Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors

Huiyang Zhou[1], Matthew D. Jennings[2], Thomas M. Conte[1]

[1]*Department of Electrical and Computer Engineering*
North Carolina State University
{hzhou, conte}@eos.ncsu.edu

[2]*BOPS Inc.*
Chapel Hill, NC 27514
MattJ@bops.com

Abstract

Global scheduling in a treegion framework has been proposed to exploit instruction level parallelism (ILP) at compile time. A treegion is a single-entry / multiple-exit global scheduling scope that consists of basic blocks with control-flow that forms a tree. Because a treegion scope is nonlinear (includes multiple paths) it is distinguished from linear scopes such as traces or superblocks. Treegion scheduling has the capability of speeding up all possible paths within the scheduling scope. This paper presents a new global scheduling algorithm using treegions called Tree Traversal Scheduling (TTS). Efficient, incremental data-flow analysis in support of TTS is also presented. Performance results are compared to the scheduling of the linear regions that result from the decomposition of treegions. We refer to these resultant linear regions as linear treegions (LT) and consider them analogous to superblocks with the same amount of code expansion as the base treegion. Experimental results for TTS scheduling show a 35% speedup compared to basic block (BB) scheduling and a 4% speedup compared to LT scheduling.

1. Introduction

Global scheduling using treegions [1,2] has been proposed to extract instruction level parallelism (ILP) at compile time. It consists of two phases: *treegion formation* and *treegion scheduling*. A treegion is a single-entry / multiple-exit nonlinear region that consists of basic blocks with control-flow forming a tree. Treegion formation does not depend on profile information, instead relying on the program's base control-flow structure with tail duplication [1,3]. This makes the treegion framework attractive for dynamic optimization, where different run-time behaviors can result in rescheduling without the need for reforming treegions. In the case of static treegion scheduling, instructions in multiple execution paths in a treegion are scheduled based on profiling information and other heuristics. Treegions provide more opportunities for ILP extraction as they provide a larger scheduling scope when compared to linear regions such as traces [4] or superblocks [3]. Treegion schedules can utilize high issue bandwidth by speeding up more than one path of execution simultaneously.

This paper extends the previous work on treegion scheduling [1,2]. First, a new treegion scheduling algorithm, *tree traversal scheduling* (TTS) is proposed to achieve higher performance by scheduling the block-ending branches as early as possible. TTS allows high resource utilization when multiple execution paths can benefit and reduces resource contention by resolving branches as early as possible. The benefit of resolving branches early is that all of the machine's execution resources are then available for each of the possible paths that result from the branch instruction. Efficient data-flow analysis is required for TTS as it results in a lot of code motion beyond basic block scope and operand renaming in support of speculation. Therefore, liveness and reaching definition analysis changes frequently but is expensive to re-compute at a procedural scope. Our approach to this problem is to use incremental adjustments to the data-flow analysis during code motion and operand renaming.

For performance analysis, we compare TTS scheduling with the scheduling of the linear regions that result from the decomposition of treegions. The decomposition of treegions into linear regions favors the more

frequently executed paths through the treegion. The most frequently executed path through the treegion makes up the first linear region and is removed from the treegion. The next most frequently executed path (from the remainder of the treegion) makes up another linear region, etc., until no paths remain. We refer to these resultant linear regions as Linear Treegions (LT) and consider them analogous to superblocks with the same amount of code expansion as the base treegion. Experimental results for TTS scheduling show that there is a 35% speedup compared to the basic block (BB) scheduling and a 4% speedup compared to LT scheduling.

Two architectural effects are also observed from the experiments:

(1) In order to fully take advantage of load speculation, the in-order processor pipeline should not stall at the execution/memory stage for a D-cache miss but stall only on the first use of the missing value at the dispatch/register read stage.

(2) Treegion scheduling has two effects on I-cache performance: fewer I-cache accesses due to fewer multi operations (mops) produced by treegion scheduling and higher I-cache and TLB miss rates due to the code expansion from treegion formation.

The remainder of the paper is organized as follows. Section 2 briefly describes the notion of the treegion and treegion formation. The TTS algorithm is discussed in Section 3 and the implementation issues for efficient data flow analysis are in Section 4. Section 5 describes the simulation environment and experimental results. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Treegion and Treegion Formation

A treegion contains of a tree of basic blocks, which is a subgraph of the control flow graph (CFG) for a program. Since the treegion depends only on the topology of the CFG, it does not change when profile information varies. This characteristic makes treegions favorable in run-time optimization environments such as dynamic optimization or dynamic re-compilation.

Figure 1 shows an example of a CFG and the treegion formation based on it. Here we use the same treegion formation algorithm in [1] and three treegions are constructed in the CFG. Since large regions are usually better for ILP extraction in scheduling than small regions, tail duplication [3] is applied as a treegion enlarging optimization. After unconditional branches are removed, the resulting treegion is shown in Figure 2. The trade-off for exposing ILP through treegion formation is the code-expansion that results from duplicates of BB6 and BB7. Other region enlarging optimization, such as branch target expansion, loop unrolling and loop peeling [9] can be used before the treegion formation.

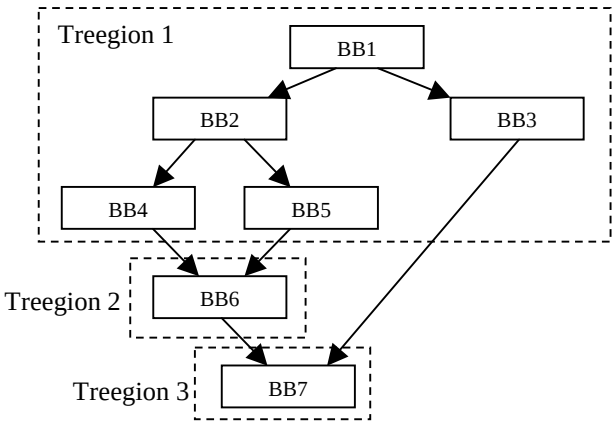


Figure 1. An example CFG and the treegions constructed (each dash line box denotes a treegion)

As each treegion contains multiple execution paths, there is more available ILP in treegions than single path regions. Treegions formation enables the scheduling of multiple paths simultaneously. The code-expansion side effect can be alleviated by compile-time transformations such as dominator parallelism [1] and the recombination of tail blocks.

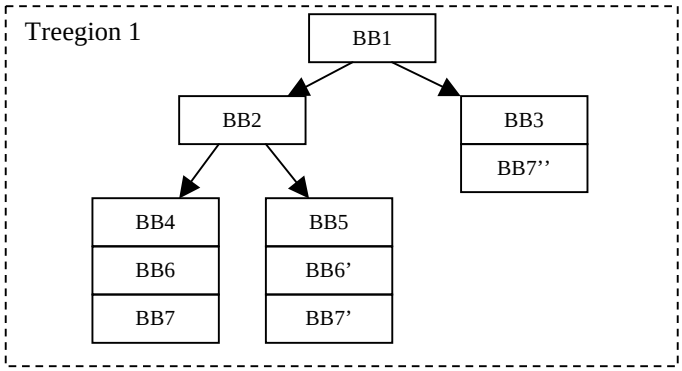


Figure 2. The treegion formed with tail duplication, where BB6' is a duplicate of BB6; BB7'and BB7'' are duplicates of BB7

3. Treegion Scheduling Algorithm: Tree Traversal Scheduling

The motivation of treegion scheduling is for the resultant schedule to speedup every execution path through the treegion. As treegions consist of multiple paths of execution, it will be necessary to prioritize the speculation of instructions. Speculative instructions originating from different paths will compete for the limited machine resources and execution frequency based on profile information is used to prioritize speculative instructions along the more frequently executed paths. TTS also limits the extent of speculation by prioritizing branch instructions so that branches will be resolved as early as possible. Executing branches early reduces contention for machine resources, as each path out of the branch instruction will see a full set of machine resources. TTS enables the execution of branches as early as possible by allowing branches to be scheduled as early as their data-dependencies allow, even if that results in downward code motion. The TTS algorithm consists of the following two steps:

Step1: Construct the control/data dependence graph and perform instruction ordering.

For each treegion prioritize instructions according to:

a) execution frequency,

b) exit count heuristic to resolve ties from (a), and

c) data dependence height to resolve ties from (b).

In this ordering, heuristic (a) prioritizes the most frequently executed paths. The exit count heuristic (b), which is adapted from the helped count priority function of speculative hedge [6], is the number of exits that follow the instruction in the treegion. This heuristic gives priority to instructions in the basic blocks that help more exits. Instructions in the root block of the treegion have the highest priority as it is on the most often executed path and helps every exit in the treegion.

Step2: Scheduling the instructions in the treegion.

First we define the following terms in our discussion:

- *current_op* denotes the instruction that has just been scheduled.
- *current_block* denotes the basic block in which current_op is scheduled.
- *candidate_op* is the instruction being considered for scheduling.

- *current_cycle* is the cycle for which *candidate_op* is considered for scheduling

Tree traversal scheduling is a cycle scheduling approach. Initially, current_op is set to a default value and current_block is set as the root block of the treegion. At each cycle candidate_op is selected from instructions according to the order determined in **Step1**. Candidate_op needs to satisfy two criteria: a) candidate_op is dominated by current_block (i.e., candidate_op is either in current_block or in a child block of current_block), and b), the source operands of candidate_op are ready before current_cycle. After candidate_op is selected, we first check to make sure there are machine resources available to schedule candidate_op in current_cycle. If the scheduling of candidate_op in current_cycle is speculative, there are additional constraints to consider. First, speculative function calls and store instructions are not allowed. Next, destination operand renaming may be required to support speculation. Finally, branch candidate_ops may require downward code motion to support the *speculation* of the branch. The *speculation* of branch candidate_ops results in the formation of a multiway branch instruction in current_block. For example, in Figure 3a (instruction semantics in this paper are as follows: Operation Destination operand, Source operand 1, Source operand 2), instructions i and i+1 are not scheduled when the branch is being scheduled. These instructions are moved downwards into their child blocks (shown in Figure 3b) as a result of the branch *speculation*.

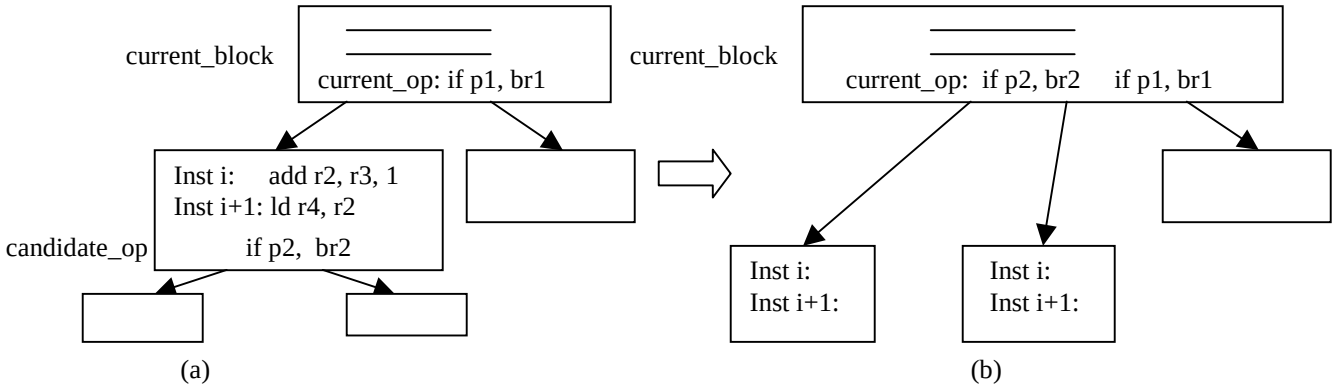
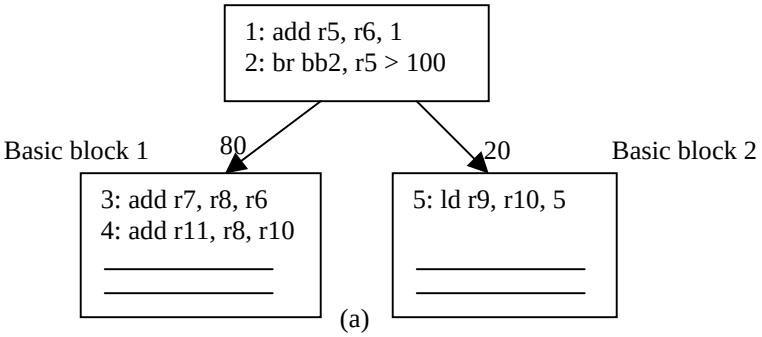


Figure 3. Scheduling a branch instruction when it is in the child block of the current block. (a) the treegion before the scheduling, (b) the treegion after the scheduling

The scheduling of block-ending branches as early as possible is an important aspect of **Step2**. For the case when many instructions in the child blocks are ready to be scheduled speculatively, we need to make sure that speculation will not be over aggressive and cause a delay in executing the branch instruction. This problem is illustrated as an example shown in Figure 4.

In Figure 4a, we assume registers r6, r8, and r10 are ready at current_cycle (cycle n) and that the instructions have been ordered in **Step1** as instructions 1, 2, 3, 4, and 5. Figure 4b shows the schedule result if we apply list scheduling on a 2-way issue machine model with one alu/branch unit and one alu/load unit. Since instructions 1, 3, 4, and 5 are ready at cycle n, the list scheduler will schedule these instructions ahead of instruction 2. But, any delay in resolving this branch will delay both paths following it. Instead of scheduling the speculations right away, the TTS algorithm ensures that speculation is not too aggressive by selecting candidate_op every cycle in the instruction list based on the predetermined order. Figure 4c shows the result of the TTS scheduling algorithm. It can be seen that the TTS algorithm schedules the branch instruction at cycle n+1, prior to instructions 4 and 5. Comparing the average execution time (assuming the latency for add/branch is 1 cycle and load latency is 2 cycles for a cache hit), the schedule in Figure 4c will have 15% speed up over the scheduling in

Figure 4b. Note in Figure 4c that after the branch is scheduled, TTS will move on to the next basic block, which may be basic block 1 or 2. The schedule at cycle n+2 is parenthesized to show this uncertainty.



(a)

Sch_Time	ALU/BR	ALU/LD
Cycle n:	add r5,r6,1	add r7, r8, r6
Cycle n+1:	add r11, r8, r10	ld r9, r10,5
Cycle n+2:	br bb2, r5>100	

Average execution time: 4 cycles

(b)

Sch_Time	ALU/BR	ALU/LD
Cycle n:	add r5,r6,1	add r7, r8, r6
Cycle n+1:	br bb2, r5>100	add r11,r8,r10
Cycle n+2:	(ld r9, r10,5)	

Average execution time: 3 + 2*0.2 = 3.4 cycles

(c)

Figure 4. (a) An example for treegion scheduling, (b) the scheduling result using list scheduling, (c) the scheduling result using TTS.

Based on this observation of branch scheduling, conceptually we can view the TTS algorithm as applying list scheduling to a sequence of basic blocks that form a path of tree traversal, as shown in Figure 5.

TTS algorithm:

1. For a treegion, sort the basic blocks according to a depth-first traversal order with the child block selected with highest execution frequency.
2. Start list scheduling at the root basic block.
3. During the scheduling of a basic block, consider speculation for instructions dominated by this basic block.
4. After scheduling the block-ending branch, traverse to the next basic block and go back to 3.

Figure 5. A conceptually simple way to view TTS algorithm

There are two aspects that we can draw from the way to view TTS in Figure 5. The first is that by use of speculation on the dominated instructions the TTS algorithm achieves high resource utilization and speeds up multiple execution paths with priority given to the most often executed paths. This is an advantage over linear scheduling methods such as trace scheduling [4] and superblock scheduling [3]. Secondly, the TTS algorithm reduces resource competition since it only attempts to schedule the instructions that are dominated by the current basic block. In the example shown in Figure 4, when scheduling basic block 1 (i.e., when current_block is basic block 1), instructions from basic block 2 will not be considered. This is an advantage of TTS over the hyperblock scheduling [10].

4. Efficient Data Flow analysis in the TTS algorithm

The TTS scheduling algorithm makes extensive uses of the data flow analysis, specifically reaching definitions and live-variable analysis [18]. As this data flow analysis is computationally expensive at procedural

scope, we need to be careful so that compile time does not become exceedingly long. In the TTS algorithm, there are several situations that make previous data flow analysis calculations obsolete. Those situations include: the change in liveness during speculative code motion, the change in reaching definitions when a copy operation is inserted during a renaming process, and the change in both liveness and reaching definitions in code downward motion. One obvious solution is to recalculate data flow information whenever there is a possible change during the scheduling phase. Unfortunately, this approach results in far too much compile time (analysis would be repeated thousands of times for benchmarks such as perl, vortex and gcc).

In the TTS algorithm, speculative code motion happens frequently for wide issue processors. To alleviate the requirement of recalculating data flow analysis with each speculative code motion, we incrementally update liveness and reaching definitions in nearly all cases. The incremental updates are not always precise, but are conservative for the scheduling problem.

In the case of the speculation without renaming, as shown in Figure 6, r1 is not live along control edge 1 or edge 2. After the candidate_op (marked with *) is moved upwards, the liveness along edge 1 is not changed and r1 is included in the liveness along edge 2. Such processing is simple: liveness is extended upward for the destination operand and added to each edge that the instruction traverses during speculation. This update is conservative as r2 and r3 may no longer be live across edge 2. The conservative liveness across edge 2 may cause unnecessary renaming, when there is a definition of r2 or r3 speculated through edge 1. As we show later, most renaming cases are actually very simple to handle and take little compile time. Such additional renaming will affect compile time very slightly.

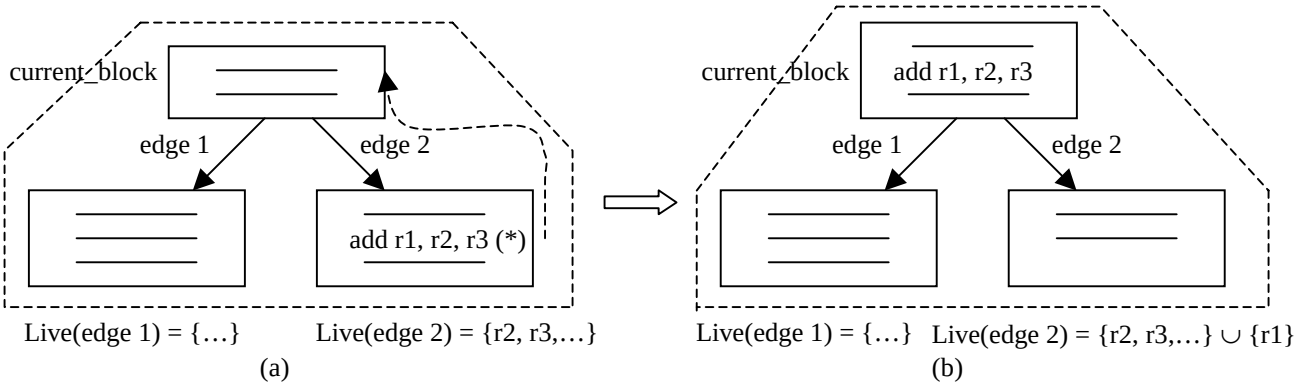


Figure 6. Liveness analysis of speculation without renaming (region enclosed in dashed line is the treegion under consideration) (a) the treegion before the current speculative scheduling of candidate_op (marked with *); (b) the treegion after the current speculative scheduling of candidate_op

If the destination operand of the candidate_op happened to be live along edge 1, then destination operand renaming must be done before the instruction can be speculated. To investigate incremental updates to data flow analysis in speculations that require renaming, we divide renaming into three categories according to the scope of renaming: *local rename*, *rename with copy*, and *global rename*. Note that although the speculative code motion is used to explain the incremental data flow analysis, the same discussion holds for the renaming cases for non-speculative code motion as well.

Local rename represents cases where the renaming scope is contained within a treegion. The characteristics that identify a local rename case are: one definition with one or more uses in the treegion and the operand is not live out of the treegion. Since treegions have a large instruction scope and include multiple execution paths, local rename is the most common case for TTS. Figure 7 shows an example of local renaming, where the

liveness of r1 along edge 1 requires the definition of r1 on the other path to be renamed before being speculated. After the definition and all the uses it reaches are renamed to rk (a new virtual register), the speculation can be performed. Since rk is a new virtual register, it has no other definitions in the program. So, we do not need to add any liveness for this operand. Also, for the same reasons as discussed before, we keep the conservative liveness of r1 along edge 3 and edge 4. In summary, in the case of local rename, all that needs to be done is renaming of the definition and all the uses it reaches (with no change to liveness information).

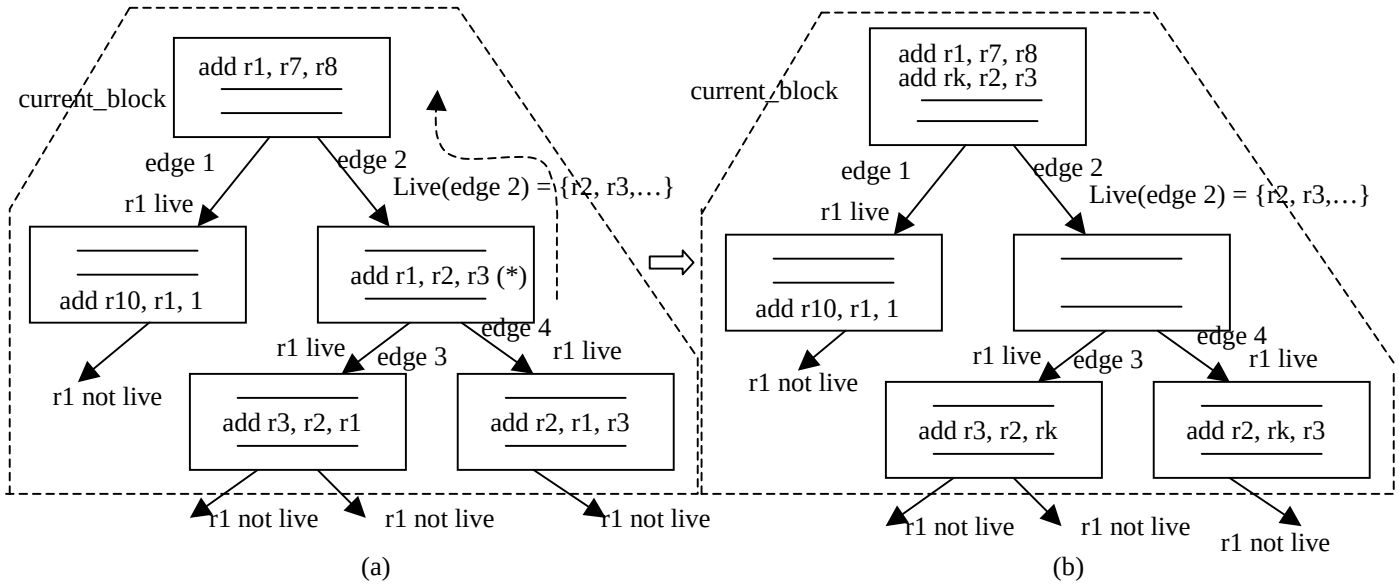


Figure 7. Liveness analysis of the speculation with local renaming (region enclosed in dashed line is the treegion under consideration) (a) the treegion before the current speculative scheduling of candidate_op (marked with *); (b) the treegion after the current speculative scheduling of candidate_op

Rename with copy is used for the cases where the operand to be renamed is live along multiple exit paths and there is a *merge problem* associated with the operand. A *merge problem* occurs when two definitions in the same treegion merge into a use and one definition dominates the other. The use in this case will be outside of the treegion scope, as treegions do not contain merge points. An example is shown in Figure 8, where both definitions of r1 inside the treegion merge into a use outside of the treegion. The definition in current_block dominates the definition in the load instruction. In rename with copy, the destination operand of candidate_op is renamed to rk and all the uses it reaches within the treegion are also renamed to rk. The instruction is speculated and a copy instruction is inserted at its original location (with the original destination operand and with rk as the source operand). Rename with copy is only applied when there are uses in the treegions that can benefit from the speculation of candidate_op. Once again, rk is a new virtual register and there are no other definitions in the program. Liveness does not change during rename with copy. However, reaching definitions need to be patched as the new copy instruction splits a previous def/use web. This update to reaching definitions is done with incremental changes and is precise.

Global rename is used for renaming def/use webs that span beyond the scope of the treegion and do not have the *merge problem*. Figure 9 shows one example of global renaming. Considering data flow analysis in the case of global renaming, reaching definitions do not change but liveness does. When global renaming affects another definition in the same treegion, we need to recalculate liveness immediately to ensure correctness for the current treegion scheduling scope. The example in Figure 9 is such a case. When global renaming only affects the

definition in the candidate_op being speculated, recalculating procedural level liveness can be delayed until after scheduling of the current treegion scope is complete.

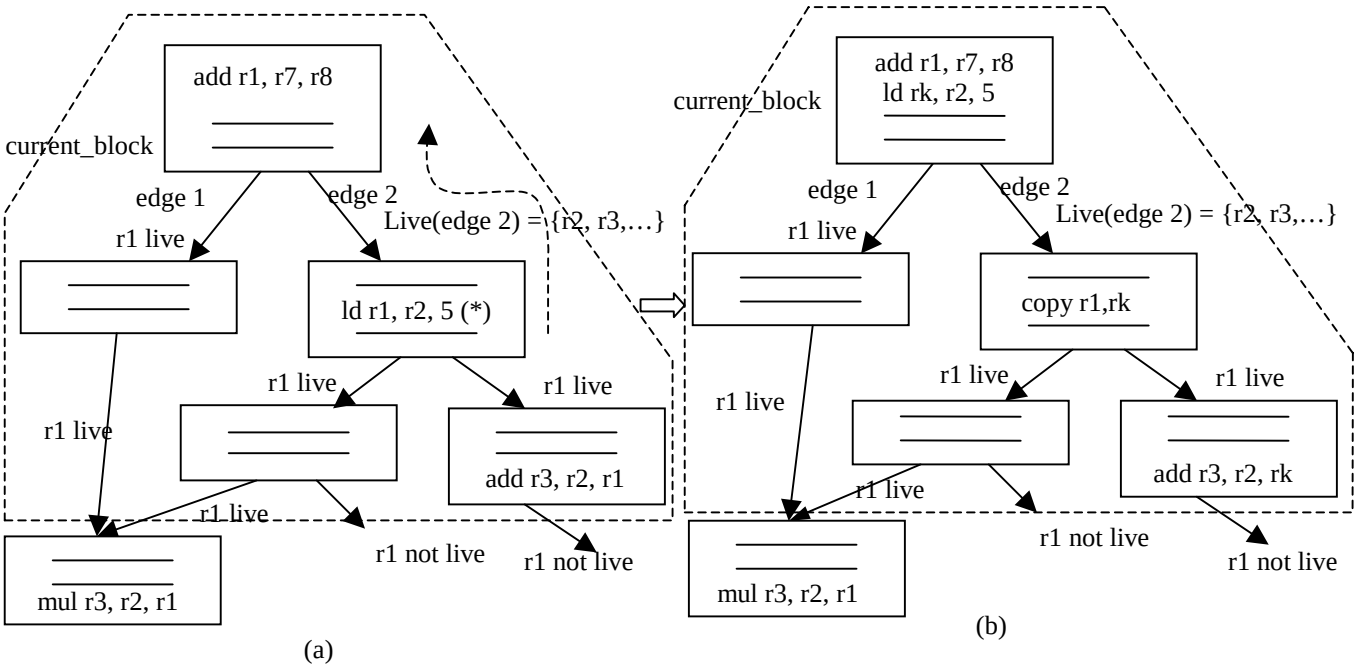


Figure 8. Liveness analysis of speculation with renaming with a copy (region enclosed in dashed line is the treegion under consideration) (a) the treegion before the current speculative scheduling of candidate_op (marked with *); (b) the treegion after the current speculative scheduling of candidate_op

Liveness information is used for the downward code motion that results from the early scheduling of block-ending branches. Downward code motion does not result in operand renaming. In downward code motion operand defining instructions are only moved down across an edge if its destination is live on that edge. Conversely, store instructions and subroutine calls are always duplicated and moved down to each successor block. An example is shown in Figure 10, where the branch instruction marked with '*' is the candidate_op and the instruction (add r1, r2, r3) is unscheduled. Since r1 is not live along edge 1, the instruction is not inserted into Block 1. Removing the destination operand from the liveness along edge 2 and adding the source operands to the liveness along edge 2 patches the liveness information. In downward code motion, the unscheduled instructions are processed in reverse program order to maintain the program semantics. Reaching definitions also need updates for downward code motion. Linking the original instruction's reaching use(s) with the downward moved instruction that dominates it does this precisely.

5. Experiment Methodology and Results

To evaluate the performance of the TTS algorithm, we implemented the TTS algorithm in the LEGO compiler [14], a research ILP compiler developed by Tinker Research Group at N. C. State University. The SPECint95 benchmark suite is used in the experiments. All programs are compiled with classic optimizations using the IMPACT compiler from University of Illinois [3] and converted to Rebel textual intermediate representation using the Elcor compiler from Hewlett-Packard Laboratories [20]. Then, the LEGO compiler is used to profile code, form treegion and schedule the instructions using the TTS algorithm. After the instrumentation for trace-based timing simulation, the scheduled Rebel code is converted to C code. Finally, a

trace-based timing simulation is run together with the functional simulation to ensure the correctness of the program and obtain the simulation results. In our experiments, all benchmarks in SPEC95int suite run to completion.

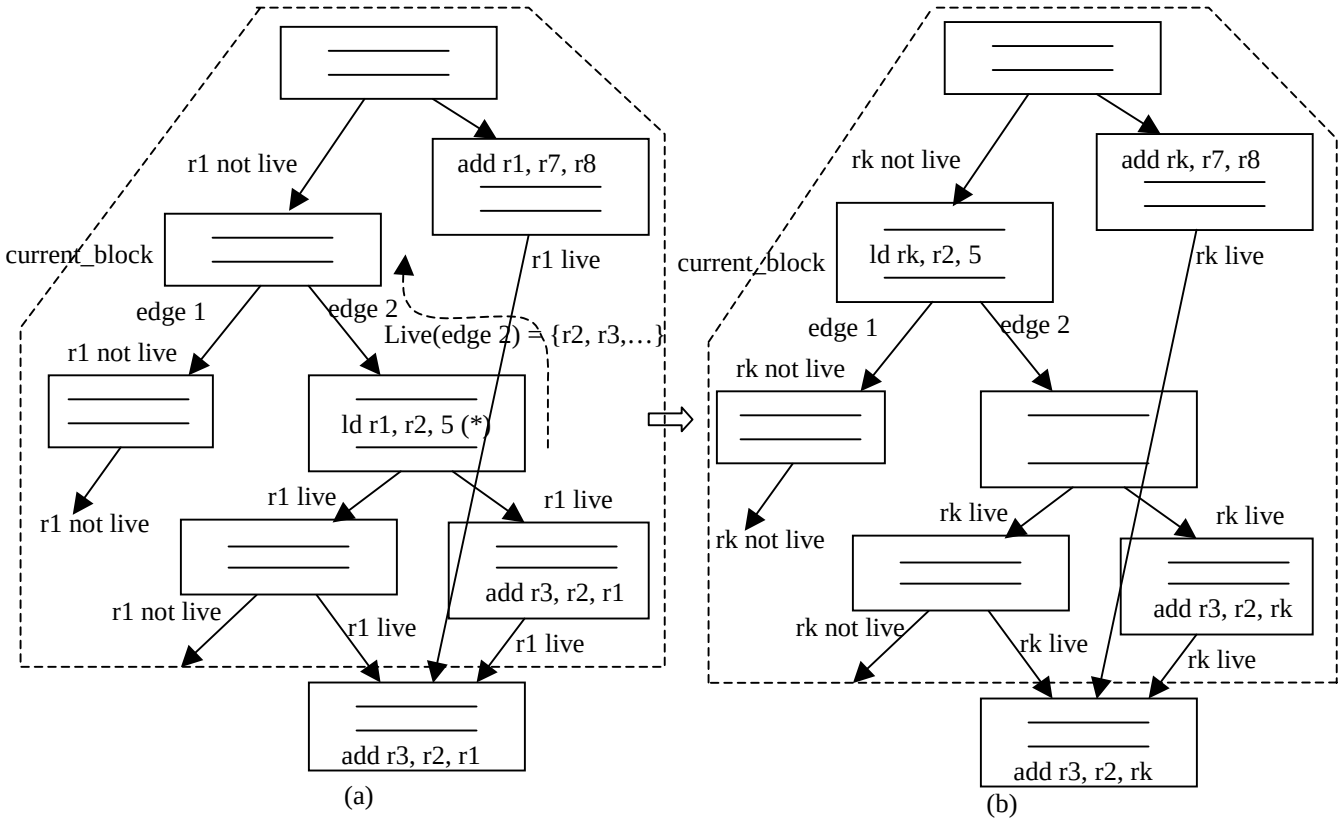


Figure 9. Liveness analysis of speculation with global renaming (region enclosed in dashed line is the treegion under consideration) (a) the treegion before the current speculative scheduling of candidate_op (marked with *); (b) the treegion after the current speculative scheduling of candidate_op

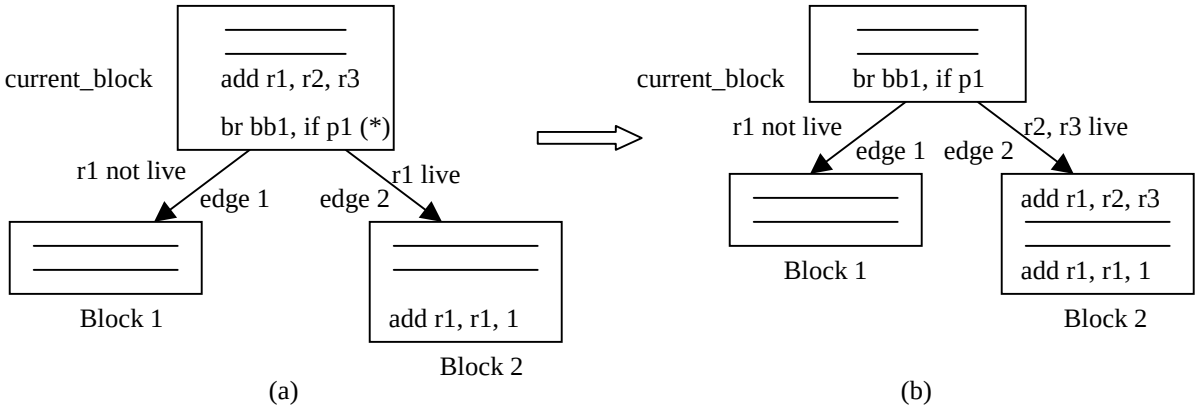


Figure 10. Liveness analysis for code downward motion (a) the treegion before the scheduling of candidate_op (marked with *); (b) the treegion after the scheduling of candidate_op

In the experimentation, the SPEC95int benchmarks are scheduled for an 8-issue VLIW machine model based on the Hewlett-Packard Laboratories HPL_PD architecture [7,8]. In this machine model, all function units are fully pipelined and all operations have a one-cycle latency except for load (two cycles for a hit), floating point add (two cycles), floating point subtract (two cycles), floating point multiply (three cycles), and floating point

division (three cycles). Then in trace simulation, the same machine model is used in the execution core with the I-cache, D-cache and a branch predictor. The detailed specification of the processor model used in simulation is shown in Table 1.

In our experiments, BB scheduling and superblock scheduling are implemented in our treegion framework and results are compared to tree traversal scheduling (TTS). In BB scheduling, list scheduling with renaming support is the major scheduling technique and treegion formation is not applied. For superblock scheduling in our treegion framework, each superblock is formed by the decomposition of treegions into linear regions, which we call *linear treegions (LT).* Figure 11 shows an example of superblock formation based on a treegion, where four superblocks (or LT) are formed from one treegion based on profile information. After superblocks have been formed, the TTS algorithm is then performed to schedule each superblock (or LT). In our experiments, LTs are used as analogous to superblocks and are compared to treegion scheduling with the same amount of initial code expansion.

Table 1. The specification of the machine model used in the experiment

	Specification
Execution	Dispatch/Issue/Retire bandwidth: 8; Universal function units: 8; Operation latency: ALU, ST, BR: 1 cycle; LD, floating-point (FP) add/subtract: 2 cycles; FP multiply/divide: 3 cycles
I-cache	Compressed (zero-nop) and two banks with 32KB each bank (Direct Mapped) [19]. Line size: 16 operations with 4 bytes each operation. Miss latency: 12 cycles
D-cache	Size/Associativity/Replacement: 64KB/4-way/LRU Line size: 32 bytes Miss Penalty: 14 cycles
Branch Predictor:	G-share style Multi-way branch prediction [15,16] Branch prediction table: 2^{14} entries; Branch target buffer: 2^{14} entries/8-way/LRU Branch misprediction penalty: 10 cycles

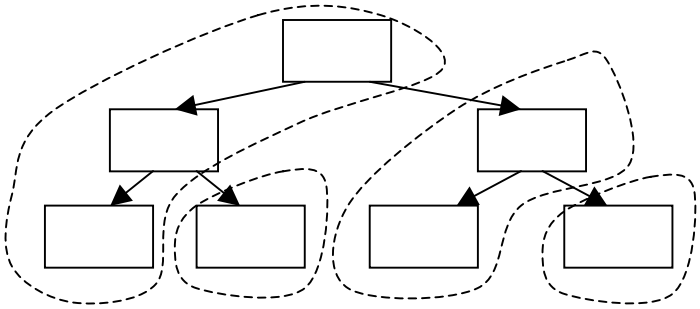


Figure 11. Superblock formation in a treegion (i.e. Linear treegion formation) (each trace enclosed in dashed lines is one superblock)

One ideal machine model and one realistic machine model are used to examine the speedup effects. Ideal instruction and data caches and perfect branch prediction are assumed for the ideal machine model. Figure 12 shows speedups for TTS and LT scheduling over BB scheduling using the ideal machine model. Figure 12 shows both TTS and LT scheduling with significant speedups over BB scheduling, 35.3% and 31.3%

respectively in average. Treegion scheduling also shows up to 9% speedup over linear treegion scheduling and 4% speedups in average.

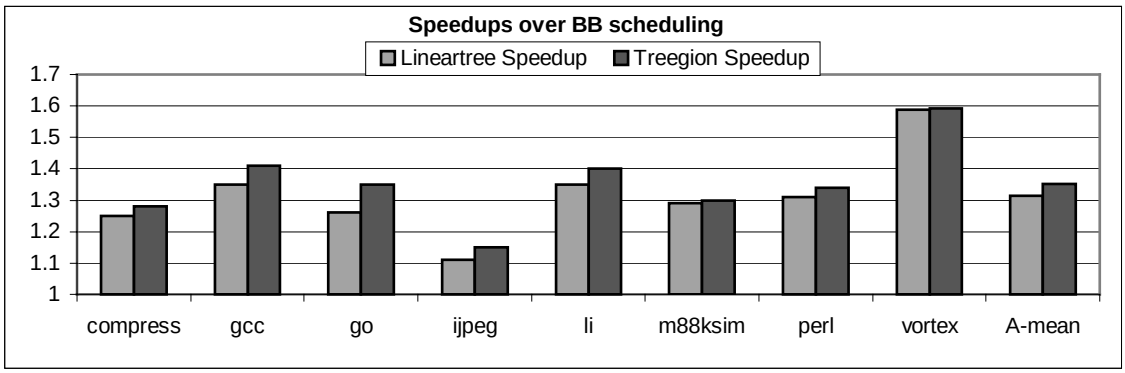


Figure 12. Speedups of treegion scheduling and linear treegion scheduling over BB scheduling on an ideal processor model (ideal branch prediction and ideal caches)

If we take cache effects and branch prediction effects into consideration with cache miss latency and branch miss prediction penalty set as in Table 1, the speedups of TTS and LT scheduling over BB scheduling are as shown in Figure 13. From Figure 13, it can be seen that TTS and LT scheduling show 28.5% and 25.8% speedups over BB scheduling. Reduced speedups for both TTS and LT scheduling are primarily the result of D-cache and I-cache performance.

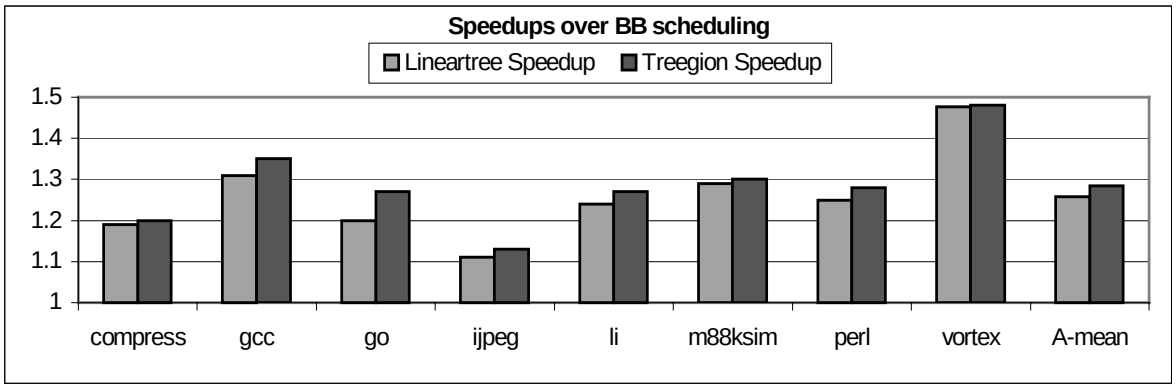


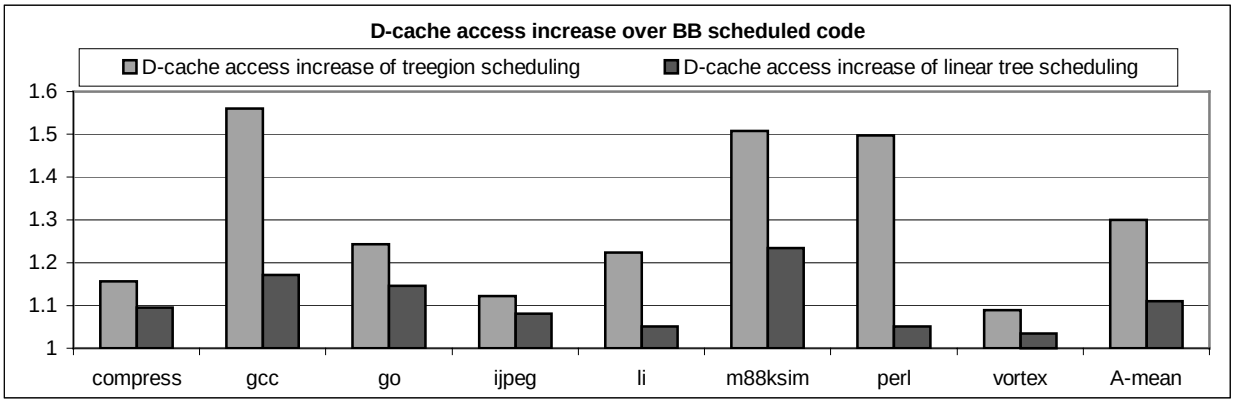
Figure 13. Speedups of treegion scheduling and linear treegion scheduling over BB scheduling on a realistic processor model

Speculative code motion, especially load speculation, is used extensively in both TTS and LT scheduling. The increased number of speculative loads results in an increased number of data cache (D-cache) accesses. The new D-cache accesses have a negative effect on the performance of TTS and LT scheduling, as the execution model is an in-order pipeline that stalls whenever there is a D-cache miss. As TTS enables more speculative loads (from multiple paths), the negative effect is more significant that for LT scheduling.

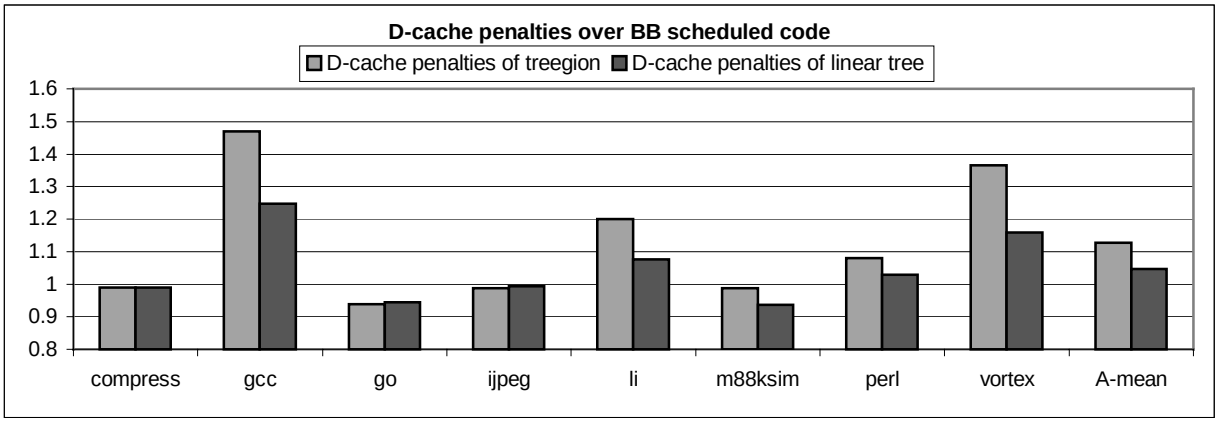
Figure 14a shows on average a 30% increase in D-cache read accesses for TTS over BB scheduled code. D-cache miss penalties are shown in Figure 14b. Although the D-cache miss rate is smaller for TTS (miss rate for TTS: 1.19%; miss rate for LT: 1.27%; miss rate for BB: 1.33%), TTS code still has the largest D-cache access penalties (13% more than BB scheduled code). This result shows that in order to take full advantage of load speculation, it is important for the execution pipeline not to stall at the execution/memory stage for each load miss but to stall only on the first use of the missing value at the dispatch/register read stage.

Code expansion due to treegion formation usually has negative effects on I-cache performance. In Figure 15, the code size increase for TTS is shown for each benchmark. Note that in examining the code size effect we only compare TTS with BB scheduling since the LT scheduling and TTS have similar code expansion.

From Figure 15 it can be seen that TTS causes an average static code size increase of 172%. (i.e., the static code size of TTS code is 72% more than the size of BB scheduled code). Since TTS usually combines more operations into its multi-ops than BB scheduling, TTS code will have fewer multi-ops than BB scheduled code. Fewer multi-op fetches result in fewer dynamic I-cache accesses, as seen in Figure 16. On average there are 15% fewer I-cache accesses for TTS. The larger size of the multi-ops for TTS and the larger overall code size will put upward pressure on I-cache miss rate, TLB miss rate and I-cache access penalties, although there are more spatial localities of the TTS code than BB scheduled code. From our simulations, the I-cache access penalties of TTS code is 12% greater than for BB scheduled code.



(a)



(b)

Figure 14. (a) The D-cache access increase of treegion scheduled code and linear treegion scheduled code over BB scheduled code, (b) The D-cache access penalty of treegion scheduled code and linear treegion scheduled code over BB scheduled code

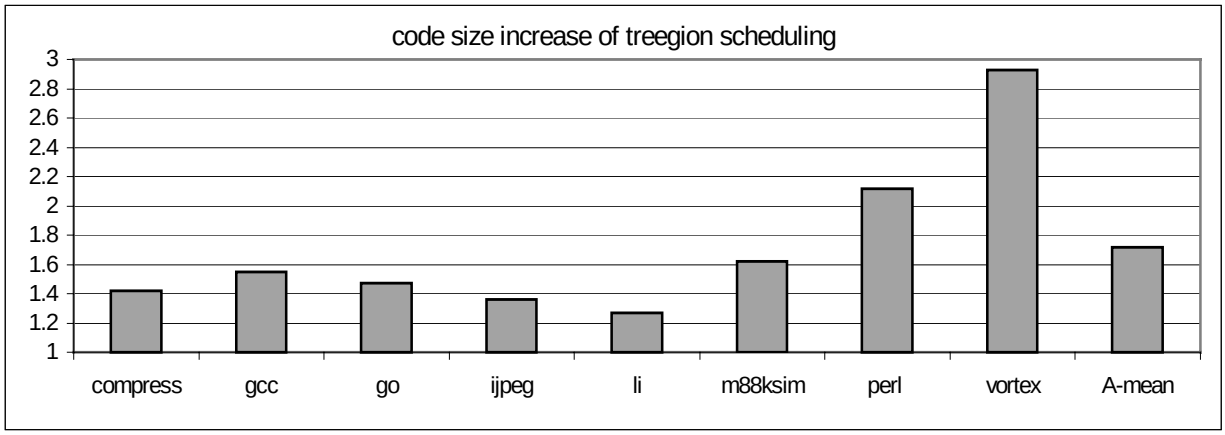


Figure 15.The treegion scheduled code size over BB scheduled code size

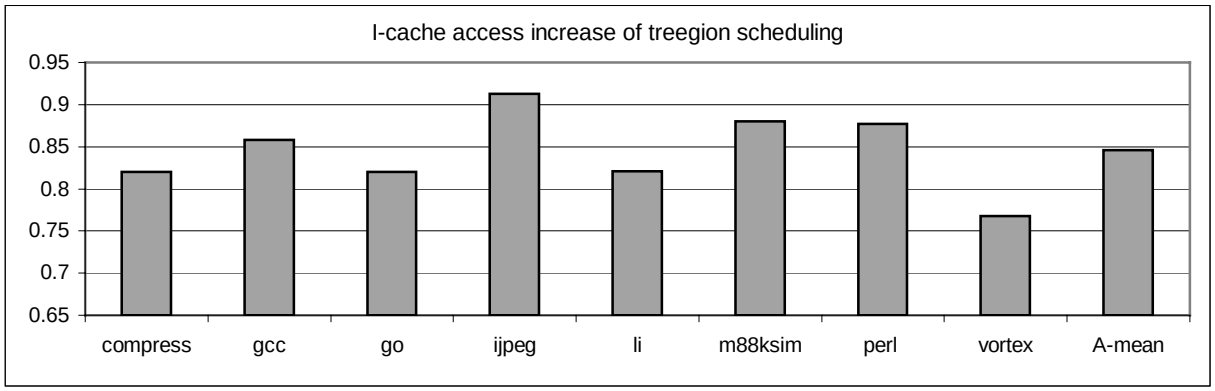


Figure 16. The I-cache access of treegion-scheduled code over the I-cache access of BB scheduled code

6. Related Work

As pointed out in [1], previous work on non-linear regions has influenced our work on treegion scheduling. Among them, decision tree scheduling (DTS) [11] can be viewed as the most direct ancestor. Based on VLIW tree instructions, a finite-resource global scheduling technique [12] was developed with the help of modified percolation scheduling [13]. Tree traversal scheduling targets wider issue ILP architectures such as EPIC [21] style machines.

Tree traversal scheduling also has many similarities with other popular global scheduling techniques, such as trace scheduling [4,5], superblock scheduling [3] and hyperblock scheduling [9,10]. As mentioned previously, the heuristics used for the candidate operation selection in the TTS algorithm were motivated by the profile variation studies on superblocks [6]. In addition, superblock scheduling also uses tail duplication as an optimization step prior to scheduling. Compared with trace scheduling and superblock scheduling, TTS has the potential to speed up multiple paths simultaneously and benefits from the larger scope of instructions in treegions. Hyperblocks use predication to schedule multiple paths together but can suffer from resource contention as if-conversion linearizes the control flow. There are also works on prioritizing the blocking-ending branches in trace scheduling [22] and superblock scheduling [23]. Comparing to them, TTS has the advantage of scheduling the branches more aggressively by allowing code downward motion and higher resource utilization by speculating instructions from more paths.

Wavefront scheduling [17] is proposed and used as a global code scheduler in Intel's reference compiler for the IA-64 architecture. It uses a path-based dependence representation to describe the data dependences and control dependences. This representation enables efficient bookkeeping for the compensation code of speculation.

7. Conclusion

This paper presents new developments for global scheduling in a treegion framework: The tree traversal scheduling (TTS) algorithm and efficient data flow analysis for treegions. The TTS algorithm achieves high performance by scheduling the block-ending branches as early as possible. TTS allows high resource utilization when multiple execution paths can benefit and reduces resource contention by resolving branches as early as possible. Profile information is used in the scheduling process to prioritize the most frequently executed paths. Implementation issues for efficient implementation of the data flow analysis associated with code motion and operand renaming are included.

Our experiments show that treegion scheduling using TTS has a speedup of 35% over BB scheduling and 4% over superblock scheduling. Two other important conclusions were drawn from the experiments:

1) To fully take advantage of load speculation, the in-order processor pipeline needs to be modified so that it does not stall for a D-cache miss at the execution/memory stage but stalls only on the first use of the missing value at dispatch/register read stage;

2) Fewer multi-ops as a result of TTS results in fewer I-cache accesses while code expansion due to treegion formation usually introduces higher miss rates.

8. Acknowledgements

This research was supported by generous cash and equipment donations from Intel, IBM, Hewlett-Packard, Sun Microsystems, Texas Instruments, and an NSF CAREER award.

9. References

[1] W.A. Havanki, S. Banerjia, and T. M. Conte. "Treegion scheduling for wide-issue processors." *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[2] S. Banerjia, W. A. Havanki, and T. M. Conte. "Treegion scheduling for highly parallel processors." *Proceeding of Euro-Par'97*, August, 1997

[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 way for VLIW and superblock compilation." *The Journal of Supercomputing*, vol. 7, pp. 229-248, January 1993.

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

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

[6] B. L. Deitrich and W. W. Hwu. "Speculative hedge: regulating compile-tim speculation against profile variations." *Proc. 29th Ann. Int'l Symp. Microarchitecture (MICRO29)*, December, 1996

[7] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0." Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, February 1994

[8] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL-PD architecture specification: version 1.1." Tech. Rep. HPL-93-80 (R.1), Hewlett--Packard Laboratories, February 2000

[9] 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

[10] 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" *Proc. 25th Ann. Int'l Symp. Microarchitecture (MICRO25)*, December, 1992

[11] P. Y. T. Hsu and E. S. Davison, "Highly concurrent scalar processing", *Proc. 13th Ann. Int'l Symp. Computer Architecture (ISCA-13)*, June 1986

[12] S. M. Moon and K. Ebcioğlu. “An efficient resource-constrained global scheduling technique for superscalar and VLIW processors.” *Proc. 25th Ann. Int’l Symp. Microarchitecture (MICRO25)*, December, 1992
[13] A. Nicolau. “Percolation scheduling: a parallel compilation technique.” Tech. Rep. TR-85-678, Department of Computer Science, Cornell University, May 1985
[14] The LEGO Compiler. Available for download at http://www.tinker.ncsu.edu/LEGO
[15] K. N. Menezes, S. W. Sathaye, and T. M. Conte. “Path Prediction for high issue-rate processors.” *Proc. Of the 1997 Conf. On Parallel Architectures and Compilation Techniques (PACT’97)*, November, 1997
[16] J. Hoogerbrugge. “Dynamic branch prediction for a VLIW processor.” *Proc. Of the 2000 Conf. On Parallel Architectures and Compilation Techniques (PACT’00)*, October, 1997
[17] J. Bharadwaj, K. Menezes, and C. McKinsey. “Wavefront scheduling: Path based data representation and scheduling of subgraphs.” *Proc. 32nd Ann. Int’l Symp. Microarchitecture (MICRO32)*, December, 1999
[18] A. V. Aho, R. Sethis, and J. D. Ullman “Compilers Principles, Techniques, and Tools.” Addison-Wesley Publishing Company, March, 1988
[19] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, “Instruction fetch mechanisms for VLIW architectures with compressed encodings.” *Proc. 29th Ann. Int’l Symp. Microarchitecture (MICRO29)*, December, 1996
[20] S. Aditya, V. Kathail, and B. R. Rau, “Elcor’s machine description system: version 3.0.” Tech. Rep. HPL-98-128 (R.1), Hewlett--Packard Laboratories, October 1998
[21] M. S. Schlansker and B. R. Rau. “EPIC: Explicitly Parallel Instruction Comupting.” *IEEE Computer*, Vol. 33, Issue 2, February 2000
[22] M. D. Smith. “Architectural support for compile-time speculation.” In D. Lilja and P. Birds, editors. The Interaction of Compilation Technology and Computer Architecture. Kluwer Academic Publishers, Boston, 1994
[23] C. Chekuri, R. Motwani, R. Johnson, B. Natarajan, B. R. Rau, and M. S. Schlansker. “Profile-driven instruction level parallel scheduling with applications to superblocks.” *Proc. 29th Ann. Int’l Symp. Microarchitecture (MICRO29)*, December, 1996