A Treegion-based Unified Approach to Speculation and Predication in Global Instruction Scheduling

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

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

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

Abstract

This paper presents a treegion-based global scheduling technique for wide issue VLIW/EPIC processors. A treegion is a single-entry/multiple-exit global scheduling scope that consists of basic blocks with control-flow forming a tree. We propose a two-phase approach to global scheduling within a treegion scope that enables speculative code motion in the first phase and uses predication of all instructions in the second phase. In the first scheduling phase, tree traversal scheduling (TTS) takes full advantage of speculation to speed up all possible paths in a treegion. Over-aggressive speculation is limited by scheduling block-ending branches as early as possible, enabled by downward code motion. A multiway branch transformation is also performed to reduce control dependence height. In the second scheduling phase, fully resolved predicates (FRPs) are used to enable branch barrier instructions, such as stores and subroutine calls, to move across branches. Selective if-conversion can also be applied to remove hard-to-predict branches in a treegion. The simulation results based on an 8-issue EPIC style machine model show an average speedup of 21% of TTS over BB scheduling, an additional speedup of 6.4% from multiway branch transformation, and another 1.9% speedup from FRP-guarded code motion. Other code transformations such as treegion code layout and the general operation combining are also presented in this paper.

1. Introduction

Treegions and the treegion scheduling have been proposed as a scheduling technique to extract instruction level parallelism (ILP) at compile time [1,2]. A treegion is a single-entry/multiple-exit

nonlinear region that consists of basic blocks with control-flow forming a tree, as illustrated in Figure 1a. Based on the control flow graph (CFG), three treegions are formed. Since large regions are usually better for ILP extraction in scheduling than small regions, tail duplication [6] is applied as a treegion enlarging optimization. After unconditional branches are removed, the resulting treegion is shown in Figure 1b. The trade-off for exposing ILP through treegion formation is the code-expansion that results from duplicates of BB6 and BB7. The detailed treegion formation algorithm was presented in [1].

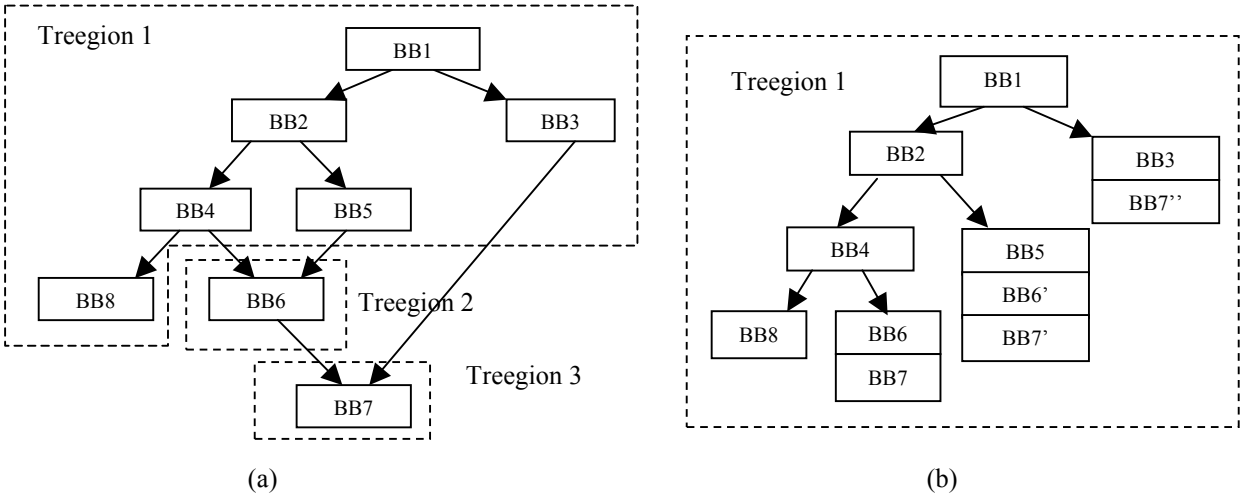

Figure 1. (a). The CFG and the treegions constructed; (b) The treegion constructed after the tail duplication

Treegion formation is based only on the programs CFG topology and is independent of profile information. This makes the treegion a well-suited scheduling scope for execution time optimization environments such as dynamic optimization or dynamic re-compilation and for cases where profile information is not representative of actual program behavior.

Because treegions are a non-linear scheduling scope, they provide an excellent framework for two popular techniques to extract ILP:

1. *speculation* as treegions provide the ability to speculate instructions from multiple execution paths instead of the one that is presumed to be the most frequently executed.
2. *predication* as fully resolved predicates (FRPs) [20] enable code motion for branch barrier instructions (e.g. stores, subroutine calls, and instructions with live-range interference that

restricts speculation) from multiple execution paths. Also, multiple paths in a treegion can be combined using if-conversion.

We propose a two-phase scheduling approach to take advantage of speculation and predication in separate phases. The motivation of this two-phase approach is based on the following observations: (a) although branch miss-prediction introduces high penalties for deep-pipelined processors, most of the branches are highly predictable and it is beneficial to keep those branches instead of removing them with if-conversion. (b) Those branches also help to reduce resource contention due to if-converted paths, as each path will see a full set of resources. (c) The treegion framework is ideal for utilizing speculative code motion (i.e., code motion across branches) to speed up multiple execution paths. If the predicates are introduced at the same time/before the speculation phase, the reverse if-conversion has to be performed to reach the same effect. As a result, the speculation is exploited in the first phase and the predication as well as if-conversion is exploited in the second phase.

In the first phase, which we call the *speculative scheduling phase*, the tree traversal scheduling (TTS) algorithm [3] is used for extensive code speculation while limiting over-aggressive speculation by scheduling block-ending branches as early as possible. Early scheduling of branches is enabled by downward code motion. In TTS, profile information is used to prioritize instructions from more frequently executed paths and the early scheduling of branches ensures no delay for less frequently executed paths.

The early scheduling of branches in TTS also results in serial branch sequences that form critical control dependence height. A multiway branch transformation is performed during scheduling to reduce such control dependence height. Treegions enable both forms of multiway branch transformation, *condition tree of (n+1)-way branches* and *condition tree of 2^n-way branches*. As a non-

linear scope, treegions have more opportunities for multiway branch transformations than linear scheduling scopes, such as traces [7] or superblocks [6].

In the second phase, which we call the *non-speculative scheduling phase*, each instruction is guarded by a predicate so that code motion introduces no additional speculation. After the initial speculative phase of scheduling, each instruction in guarded by the FRP for the block that it now resides in. The FRP for a block is a Boolean which is true if and only if control flow will reach that block. Non-speculative code motion above branches is possible for branch barrier instructions when they are guarded by their FRP. Additionally, *selective if-conversion* can be applied to remove branches in a treegion based on their run-time predictability.

This paper also presents work on two other important aspects of performance for treegion schedules. General operation combining is used to merge together two similar or identical operations in a treegion if they can be scheduled in the same block and location. Treegion code layout is performed to utilize the spatial locality in treegion-scheduled code to improve I-cache performance.

Using an 8-issue VLIW/EPIC style machine model, we analyzed the run time based on the following categories: *non-stall execution time* (i.e., the execution time taken by continuous pipeline execution), *I-cache stall time*, *D-cache stall time*, *branch misprediction stall time*, and *other stall time* (due to necessary stalls between scheduling scopes). Simulation results show that compared to basic block (BB) scheduling, TTS reduces non-stall execution time significantly while keeping other stall time in the same range with the help of the architectural features of the pipeline model. Overall, TTS has up to 40% speedup and 21% average speedup over BB scheduling. The multiway branch transformation, which requires a multiway branch predictor [14,15] in the processor, has two effects on the run time. Non-stall execution time is reduced as the multiway branch transformation reduces the critical control dependence height exposed by TTS. Branch misprediction stalls are also reduced as the

multiway branch transformation reduces aliasing among different multiway branches when indexing the branch prediction table. In total, the multiway branch transformation results in an average of 6.4% speedup over TTS. When the non-speculative scheduling phase is added, another 1.9% speedup is seen from FRP-guarded code motion. Selective if-conversion/reverse if-conversion requires accurate run-time predictability of branches, which make it most beneficial if the architecture has support for dynamic optimization [17].

The rest of the paper is organized as follows. Section 2 describes the TTS algorithm, our simulation methodology, and the results of TTS. Multiway branch transformation and its results are discussed in Section 3. Section 4 presents the non-speculative scheduling phase in treegion scheduling, including FRP-guarded code motion and selective if-conversion. General operation combining and code layout are introduced in Section 5. Section 6 concludes the paper.

2. Tree Traversal Scheduling

2.1. The Tree Traversal Scheduling algorithm

In treegion-based global scheduling, Tree Traversal Scheduling (TTS) takes full advantage of speculation in the speculative scheduling phase. Treegions are well suited to speculation as they enable instructions from multiple paths to be speculated. The main motivation of TTS is to utilize the large scheduling scope provided by treegions for effective speculative code motion. Speculation needs to be performed carefully as over-aggressive speculation may result in a negative impact on performance, as shown in Figure 2 (assuming the latency for add/branch is 1 cycle and load latency is 2 cycles for a cache hit).

In the example in Figure 2a, we assume registers r6, r8, and r10 are ready at the current cycle (cycle n). Figure 2b 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, 5, and 6 are ready at cycle n, the

list scheduler may schedule these instructions ahead of instructions 2 and 4. The average execution time is computed as follows: execution time is 4 cycles if control edge 1 is taken and 3 cycles if control edge 2 is taken; the average is then (4*0.8 + 3*0.2) = 3.8 cycles. The speculation of instructions 5 and 6 results a one-cycle delay for the block-ending branch (which delays both paths following it) and a one-cycle delay for instruction 4 as it is ready to be issued at cycle n+1.

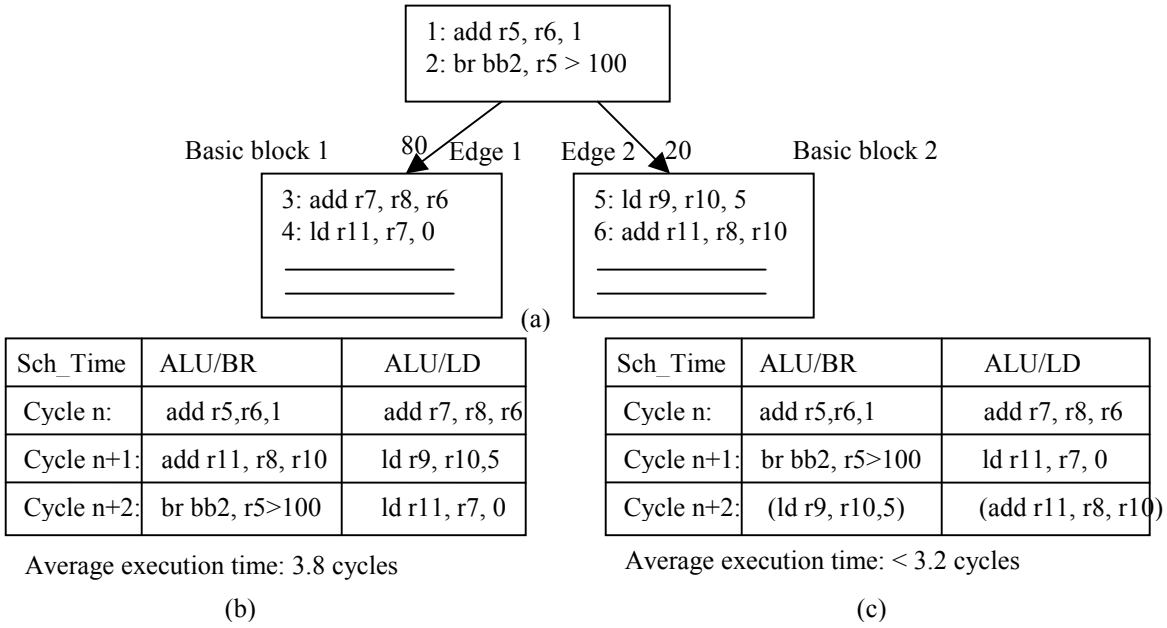

Figure 2. (a) An example for treegion scheduling, (b) the scheduling result using list scheduling, (c) the scheduling result using TTS

TTS solves over-aggressive speculation in the above example. At each cycle, the candidate instruction is chosen for scheduling based on criteria in the following order:

a) execution frequency,

b) exit count heuristic [8] to resolve ties from (a), and

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

This ordering gives priority to block-ending branches as parent blocks always have higher execution frequency and exit count than child blocks. It also prioritizes more frequently executed instructions. For the example in Figure 2, the scheduling result using TTS is shown in Figure 2c, where the average

execution time is less than 3.2 cycles since all the machine resources will be available to basic block 1 at cycle (n+2) if it is on the actual execution path. Note that TTS moves to one of the child blocks after the block-ending branch of the current BB is scheduled. The schedule at cycle n+2 is parenthesized to show that it is the scheduled result if TTS chooses basic block 2 as the next scheduling block.

TTS prioritizes instructions from the most frequently executed path according to profile information, as seen from the candidate selection heuristic (a). When profile information is inaccurate (or not available), the exit count heuristic (b) provides some protection against profile variation. The exit count heuristic is adapted from the helped count priority function of speculative hedge [8] and is the number of the exits that follow the instruction in the CFG. Also, the ability of treegions to allow speculative instructions from more than one execution path inherently makes them less sensitive to profile shifts.

Figure 3 describes the TTS algorithm. The distinguishing characteristics of TTS are that branch instructions are scheduled as early as possible and that speculation from more than one path of execution is based on the priority heuristic. Step 1 forms the scheduling order of the basic blocks, which is determined by a depth first traversal based on execution frequencies. For the example treegion in Figure 4, the basic block order is: BB1, BB2, BB4, BB7, BB6, BB5, and BB3. In steps 2-4, blocks are scheduled in the order determined in step 1. For each basic block currently being scheduled, only instructions contained in the blocks that it dominates are considered for speculation. In the same example, when scheduling BB1, instructions from all other basic blocks are considered for speculation, but when scheduling BB4 only the instructions remaining in BB6 and BB7 are considered for speculation. In TTS, the early scheduling of branch instructions may result in downward code motion. A detailed discussion of TTS and implementation issues for data flow analysis is found in [3].

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 with priority given to the block-ending branch.
3. During the scheduling of a basic block, consider speculation for instructions dominated by this basic block according to their execution frequency, exit count and dependence height.
4. After scheduling the block-ending branch, traverse to the next basic block and go back to 3.

Figure 3. The Tree Traversal Scheduling (TTS) Algorithm

The TTS algorithm achieves high resource utilization and speeds up multiple execution paths (with priority given to highest frequency paths) by speculating instructions from all possible execution paths. This is an advantage over linear scheduling methods such as trace scheduling [7] or superblock scheduling [6]. The TTS algorithm reduces resource competition by the early resolution of branch instructions and only scheduling instructions that are dominated by the current basic block. The early resolution of branch instructions results in all the machine resources being available to each path out of a branch instruction. For example in Figure 4, when scheduling basic block 2, instructions from basic block 3 will not compete simultaneously for the machines execution resources. This is an advantage of TTS over hyperblock scheduling [12] since in hyperblock scheduling instructions from disjoint control flow paths may compete for resources in any given cycle.

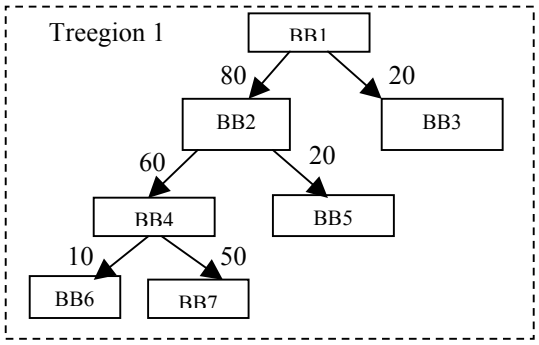

Figure 4. The tree traversal order of a treegion: BB1, BB2, BB4, BB7, BB6, BB5, BB3 (the number along the control edge represents the profiled execution frequency)

As the algorithm in Figure 3 implies, TTS enables chains of instructions to be speculated including the exception potential instructions, such as loads, divisions, etc., and their dependent instructions. The propagation of NaT [11] is used to catch such an exception if it happens at run time.

2.2. Simulation Methodology

The TTS algorithm is implemented in LEGO compiler [13], a research ILP compiler developed for high performance VLIW/EPIC style microprocessors, and is evaluated using the SPECint95 benchmark suite. The compiling process is as follows. All programs are first compiled with classic optimizations using the IMPACT compiler from University of Illinois [21] and converted to Rebel textual intermediate representation using the Elcor compiler from Hewlett-Packard Laboratories [19]. Then, the LEGO compiler is used to profile code, form treegions and schedule the instructions using the TTS algorithm. After instrumentation is added for trace-based timing simulation, the scheduled Rebel code is converted into an inline execution simulator that is emitted as C code. Finally, a trace-based timing simulation runs together with the execution simulation to obtain the simulation results while ensuring the correctness of the program. In our experiments, all benchmarks in SPEC95int suite run to completion demonstrating that LEGO is a functioning compiler.

In this study, the SPEC95int benchmarks are scheduled for an 8-issue VLIW machine model based on the Hewlett-Packard Laboratories HPL_PD architecture [9,10]. 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). Also, the execution pipeline is modeled such that it stalls only at dispatch/register read stage when any of the source operands are not available. Then, in trace-based timing simulation, the same machine model is used with an I-cache, a D-cache and a branch predictor. The detailed specification of the processor model is shown in Table 1.

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 2-way 32KB each bank [18]. 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 Multiway branch prediction [15] Branch prediction table: 2^{14} entries; Branch target buffer: 2^{14} entries/8-way/LRU Branch misprediction penalty: 10 cycles

2.3. Results

Performance results for TTS are compared to basic block (BB) scheduling for the same machine model. For BB scheduling, list scheduling with software renaming support to remove output- and anti-dependences is applied. Treegion formation is skipped for BB scheduling so that I-cache effects are appropriately modeled. General operation combining and code layout (described in Section 5) are applied to both BB scheduled code and TTS scheduled code to improve execution time performance.

Figure 5 shows the speedup of TTS scheduled code over BB scheduled code. It can be seen that the speedup of TTS over BB scheduling is up to 40% for benchmark *vortex* and the average speedup is 21%.

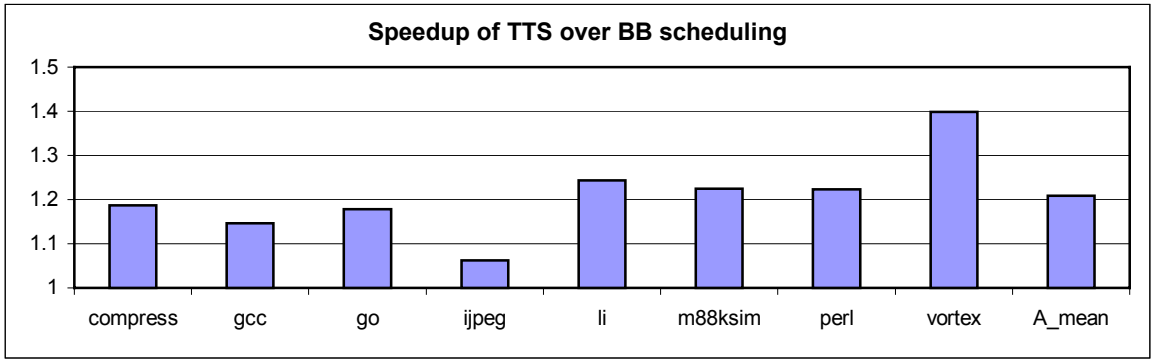

Figure 5. The speedup of TTS over BB scheduling

A breakdown of the components of execution time helps us better understand the scheduling impacts on performance. For our machine model we categorize the execution time as follows:

- pipeline execution time (non-stall cycles)
- I-cache stall time
- D-cache stall time
- branch misprediction stall time
- other stall time (e.g. stalls between scheduling scopes)

Inter-region/inter-procedural stalls may happen when a computation of a live variable is scheduled in the same multi-op as the region-exit branch/return operation. If the computation latency is more than 1 cycle and there is an immediate use of the live variable, the execution pipeline will be stalled.

Figure 6 shows each execution time by category for the BB scheduled result. Most of the execution time is spent on pipeline execution (81.6% on average). An average of 12.2% of execution time is due to branch mispredictions. D-cache misses and I-cache misses account for 4% and 1.2% of execution time, respectively. Less than 1% is from other stalls.

Figure 7 shows each execution time by category for the TTS results. The stall time categories (non pipeline execution time categories) for TTS and BB scheduling are very similar. However, the stall cycles represent a larger percentage of execution time for TTS as the overall execution time has decreased significantly.

A detailed look at each execution time category provides insights of TTS and motivates the discussion in the following sections.

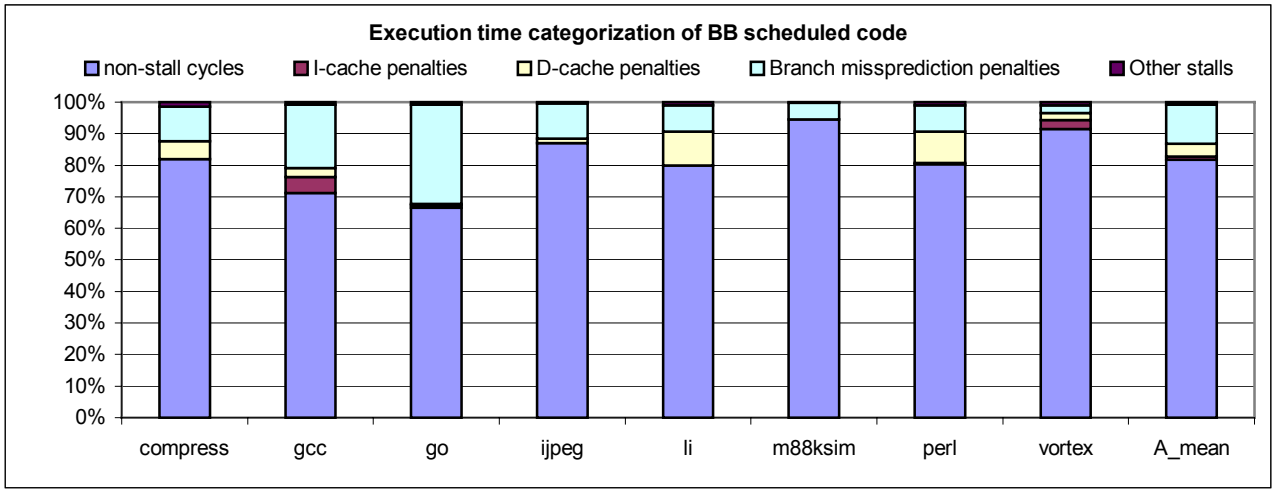

Figure 6. Execution time categorization of the BB scheduled code

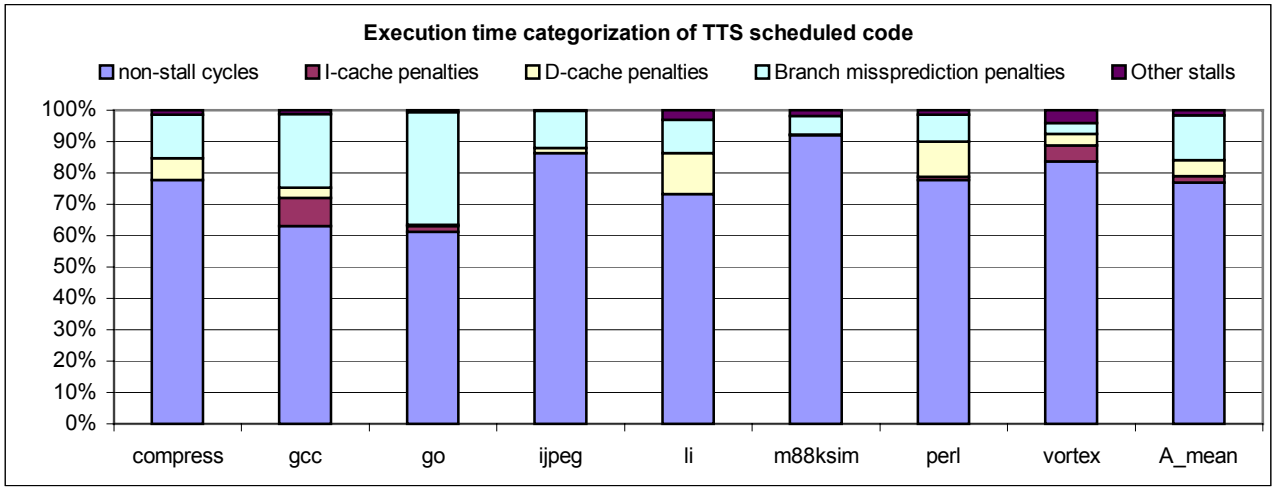

Figure 7. Execution time categorization of TTS scheduled code (The stall cycles represent a larger percentage of execution time for TTS as the overall execution time has decreased relative to Figure 6)

- **Non-stall cycles for pipeline execution**

Speculation in TTS reduces program dependence height and the early resolution of branches provides a full set of machine resources to each path out of the branch. Both effects reduce non-stall execution time as more ILP is utilized. The ratio of non-stall cycles of TTS over BB scheduling is shown in Figure 8. We see that TTS reduces non-stall cycles for benchmark *vortex* by 35% and reduces non-stall cycles for benchmark *ijpeg* by 11%, as reflected by the maximum and minimum speedups, respectively in Figure 5. Overall, TTS reduces non-stall cycles when compared to BB

scheduling by an average of 25%. Even with these reductions in non-stall execution times, Figure 7 shows that non-stall execution time still dominates overall execution time.

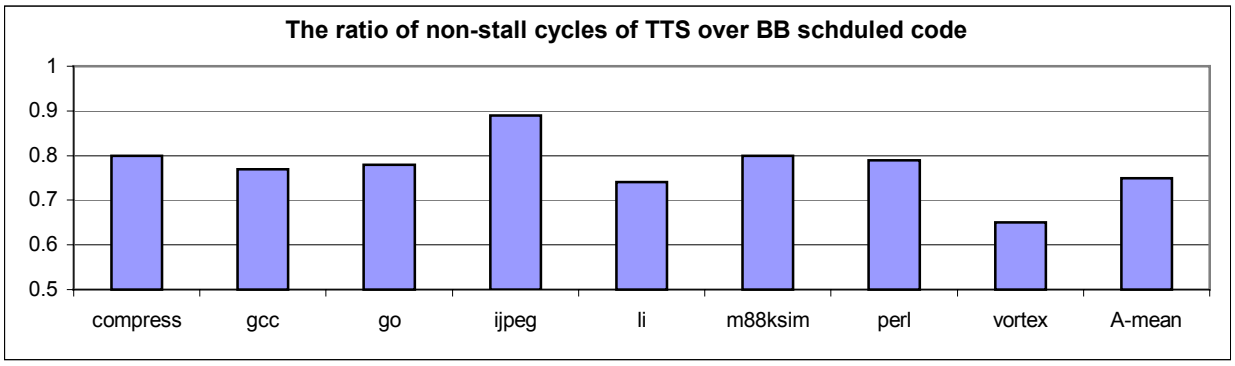

Figure 8. The ratio of non-stall cycles of TTS scheduled code vs. BB scheduled code

- **I-cache stalls**

Treegion formation and downward code motion results in an average code size expansion of 72% more than the original code size. Also, the TTS algorithm impacts I-cache performance in the following ways:

1) Tail duplication increases code spatial locality.
2) Number of I-cache accesses is reduced (fewer multi-ops for each execution path).
3) More operations per multi-op.

The increased code size and the larger number of operations per multi-op results in a larger I-cache miss rate and TLB miss rate. This effect can be minimized by an I-cache with larger size/associativity. Although I-cache miss-rates increased for TTS scheduled code, the resulting miss rate was still very small (maximum of 1.33% for gcc and 0.27% on average). The impact on overall execution time is around 2%. The low miss rate is the result of the optimized I-cache structure that we used in our machine model.

- **D-cache stalls**

Extensive load speculation in TTS results in an average of 30% more D-cache accesses. The increased D-cache accesses have the following effects: more memory traffic, more D-cache misses,

and decreased D-cache miss rate in most benchmarks (due to the temporal locality of the data and the prefetch effect of the speculative loads). Our machine model is tolerant of D-cache access penalties resulting from load speculation in that the pipeline will not stall at the execution/memory stage for a load miss. Instead, it stalls on the first use of the missing value at the dispatch/register read stage. This is similar to the behavior of the Intel Itanium processor [22]. The effect for this type of pipeline optimization on D-cache stalls is shown in Figure 9. Compared with D-cache stalls of BB scheduled code (obtained with optimized machine model), the TTS code shows 23% more D-cache stalls with the simple in-order pipeline model that stalls at each load miss. When the pipeline model is optimized, the D-cache penalties of TTS scheduled code is in the same range as the D-cache stalls of BB scheduled code.

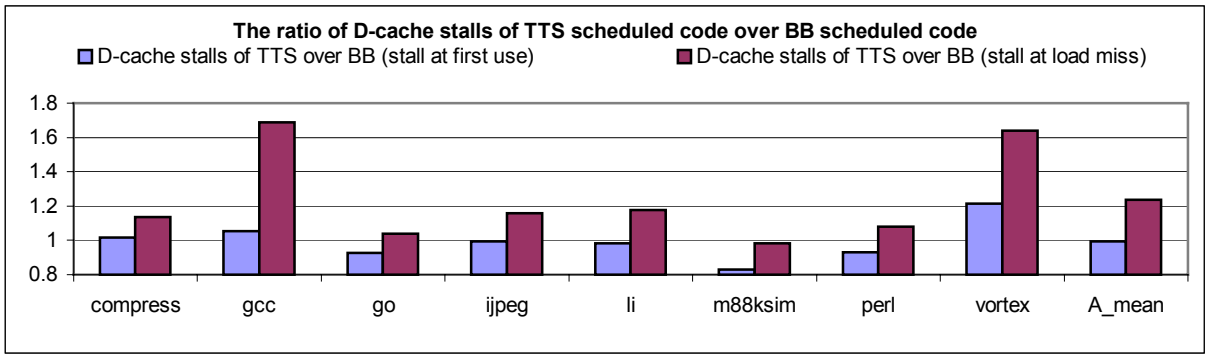

Figure 9. The ratio of D-cache penalties of TTS scheduled code over BB scheduled code

- **Branch misprediction stalls**

TTS does not significantly change the characteristics of branches in the program except by duplicating branches and removing some unconditional branches during tail duplication process. As a result, the branch misprediction penalties remain in the same range as for BB scheduled code. As seen from Figure 7, branch misprediction stalls account for around 15% of the total execution time, which suggests it is a good objective to reduce branch related stalls.

3. Multiway Branch Transformation in a Treegion

As TTS extensively performs speculation and the early scheduling of block-ending branches with downward code motion, a treegion usually results that contains a sequence of branch-only BBs (i.e., a BB consists of only a branch) followed with many stores and subroutine calls (as those instruction are unable to be speculated due to their unknown side effects). In such a treegion, control dependence becomes the critical path as it enforces the issue of one sequential branch at each cycle. Two types of such control structures, a condition tree of (n+1)-way branches and a condition tree of 2^n-way branches, are shown in Figure 10a and 10b respectively. In these control structures, the sequential branches have to be executed in serial to ensure the correct program semantics. In order to solve this problem, the multiway branch transformation is used. It transforms sequential branches into one multi-op containing several branch operations. Architectural support for multiway branch execution is present in several VLIW/EPIC architectures [15,22].

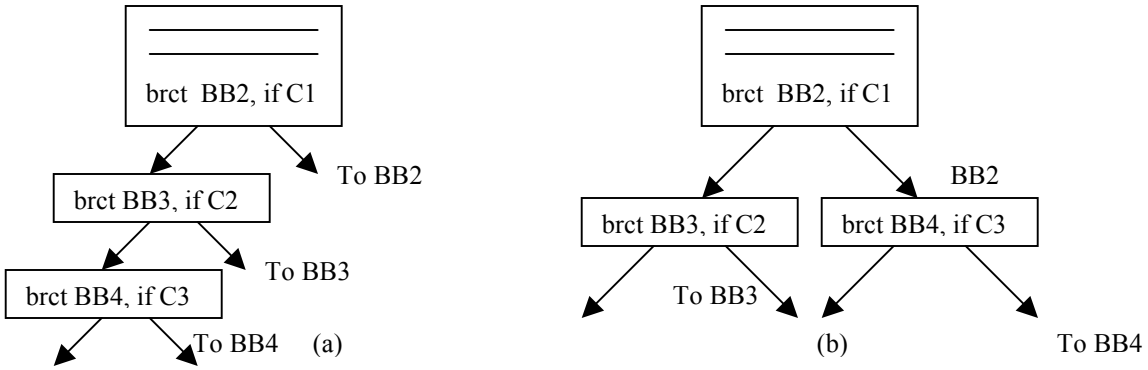

Figure 10. (a) A condition tree of (n+1)-way branch; (b) a condition tree of 2^n-way branch

With multiway branch support in the architecture, multiple branches can be executed at the same time so as to minimize the control dependence height. Treegions are well suited to the multiway branch transformation for two reasons. First, TTS maximizes the multiway branch opportunities due to speculation, early branch scheduling and downward code motion. Secondly, both types of condition

tree shown in Figure 10 can be exploited to form a multiway branch. This is an advantage of treegions over linear scopes as the condition tree of 2^n-way branches is excluded for linear scopes.

In treegion scheduling, the multiway branch transformation is integrated into TTS: when scheduling a branch instruction, TTS investigates the chances of merging it with its predecessor branch instruction. The branch-merging/multiway-branch formation process involves two steps: preparation of the correct branch targets, and condition manipulation for each branch operation in the multiway branch to maintain the semantics of the program. The following cases are considered according to whether the branch is conditional or unconditional and whether it is on the taken path or untaken path of its predecessor branch. Here, note that when scheduling a branch in a treegion, its predecessor branch cannot be an unconditional branch; otherwise it would have been removed during the treegion formation process.

Case 1: The branch is unconditional and it is on either the taken or untaken path of its predecessor branch. The branch merging is straightforward in this case: remove the unconditional branch and modify the taken/untaken branch target. Figure 11 shows the branch merge process when the unconditional branch is along the taken path of its predecessor branch. The taken target of the predecessor branch is modified to the target of the unconditional branch, which is then removed after the merging process.

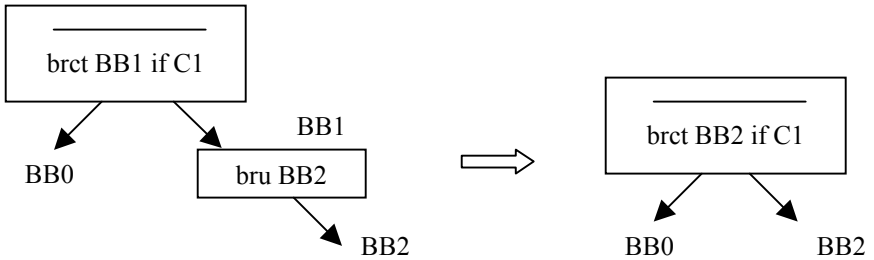

Figure 11. The branch merging when the unconditional branch is along the taken path of its predecessor branch

Case 2: The branch is conditional and it is on the taken path of its predecessor branch. The branch merges with the predecessor branch operation into two conditional branches with changes in both

branch conditions and branch targets, as shown in Figure 12. Note that as the LEGO ISA is based on the HP-PLD architecture, it requires the branch condition to be specified in a predicate (as shown in Figure 12). In previous examples, we simplified so that the branch condition is included in the branch instruction and the ideas still hold when the condition is separated from the branch operation. From Figure 12, it can be seen that the branch condition manipulation can be implemented in the corresponding predicate computation with no additional predicate define operation introduced for multiway branch transformation.

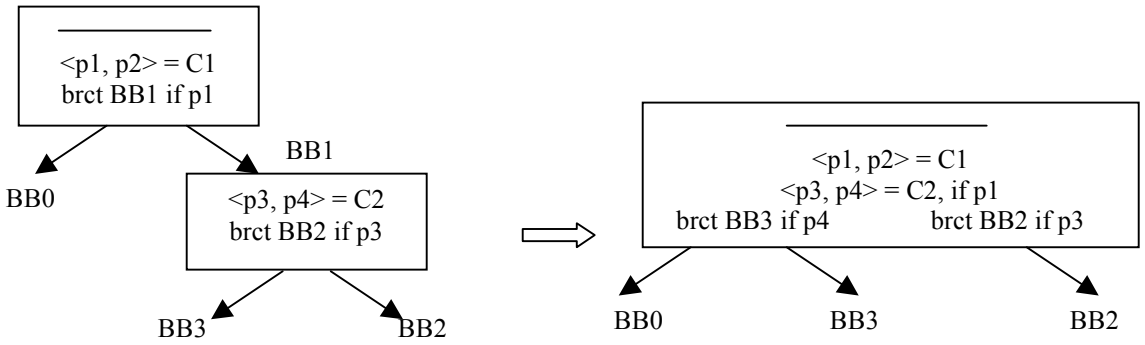

Figure 12. The branch merging when the conditional branch is along the taken path of its predecessor branch

Case 3: The branch is conditional and it is on the untaken path of its predecessor branch. The branch merges with the predecessor branch operation into two conditional branch operations with changes in condition only, as shown in Figure 13.

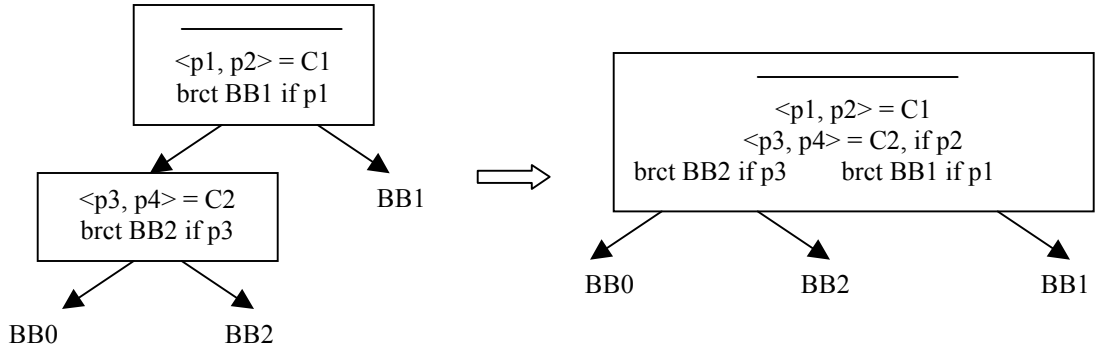

Figure 13. The branch merging when the conditional branch is along the untaken path of its predecessor branch

As branch merging is performed at the same time as the branch is being scheduled in TTS, all branches are investigated sequentially for eligibility according to their scheduling order. This is

desirable as it gives priority to reducing the control dependence height along the more frequently executed paths. Every possible merge is a subset of one of the three cases described above. Note that although predecessor branches appear as a single conditional branch in the description of the cases, the same transformation applies when a predecessor is one branch operation of a multiway branch.

Figure 14 shows the speedup resulting from using the multiway branch transformation in TTS. Speedup is always positive and is up to 13.7% for benchmark *go* and averages 6.4%, compared to TTS without multiway branch transformation.

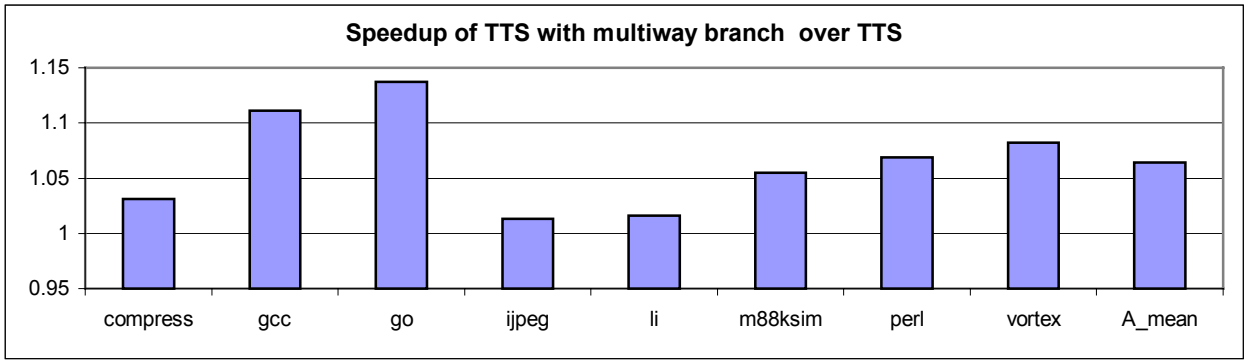

Figure 14. The speedup of TTS with multiway branch over TTS

The speedup mainly comes from two execution time categories: reduced non-stall execution time and reduced branch misprediction stall time. Figure 15 shows the contribution of each execution time category to the overall speedup. Taking the benchmark *compress* as an example, it shows that the reduction in non-stall cycles accounts for 88% of its speedup and fewer branch mispredictions result in 10% of its speedup. From Figure 15, it can be seen that the multiway branch transformation always produces positive reduction in non-stall execution time, which is a direct result of the reduction of control dependence height and it contributes 76% of the speedup on average. For branch misprediction stalls, however, one exception is observed for benchmark *li*, where the non-stall cycle reduction is offset by performance degradation due to increased branch mispredictions. The multiway branch transformation has several contrary effects on branch misprediction stalls. First, as multiple branch

operations combine into one multiway branch, the number of resulting multiway branches is smaller than the number of original branches. This reduction is beneficial as the number of branch predictor accesses is reduced. The smaller number of multiway branches also reduces the possibility of aliasing among multiway branches when indexing the branch prediction table. Aliasing among branches inside a multiway branch, however, results in a small increase in branch misprediction rate (maximal increase happens for benchmark *li* whose branch misprediction rate increases from 3.98% to 4.85%). Overall, branch misprediction stalls are reduced by 12.3% and non-stall execution time is reduced by 5.1% in average by the multiway branch transformation.

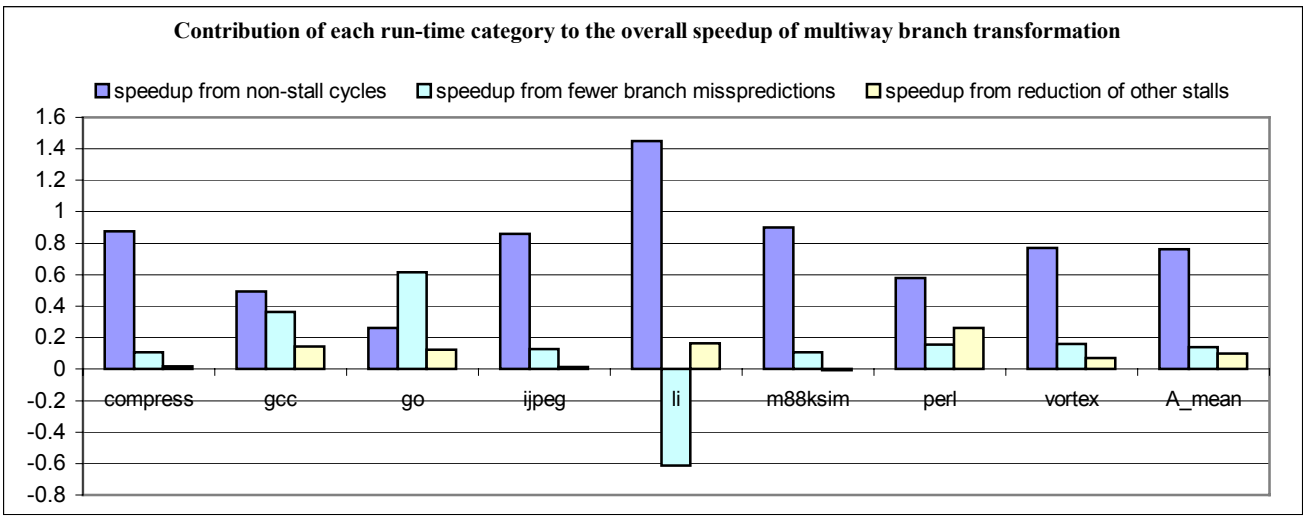

Figure 15. The speedup of the multiway branch transformation from different run-time categories

4. Predication Support in Treegion Scheduler

In treegion scheduling, predication support is introduced in the second phase, the non-speculative scheduling phase. Since TTS produces code with high resource utilization by speculation and multiway branch transformation, compiling for predicated execution in treegions must be performed carefully so as to not offset the performance gained from the speculative scheduling phase. In this paper, two types of predicate support are used: (1) fully resolved predicates (FRPs) to enable code motion of stores and subroutine calls across branches, and (2) if-conversion to remove hard-to-predicted branches. Based on

the scheduling results of the first phase, each instruction is guarded by the FRP for the block that it now resides in. (As pointed out in Section 3, LEGO ISA uses predicates to specify the branch condition. No additional FRP computation is required as those predicates can be used directly as FRPs.) Then, FRP-guarded stores, subroutine calls and their dependent operations can be moved across branches to fill in empty slots from the first scheduling phase. Those empty slots resulted from such FRP guarded code motion are then filled with rescheduling the operations that are dominated by the basic block containing those empty slots. One result from the FRP-guarded code motion is the improved static resource utilization. Static and dynamic resource utilization (i.e., the run-time resource utilization) needs to be distinguished in the presence of predication. FRP-guarded code motion improves static resource utilization but not necessarily dynamic resource utilization, which, however, can be improved using a multithread architecture in the treegion framework [4]. FRP-guarded code motions may also result in branch-only basic blocks, which create additional chances for the multiway branch transformation.

The effectiveness of FRP-guarded code motion depends on available machine resources, especially for treegion scheduling, as many resources have already been consumed by the speculative scheduling phase. With our 8-way issue machine model, FRP-guarded code motion results in up to 6.1% speedup in benchmark *vortex* and an average of 1.9% speedup, as shown in Figure 16.

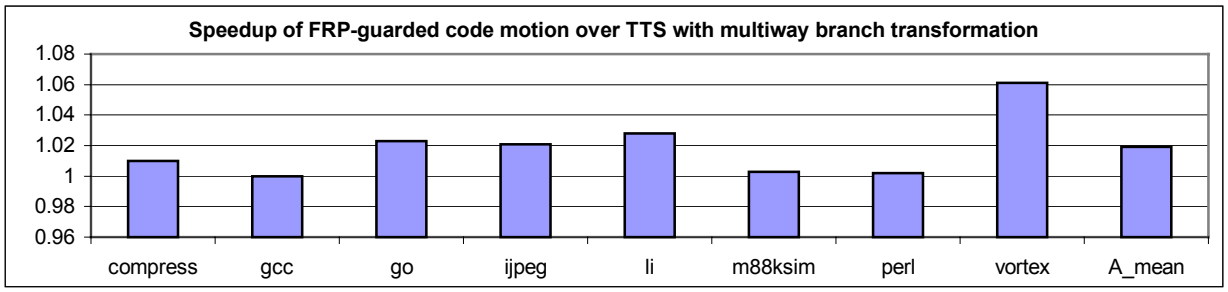

Figure 16. The speedup of FRP-guarded code motion over the TTS with multiway branch transformation

When compared to FRP-guarded code motion, *selective if-conversion* is more complicated as two critical issues are involved: *what to if-convert* and *when to if-convert*. The goal of selective if-conversion in treegions is to remove only hard-to-predict branches. Hard-to-predict branches are not the same as static unbiased branches [20]. Instead, they represent the branches that have high run-time misprediction rates. If-conversion could cause significant performance loss from several factors such as resource contention, dependence height imbalance, and execution frequency imbalance of if-converted paths [5]. Therefore, if-conversion should only be applied when the delay resulting from additional resource contention is smaller than the branch misprediction penalty. As branch prediction behavior is difficult to determine at compile time (i.e., it may vary with some patterns in run time), a reasonable approach is to delay if-conversion decisions until run-time software optimization [17].

As the control flow graph of a treegion contains no merge points, if-conversion in a treegion is different from hyperblock formation [11,12] and additional multiway branches can result from if-conversion. One example is shown in Figure 17.

In Figure 17, suppose branches 1 and 3 are chosen for if-conversion. After branch 1 is removed, BB1 and BB2 are merged into BB0. Then, branches 2 and 3 are transformed into a multiway branch containing three branch operations (two of them are created from branch 3), as shown in Figure 17b. The if-conversion of branch 3, shown in Figure 17c, combines those two branch operations into one "unconditional" branch guarded by a predicate that is true if and only if the condition of branch 1 is true. As a result, the "unconditional" branch operation has similar branch prediction characteristics as branch 1. However, such a combination helps multiway branch prediction as aliasing among branch operations inside the multiway branch is reduced. The whole process is also similar to partial if-conversion [5] as the two branches to be converted are sequential (i.e., along the same execution path).

After the if-conversion, the rescheduling is performed on the if-converted basic block and its dominated basic blocks with the same methodology as TTS.

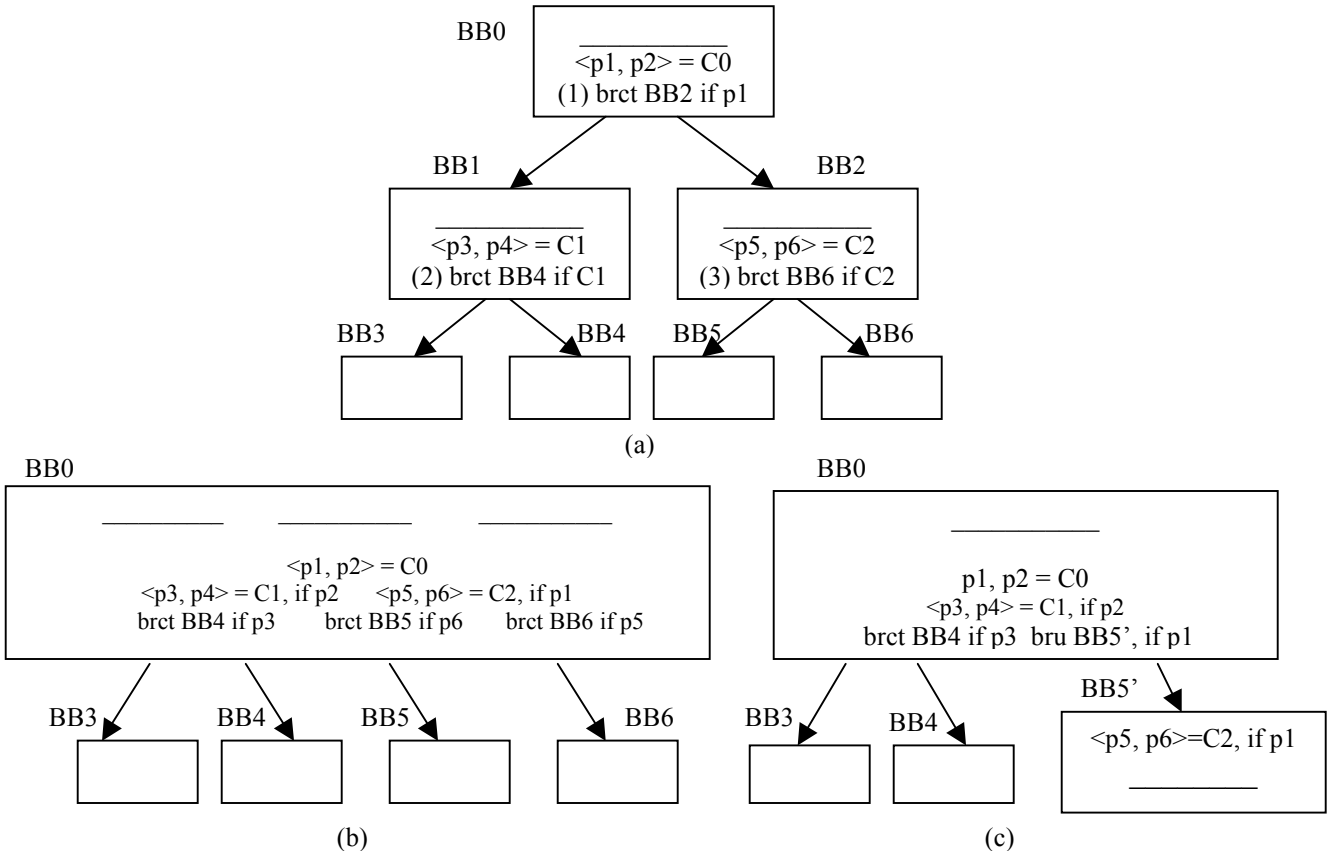

Figure 17. An example of if-conversion in a treegion (a) the CFG before if-conversion (b) the CFG after if-conversion of branch 1 (c) the CFG after if-conversion of branch 3

5. Other Treegion Transformations

In addition to the scheduling techniques discussed in previous sections, two other transformations have significant effects on the performance of treegion schedules: *general operation combining* and *treegion code layout*.

General operation combining is motivated from dominator parallelism [16]. Dominator parallelism is exhibited by identical operations from different paths, which are speculated into a block that dominates each operation. In previous work on treegion scheduling [1], dominator parallelism was

mainly used to remove the redundant copies of operations that resulted from tail duplication. In TTS, the use of dominator parallelism is extended into general operation combining such that two similar or identical operations will be merged if they can be scheduled in the same block and location. General operation combining is performed during scheduling. When an operation (*A*) is selected for scheduling, it is compared with other operations that have already been scheduled in the same cycle. If another scheduled operation is found to have the same opcode and source operands, operation *A* is then merged into it and, if necessary, subsequent uses of *A*'s definitions are renamed. Treegions provide many opportunities for operation combining due to their large scope of instructions. In addition to duplicate operations resulting from tail-duplication, load address generation and load operations also are good candidates for operation combining. In treegion scheduling, general operation combining reduces the static code size by an average of 12.8%.

As treegions are nonlinear regions containing multiple execution paths, appropriate code layout is important for treegion-scheduled code to obtain good I-cache performance, especially when cache size/associativity is limited (up to 40% difference in I-cache penalties is observed). In our treegion scheduling framework, the basic blocks of a treegion are laid out in the same order as block ordering for tree traversal scheduling: a depth first traversal with the child block selected with highest execution frequency. During layout, the polarity of conditional branches may be changed so that the branch is taken when off-trace execution happens. This transformation helps to increase the spatial locality of instruction traces and reduces branch misprediction rates slightly (0.5% on average) since branches are taken less frequently.

6. Conclusion

This paper presents several extensions to treegion-based global scheduling. A two-phase approach to utilize both speculation and predication was presented. In the first phase, the TTS algorithm speeds up

multiple execution paths using speculation and early scheduling of block-ending branches. The critical control dependence height exposed from TTS is reduced by the multiway branch transformation. In the second scheduling phase, fully resolved predicates are used to allow code motion of stores and subroutine calls across branch barriers. In addition, selective if-conversion can be applied to remove hard-to-predict branches. The speedups from these scheduling techniques are analyzed for different run-time categories using an 8-issue VLIW/EPIC style machine model. The simulation results show that while all those techniques reduce non-stall pipeline execution time, the multiway branch transformation also benefits from fewer branch misprediction stalls. Although treegion scheduling introduces a significant amount of load speculation, the D-cache stall times remain approximately in the same range as BB scheduling if a stall-on-use pipeline model is implemented. The code expansion introduced by treegion scheduling shows no significant impact on I-cache performance due to an optimized I-cache structure and careful treegion code layout.

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

7. 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] H. Zhou, M. Jennings, and T. M. Conte. "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors". *Proceedings of the* 14th *Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, *LNCS*, *Springer Verlag*, August, 2001

[4] E. Ozer and T. M. Conte, "Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors," *The 8th International Conference on High Performance Computing (HiPC'01)* (Hyderabad, India), December 2001.

[5] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication", *Proc. 30th Ann. Int'l Symp. Microarchitecture (MICRO30)*, December, 1997

[6] 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.

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

[8] 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

[9] 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

[10] 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
[11] 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
[12] 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
[13] The LEGO Compiler. Available for download at http://www.tinker.ncsu.edu/LEGO
[14] 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
[15] J. Hoogerbrugge. "Dynamic branch prediction for a VLIW processor." *Proc. Of the 2000 Conf. On Parallel Architectures and Compilation Techniques (PACT'00)*, October, 1997
[16] A. V. Aho, R. Sethis, and J. D. Ullman "Compilers Principles, Techniques, and Tools." Addison-Wesley Publishing Company, March, 1988
[17] K. M. Hazewood and T. M. Conte, "A light weight algorithm for dynamic if-conversion during dynamic optimization", *Proc. Of the 2000 Conf. On Parallel Architectures and Compilation Techniques (PACT'00)*, October, 2000
[18] 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
[19] 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
[20] M. S. Schlansker and B. R. Rau. "EPIC: An architecture for instruction-level parallel processors" Tech. Rep. HPL-99-111, Hewlett--Packard Laboratories, February 2000
[21] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors", *Proc. 18th Int'l Symp. On Computer Architecture (ISCA18)*, 1991
[22] H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture", *IEEE Micro*, Vol. 20, Num. 5, Sept/Oct 2000