallelism in the presence of tail duplication can be used to eliminate the scheduling of redundant ops introduced during tail duplication, allowing for the scheduling of more useful calculations. When this is done, treegion scheduling outperforms superblock scheduling by between 5% and 9% for a four-issue machine and by about 12% for a wider eight-issue machine.

Treegion scheduling is still a relatively new technique, and there is more to be done before the subject is truly mature. However, the work described here demonstrates the potential for treegion scheduling to be a powerful tool for high-performance scheduling of VLIW programs.

Bibliography

[1] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: an introduction and motivation," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture* [27], pp. 158–168.

[2] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[3] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[4] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Ann. Int'l. Symp. on Microarchitecture*, (Portland, OR), pp. 45–54, Dec. 1992.

[5] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proc. 13th Ann. Int'l Symp. Computer Architecture*, (Tokyo, Japan), June 1986.

[6] S.-M. Moon and K. Ebcioğlu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proc. 25th Ann. Int'l Symp. on Microarchitecture*, (Portland, OR), pp. 55–71, Dec. 1992.

[7] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[9] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software–Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[10] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Ann. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[11] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[12] B. L. Deitrich and W. W. Hwu, "Speculative hedge: regulating compile-time speculation against profile variations," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture* [28].

[13] T. M. Conte, K. N. Menezes, and M. A. Hirsch, "Accurate and practical profile-driven compilation using the profile buffer," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture* [28].

[14] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *Int'l Journal of Parallel Programming*, vol. 24, Feb. 1996.

[15] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *ACM Trans. Comput. Sys.*, vol. 11, pp. 376–408, Nov. 1993.

[16] E. Altman and K. Ebcioğlu, "DAISY: Dynamic Compilation for 100% Architectural Compatibility." June 1997.

[17] R. Cytron and J. Ferrante, "What's in a name? -or- The value of renaming for parallelism detection and storage allocation," in *Proc. of the 1987 Int'l Conference on Parallel Processing*, pp. 19–27, Aug. 1987.

[18] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three architectural models for compiler-controlled speculative execution," *IEEE Trans. Comput.*, vol. 44, pp. 481–494, Sept. 1995.

[19] M. S. Schlansker and V. K. Kathail, "Critical path reduction for scalar programs," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[20] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Trans. Comput.*, vol. 42, pp. 1045–1057, Sept. 1993.

[21] S. A. Mahlke, *Exploiting instruction level parallelism in the presence of branches*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1996.

[22] R. A. Bringmann, *Enhancing instruction level parallelism through compiler-controlled speculation*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.

[23] A. Nicolau, "Percolation scheduling: a parallel compilation technique," Technical report TR-678, Department of Computer Science, Cornell University, Ithaca, NY, May 1985.

[24] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th Ann. ACM Symp. on Principles of Programming Languages*, Jan. 1983.

[25] K. Ebcioğlu, "Some design ideas for a VLIW architecture for sequential-natured software," in *Proceedings of the IFIP Working Group 10.3 Working Conference on Parallel Processing*, (Pisa, Italy), pp. 3–21, North Holland, 1988. (published as *Parallel Processing*, M. Cosnard, et al., (eds).).

[26] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace Scheduling-2," Tech. Rep. HPL-93-43, Hewlett-Packard Laboratories, June 1993.

[27] *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Dec. 1995.

[28] *Proc. 29th Ann. Int'l Symp. on Microarchitecture*, (Paris, France), Dec. 1996.