Treegion Scheduling for Wide Issue Processors

William A. Havanki   Sanjeev Banerjia*   Thomas M. Conte

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7911
(919) 515-5067, conte@ncsu.edu

Abstract

Instruction scheduling is one of the most important phases of compilation for high-performance processors. A compiler typically divides a program into multiple regions of code and then schedules each region. Many past efforts have focused on *linear* regions such as traces and superblocks. The linearity of these regions can limit speculation, leading to underutilization of processor resources, especially on wide-issue machines. A type of *non-linear* region called a *treegion* is presented in this paper. The formation and scheduling of treegions takes into account multiple execution paths, and the larger scope of treegions allows more speculation, leading to higher utilization and better performance. Multiple scheduling heuristics for treegions are compared against scheduling for several types of linear regions. Empirical results illustrate that instruction scheduling using treegions — *treegion scheduling* — holds promise. Treegion scheduling using the *global weight* heuristic outperforms the next highest performing region — superblocks — by up to 20%.

1 Introduction

Contemporary high performance microprocessors detect and extract instruction-level parallelism (ILP) by using a combination of hardware and compiler techniques. Within the compiler, instruction scheduling is a phase that can play a key role in exploiting ILP. A compiler typically divides a program into multiple *regions*, areas of code that tend to execute together, and then schedules each region.

Many past efforts have focused on *linear* regions, that is, regions which contain one control path. Trace scheduling [1] forms regions in which the basic blocks of each region, called a trace, may execute in order from top to bottom. Superblock scheduling [2] forms superblocks, regions with a single entrance and (possibly) multiple exits. Both of these techniques use profile information to guide the region formation process [3], so that the most often executed paths reap the greatest benefit.

The linearity of these regions and their dependence on profile information can be detriments to performance under some conditions. The actual execution patterns of a program may vary from the predictions inferred from profile information, which can cause a reduction in program performance [4], [5]. Also, restricting regions to linear sets of basic blocks intrinsically limits speculation and can lead to underutilization of processor resources, especially on wide-issue machines. An additional complexity of traces is the presence of *merge points*, basic blocks which have multiple incoming control flow edges. The use of speculative execution in this case requires special care such as bookkeeping code, which consumes machine resources and adds more complexity to the compiler.

This study presents an analysis of a *non-linear* region called a *treegion*. A treegion encompasses a decision-tree subgraph of a program's control flow graph (CFG). Treegion formation is performed without profile information, and instruction scheduling across a treegion – a process that is termed *treegion scheduling* – can use or ignore this information. Both processes take into account multiple execution paths, not just a single one. The larger scope of treegions gives the scheduler more chances to speculate operations, leaving less idle resources. Since they are tree structures, the complexity of merge points is avoided as well. This study builds upon our earlier work in treegion scheduling [6], [7].

This paper is organized as follows. Section 2 defines a treegion and describes the treegion formation process. Section 3 introduces treegion scheduling and several heuristics that can be applied to the process. Section 4 shows how tail duplication during treegion

*The author is now with Hewlett-Packard Laboratories, Cambridge, MA.

formation and the use of *dominator parallelism* during scheduling can improve the performance of treegion schedules. Section 5 reviews related work in nonlinear regions, and Section 6 concludes the paper.

2 Treegion Formation

The treegion is named for the fact that it is a region containing a tree of basic blocks which is a subgraph of the CFG of a program. Therefore, formation of a treegion is dependent only on the CFG topology. A treegion can contain multiple, independent control paths that diverge from the *root* (block) of the tree. Since it is a tree, a treegion is acyclic and contains no merge points except possibly the root itself.

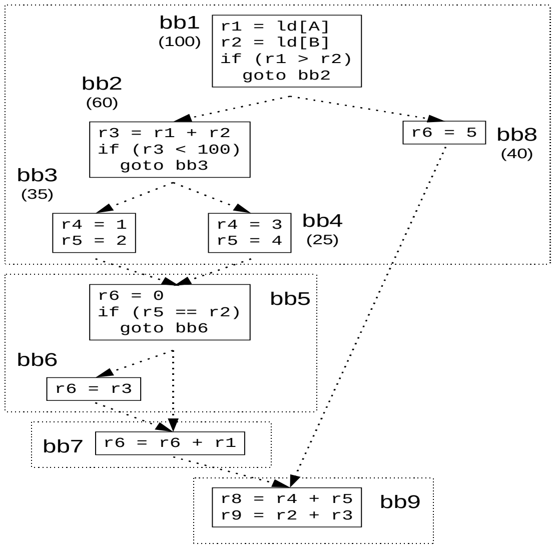


Figure 1: A CFG divided into treegions.

Figure 1 shows a CFG after treegion formation, where each treegion is surrounded by dotted lines. Note that some treegions contain only one basic block, while others contain several. The treegion formation algorithm is shown in Figure 2. Treegions are grown across a CFG starting from the entry points, each of which roots a new treegion. From a given root, the CFG is traversed, and basic blocks are absorbed into the root's treegion if they are not merge points. Eventually, only merge points remain following a treegion's leaf blocks. These are called *saplings* of the treegion and become the roots of new treegions. The process continues until the entire CFG has been consumed, at which time each basic block is in exactly one treegion.

The programs of the SPECint95 benchmark suite were run through the treegion formation process. The programs had classic optimizations and a profiling

```
treeform (CFG)
{
  Add top node(s) of CFG to unprocessed queue
  while unprocessed queue is not empty {
    Get first node in unprocessed list
    if node is already in a treegion continue

    Make a new empty treegion
    absorb-into-tree (treegion, node)

    for each sapling of current treegion
      if sapling is not in a treegion
        add sapling node to unprocessed queue
  }
}

absorb-into-tree (treegion, node)
{
  Add node to candidate queue
  while candidate queue is not empty {
    Get first node in candidate queue
    if node is already in treegion continue
    if node is a merge point and not the root
      continue

    Move node into treegion
    Add each successor of node to (front of)
       candidate queue
  }
}
```

Figure 2: 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.

run using training inputs applied to them using the IMPACT compiler [2]. The benchmarks were then converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories [8]. Region formation was performed using the LEGO compiler, a research compiler developed at N.C. State University [6]. The results are tabulated in Table 1. The average treegion in SPECint95 contains two to four basic blocks and between 20 and 25 instructions. Treegions clearly contain more operations than basic blocks and thus can enable the extraction of high levels of ILP.

3 Treegion Scheduling

The motivation for building treegions is the ability to schedule multiple execution paths simultaneously. This provides more opportunity for speculation and also allows the scheduler to better balance the needs of the different paths.

The speculative hedge heuristic [4] inspired many of the particulars of treegion scheduling. This technique provides various ways to prioritize instructions in superblocks, based on the weights of the exits they precede, the number of exits they precede, and other factors. The goal of speculative hedge is to avoid delaying execution paths even if they do not execute fre-

Program	avg # bb	max # bb	avg # instrs
compress	2.43	8	17.63
gcc	2.85	384	21.54
go	2.75	89	20.95
ijpeg	2.39	69	20.87
li	2.56	44	18.29
m88ksim	3.38	146	25.68
perl	3.14	774	23.45
vortex	3.30	39	33.53

Table 1: Treegion statistics. The data presented, from left to right, are average basic block count per treegion, maximum basic block count in a treegion, and average number of operations per treegion.

quently; the ideas presented in this technique adapt nicely to treegion scheduling.

First, treegion scheduling will be described, and an example will be presented which shows how treegion scheduling can produce a more efficient schedule than superblock scheduling. Then, the results of experiments using four different treegion scheduling heuristics will be presented.

Basic treegion scheduling algorithm

```
scheduleTreegion (treegion)
{
  Form DDG for treegion
  sortDDGNodesBy*** (DDG)
  listSchedule (DDG)
}
```

Figure 3: Treegion scheduling algorithm. The sort function can be replaced with various sorting schemes, each representing a different heuristic.

Figure 3 shows the basic treegion scheduling algorithm. The first step of the three-step process is the construction of a data dependence graph (DDG) for the treegion being scheduled. The second step is the sorting of the nodes in the DDG into a list for the third step, list scheduling. The sort function directly affects the quality of the schedules produced, and the heuristics presented in the rest of this section vary in how they perform this sorting.

Because the DDG may contain instructions from separate paths, the list scheduler may place instructions from multiple paths into the same cycle in the schedule. If the instructions do not conflict in their use and/or definition of data items, this causes no problems. If they do conflict, compile-time register renaming [9] is used.

The treegion scheduler can speculate instructions above branches. Speculating an instruction above a branch may cause incorrect execution if the instruction defines data that is used on another exit from the branch. The treegion scheduler uses register renaming to prevent such live-out violations.

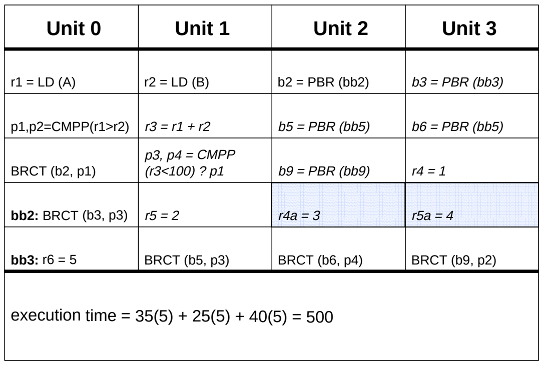
The treegion scheduler often produces schedules where several branches are scheduled in one cycle (providing the architecture allows it). Predication [10] is used to ensure that the proper branch is executed. The combination of predicated branches and speculation is similar to techniques used for critical path reduction [11].

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			

Figure 4: An example superblock schedule. Instructions in italics are speculated above branches upon which they are dependent.

Figures 4 and 5 show schedules for the topmost treegion in Figure 1 using superblock scheduling and treegion scheduling, respectively[1]. The estimated execution times for both schedules are derived by multiplying the schedule height of each path by the number of times the profile weights indicate that the path will be executed. A four-issue processor with universal functional units and unit instruction latencies is assumed. The schedules use operations similar to those found in Hewlett-Packard Laboratories PlayDoh specification [12] that are also common in other experimental architectures: speculation, predication, and a compare-to-predicate operation. Registers beginning with "r" are general-purpose integer registers, those beginning with "p" are predicate registers, and those beginning with "b" are branch target registers (BTRs). The PBR operation initializes a BTR by

[1]For simplicity, the examples only consider the topmost treegion.



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			

Figure 5: An example treegion schedule. Instructions in italics are speculated above branches upon which they are dependent. Instructions in shaded blocks show the effect of register renaming.

setting it to a target address, the CMPP operation saves the result (and optionally, the complement of the result) of a comparison into destination predicate registers, and the BRCT, BRCF, and BRU operations are branches if condition true, if condition false, and unconditionally, respectively.

The superblock schedule (Figure 4) is split into three sections. The top section is for the superblock created from (bb1, bb2, bb3), the middle section is for bb4, and the bottom section is for bb8. The estimated execution time for this section of code is 525 cycles. The treegion schedule (Figure 5) executes in 500 cycles, 25 less than the superblock schedule. This is primarily due to the speculation of additional instructions from bb4.

On a very wide machine, both schedulers are able to speculate more instructions. However, the treegion scheduler has access to multiple paths, allowing even more speculation and facilitating the completion of additional paths.

Note that the first operation in the fifth cycle of the treegion schedule (`r6 = 5`) is executed unconditionally instead of being predicated. This is permissible since r6 is not live-out on any paths out of the treegion, so no conflicts arise. If r6 were live-out, register renaming can be used as described above.

The different scheduling heuristics presented in the remainder of this section are compared with basic block scheduling and with scheduling for *simple linear regions* (SLRs). Simple linear regions are formed in the same manner as superblocks, but tail duplication is not permitted. In fact, their formation is implemented as a special case of treegion formation, where for a given node (basic block) placed into an SLR, the successor node with the highest profile weight is selected next for possible inclusion rather than all successors of the node. The result is a single-entry, multiple-exit region formed without tail duplication. Treegions will be compared with superblocks in Section 4, after the application of tail duplication to treegions has been presented.

Program	avg # bb	max # bb	avg # Ops
compress	1.30	3	9.43
gcc	1.26	54	8.98
go	1.20	22	9.16
ijpeg	1.32	18	11.58
li	1.44	7	10.25
m88ksim	1.34	9	10.19
perl	1.27	24	9.29
vortex	1.25	8	12.71

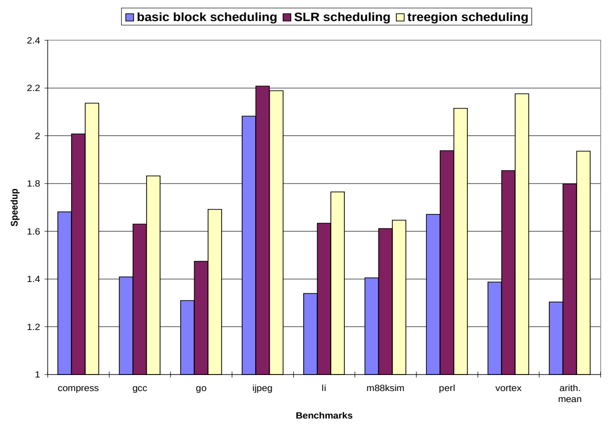
Table 2: SLR statistics. The data presented above, from left to right, are average basic block count per SLR, maximum basic block count in an SLR, and average number of operations per SLR.

Characteristics for the regions resulting from SLR formation across SPECint95 are presented in Table 2. The SLRs typically include 1-2 basic blocks and 8-12 instructions, which is less than treegions formed over the same programs. Tables 1 and 2 illustrate that a treegion provides a scheduler with more instructions (as well as more paths) to speculate and schedule.

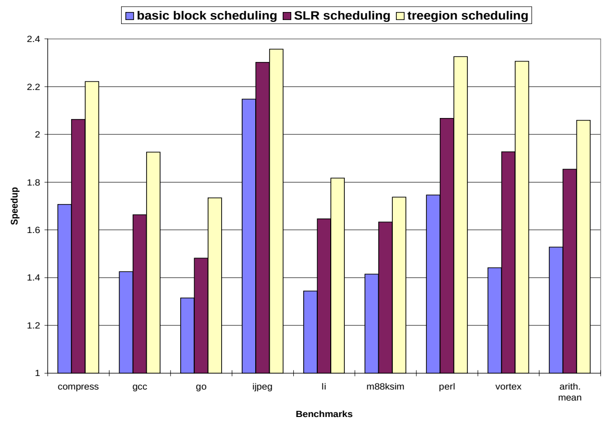
Before presenting the heuristics used for treegion scheduling and the results, our framework for these experiments will be described. Programs from the SPECint95 benchmark suite were used, as explained in Section 2. Two machine models were used for this study: a 4-issue processor (4U) and a 8-issue processor (8U), both with universal units. Both machines are statically-scheduled, very long instruction word (VLIW) architectures [1] (A VLIW instruction can also be referred to as a *MultiOp*, with each individual operation in the MultiOp termed an Op [13]. The Op/MultiOp terminology is used in the remainder of this paper.) All operations in the two machines are unit latency except for load (2 cycles), floating-point multiply (3 cycles), and floating-point divide (9 cycles). Memory operations are serialized (loads cannot bypass stores) since no aliasing information is available, and since the machine models are Playdoh-style machines a store and any dependent memory operations can be scheduled in the same cycle. All functional units are fully pipelined. Program performance was measured by using the profile count and schedule height of each region to estimate execution time. The effects of instruction and data caches were ignored, and perfect branch prediction was assumed, in order to determine the maximum benefit of the scheduling heuristics. Speedup over basic block scheduling on a single-issue, pipelined universal unit machine was the performance metric used. Copy Ops added due to renaming were not used in computing speedup.

Dependence height treegion scheduling

The dependence height heuristic — also commonly referred to as critical path scheduling — is the simplest of the heuristics discussed here. In the second step of treegion scheduling, the DDG nodes are sorted by their heights. This provides the most opportunity for speculation since Ops very far down in the treegion that have a large dependence height are given the same priority as those nearer the treegion root. On a very wide machine a large amount of speculation will occur due to abundant processor resources.



a) 4U machine model



b) 8U machine model

Figure 6: Dependence height treegion scheduling.

The results of the experiments with dependence height treegion scheduling are shown in Figure 6.

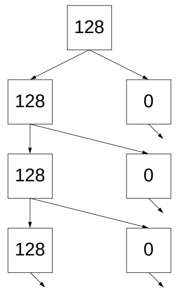


Figure 7: A biased treegion. The block numbers indicate profile weight. The leftmost path is the only path executed in the treegion.

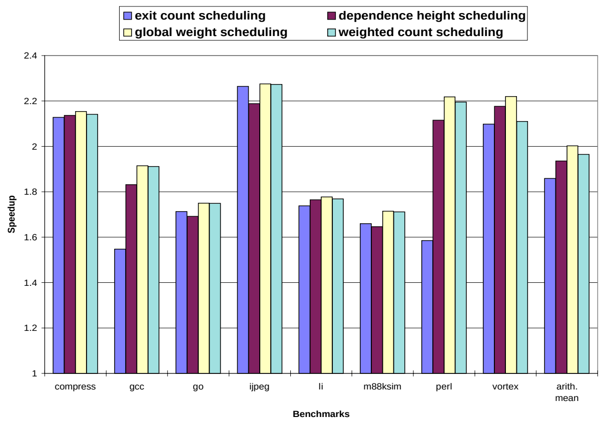
The results show that this heuristic performs well. The speedup for treegion scheduling here exceeds that of ordinary basic block scheduling by 48% and 35% for the 4U and 8U machine models, respectively, and it also exceeds the speedup of SLR scheduling by 8% and 11% (SLR scheduling was performed as treegion scheduling using the dependence height heuristic as well). The speedup for treegion scheduling is always the greatest except for the 4U schedule for **ijpeg**; this is because that particular benchmark has a preponderance of *biased* treegions, where a single path is executed 100% of the time. An example of such a treegion in **ijpeg** is shown in Figure 7. SLRs can focus on this path by itself, while treegions allow their schedule heights to stretch a bit in order to give several paths a chance to execute.

Though the results for this heuristic at first seem promising, there is a danger of overaggressive speculation for heavily-executed large treegions. Exits from these treegions may be delayed because Ops from less frequently executed paths which happen to have a greater dependence height may be speculated into the top of the treegion schedule. This would lead to poor usage of resources since the results of these speculated Ops are rarely used while all execution paths in the treegion are delayed. The next heuristic attempts to avoid this danger.

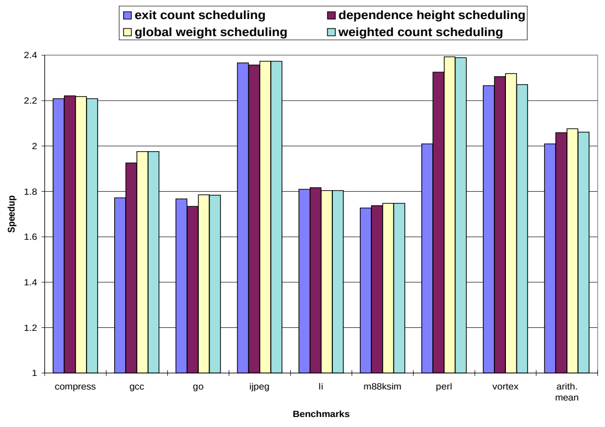
Exit count treegion scheduling

The scheduling results for the remaining three heuristics are presented in Figure 8. Each heuristic will be described in turn, accompanied by an analysis of the heuristic's results.

The exit count heuristic is adapted from the helped count priority function of speculative hedge [4]. In



a) 4U machine model



b) 8U machine model

Figure 8: The four treegion scheduling heuristics.

this scheme, the priority of an Op is equal to the Op's *exit count*, which is the number of exits that follow the Op in control flow in the treegion. This heuristic gives highest priority to Ops in the root of the treegion, since by definition they help every exit in the treegion. The nodes in the treegion DDG are sorted first by exit count; then, Ops with the same exit count are sorted by dependence height.

This heuristic attempts to curtail any overaggressive speculation by accounting for the treegion topology; however, the results are mixed, and overall the dependence height heuristic provides 2–4% higher speedup. Notably, this heuristic performs very poorly on **gcc** and **perl**.

An explanation for the surprising results for the exit count heuristic was found by examining particular treegions in the two problem benchmarks. These treegions were each rooted by a very wide multiway branch, and their leaves were the various choices for the branches. Furthermore, the treegions were shallow; only a few treegion exits resided along any one branch destination. A simplified example of such a treegion, taken from **gcc**, is shown in Figure 9.

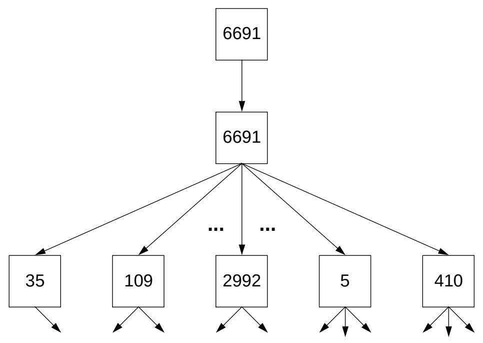


Figure 9: A wide, shallow treegion. The block numbers indicate profile weight. The branch destinations with the highest exit count are not necessarily the most often executed.

The fundamental flaw of the exit count heuristic exposed by these treegions is the assumption that an area of code with a higher exit count typically has a higher profile weight. In these treegions, each area of code at a branch destination had a roughly equal (small) exit count. However, most of them (not shown) had zero profile weight, and those with the highest exit counts did not have the highest weight.

As a result of this flaw, those branch destinations with low exit count and high profile weight were given less priority than those with a high exit count but a low profile weight. The most often executed sections of the treegion away from the root were therefore delayed. This behavior was aggravated by the large width and symmetry of the treegions. On the other hand, the dependence height heuristic is more democratic in these situations, and tends to schedule all destinations at an equal priority. Hence it produces better schedules for these treegions.

Global weight treegion scheduling

The global weight heuristic uses profile weights to prioritize Ops. This gives an advantage to Ops executed more often, which may help the most frequently executed paths be scheduled more quickly. The priority value assigned to an Op is the profile weight of the original basic block which contains it. Ops with the same weight are sorted by dependence height. This heuristic stems from the helped weight heuristic of speculative hedge, which uses the total weight of all exits helped by an Op as a priority value. Since a treegion is fundamentally a tree of execution paths, the weight of all exits reached through an Op in a treegion is equal to the weight of that Op's basic block.

The results in Figure 8 show that this heuristic has the best overall performance. It provides 3% higher speedup than the dependence height heuristic for the 4U machine model, and about 1% higher speedup for the 8U model. There is greater improvement for the 4U model because the global weight heuristic targets heavily executed Ops aggressively, and this tactic pays off more when resources are less available.

Note that the global weight heuristic avoids the flaw of the exit count heuristic, at the expense of relying on profile information. By targeting those areas of code with high weight, the global weight heuristic selects the proper branch destinations of the multiway branches and gives them highest priority.

Weighted count treegion scheduling

The weighted count heuristic combines the global weight and exit count heuristics. The focus on profile weight of the former heuristic is tempered by the restrictions on speculation imposed by the latter. The primary sorting criterion is profile weight, to give heavily executed Ops an advantage. Ops with the same weight are sorted by exit count to avoid overaggressive speculation. Ops with both the same weight and exit count are sorted by dependence height.

The hope for this heuristic is that the flaw in the exit count heuristic will be avoided by prioritizing Ops primarily by weight. However, it does not improve over the dependence height heuristic as much as the simpler global weight heuristic. In particular, the speedup of **vortex** is reduced.

The reason for the slight degradation in performance is again due to sorting by exit count. By giving highest priority to Ops early in a treegion, the heuristic naturally retires exits higher in the treegion earlier. However, the most taken exits may be lower in the treegion, and thus are delayed. The sorting by weight that the weighted count heuristic performs partially solves this problem, but when the weights in a treegion are equal or very close, and the exits near the bottom are taken most often, the performance degradation surfaces.

This effect is most noticeable in biased treegions that are also *linearized*, i.e., that contain a single execution path. An example of a linearized treegion in **vortex** is shown in Figure 10. Since the weights of the blocks are the same, the weighted count heuristic behaves like the exit count heuristic, and so the only taken exit at the bottom of the treegion is delayed. On the other hand, the global weight heuristic treats all of the blocks equally, and manages to schedule the taken exit in less time.

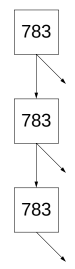


Figure 10: A linearized treegion. The block numbers indicate profile weight. The most frequently taken exit is at the bottom of the treegion; the weighted count heuristic instead focuses on the intermediate exits, which are never taken.

In summary, the dependence height and global weight heuristics produce the best overall speedup. The former heuristic is useful when profile information is unavailable or unreliable, while the latter can make use of reliable profile information. The exit count and weighted count heuristics suffers from some flaws; however, it remains to be seen whether they perform well in the face of profile variations, where they may preserve performance better.

4 Applying Tail Duplication to Treegions

Intuition suggests that large treegions are better for producing high-performance schedules than small treegions, since large treegions provide more opportunity for speculation. This section examines the use of tail duplication in expanding treegions and compares the schedules derived from them with superblocks.

Tail duplication heuristics

The fundamental goal of any attempt to expand treegions beyond their normal size is to eliminate merge points, since they delimit treegion boundaries. Tail duplication, a process applied during superblock [2] and hyperblock [14] formation, can be used in treegion formation to convert saplings (which are merge points) into a set of single entry blocks which can be absorbed into surrounding treegions. Before describing the algorithm for treegion formation with tail duplication, the heuristics applied to tail duplication will be described.

A code expansion limit is the primary heuristic. Uncontrolled code expansion will create large treegions at the expense of treegions following them in the code. Limiting code expansion creates more balanced treegions and keeps the code size and compile time under control. The number of distinct execution paths in a treegion (*path count*) is also limited for similar reasons. A large number of paths in a treegion will lead to increased interference between paths when competing for schedule slots, so limiting their number prevents some from being delayed too long.

Finally, a limit on a tail duplicated sapling's *merge count*, or number of incoming edges, is used. Tail duplicating a sapling with a large merge count is expensive in terms of code size and compile time. Therefore, saplings with high merge counts are not tail duplicated unless they are merge points with no successors in the CFG, such as function exits.

Modified treegion formation algorithm

```
treeform-td (CFG)
{
  Add top node(s) of CFG to unprocessed queue
  while unprocessed queue is not empty {
    Get first node in unprocessed list
    if node is already in a treegion continue
    Make a new empty treegion
    absorb-into-tree (treegion, node)
    do {
      if treegion path count exceeds limit break
      for each sapling of current treegion {
        if sapling is in another treegion
          continue
        if code expansion limit might be exceeded
          continue
        if sapling merge count exceeds allowed
          limit continue
        use this sapling
      }
      if sapling selected for tail duplication {
        if sapling is merge point
          tail-duplicate (sapling onto node
                          in tree)
          absorb-into-tree (treegion, duplicate)
        else
          absorb-into-tree (treegion, sapling)
      }
    } while some sapling is selected for tail
      duplication
    for each sapling of current treegion
      if sapling is not in a treegion
        add sapling node to unprocessed queue
  }
}
```

Figure 11: 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.

The treegion formation algorithm can be modified to use tail duplication; the result is shown in Figure 11. Tail duplication is performed on each treegion as it is formed, before the next treegion is created. A sapling which qualifies is found and tail duplicated. The duplicate is then absorbed into the treegion. This continues until no qualifying saplings remain. Often a sapling that has been tail duplicated will have only one remaining incoming edge. The algorithm takes this into account and absorbs such saplings directly.

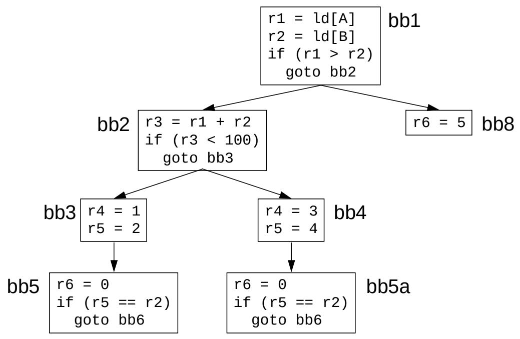


Figure 12: A treegion after tail duplication. The topmost treegion in Figure 1 is shown above after bb5 has been tail duplicated.

Figure 12 shows the topmost treegion of Figure 1 after basic block 5, originally a merge point, has been tail duplicated, and both the original block and its duplicate have been absorbed into the treegion. This process could be continued through the CFG in Figure 1 to the leaf node at basic block 9, resulting in one large treegion where each execution path through the original CFG has been converted into a unique path through the treegion. The entire CFG could then be scheduled as a single unit.

Dominator parallelism in treegion scheduling

A primary drawback of tail duplication is the introduction of redundant operations which may waste resources and extend execution time when speculated. In some cases the scheduler can take advantage of *dominator parallelism* to eliminate redundant Ops from the schedule.

Dominator parallelism is exhibited by identical Ops from different execution paths which are speculated into a block that dominates each Op [15]. When this occurs, all but one of the Ops can be safely eliminated from the schedule. The remaining Op will perform the operation for all paths concerned. The Op often is still speculative, as varying data dependencies may have prevented identical Ops from *all* paths from being speculated into a single dominator, but when a complete set of identical Ops has moved into a dominator, the single Op can be made non-speculative. Dominator parallelism has been used and discussed in both research and commercial contexts [16], [17].

The treegion scheduler can easily detect dominator parallelism between tail duplicated Ops. Because of the treegion's structure, any block in a treegion dominates all blocks below it. Ops speculated upwards are always speculated into dominators. Therefore, if a tail duplicated Op A' is speculated into a block where one of its duplicates A'' is already scheduled, A' can be eliminated.

As a simple example, refer to basic blocks 5 and 5a in Figure 12. The operation `r6 = 0` in both blocks may be speculated into basic block 2 (or 1). If this occurs, then only one copy of the Op needs to be scheduled and the work will be done for both execution paths.

Comparison with superblocks

Experiments were run to show the effects of tail duplication and dominator parallelism on treegion scheduling. The results are compared to superblock scheduling[2]. In these experiments, the code expansion limit is set to 2.0 and 3.0 times the original code size per treegion. The merge count limit is held at four and the path count limit at twenty.

Tables 3 and 4 present statistics on the characteristics of the superblocks and treegions with tail duplication formed by the LEGO compiler. Figure 4 shows the sizes of the regions formed. For most of the programs, treegions contain more basic blocks and Ops than superblocks. This is an intuitive result, since treegions consider multiple paths. Table 4 shows the amount of code expansion that occurs when forming superblocks and treegions. Treegions experience more code expansion than superblocks, another intuitive result, since tail duplication can occur along multiple paths within a treegion. Overall, the amount of code duplication is moderate for both types of regions. This implies that the instruction cache behavior of the programs will not be dramatically affected by the increase in code size.

Figure 13 shows the results of treegion scheduling using the global weight heuristic versus superblock scheduling. For both machine models, the speedup of treegion scheduling exceeds that of superblock scheduling by 15% with a code expansion limit of 2.0 (actual code expansion 1.32), and by 20% with a code expansion limit of 3.0 (actual code expansion 1.44). Overall, just as with treegions without tail duplication versus SLRs, the additional paths and Ops available within a treegion enable the extraction of higher levels of ILP.

[2]Every attempt was made to produce superblocks within the LEGO compiler as described in the literature [2], [18], [19]; differences between the LEGO compiler and the IMPACT compiler are unavoidable, but minimized.

Program	**Region type**		
	sb	tree (2.0)	tree (3.0)
compress	1.26	1.34	1.62
gcc	1.14	1.32	1.43
go	1.21	1.33	1.40
ijpeg	1.15	1.26	1.38
li	1.20	1.26	1.31
m88ksim	1.19	1.34	1.49
perl	1.07	1.30	1.38
vortex	1.17	1.37	1.45
average	1.18	1.32	1.44

Table 3: Code expansion statistics. The data presented indicate the factor by which code size increased for superblocks (sb) and treegions (tree). The numbers in parentheses indicate treegion code expansion limits.

5 Related work

Previous work on non-linear regions have influenced the study of treegions. The most direct ancestor to this work is decision tree scheduling (DTS) [20]. DTS recurses down the paths of a decision tree (similar to a treegion) producing schedules. Guarded instructions, the predecessors of predicated instructions [10], occupy branch delay slots. Ops are prioritized by the weight of the execution paths running through them and by their position along the critical path, a heuristic similar to global weight. Although DTS shares the same spirit as treegion scheduling, the former addresses pipelined processors, while the latter is more concerned with ILP. Also, treegion scheduling has the advantage of using speculation to move operations above branches and thereby exploiting dominator parallelism.

The IBM VLIW architecture is based on a *tree-instruction* which contains a decision-tree set of operations. The machine evaluates the various conditionals and executes all necessary operations in a single cycle. The finite-resource global scheduling technique [21] schedules ordinary code into tree-

Benchmark	Region type					
	# regions		avg # bb		avg # Ops	
	sb	tree (2.0)	sb	tree (2.0)	sb	tree (2.0)
compress	19	87	5.263158	5.195402	31	35.632184
gcc	3471	15186	5.576491	6.152575	32.033708	41.123864
go	1644	3280	3.753041	5.608232	24.63017	39.246646
ijpeg	347	1575	3.962536	4.795556	26	37.392381
li	180	1053	4.366667	4.584995	23.7	30.91453
m88ksim	129	1483	5.844961	6.923803	72	48.937964
perl	144	3527	6.659722	6.198469	38.659722	42.955486
vortex	184	1175	9.054348	7.724255	74.945652	72.055319

Table 4: Superblock and treegions with tail duplication statistics. The data described, from left to right, are total region count, average basic block count per region, and average number of Ops per region. 'sb' indicates a superblock, 'tree' indicates a tree.

instructions with help from a modified version of percolation scheduling [22]. The degree of speculation (DOS) heuristic controls how operations are moved through and between tree-instructions.

Hyperblocks [14] are an extension of superblocks that contain predicated code. Multiple paths are scheduled together, and predication ensures that only instructions on the taken path write back their results. Trace scheduling-2, or TRACE-2 [17], performs scheduling across a *cluster* of basic blocks. This allows the technique to detect dominator parallelism and to generate compensation code as scheduling proceeds. A speculative yield function contributes to prioritizes the operations. TRACE-2 allows merge points, but requires some scheduling complications treegion scheduling avoids.

6 Conclusion

This paper presented treegions and treegion scheduling. A treegion is a profile-independent tree-shaped subgraph of a program CFG. The tree topology may contain multiple paths of control. To an instruction scheduler, this presents greater opportunities for utilizing processor resources and more operations for speculative execution. The process of scheduling the operations in a treegion is termed treegion scheduling. Four scheduling heuristics were presented that can be used with treegion scheduling: dependence height, exit count, global weight, and weighted count. The use of tail duplication to expand the scope of a treegion and the exploitation of dominator parallelism to increase schedule inefficiency were also discussed. Treegion scheduling using the various heuristics and both with and without tail duplication were compared to scheduling with three types of linear regions – basic blocks, simple linear regions, and superblocks. The resulting performance is highly dependent on the heuristic used. The heuristic that performs best proved to be global weight, and treegion schedules obtained with tail duplication and dominator parallelism performs 12% and 20% better than SLRs and superblocks, respectively. Based on these results, it can be concluded that treegions are a promising alternative to more traditional, linear regions.

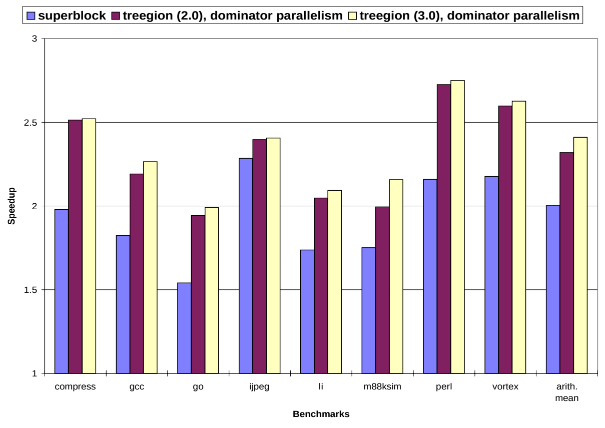
There are several items of interest for future work in treegion scheduling. First, we would like to investigate the performance of treegion schedules across different sets of inputs, to see the effects of profile variations using the various heuristics, and on dynamically scheduled processor models. Second, this study did not employ any software pipelining techniques, which would surely improve the performance of loops in the code [21]. Third, we are currently investigating the benefits of integrating code in the form of if-then and if-then-else statements into treegions. The serialization of code using predication as in hyperblocks [14] is an alternative to using tail duplication to eliminate merge points. We also plan to compare the tradeoffs between hyperblocks and treegions directly and to evaluate the merits of predication versus speculation for scheduling. Work on a complete implementation of hyperblocks in the LEGO compiler is currently underway.

Acknowledgements

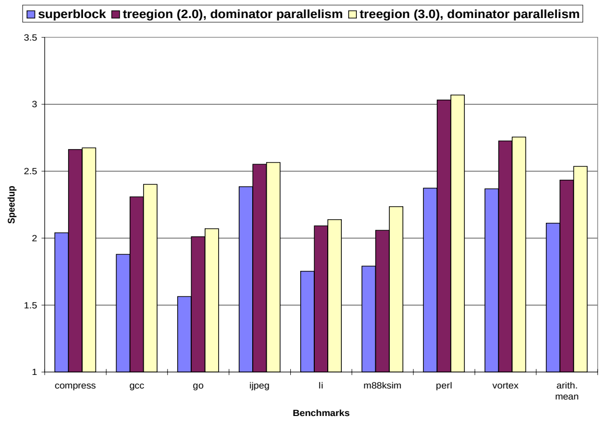
This work was supported by IBM, Hewlett-Packard, Intel, and the National Science Foundation under grants MIP-9625007 and GER-9454175.

References

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



a) 4U machine model



b) 8U machine model

Figure 13: Results for global weight tail duplicated treegion scheduling.

[2] 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.

[3] 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.

[4] B. L. Deitrich and W. W. Hwu, "Speculative hedge: regulating compile-time speculation against profile variations," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [23].

[5] 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. Microarchitecture* [23].

[6] S. Banerjia, W. A. Havanki, and T. M. Conte, "Treegion scheduling for highly parallel processors," in *Proc. Euro-Par'97*, (Passau, Germany), Aug. 1997.

[7] W. A. Havanki, "Treegion scheduling for VLIW processors," Master's thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, July 1997.

[8] R. Johnson and M. Schlansker, "Analysis techniques for predicated code," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [23], pp. 100–113.

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

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

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

[12] 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.

[13] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *Computer*, vol. 22, pp. 12–35, Jan. 1989.

[14] 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. Microarchitecture* [24], pp. 45–54.

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

[16] R. Gupta and M. L. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *IEEE Trans. Soft. Engg.*, vol. 16, no. 4, pp. 421–431, 1990.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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. Microarchitecture* [24], pp. 55–71.

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

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

[24] *Proc. 25th Ann. Int'l Symp. Microarchitecture*, (Portland, OR), Dec. 1992.