Treegion Instruction Scheduling in GCC

Michael C. Rosier and Thomas M. Conte
Center for Embedded System Research
Department of Electrical and Computer Engineering
North Carolina State University
{mcrosier,conte}@ncsu.edu

Abstract

Instruction scheduling is a critical compilation phase for extracting significant amounts of parallelism within a program. The first step of instruction scheduling is region formation; the size and characteristics of the region play an important role in determine the amount of available ILP. In this work the status of the implementation of an architecture-independent, aggressive global instruction scheduler based on Treegions is presented. A Treegion is a a tree-shaped subgraph of the control-flow-graph (CFG). Unlike other region formation algorithms, such as Traces or Superblocks, Treegions take into account multiple execution paths, producing more opportunities for parallelism. Unlike Hyperblocks, Treegions do not require predicates. Treegion formation can use tail duplication. To limit the possible negative side effects of code expansion, our Treegion scheduler uses the Instantaneous Code Size Efficiency (ICSE) heuristic of Zhou for conservative tail duplication. Experimental results show that Treegion formation dramatically increases the average region size as compared to the current region formation code. The implementation currently resides on the sched-treegion-branch.

1 Introduction

High performance microprocessors use complex hardware techniques (*e.g.*, out-of-order execution, branch prediction, prefetching) to exploit parallelism within a program. Alternatively, instruction scheduling is a compile-time technique for extracting instruction-level parallelism (ILP). For wide-issue statically scheduled processors (*e.g.,* EPIC, VLIW), instruction scheduling plays an exceedingly important role in improving performance.

During global instruction scheduling the compiler divides a program's control flow graph (CFG) into multiple regions and then schedules each region separately. The scope of a region may be limited to a basic block (*i.e.*, basic block scheduling) or encompass the entire CFG. Past work has predominantly focused on *linear* regions, *i.e.*, regions containing a single control path, which often limits speculation, resulting in an under utilization of processor resources. These types of global instruction scheduling techniques include Trace scheduling [1], Superblocks [2], and Hyperblocks [3]. Trace scheduling forms linear regions, called traces, of basic blocks that execute sequentially. Loop-unrolling is commonly used as a trace enlarging optimization. Similar to Trace scheduling, Superblocks form single-entry, (possibly)

multiple-exit linear regions. After Superblock formation side entrances are removed via tail duplication. Finally, Hyperblocks extend upon Superblocks by using hardware predication to reduce the need for tail duplication. These techniques suffer from a number of pitfalls. First, formation is based on profile information. Often when scheduling for the more probable path the less likely path suffers. Variation in input sets or lack of profile information can result in a significant performance penalty. Furthermore, the linearity of these regions limit the opportunity for speculation.

A Treegion is a non-linear, single-entry, multiple-exit region of code containing basic blocks that constitute a tree-shaped subgraph of the control-flow-graph (CFG). Building large regions is a critical aspect of instruction scheduling that enables the compiler to extract parallelism. Unlike other region formation algorithms, such as Traces and Superblocks, Treegions include multiple paths of execution, producing larger regions and more opportunities for speculation. In addition, Treegions do not require special architectural features for region formation.

GCC currently supports both linear and non-linear regions. Linear regions are supported in the form of Superblocks (`tracer.c`) and Extended Basic Blocks (EBB) (`sched-ebb.c`). Meanwhile, support for non-linear regions (`sched-rgn.c`) are limited to loop-free procedures and reducible inner loops. Treegions have the advantage that unlike Superblocks and EBB, their formation includes multiple paths of execution and do not require profile information. Generally, Treegions can realize significantly larger regions than other region formation techniques.

The remainder of this paper is organized as follows. Section 2 describes the current GCC global instruction scheduler. Section 3 describes natural treegion formation, or treegion formation without tail duplication. Section 4 describes an efficient technique for the tail duplication of treegions. Sections 5 and 6 discuss Treegion scheduling and experimental results, respectively. Finally, section 7 gives a brief conclusion.

2 GCC Instruction Scheduling

The GCC instruction scheduler is a list-based instruction scheduler derived from work originally developed at IBM Haifa Labs. The generic parts of the scheduler are found in `haifa-sched.c`. The goal of list-scheduling is to minimizing the length of the critical path while maximizing the opportunity for parallelism. The steps to list scheduling are as follows:

1. Build the data dependence graph.
2. Calculate priorities for each instruction.
3. Iteratively schedule ready instructions.

The scheduler is invoked before and after register allocation. Treegion scheduling extends upon the interblock scheduling pass, found in `sched-rgn.c`, performed prior to register allocation. Instructions may be speculatively scheduled during the first pass with much greater ease than during the second pass. After register allocation each pseudo-register has been assigned a physical register, introducing anti- and output-dependencies. These dependencies greatly restrict scheduling.

Region formation is the first step of interblock scheduling. In this work, treegions are the chosen region type. Treegion formation is a two step process involving natural treegion formation and tail duplication, which are discussed in sections 3 and 4, respectively.

Prior to scheduling, dependencies between instructions are found for each basic block within the region. Such dependencies include those between registers (*i.e.,* true-, anti-, and output-dependencies), memory dependencies, dependencies to maintain function call ordering, and the dependence between a conditional branch and the setting of the condition code. Routines for building the data dependence graph are found in `sched-dep.c`.

Next, instruction priorities are calculated. The priority of an instruction dictates the order in which it may reside on the *ready list*, or the list of instructions whos dependencies have been resolved and are available for scheduling. Priorities are calculated in reverse order beginning with a basic block's tail instruction and ending with the head instruction. The priority of an instruction is found by summing the latency of the instruction and the maximum priority of any dependent successor instruction. This has the effect of exposing the longest dependency chain, giving those instruction along the critical path highest priority.

Finally, after finding dependencies and calculating priorities, `schedule_block()` is called for each basic block within the region to perform list-scheduling. During the scheduling process instructions are added to the ready list when their dependencies are resolved. Dependent instructions that become ready, but do not reside in the current block, may be added to the ready list if the current block dominates the block in which the potentially speculative instruction resides. The flow probability of a speculative instruction is an important factor to consider when performing interblock motion. Over speculation may delay the critical path or increase contention for resources, while under speculation may result in missed opportunities for increasing parallelism.

The ordering of the ready list is an important factor to consider when list-scheduling. If multiple instructions share the same priority, attributes of these instructions, such as register pressure, affect later scheduling decisions. Choosing between these instructions plays a critical role in finding the optimal schedule. The algorithm for sorting instruction in the ready list is as follows:

1. select the instruction with the highest priority, ties broken by
2. select the instruction which least contributes to register pressure, ties broken by
3. prefer in-block upon interblock motion, ties broken by
4. prefer useful upon speculative motion, ties broken by
5. choose the instruction with the highest flow probability, ties broken by
6. choose the instruction which is least dependent upon the previously scheduled instruction, ties broken by
7. choose the instruction which has the most instructions dependent upon it, or finally
8. choose the instruction with the lowest UID.

Sorting instructions based on this algorithm maximizes the opportunity for parallelism while minimizing the length of the critical path.

3 Treegion Formation

This section describes the two step process of treegion formation. First, natural treegions based on the original CFG are formed. Then, the ICSE heuristic is applied to perform tail duplication.

3.1 Natural Treegion Formation

Natural treegion formation begins at the entry block of a procedure, which forms the root of a new treegion. Starting at this root, the CFG is traversed and successor basic blocks are absorbed into the treegion if they are not a *merge point* (*i.e.*, have multiple predecessor edges). Eventually all successor blocks that do not contain merge points are consumed by the treegion and only leaf nodes remain. These leaf nodes, referred to as *saplings*, are then added to a *saplings list*. Saplings form the roots of new treegions. For each sapling the same process is applied until all basic blocks in the CFG have been consumed.

Figure 1 shows pseudo-code for finding natural treegions. Initially the saplings list includes only the successor to the `ENTRY_BLOCK` of the current procedure. This basic block is then removed from the saplings list to form the root of a new treegion. Next, all successors of the root node are added to the successor edge list. Each edge in the edge list is then traversed in breadth first order to absorb successor blocks into the newly formed treegion. Backedges are not traversed to prevent the forming of cyclic regions. Traversal also ends at the `EXIT_BLOCK`. If the current basic block has multiple predecessor edges (*i.e.*, `EDGE_COUNT(edge->preds) > 1`) the node is added to the saplings list and its successor basic blocks are not considered for inclusion in the current treegion. Finally, if the current node is absorbed into the treegion all its successor edges are added to the successor edge list.

```
find_treegions (void)
{
 add ENTRY_BLOCK->succs to saplings;
 while(more saplings)
 {
  node = first set bit (saplings);
  treegion += node;
  edge_list += node->succs;

  while(more edges)
  {
   curr_edges = edge_list[];
   curr_node = curr_edges[]->dest;

   while(more succ in curr_edges[])
   {
    /* Dont traverse backedges */
    if(edge->flags & BACK_EDGE)
     continue;

    /* Skip Exit Block */
    if(curr_node == EXIT_BLOCK)
     continue;

    /* Add merge to saplings */
    if(EDGE_COUNT(edge preds) > 1)
    {
     SET_BIT (saplings, curr_node);
     continue;
    }

    /* Add node to treegion */
    treegion += curr_node;

    /* Add succs to edge list */
    edge_list += curr_node->succs;
   }
  }
 }
}
```

Figure 1: Pseudo-code for natural treegion formation

Figure 2 shows an example CFG after treegion formation. The size and number of treegions is based on the layout of the CFG, not profile information. From figure 2 it can be seen that for any block in a treegion all predecessor blocks

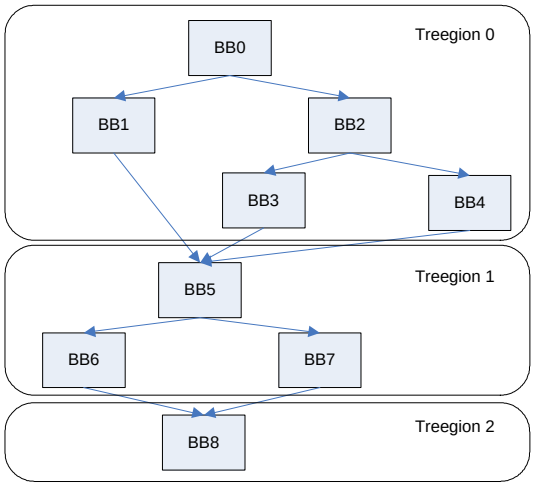

Figure 2: CFG after treegion formation

dominate it. In section 5.2 further optimizations based on dominator parallelism are discussed. It is also important to note that speculatively scheduled instructions are never duplicated because treegions do not contain merge points. For our chosen benchmark suite the average natural treegion contains 2.65 blocks and 20.89 instructions. For these regions, on average 3.65 instructions are speculatively scheduled.

4 Tail Duplication

Tail duplication is performed in order to increase region size providing more opportunity for speculation. However, overly aggressive duplication has the potential to negatively impact the performance of the instruction cache and TLB. This section begins by presenting the tail duplication implementation, with treegions being the unit of duplication. Then an efficient technique for deciding upon when to apply tail duplication is presented. This metric, referred to as the Instantaneous Code Size Efficiency (ICSE), is defined as the change in IPC relative to the change in code size after tail duplication. For each edge between a pair of treegions the ICSE is calculated to determine if the duplication of the child treegion will be beneficial.

4.1 Tail Duplication Example

The tail duplication process begins by calculating the ICSE of each candidate, discussed in subsection 4.2. Each control edge between a parent and child treegion is a potential candidate with the child treegion being the target for duplication. After calculating all ICSEs, the best candidate is selected for duplication if it is above the ICSE threshold. If no more candidates are available for duplication then the scheduling process may begin.

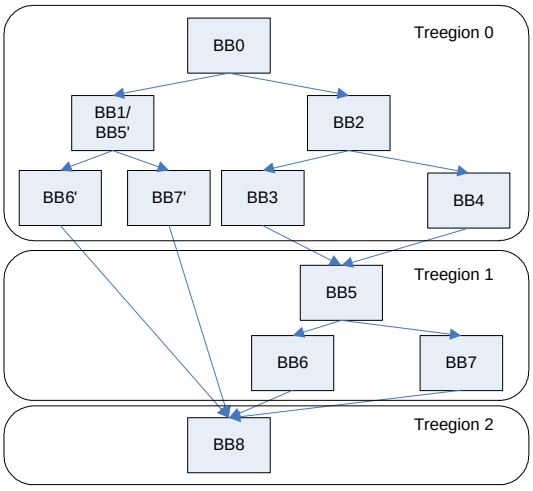

Figure 3: Duplication of candidate edge between BB1 and BB5

Continuing with the example in figure 2, figure 3 depicts the result of selecting the candidate edge between basic blocks 1 and 5. Blocks 5, 6, and 7 in the child treegion, treegion 1, are duplicated. These duplicated blocks, denoted with tick marks, are then absorbed into treegion 0. After calling `cleanup_cfg()` basic blocks `BB5'` and `BB1` are merged into a single block. Tail duplication continues until either no more candidates exist or no more can-

didates are above the ICSE threshold. Under code size or compile time constraints, treegion size may also be limited by the number of basic blocks and/or the number of instructions contained with in the treegion. Compilation flags for constraining tail duplication and region formation are discussed in subsection 4.4.

4.2 Instantaneous Code Size Efficiency

In previous work Zhou *et al.* [4] have shown that for a minimal code size increase (~2%) a significant speedup can be obtained. Furthermore, duplication beyond that of the initial code size increase produces only small additional gains in performance. Due to these facts the ICSE equation was developed and is as follows:

$$Efficiency = \frac{IPC_{after\_td} - IPC_{before\_td}}{code\_size_{after\_td} - code\_size_{before\_td}} \quad (1)$$

In equation 1, IPC_{before_td} and IPC_{after_td} refer to the instruction-per-cycle (IPC) ratio of a treegion before and after the application of tail duplication, respectively. $code\_size_{after\_td} - code\_size_{before\_td}$ refers to the change in code size due to tail duplication. Equation 1 requires the IPC of a region to be known at compile time. Since this information is not available, a heuristic is used to estimate the execution time of a treegion, defined as follows:

$$Exec\_Time = \sum_{path_i} [Max(ddb_{path_i}, rb_{path_i}) * freq_{path_i}] \quad (2)$$

The estimated execution time of a multi-path treegion is defined as the sum of the expected execution time of each path through a treegion biased by the execution frequeny of each path. The execution frequency of each path, $freq_{path_i}$, is determined through profiling. If profile information is not available, GCC uses a number of heuristics to approximate the execution frequencies. The expected execution time of any path is the maximum of the data dependence bound, ddb_{path_i}, and the resource bound, rb_{path_i}.

The data dependence and resource bounds are found using similar techniques as those used during modulo scheduling [5] to find the minimum initiation interval (MII). For a given treegion, the data dependence bound is calculated as the height of the longest true-dependency chain in the DDG. The resource bound is computed as the number of instructions in the treegion divided by the issue width of the target machine.

4.3 Tail Duplication Implementation

Figure 4 shows a partial call graph for the main tail duplication function, `td_treegions()`. The `td_init_candidates()` function is first called to calculate the ICSE for all possible tail duplication candidates. Prior to calling `td_add_candidate()`, candidates that exceed user defined parameters (*e.g.*, maximum number of basic blocks per region) are eliminated to restrict the formation of excessively large regions. This prevents compile time from becoming exceedingly long.

Next, `td_classify_candidate()` is called to classify the candidate into one of four possible types. The classification is based on two factors: (1) the number of predecessor edges entering the child treegion and (2) the number of parent treegions the child possesses. These two factors strongly influence efficiency. For example, assume there exists two edges between a parent treegion *A* and a child treegion *B*. No additional predecessor edges are

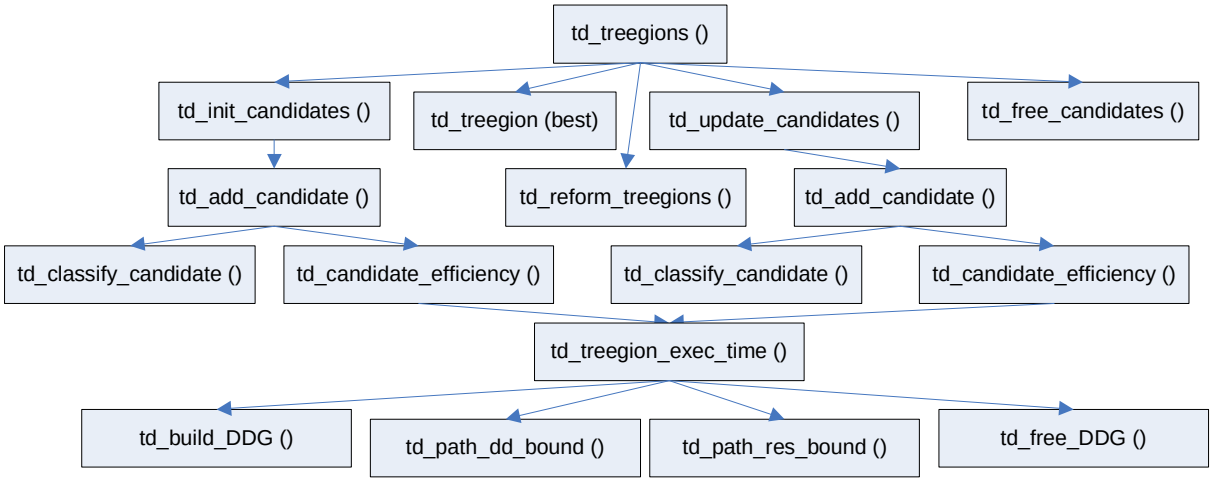

Figure 4: Partial call graph for tail duplication code

entering treegion *B*. This implies treegion *A* is the lone parent of treegion *B*. After duplicating the child treegion, denoted *B'*, both treegion *B* and treegion *B'* can be merged into the parent treegion *A*. This type of duplication produces much larger regions relative to a small code size increase. Alternatively, if three edges are shared between the parent and child only the duplicated treegion *B'* can be merged into the parent treegion *A*.

After classifying the candidate, `td_candidate_efficiency()` is called to calculated ICSE. Based on the type of candidate the estimated execution time, as defined in equation 2, is calculated by `td_treegion_exec_time()`. The estimated execution time is used to approximate the change in IPC before and after tail duplication. To find the resource and data dependence bounds of a treegion the DDG must first be built. This is done using the routines found in `sched-dep.c`. The treegion is then traversed in depth-first order. For each unique path through the treegion the `td_path_res_bound()` and `td_path_dd_bound()` functions are called to find the maximum bound.

Once all ICSEs have been calculated the best candidate is selected for duplication. After duplication, `td_reform_treegions()` is called to incrementally update the data structure of each effected treegion. `td_update_candidates` is then called to recalculate the ICSE for each effected treegion. This incremental updating process is critical for minimizing compile time. After all possible candidates have been duplicated, `td_free_candidates()` is called to free all tail duplication related data structures. Finally, `cleanup_cfg()` is called to optimize the CFG and merge basic blocks. Scheduling then begins after calling `find_treegions()` again due to the fact the calling of `cleanup_cfg()` invalidates all region related data structures.

4.4 Compilation Parameters

Compile time is an important consideration for a production level compiler. Various compilation parameters can be used to limit compile time as well as fine tune the performance of the application being compiled. These parameters are as follows:

1. *max-sched-region-blocks* - limit the size of the region based on the number of basic blocks.

2. *max-sched-region-insns* - limit the size of the region based on the number of instructions.

3. *treegion-max-code-growth* - limits tail duplication based on a maximum amount of code growth.

4. *treegion-icse-threshold* - sets the ICSE threshold. Prior work [4] has shown the optimal range to be between 0.268 and 0.577. A higher threshold results in less duplication.

5. *min-spec-prob* - the minimum probability of reaching a source block for interblock speculative scheduling.

5 Treegion Scheduling

Due to the acyclic nature of treegions, the Haifa scheduler does not require any modifications to accomodate treegions. However, in this section various modifications are proposed to enhance the performance of the scheduler.

5.1 Tree Traversal Scheduling

The goal of Tree Traversal Scheduling (TTS) [6] is to speedup every execution path through the treegion. This is accomplished by prioritizing speculative instructions from different paths which compete for limited resources. Profile information is used to prioritize the scheduling of basic blocks within a treegion.

The algorithm for tree traversal scheduling is as follows:

1. For a treegion, sort the basic blocks according to a depth-first traversal order with the child block selected with the highest execution frequency.

2. Begin list scheduling blocks at the root basic block.

3. During the scheduling of a basic block, consider speculation for instructions dominated by this basic block.

4. Repeat step 3 until all basic blocks in the treegion have been scheduled.

The primary strength of Tree Traversal Scheduling is that the frequently executing path is given highest priority, while the less frequently executing paths are not severly penalized.

5.2 Operation Combining (Future Work)

The application of tail duplication enables the removal of merge point between treegions, producing larger regions. However, despite the benefits, tail duplication has the potential to decrease the performance of the instruction cache and TLB due to the creation of many redundant instructions. In some instances the instruction scheduler can take advantage of *dominator parallelism* to remove redundant operations at schedule time.

Dominator parallelism [7] presents itself when an instruction is speculatively scheduled into a predecessor block that dominates blocks containing redundant copies of the scheduled instruction. In these instances, a form of schedule-time partial redundancy elimination (PRE), also referred to as operation combining, may be applied to remove all but the speculatively scheduled copy of the instruction. The

single remaining instruction performs the operation for all paths. If the instruction is redundant in every control path below the target block the instruction can be made non-speculative.

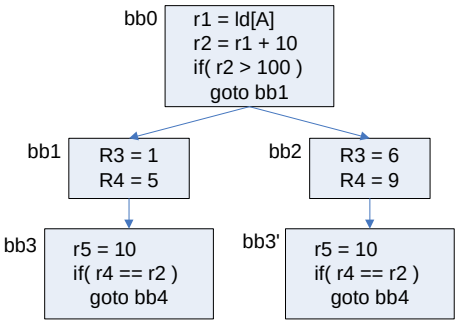

Figure 5: Example of operation combining within a treegion

Figure 5 depicts an example treegion after the duplication of basic block `bb3`, denoted `bb3'`. During the scheduling of `bb0`, the instruction `r5 = 10` can be speculatively moved from both blocks `bb3` and `bb3'` into basic block `bb0`. Assuming `r5 = 10` has already been hoisted from `bb3`, the redundant copy from `bb3'` may be safely eliminated. The scheduler can easily detect this optimization due to the characteristic that any basic block within a treegion dominates all its successor blocks. Any instruction speculated upward is always moved into a dominator. Therefore, if an instruction is speculated into a block where a redundant copy of the instruction has already been scheduled, one copy can be removed.

6 Experimental Results

Experiments were performed to evaluate the performance of the Treegion instruction scheduler. All tests were conducted on an Itanium 2 processor. The benchmark suite consisted of a subset of benchmarks from the SPEC2K suite including: *gzip*, *mcf*, *crafty*, *parser*, *gap*, *bzip2*, *twolf*, *wupwise*, *swim*, *mgrid*, *applu*, *equake*, *ammp*, *sixtrack*, and *apsi*. Profile information was generated using the `-fprofile-arcs` flag and all benchmarks were compiled using the `-O3` and `-fbranch-probabilities` flags. Flags set to control speculation include `-fsched-interblock`, `-fsched-spec`, and `-fsched-spec-load`.

Table 1 presents various region related statistics for the original region formation code, natural treegion formation, treegion formation with tail duplication bounded by an ICSE threshold of 0.577, and treegion formation bounded by a maximum of 100 instructions per region, respectively. The current region formation code produces and average region size of 1.10 basic blocks, containing 8.66 instructions of which 0.09 were speculatively scheduled. Due to the limited scope of the region the opportunity for speculation is limited. Natural treegions, *i.e.,* treegions without tail duplication, produce an average region size of 2.65 basic blocks, containing 20.89 instructions of which 3.65 were speculatively scheduled. Even without the application of tail duplication natural treegions provide greater opportunity for parallelism. Limiting duplication to a ICSE threshold of 0.577 produces only slightly larger regions beyond that of natural treegions. Finally, applying unlimited tail duplication while limiting region size to a maximum of 100 instructions produces an average regions size of 5.70 basic block, containing 35.95 instructions of which 6.00 were speculatively scheduled.

Table 2 shows the speedups for the various region formation techniques. Speedups are relative to basic block scheduling. The execution time of each SPEC benchmark was found by averaging five runs using the *ref* input set. The speedup results vary across benchmarks. The original region code produces speedup for *parser*, *twolf*, and *ammp* while *gap* and

Table 1: Region statistics

	Region	Natural Treegion	Treegion (k = 0.577)	Treegion (100 insns)
# Basic Blocks	1.10	2.65	2.79	5.70
Instructions	8.66	20.89	21.70	35.95
Interblock	0.09	3.65	3.81	6.00

Table 2: Speedup results

	Region	Natural Treegion	Treegion (k = 0.577)	Treegion (100 insns)
gzip	1.00	0.96	0.96	1.03
mcf	1.00	1.00	1.00	1.00
crafty	1.00	0.99	1.00	1.00
parser	1.01	1.01	1.01	1.01
gap	0.99	1.01	1.00	1.00
bzip2	0.99	1.06	1.06	1.06
twolf	1.03	1.01	1.01	1.03
wupwise	1.00	0.99	1.02	1.01
swim	1.00	1.02	1.04	1.01
mgrid	1.00	0.99	0.99	1.00
applu	1.00	1.00	1.00	1.00
equake	1.00	1.00	1.01	1.00
ammp	1.01	1.01	1.00	1.00
sixtrack	1.00	0.98	0.98	0.98
apsi	1.00	1.01	1.01	1.01
average	1.00	1.00	1.01	1.01

bzip slowdown. Natural treegions produce a speedup for seven of the fifteen benchmarks, slowdowns for five of the benchmarks, while three benchmarks remain uneffected. The most significant speedup (6%) is for *bzip2*. For *wupwise*, *swim*, and *equake* the best performance gain is realized using the ICSE threshold. Applying unlimited tail duplication while limiting region size to 100 instructions produce a speedup for seven of the fifteen benchmarks, slowdown for only *sixtrack*, while seven benchmarks remain uneffected. On average the original region formation code and natural treegion formation provide no speedup, while treegion formation with tail duplication bounded by ICSE and treegion formation bounded by instruction count produce and average speedup of 1%.

7 Conclusions

This paper presents the status of the implementation of an architecture-independent, aggressive global instruction scheduler based on Treegions. Natural Treegion formation and tail duplication have been completed and are currently maintained on the sched-tree-branch. The ICSE heuristic has also be implemented as

a means of judiciously applying tail duplication. To ensure compile time does not become exceedingly long fine tuning of this code is an ongoing process. Also, while tail duplication has the benefit of increasing region size, it does introduce redundant instructions. Finally, operation combining is presented as future work for eliminating redundant instructions as schedule time.

Our results show that treegion formation dramatically increases the average region size as compared to the current region formation code. This in turn results in a significant increase in the number of speculatively scheduled instructions. Our results show performance benefits for a few benchmarks (i.e., *parser*, *bzip2*, *wupwise*, *twolf*, *swim*, and *apsi*) while others show little improvement because of architectural features such as memory latencies that hide scheduling improvements. Techniques such as software prefetching should be able to alleviate such issues resulting in future performance gains from Treegion scheduling.

8 Acknowledgments

Thanks go to Diego Novillo and Gerald Pfeifer for their assistance during the opening of the Treegion scheduling branch. Thanks also to TINKER members Balaji Iyer, Paul Bryan, Jesse Beu, and Saurabh Sharma, for their helpful insight.

References

[1] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," in *IEEE Transactions on Computers*, pp. 478–490, 1981.

[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 Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, 1993.

[3] 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 *25th Annual International Symposium on Microarchitecture*, 1992.

[4] H. Zhou and T. M. Conte, "Code Size Efficiency in Global Scheduling for ILP Processors," in *Proceedings of the 6th Annual Workshop on the Interaction between Compilers and Computer Architectures (INTERACT-6) held in conjunction with the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, (Cambridge, MA), February 2002.

[5] M. Hagog and A. Zaks, "Swing Modulo Scheduling for GCC," in *The 2004 GCC Developers' Summit*, (Ottawa, Canada), June 2004.

[6] H. Zhou, M. D. Jennings, and T. M. Conte, "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors," in *Proceedings of the 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, (Cumberland Falls, KY), August 2001.

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