A New Framework for Integrated Global Local Scheduling

Srinivas Mantripragada Suneel Jain Jim Dehnert
High Performance Programming Environment Group
Silicon Graphics Computer Systerms
2011 N. Shoreline Blvd., Mountain View, CA 94043
srinivas@engr.sgi.com

Abstract

Global Instruction Schedulers can be classified as either structure or profile driven. Structure driven approaches attempt to find instruction level parallelism by redistributing instructions along all possible execution paths. When resources are limited, poor choices may penalize the frequently executed paths. By contrast, profile driven approaches use feedback information to identify frequently executed (hot) regions, and attempt to improve their performance. This may be at the expense of less frequently executed (cold) regions, for instance by inserting fixup code. The overall performance improves if the frequency information is accurate and there is a dominant trace in the program. If either of these conditions does not hold, performance may degrade.

We present a novel algorithm that attempts to combine the individual merits of the above two approaches while avoiding some of their drawbacks. We have also incorporated several techniques which improve the global scheduling performance on out-of-order (OOO) processors. Our algorithm is integrated with a parametric resource model and can be applied both before and after register allocation. It has been implemented in the SGI MIPSpro compiler, and the results have been evaluated on the MIPS R8000 and R10000 processors.

1 Introduction

The ever-increasing complexity of superscalar processors poses a serious challenge for compilers to generate code which can achieve the full potential of the machine. Earlier studies [9] have shown that extracting instruction level parallelism at the basic block level is not enough. Several strategies have been proposed to extend the parallelism extraction phase beyond basic block boundaries, taking into account the constraints of latency hiding, resource availability, and data dependencies. We call such a phase, which moves instructions globally for the purpose of increasing ILP, a global scheduler.

We classify global scheduling algorithms as either *profile* or *structure* driven. A structure driven scheduler attempts to identify parallelism along all execution paths within a region, with priorities based on program structure. Region scheduling [3], percolation scheduling [11], global instruction scheduling [1], all fall in this category. These approaches attempt to increase parallelism by moving operations between basic blocks without considering execution frequency. As a result, their decisions may not always be profitable.

Recently, there has been more focus on feedback-directed compilation. Profile driven schedulers attempt to identify the most frequently executed regions in the procedure (or loop), by using either feedback data or static estimates. Trace scheduling [2], superblock scheduling [4], and hyperblock scheduling [8] fall in this category. The essence of these approaches is to identify the most frequently executed paths (i.e. sequence of basic blocks), coalesce them into an extended block, apply a local scheduling algorithm, and make any necessary corrections by inserting compensation code at side exits and entries.

Both approaches use a similar infrastructure. The primary data structure used is a dependence graph where the nodes are the operations and the arcs represent the minimum latency which must separate the two operations it connects. These latencies are usually well defined, and model optimistic program behavior for instructions with variable real latencies. This works well when the assumptions match real runtime latencies, but events such as branch misprediction, cache misses, etc., can render the compiler's schedule quite inaccurate. Balanced scheduling [6] is an example of a structure directed approach which attempts to address some of the issues at the basic block level.

We believe that the best results will be obtained by balancing profile and structure driven techniques, while modifying the traditional dependence graph model to better reflect out-of-order (OOO) processor behavior. We present an **I**ntegrated **G**lobal **L**ocal **S**cheduling (IGLS) framework based on this balanced approach. Our framework supports IGLS phases both before and after global register allocation (GRA). Our approach has been to treat register allocation and scheduling separately. This minimizes the additional complexity required to combine the two in a single phase. By applying the IGLS phase both before and after register allocation, some of the weaknesses due to phase-ordering between the two can be minimized, and each can perform specialized subtasks.

The remainder of the paper is organized as follows. Section 2 reviews earlier work in this area. Section 3 presents the background of our work. Section 4 describes the IGLS framework in detail. Section 5 presents the OOO issues and the adjustments made. Section 6 describes the compile speed issues that were addressed. Finally, Section 7 presents the experimental results and Section 8 summarizes this work.

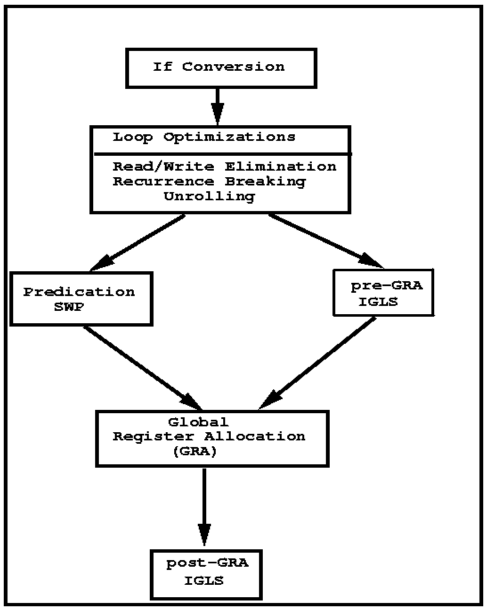


Figure 1: **Code generator framework indicating invocation of IGLS phases.**

2 Related Work

Trace scheduling [7, 2] starts by picking the main (or the most frequently executed) trace in an acyclic flow-graph. Operations are then allowed to move past branches or other independent operations within the trace. Compensation copies are inserted wherever necessary to maintain the original program order. As a result, the main traces are optimized at the expense of off-trace paths even though both paths may have similar performance impact. This can work well in scientific programs where the main trace is the dominant path, but may not work so well in non-numeric programs. *Superblock scheduling* [4] attempts to address the problem by duplicating the off-trace paths, essentially creating one (or more) superblocks with a single entry and multiple exits. The problem of bookkeeping and inserting compensation copies (as in trace scheduling) is avoided at the expense of code duplication. This can result in code bloat and other undesirable effects. *Hyperblock scheduling* [8] attempts to alleviate the problem further by *if-converting* the paths with similar execution behavior. This is a good technique for architectures supporting a gamut of predicated instructions and where the execution costs of predication are low. In the MIPS architecture there is limited support for predicated execution (in the form of conditional moves). Emulating the effect of full predication is expensive and the approach is not viable.

Among structure driven approaches, Bernstein [1] used the program dependence graph for global scheduling. Their global scheduling approach included reordering of instructions among equivalent blocks or 1-level speculative blocks within a given loop body. Region scheduling [3] also used the PDG and provided a more powerful set of code transformations. Regions are classified according to their parallelism content and are used to drive the transformations. However, the approach is applied very early in the code generation process and isn't integrated with any resource model.

We feel that profile driven approaches are important to direct the effort to the most profitable sections in the program. However, the additional expense of either bookkeeping or tail duplicating the off-trace paths to preserve some desired order are the artifacts of the current approaches and not the general problem inherent in profile driven approaches. Also, the current profile driven approaches apply all code reordering transformations within the trace/block(s) initially formed. This makes them highly sensitive to the initial selection of traces/blocks. We present an IGLS framework that does not have this limitation. Frequency weighted estimates are used to prioritize the selection of blocks but the code reordering transformations from/to them are not limited to any trace/extended-block. This is effectively how structure directed approaches work except that now the direction of code transformations and the order of block selection is more organized. As a result, the algorithm is less sensitive to the initial selection of hot regions.

Also, the selection of the appropriate motion type and target block selection is made flexibly depending on the block's current properties. This is untrue for most structure directed approaches, as the order of applying the transformations is pre-determined.

Previous global scheduling work has not addressed the issues related to OOO processors. We feel that static sched-

FOREACH procedure (region) {
  Determine the dominator/post-dominator information
  Detect all loops and create a loop list **L** inside this region.
  Invoke **LOCS** for an initial local scheduling pass.
  WHILE the loop list **L** is not empty {
    Sort the blocks in **L** using the **priority_selector**.
    WHILE the priority_list is not empty {
      Invoke the priority_function P_i to request block B_s
      Invoke **motion_selector** to describe motion_type M_k for B_s
      Invoke **target_selector** to then determine target blocks **T**.
      (The selection of target blocks conform to motion_type M_k.)
      FOREACH block T_t in target block list **T**
        Select one (or more) instruction(s) I_l to move from
        B_s to T_t or vice-versa.
    }
  }
}

Figure 2: **IGLS Framework**

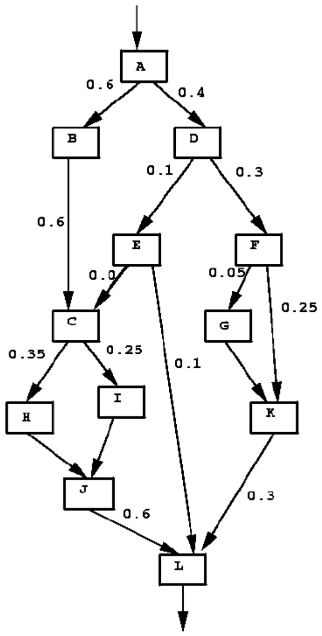


Figure 3: **A Sample Control Flow Graph to compare the Profile-driven, Structure-driven and the proposed Balanced Approach.**

ule estimates are not sufficient to decide the profitability of global scheduling in OOO processors. This led us to devise new heuristics to balance the variations in schedules due to run-time (or other) effects.

3 Background

The IGLS phase in the compiler is part of the code generator, which is invoked after all the machine independent optimizations have been performed. Figure 1 shows how IGLS fits in with the other code generator phases. The unit of compilation is a *region*, which is a single entry, multiple exit, code fragment. A region may either be the whole procedure, or a part of a procedure. The if-conversion phase is responsible for converting control dependences into data dependences where profitable. One example of a profitable conversion is the computation of the maximum of two elements. A loop optimization phase comes next, removing redundant cross-iteration read/write operations, eliminating cross-iteration common subexpressions, breaking recurrences, and unrolling the loop body. Finally, it estimates the minimum initiation interval of a loop, and decides whether to attempt software pipelining [12]. Code which is not software pipelined is passed to the *pre*-GRA IGLS phase. After register allocation, a *post*-GRA IGLS phase is invoked.

The intermediate language used by the IGLS phase represents a sequence of operations closely matching the machine code of the target machine. In the pre-GRA IGLS phase, the operands and results of these operations are abstract symbols called TNs (Temporary Names). Register allocation binds TNs to physical registers, so the post-GRA IGLS phase deals with machine instructions with the registers allocated.

4 The IGLS Framework

The IGLS framework is shown in Figure 2. It is split into three separate components: the *Priority selector*, the *Motion selector* and the *Target selector*. Decoupling the IGLS phase into these three units allows us to leverage the individual benefits of profile/structure approaches at these component level. The local instruction scheduler (LOCS) is embedded within the IGLS framework. Before global scheduling, a local scheduling pass is done to estimate the profitability of code reordering decisions. The dominator/post-dominator information is then calculated. All the loops in the region are then detected and a loop-list is created. The outermost body (i.e. code present outside the outermost loop but within the region) is also kept in the loop-list. The properties of the loop including nesting level, presence of multiple blocks, number of side-exits, presence of calls, etc. are then determined. This helps to understand the behavior of the loop and select appropriate global scheduling transformations.

The IGLS phase traverses the loop-list, starting with the innermost loops and proceeding outwards. While processing outer loops, we ignore blocks that have already been processed (i.e those comprising inner loops). Since some of the global decisions may not always be optimal at the local level, a second pass of the LOCS is invoked after IGLS phase.

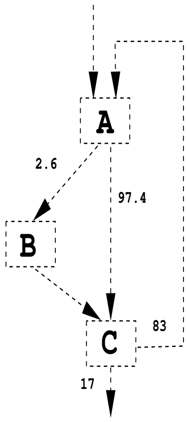
4.1 Block Priority Selection

The first IGLS phase is responsible for picking the desired order of scheduling blocks within a PU (or region). The intent is to identify the hot or critical blocks and schedule them first before processing blocks which are less important. The key to the proposed balanced approach lies in the *selection* of which blocks to schedule next. The approach is to schedule the most likely executed block among all unscheduled blocks. The processing order of two blocks need not follow any particular program order.

Block frequencies are used first in the prioritized selection. Block parallelism (measured as a ratio of the number of operations present in the block divided by the length of the longest critical path) is then used to give slightly higher priority to blocks with excessive or insufficient parallelism. Blocks whose sizes are greater than a threshold value are given lesser priority.

Figure 3 shows a sample control flow graph used to compare the three different approaches. The blocks are labeled from *A-L* and the arrows between blocks represent control flow edges. The edges are annotated with the overall probability of executing that path. In the profile driven approach, traces starting from the seed block A, result in 5 distinct traces (*A-B-C-H-J-L; D-F-K; I; E; G*). Once these traces are selected, the code motion transformations are limited to that trace and are not applied between traces. This may result in empty instruction slots if there is no more ILP along that trace to exploit.

Often, critical blocks in a trace contain enough opportunities of downward instruction movement by replicating them to off-trace blocks (if necessary). This is clearly not possible if trace boundaries are strictly enforced. In addition, off-trace paths with higher probability are given lesser priority in the scheduling process just to enforce a desired order of block selection. For example, block *H* with a lesser cumulative probability *cp* (0.35) is scheduled before block *D* with a higher *cp* (0.4). Such limitations can have negative effects. The instruction scheduling trace for the structure driven approach is *A-B-D-C-E-F-H-I-G-J-K-L.* Since both paths are considered as equally probable, the selection criterion is to pick lexicographically adjacent blocks at the same level before proceeding to the next level and so on. If there are limited resources, this can result in poor decisions, often penalizing the frequent blocks. The order of selection of blocks for the proposed balanced approach is *A-L-B-C-J-D-H-F-I-K-E-G*. As is evident, no artificial barriers to code motion are enforced and the selection criteria are purely based on block properties.




```
unsigned char window [0x8000];
unsigned strstart;


int longest_match(cur_match)
{
  unsigned cur_match;
  unsigned chain_length = 32;
  unsigned char *scan = window + strstart;
  unsigned char *match;
  register int len;
  int best_len = prev_length;

  unsigned char *strend = window + strstart + 258;
  unsigned char scan_end1 = scan[best_len-1];
  unsigned char scan_end = scan[best_len];

  do {
    ------------------------------------------------
    match = window + cur_match;
                                                            A
    if (match[best_len] != scan_end || match[best_len -1] != scan_end1 ||
        *match != *scan || *++match != scan[1]) continue;

    scan += 2;
    ------------------------------------------------
    match++

    do {
    } while (*++scan == *++match && .......)                B

    .....
    .....
    ------------------------------------------------
  } while ((cur_match = prev[cur_match & (0x8000-1)]) > 0
          && --chain_length != 0);                          C

  return best_len;
    ------------------------------------------------
}
```

Figure 4: **Source code from example loop from** *gzip***.**

4.2 Motion Selection

This phase is responsible for picking the desired order of applying code transformations once the block is selected for scheduling by the priority descriptor. The properties of the block, including block size, block parallelism, or presence of certain specific constructs (e.g.. entry, exit or calls) in the block, are first determined. The order of applying transformations is then selected to improve the existing schedule of the block. For example, if a block has more parallelism than is provided by the architecture, priority is given to transformations which will move code out of the block. In contrast, code motion transformations for blocks with unused parallelism result in moving code into the block. Relative functional unit usage summaries are calculated which help in setting instruction movement priorities. For example, the entry/exit blocks have high load parallelism when compared to other types (say integer, shift,...), so the priority will be to move loads out of this block to candidate blocks (where the load parallelism level is very low). The default order is to perform code motion among control equivalent blocks first since it is always *useful*. Code motion across calls is also included in this category.

Speculative movement is then performed for instructions that will not cause any exceptions. Additional contextsensitive analysis is performed to identify more common patterns which can be assumed to be safe. Most notable among them is NULL pointer speculation in which a pointer dereference is speculated past the condition which checks for the NULL condition (e.g.. $if(p \;\; \&\& \;\; *p \; < \; 10)$ where p is a pointer and the load of the pointer address is speculated past the test for zero condition). This requires kernel collaboration to map page zero as readable and is provided as an user option. Another example of directed speculation is to move a load (e.g.. $lw \;\; r_x, k1(r_z)$) from a source to the destination block containing $lw \;\; r_y, k2(r_z)$ if it can be detected that register r_z is not modified in between.

Circular scheduling [5, 10], is a loop scheduling scheme by which instructions from the top portion of the loop body are moved to the bottom portion of the loop body. This can improve performance if, (1) the instructions in the top portion form a critical set of instructions and by moving them can remove unnecessary stalls, and, (2) the bottom half has enough slack cycles that the instructions moved from the top can be accomodated without introducing delays. This also helps in creating additional opportunities for moving instructions from the bottom half higher. Since the profitability decisions are no different from those made by other transformations performed by IGLS, it is a natural extension. In our current implementation, circular scheduling is applied after all pre-IGLS transformations have been applied. It is the first transformation performed by the post-IGLS phase. The loop is transformed into a circular scheduled form before other post-IGLS transformations are applied.

```
.L.2.25.temp:
  addu $14,$4,$24              [0]
  lw $11,%got_disp(prev)($1)  [1]
  addu $13,$2,$14              [1]
  lbu $3,0($13)                [2]
  bne $3,$5,.L.2.40            [5]
  nop
.BB3.longest_match:
  ld $3,24($sp)                [0]
  bne $12,$3,.L.2.40           [3]
.L.2.34:
  lbu $6,0($9)                 [0]
  lbu $5,0($14)                [1]
  beq $5,$6,.L.2.31.temp       [4]
  nop                          [4]
.L.2.40.temp:
  addiu $12,$0,1               [0]
.L.2.31.temp:
  bne $12,$0,.L.2.41.temp      [0]
  nop                          [0]
...
...
...
...
.L.1.41.temp
  lw $5,%gp_rel(prev)($gp)    [0]
  andi $25,$4,32767
  addu $10,$25,$25             [1]
  addu $13,$11,$10             [2]
  lhu $4,0($13)                [3]
  beq $4,$0,.BB42.             [6]
  addiu $8,$8,-1               [6]
.BB36.
  bne $8,$0,.L.1.25.temp+4    [0]
  addu $11,$4,$10              [0]
.BB42.
  jr $31                       [0]
  nop                          [0]
```

```
.BB58.
  addiu $14,$4,$24             [0]
  lw $11,%got_disp(prev)($1)  [1]
  addu $13,$2,$14              [1]
  lbu $3,0($13)                [2]
  andi $25,$4,32767            [3]
  lbu $12,-1($13)              [3]
  addu $10,$25,$25             [4]
  lw $11,0($11)                [4]
.L.2.25.
   bne $3,$15,.L.2.40          [0]
   nop                         [0]
.BB3.longest_match
   bne $12,$3,.L.2.40          [0]
   nop                         [0]
.L.2.34:
   lbu $6,0($9)                [0]
   lbu $5,0($14)               [1]
   beq $5,$6,.L.2.31           [4]
   or $12,$0,$0                [4]
.L.2.40:
   addiu $12,$0,$1             [0]
.L.2.31:
   bne $12,$0,.L.2.41          [0]
   addu $13,$11,$10            [0]
   ...
[0]          ...
   ...
   ...
.L.2.41
   lhu $4,0($13)               [0]
   addiu $8,$8,-1              [2]
   beq $4,$0,.BB42.            [3]
   addu $14,$4,$24             [3]
.BB36.
   andi $25,$4,32767           [0]
   addu $13,$2,$14             [0]
   lbu $12,-1($13)             [1]
   lw $11,%got_disp(prev)($1) [2]
   lbu $3,0($13)               [3]
   addu $10,$25,$25            [4]
   bne $8,$0,.L.2.25           [4]
   lw $11,0($11)               [4]
.BB42.
   jr $31                      [0]
   adiu $sp,$sp,48             [0]
```

Figure 5: **Generated Assembly before/after IGLS for the example loop.**

4.3 Target Block Selection

This phase is responsible for selecting the prioritized list of candidate blocks once the source block is determined by the block priority selector and the order of applying code motion transformations is selected by the motion selector. The selection of candidate blocks is made to supplement the decisions made by prior selectors. Useful code movement from an infrequent block to a frequent block is rare.

The following assumptions are observed when deciding which blocks should be considered as candidate blocks. If the source block has excessive parallelism (or is bottlenecked by load parallelism), suitable candidate blocks are selected which have unused capacity in either (or both) of them. Empty blocks are never considered as candidates since they, along with branches around them, can be eliminated by the control flow optimizer invoked after IGLS phase. Candidate blocks which are below a frequency threshold relative to the source block are not considered. The threshold value is set higher for profiled frequency than for estimated frequency. Special cases are used for blocks with very few instructions, giving them higher priorities. This sometimes can result in empty blocks, which along with the branch (if any) around them can be removed, at a later phase.

Once the candidate blocks are selected, the operations are moved across multiple blocks. The block characteristics are recalculated as a result of the movement. A block can serve as a *target* block several times but as a *source* block for a specific motion type only once.

4.4 A Program Example

We present a sample example routine to illustrate the effect of IGLS. The $longest\_match$ routine constitutes about 50% of the execution time in *gzip*, a compression utility program. The source code for the loop is shown in Figure 4. The important sections of the code fragment are labelled as **A,B,** and **C** with runtime execution frequencies as shown in the graph. Sections **A** and **C** constitute 97.4% of the execution time in the loop. The optimized generated assembly is as shown in Figure 5. The code for section **B** has been ignored for clarity. Assuming 2 ALU, 1 memory and 1 branch instruction per cycle, the cycles are shown in the right. In Figure 5(a), the blocks labelled from .L.2.25.temp to .L.2.31.temp test for four early exit conditions in section **A**. The blocks labelled from .L.1.41.temp to

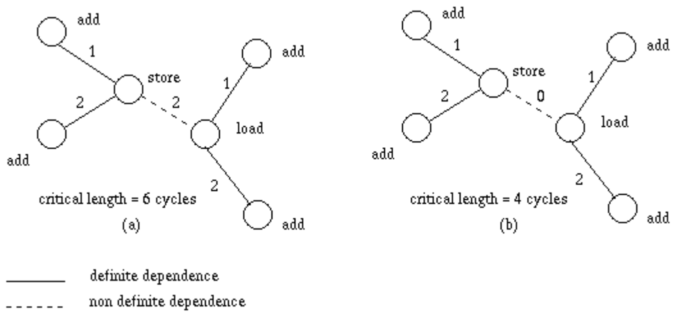


Figure 6: **(a) An Example showing non-definite memory dependence (b) Relaxation using the proposed approach.**

.BB36 constitute the loop exit conditions in section **C**. The schedule length of the loop is 25 cycles. As a result of pre-IGLS transformations, the instructions which do *cur_match & (0x80000-1)* have been moved from section **C** to section **A**. This results in a reduction of 3 cycles in section **C**. Circular scheduling is then applied at the beginning of the post-IGLS phase. As a result, the loads of *match[best_len]*, *match[best_len-1]* and instructions earlier moved from section **C** (*cur_match & (0x80000-1)*) have been speculated from section **A** and circular scheduled with the instructions in **C** section. The other post-IGLS transformations didn't result in any further profitable movement. The net number of cycles, as result of combined pre-IGLS and post-IGLS transformations, has been reduced to 18 (28% improvement in performance).

5 OOO Issues and Machine Adjustments

We make a number of adjustments to the traditional underlying machine model and dependence graph for OOO processors. Many of these adjustments allow static scheduling to better model OOO execution by modifying the usual static execution model.

Memory Dependence Adjustments : The freedom of global scheduling is often limited by memory dependences. These dependences disallow the movements of loads past stores. Since the memory instructions constitute a significant portion of the dynamic trace and often constitute the critical path, the inability to determine independence at compile time can seriously limit the scope of static scheduling. This led us to treat memory dependences as two separate categories: *definite* or *non-definite*. A non-definite memory dependence is a dependence relationship which cannot be proved to be valid all the time, but can potentially exist under some condition(s) which cannot be determined statically. We convert all *non-definite* memory dependences to arcs with zero latencies. Since, there is a high probability that this dependence will turn out negative at run-time, biasing such dependences to zero latencies produces a better estimation of critical lengths, early/late start times etc. This may enhance the applicability of some optimizations (e.g. if-conversion, unrolling, scheduling, etc.) which rely heavily on critical path lengths, and operation slack periods in deciding the profitability of transformations. For example, in Figure 6, the critical height is reduced from 6 to 4 cycles as a result of the relaxation.

Load Predictability Issue : Another consideration is the difference between loads which have high predictability and those which have high cache miss rates. The loads which fall in the later category usually are indirect loads whose address expressions are unknown at compile time, or, whose base addresses are variant expressions of induction variable updates in a loop. We identify such loads and mark them with slightly higher latencies than regular loads. This induces the scheduler and software pipeliner to reorder enough instructions to keep the processor busy while a possible cache miss is being serviced.

Branch Misprediction Penalty : In today's OOO microprocessors with deep pipelines, the cost of a branch misprediction is very high. In the R10000, the cost of a branch misprediction is 8 cycles. We factor in branch misprediction costs in addition to other heuristics in deciding the profitability of code movement. If an operation is moved from a block that is the target of such misprediction to above the branch, the instruction can be executed while a branch misprediction is being serviced. This factor allows us to move instructions from a lower probability block to the predecessor block.

Conversion to Branch-likelies : Most OOO processors provide flexibility to the compiler to provide hints to the hardware regarding the nature of a branch that can be predicted accurately at compile time. For example, the behavior of most loop-back branches, branches to error handling code, assertions, etc., can be predicted quite accurately by the compiler and the information can be passed to the hardware. Since the branch prediction table is often limited in size, time is often lost due to unnecessary branch cache collisions. This may also result in poor prediction of branch behavior as insufficient global history has been gathered before the branch is predicted. These collisions can be avoided by the compiler, either by removing the poorly-predicted branches (e.g. if-conversion), or by converting highly predictable branches to a different form. In our case, the MIPS architecture supports *branch-likely* instructions which don't occupy any slots in the prediction table and are always predicted as *taken*. We generate all branches which have taken probability greater than 0.95 as branch-likely instructions.

Branch Target Alignment : We devised a new algorithm to incorporate branch target alignment within the IGLS framework. Our previous implementation performed the alignment phase as a postpass to all the optimizations. This sometimes resulted in too many *noops* being inserted in order to align the hot regions. By embedding the alignment decisions into the IGLS framework, some of the decisions for code movement can either be avoided or be done based on alignment objectives avoiding *noops* later. A second pass of alignment is done during the code layout phase to correct some of the earlier assumptions.

The heuristics that we use to decide whether to align the block includes the following assumptions. Infrequently executed blocks ($<$0.1) are not aligned. Noops are not added to a previous block unless the current block is at least five times more frequent. Typically this happens to *loop head* blocks which are at least five times more frequent than the loop prolog block. For all predecessors of the current block, the taken_freq fall_thru_freq is calculated. The target PC value is then calculated determining whether the ratio of taken_freq/fall_thru_freq or its inverse is greater than five times the threshold value. The number of ops is then determined that need to be added (using the computed PC value and the number of words present in the cache line) in order to align this block.

The same interface routine is used both by the IGLS phase and the final code layout phase when noops are inserted to align the hot regions. The IGLS calls this routine to determine the number of *noops* that need to be added in case the decision is made to align the source and/or the target block. The *vacant* slots are factored into the code reordering decisions.

Local Scheduling Alterations : OOO adjustments have been made to the local scheduler as well. One example is prefetch operations. Prefetch operations are scheduled like any other memory instruction. Usually, the prefetch nodes do not have any successors (at least in the acyclic domain) and end up with huge slack values. As a result, these operations end up being clustered at the end of the loop body. Too many concurrent prefetch accesses can cause dirty misses, resulting in extra bubbles in the memory pipeline, and additional stalls due to further cache misses. The OOO adjustments give them higher priority and spread them out in the block.

Another technique is to schedule the operations which define the operands of a branch as far away from the branch as possible. This can help in early computation of the condition and can help minimize speculations beyond the branch which occur before the condition is evaluated.

6 Compile Speed Issues

Several heuristics have been implemented to limit the effort to the most profitable sections of a loop. Loops whose nesting levels are less than the optimization levels are given lower priority. Basic blocks which can be unreachable as a result of intermediate optimizations are avoided. Large basic blocks whose sizes are greater than a threshold value are not processed. Low frequency ($<$ 0.2) blocks are ignored. Appropriate circuit breakers are added (wherever necessary) to reduce the compile-time effort. The intensity of the circuit breakers varies depending on the nesting level of the loop and the optimization level used. During pre-IGLS phase, legality checking for code movements is limited to comparisons with global TNs (since local TNs are not live out of the block) and live-in/live-out sets on off-trace paths. The additional overhead compiling the SPEC integer benchmarks averaged 2%-3%.

7 Performance Evaluation

In this section, we report experimental results on the applicability of the proposed technique on MIPS R8000 (75 Mhz) and MIPS R10000 (195 Mhz) processors. The MIPS R8000 is a 4-issue superscalar RISC processor. It contains two arithmetic logic units (ALU), two floating point units (FPU) and two load store units. The processor provides in-order dispatch of up to four instructions each cycle. The MIPS R10000 is a more complex processor providing OOO execution and complex branch prediction hardware. The processor can issue up to four instructions per cycle. The chip provides two arithmetic logic units (ALU), three floating-point units and an address calculation unit. The integer and the floating point register files have 64 registers each 64-bits wide, of which only 32 registers are visible to the external user. The chip uses the other 32 registers for its internal use which includes supporting the speculative state of the machine, performing OOO execution etc. Both the MIPS R8000 and MIPS R10000 support the MIPS IV instruction set. The results were studied on both processors to measure the effectiveness of global scheduling techniques on processors which support in-order and out-of-order execution respectively.

Table 1 shows the percentage improvement in runtime performance obtained by enabling the pre-GRA and post-GRA IGLS phases in the MIPSpro compiler. The baseline measurement was done using the *-O3* options for all the SPECint95 benchmarks. The baseline compiler includes a powerful set of loop optimizations by the loop nest optimizer, machine independent optimizations by the global optimizer and modulo scheduling [12]. The only form of scheduling enabled for the baseline compiler is from the local scheduling (LOCS) phase. The LOCS phase is implemented as a backward cycle-driven list scheduler and is invoked both before and after register allocation phase. Columns 2 and 3 show that the combined IGLS phases (pre-GRA and post-GRA) are able to improve the performance for all the benchmarks in both the R8000 and R10000 processors. In the R8000 case, most notable among the improved ones are *go, compress, ijpeg* and *vortex* where the performance improvements are 11.4%,16.71%, 16.92% and

Benchmarks	R8000 Speedup (%)	R10000 Speedup (%)	R10000 (OOO) Speedup (%)
099.go	11.4	6.1	2.1
124.m88ksim	1.2	4.8	1.9
126.gcc	4.0	4.9	1.5
129.compress	16.71	3.8	1.2
130.li	4.11	4.2	1.5
132.ijpeg	16.92	3.4	1.0
134.perl	3.4	2.8	1.7
147.vortex	15.42	6.1	3.8

Table 1: **Performance Improvements for SPEC95 integer benchmarks using IGLS. Columns 2 and 3 show the speedup (in percentages) over results obtained with IGLS disabled. Column 4 shows the additional performance improvements over the results obtained when the OOO heuristics are not used.**

15.42% respectively. The overall speedup ranges between 1%-17%.

In the R10000 processor case, the speedup with IGLS enabled ranges between 3%-6%. Most notable among the improved ones are *go* and *vortex* where the performance improvement is approximately 6%. Smaller performance improvement relative to the R8000 is expected since the R10000 processor provides dynamic scheduling with OOO execution, which makes it less sensitive to compile-time (or static scheduling) techniques. Column 4 shows the additional performance improvements obtained when the proposed OOO heuristics are added to the scheduling framework. This results in an additional 1%-4% improvement over results obtained when the OOO heuristics are not used. This shows that the performance of an OOO processor can be improved further if assumptions to better reflect OOO processor behavior can be integrated with code reordering decisions.

We also observed that the IGLS phase done before register allocation was able to improve the average performance of SPECint95 benchmarks by an additional 9.6% for R8000 processor and an additional 3.2% for the R10000 processor case, when combined with the IGLS phase done after register allocation. This shows that adding an additional global scheduling phase before register allocation supplements the scheduling phase after register allocation.

8 Summary

In this paper, we present the IGLS framework, which consists of three major components: the priority selector, motion selector and target selector. Working together, the three components allow us to combine the individual merits of the structure and profile driven approaches while avoiding some of their drawbacks. In addition, we propose new techniques which improve the global scheduling performance on a OOO processor. The IGLS scheme yields good improvements over already optimized codes and incur only a modest compile-time overhead. The benefits are greater for the MIPS R8000 processor as compared to the R10000 processor when similar scheduling heuristics to determine profitability are used. The performance on an R10000 processors improves further when the proposed OOO heuristics are integrated with the scheduling decisions.

References

[1] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Conference Record of SIGPLAN Programing Language and Design Implementation*, pages 241–255, 1991.

[2] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[3] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, SE-16(4):421–431, March 1990.

[4] W. W. Hwu and et al. The Superblock: An Effective Structure for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 1993.

[5] S. Jain. Circular Scheduling: A New Technique to Perform Software Pipelining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–228, June 1991.

[6] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June, 1993.

[7] P. G. Lowney and et al. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, pages 51–142, 1993.

[8] S. Mahlke and et al. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, December 1-4, 1992.

[9] S. Moon and K. Ebcioglu. An Efficient Resource Constrained Global Scheduling Technique for Superscalar and VLIW processors. In *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, December 1-4, 1992.

[10] S. Moon and K. Ebcioglu. Parallelizing Non-Numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, 1997.

[11] A. Nicolau. A Fine-Grain Parallelizing Compiler. Technical Report Tech Report No. 86-792, Cornell University, 1986.

[12] J. Ruttenberg, G. Gao, and et al. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1996.