Instruction scheduling in the TOBEY compiler

by R. J. Blainey

The high performance of pipelined, superscalar processors such as the POWER2™ and PowerPC™ is achieved in large part through the parallel execution of instructions. This *fine-grain* parallelism cannot always be achieved by the processor alone, but relies to some extent on the ordering of the instructions in a program. This dependence implies that optimizing compilers for these processors must generate or *schedule* the instructions in an order that maximizes the possible parallelism. This paper describes the parts of the TOBEY compiler which address the instruction scheduling issue.

Introduction

The TOBEY[1] family of compilers is designed to optimize code for the superscalar IBM RISC System/6000® (RS/6000) computers based on the POWER, POWER2™, and PowerPC™ architectures. All of these machines have pipelined, superscalar implementations that manage instruction-level parallelism in hardware. The implementations differ in the degree of superscalar parallelism, the depth of pipelines, and the latencies of instructions. Advanced models also include register renaming and true out-of-order execution. This variety of implementations has placed great demands on the compiler and particularly on the instruction scheduler. The scheduler is the compiler component responsible for the reordering and replicating of instructions for the purpose of minimizing execution time for a given target machine. The scheduler must be flexible enough to generate code optimized for machines with a wide range of capabilities. The scheduler must also be portable, to ease the application of TOBEY compiler technology to a larger class of target machines. This paper describes the fundamental algorithms used in the TOBEY instruction scheduler, along with the engineering solutions designed to make them work somewhat independently of the target machine. The IBM Haifa Scientific Center, the IBM Toronto Laboratory, and the IBM Thomas J. Watson Research Center jointly developed the instruction scheduler.

Overview of the TOBEY compiler

Figure 1 depicts the TOBEY compiler organization, highlighting the role of instruction scheduling. One of several language processors translates the source programming language, such as C, C++®, or FORTRAN, into a common intermediate language. The intermediate form is analyzed and transformed by a suite of global optimizations. The subsequent compiler stages include instruction scheduling, register allocation, and final code generation.

[1] Toronto Optimizing Back End with Yorktown. TOBEY is a common component in the AIX® XL FORTRAN/6000 and CSet++® compilers.

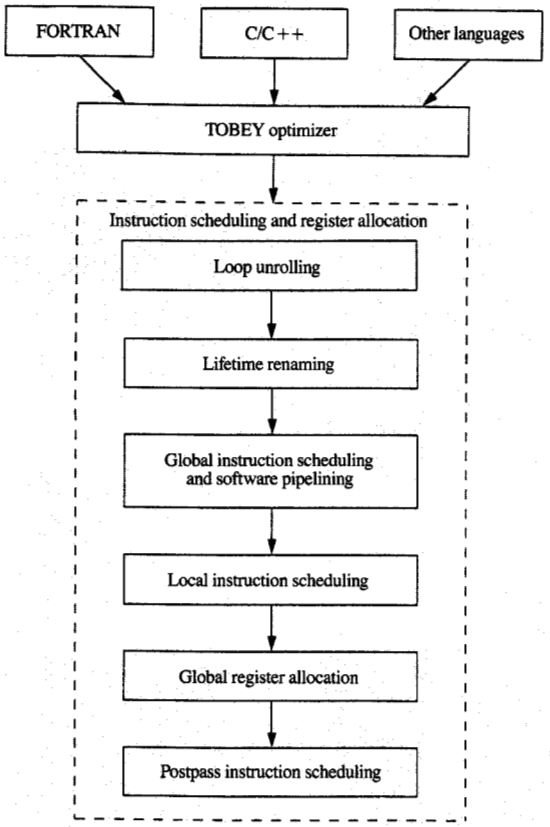


Figure 1

Instruction scheduling in TOBEY.

The program undergoes two transformations before the first phase of instruction scheduling is done. First, inner loops are unrolled to expose more independent instructions per iteration. Next, the lifetimes[2] of variables are analyzed and renamed so that each unique lifetime has a unique name. The *global scheduling and software pipelining* phase performs global code motion and high-level loop transformations, providing greater flexibility and opportunity to the two scheduling phases which follow. The *local scheduling* phase creates basic block instruction schedules using a sophisticated model of the target machine. *Global register allocation* assigns registers to variables using an enhanced implementation of the Chaitin algorithm [1]. Finally, the *postpass instruction scheduling* phase schedules code generated by the register allocator, schedules interlocks due to register assignment, and performs some special-case scheduling of branch instructions.

[2] A lifetime of a variable is a portion of the program over which the variable contains a useful value. The lifetime begins with the definition and ends with the last use of the variable.

Local instruction scheduling

The local instruction scheduler processes linear, branch-free segments of a program (*basic blocks*), reordering to optimize the use of the target machine. In general, instruction scheduling reorders instructions subject to control flow and data dependence constraints. Limiting the scope to basic blocks allows the local scheduler to move instructions without being concerned about legality of code motion across control flow. This leaves data dependence as the only constraint to reordering.

The local scheduler divides a basic block into *windows* of instructions. A window is delimited by reaching a maximum size, by reaching a boundary instruction (for example, a trap or special *fence* instruction), or by reaching the end of the block. The scheduler processes windows of limited size primarily to avoid the excessive compile time of reordering long sequences of instructions. To improve the scheduling of instructions which are near window boundaries, the windows are overlapped. For each window, the local scheduler builds a *dependence graph* in which the nodes are instructions and the directed edges correspond to some type of data dependence between instructions.

Using the dependence graph, the scheduler executes an algorithm called *list scheduling* to issue all nodes in the graph in an order which minimizes pipeline delays and idle processor cycles. The algorithm is essentially a time-driven simulation of the target machine where, in each cycle, one or more instructions may be issued.

- *Dependence graph*

The scheduler analyzes the sequence of instructions to be reordered, identifying any interesting data dependences. Three types of dependence are typically of interest: *true dependence*, *antidependence*, and *output dependence*. An instruction has a true dependence on a previous instruction if it uses a value generated by the other instruction. An instruction has an antidependence on a previous instruction if it writes to a register or memory location which is used by the other instruction. Finally, an instruction has an output dependence on a previous instruction if it writes to a register or memory location which is also written by the other instruction. **Figure 2** shows examples of each type of dependence.

The scheduler labels or *weights* the dependence graph edges with nonnegative integers representing the pipeline delay that would result if the two instructions represented by the incident nodes were issued in sequence. For example, if the edge represents the dependence of a fixed-point instruction on a value loaded from memory, the edge

is labeled *1* to represent the single cycle typically required to access the data cache. Edge weights may also be assigned the special value *weak* if the incident nodes represent instructions that may be executed in parallel on the target machine. Weak edges represent an ordering relation (that is, the dependent instruction may not be issued before the instruction on which it depends), but do not require the incident instructions to be issued in different cycles. A weak edge is typically used when there is an antidependence or output dependence between two instructions.

The dependence graph used by instruction scheduling also includes information about the *execution time* of each instruction. In our model, the execution time of an instruction is a list of execution resources (such as functional units and register files) required and the number of cycles consumed on each functional unit. The number of consumed cycles on a functional unit is the minimal number required to compute the result, not including rename, decode, or writeback stages. As expected for a RISC machine, this time is usually one cycle. However, the RS/6000 machines include complex instructions, such as integer multiply and divide, which may require many cycles to complete execution. Each node in the dependence graph is labeled with a resource list representing the execution time.

Figure 3 shows a program segment and its corresponding dependence graph for the PowerPC 601™ target machine.

- *List-scheduling algorithm*

List scheduling is a well-known algorithm originally designed to solve the microcode compaction problem for horizontal microarchitectures [2], and more recently applied to the reordering of instructions for optimization of RISC programs [3–5].

The algorithm is a time-driven machine simulation which issues a group of instructions in each cycle which are expected to issue in parallel on the target machine. The group to be issued is based upon a working set of instructions called the *tentatively scheduled* set. The tentative set is formed by considering all possible combinations of instructions which are available to be issued (according to data dependences) and comparing these alternatives using a preference function $\mathcal{H}$. The *committed set* is the final form of the tentative set and represents the instructions actually issued in the cycle.

The preference function used to decide the final form of the issue group is the most important factor in the success of the algorithm. The heuristics built into this function include the earliest time that an instruction may be issued and the latest time it may be issued without delaying the execution of the window of instructions. These two measures determine the freedom of motion of the instruction. Given these bounds, the scheduler determines

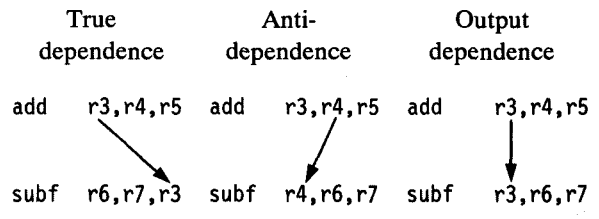


Figure 2

Types of data dependence.

FORTRAN program:

```
DO I = 1, N
  T5 = (A(I) + B(I))

  T0 = (A(I) - B(I))
  T0 = T0 * C(I)

  E(I) = T5 / T0

END DO
```

Generated PowerPC code:

```
lwz   gr0,-1596(gr3)  1
lwz   gr4,-1196(gr3)  2
addc  gr5,gr0,gr4     3
subfc gr0,gr0,gr4     4
lwz   gr4,-796(gr3)   5
mullw gr0,gr0,gr4     6
divw  gr0,gr5,gr0     7
stwu  gr0,4(gr3)      8
```

Dependence graph:

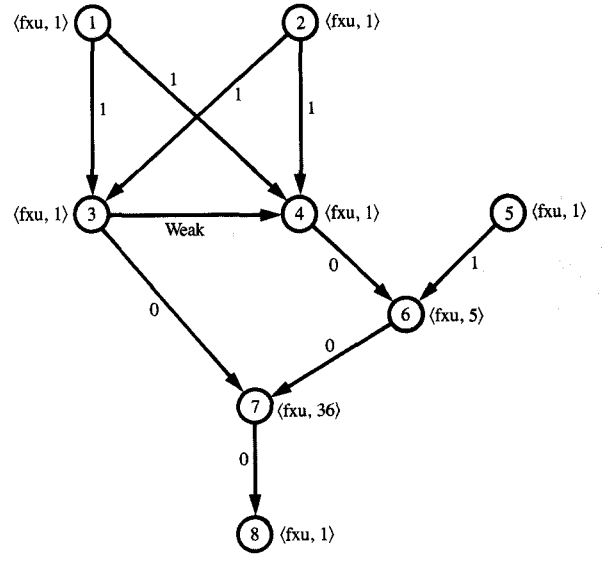


Figure 3

Example dependence graph for the PowerPC 601.

an ordering of the instructions which is optimal with respect to a set of additional measures based on the dependence graph and target machine resource constraints.

The algorithm proceeds roughly as follows:

1. Compute the dependence graph and the heuristic values needed to evaluate instruction preferences.
2. Initialize the *ready list* of instructions. This is the list of instructions that are candidates to be issued. To enter the list, a node in the dependence graph must either have no incoming edges or have only weak incoming edges from nodes already in the ready list. Initialize the machine cycle counter to 0.
3. Let $\mathcal{T}$ be the tentatively scheduled set. Set $\mathcal{T} = \emptyset$. For each instruction v in the ready list, do step 4.
4. If v is eligible for issue, then attempt to allocate machine resources for v. If resources are available for v, then let $\mathcal{T} = \mathcal{T} \cup \{v\}$. Otherwise, let $\mathcal{D} \subseteq \mathcal{T}$ be the set of instructions in the tentatively scheduled set which compete for resources required by v. If $\mathcal{H}(v, w)$ prefers v, $\forall w \in \mathcal{D}$ then *displace* the instructions in $\mathcal{D}$. Displacing an instruction places it back on the ready list but locks out the resources which were assigned to v for future allocation in this cycle. This is a guarantee of forward progress.
5. Commit all tentatively scheduled instructions for this cycle. Update the ready list to include any nodes *uncovered* by the issuing of instructions in this cycle (that is, any instructions which have become candidates as defined above because of the issuing of instructions in this cycle). If all nodes in the graph have been issued, then exit.
6. Increment the cycle counter. Go to step 3.

In step 4, we need conditions for deciding whether instruction v is *eligible* to be issued in the current cycle. The following conditions are checked:

1. If v was entered into the ready list with incoming weak edges, check that all weak predecessors (*triggers*) have either been issued or are tentatively scheduled to be issued.
2. Check that the time clock has reached the *earliest time* for v.
3. Check that the execution-unit synchronization counter does not exceed the machine tolerance (for example, for the RS/6000, check that the FXU is not executing too far ahead of the FPU or vice versa).

The key to the success of this algorithm is the nature and ordering of the heuristics encoded in the preference function $\mathcal{H}$. The heuristics attempt to balance the many factors involved in the prioritization of instructions for issue. They are divided roughly between those that check for violation of machine resource constraints and those that prefer instructions based upon calculations on the dependence graph. For each heuristic, if two instructions are equal according to its measurements, the test proceeds to the next heuristic; therefore, the order given is significant. Note that the heuristics designed to prevent resource exhaustion (store queue and register files) are checked first, but in practice are rarely used to decide schedules.

Given a pair of instructions (v_1, v_2), the heuristic set for local scheduling is the following:

1. Prefer not to issue floating-point store instructions when more consecutive floating-point stores have been issued than can fit in the store queue.
2. Prefer floating-point stores to other instructions if the stores remaining to be issued exceed the *critical store count* for the remainder of the scheduling window. The critical store count is roughly $(q \cdot r)/(q + 1)$, where q is the size of the store queue and r is the number of instructions remaining to be issued.
3. If register pressure (approximately equal to the number of live variables) is greater than the number of available registers for some register file,[3] then prefer the instruction that has the smallest positive or largest negative net effect on register pressure.
4. Each instruction has a *deadline* cycle which indicates that issuing the instruction after that cycle will cause delays in completing the scheduling window. If issuing v_1 first will cause v_2 to miss its deadline, then prefer v_2. Similarly, if issuing v_2 first will cause v_1 to miss its deadline, then prefer v_1.
5. Each instruction is assigned a *weight* that includes the net effect on register pressure combined with various heuristic measures designed to moderate the upward code motion effect of scheduling. Prefer the instruction with lower weight.
6. The *uncover count* of an instruction is the number of instructions that would become available for entry to the ready list if it were issued. Prefer the instruction with a larger uncover count.
7. Each instruction v has a *sum-delay*, S_v, which is defined as

$$S_v = \begin{cases} \max\limits_{i \in Succ(v)} (W_{vi} + S_i) & \text{if } Succ(v) \neq \emptyset, \\ 0 & \text{otherwise,} \end{cases}$$

where $Succ(v)$ is the set of successors of v in the dependence graph and W_{vi} is the weight of edge (v, i). The sum-delay is the maximum number of idle processor cycles (represented as nonzero edge weights)

[3] The RS/6000 machines include separate register files for integer and address values, floating-point values, and condition codes.

on any path to the completion of the window. Prefer the instruction with the larger sum-delay.

8. Each instruction v has a *critical path*, C_v, which is defined as

$$C_v = E_v + \begin{cases} \max_{i \in Succ(v)} (W_{vi} + C_i) & \text{if } Succ(v) \neq \varnothing, \\ 0 & \text{otherwise,} \end{cases}$$

where E_v is the execution time of instruction v. The critical path is similar to the sum-delay measure but includes the execution time of instructions in the path. Prefer the instruction with the larger critical path.

9. Prefer the instruction appearing first in the original sequence of code.

• *Machine simulation*

The allocation of machine resources to instructions is a critical mechanism in the scheduling of instructions for superscalar machines. To efficiently utilize the resources of the machine, particularly multiple execution units, the scheduler models the dynamic behavior of the machine as closely as possible. The scheduler manipulates data structures corresponding to real machine resources such as functional units, registers, register read and write ports, synchronization counters, and store queues. Most of these attributes (for example, the number and type of functional units) are specified using an abstract machine description language so that the scheduling simulation can be made as independent of the target architecture as possible.

One of the significant features of the TOBEY instruction scheduler is that it is almost entirely driven by tables which represent the target machine organization. The tables are generated by a software tool which understands the high-level machine description language. This allows for greater flexibility in the application of the scheduler to new machines and easier modification of machine attributes. The high-level description language also serves as an effective communication device between compiler developers and chip engineers. The edges of the dependence graph are constructed and weighted using a machine-independent algorithm. The algorithm obtains machine-specific execution time and delay values by lookup in the target machine tables.

To take advantage of the tables generated from machine descriptions, the machine simulation is designed to handle a generic superscalar RISC processor. The *dispatcher* is used to allocate machine resources for an instruction based on the target machine tables. If some resource is not available because some other instruction is tentatively assigned the resource, it determines which instruction is to be issued.

When instructions compete for execution resources, a preference function is evaluated to determine which instruction is better to issue. If the instruction being considered is *better* than the one tentatively assigned the resource, the latter is *displaced* in favor of the former. During the allocation of resources to an instruction, one or more instructions may be displaced. After successfully allocating resources to an instruction, the dispatcher tries to allocate resources for any displaced instructions (since they may be better than some other tentatively scheduled instructions). This process is guaranteed to terminate by preventing the allocation of a resource to a displaced instruction when the resource is tentatively assigned to the instruction which caused the displacement.

Global instruction scheduling

Many techniques have been successful in addressing the problem of local instruction scheduling. By comparison, the scheduling of instructions across basic block boundaries is a much more difficult problem, with far fewer solutions that work well in practice. Global scheduling is complicated by the need to make decisions about the benefit and correctness of moving instructions between basic blocks. Optimal scheduling also requires that appropriate loop-level transformations such as loop unrolling are performed. Previous work in this area includes the trace-scheduling technique developed by Fisher and Ellis [6–8], the global parallelization techniques developed by Ebcioğlu [9–11], and a collection of algorithms for software pipelining of loops [12–16].

The TOBEY compiler implements a global scheduling algorithm that is a generalization of list scheduling [17]. To capture the constraints imposed by control flow, the global scheduler replaces the dependence graph used in local scheduling with the *program dependence graph* (PDG) [18]. The PDG represents both the control and data dependence relationships between statements or instructions in a program.

• *Control dependence and classification of code motion*

The *control dependence graph* (CDG) is a part of the PDG that summarizes the relationship of instructions in the control flow of the program. We simplify the graph to group instructions belonging to the same basic block so that the nodes of the CDG are simply basic blocks. The global scheduler processes the program one loop or *strongly connected region*[4] at a time, beginning with the innermost (that is, those loops not containing other loops). Loops nested within other loops are treated as a single control flow node for the purpose of scheduling the containing loop. A PDG is built for each loop as it is processed. Since we may assume that we are building control dependence relations for a strongly connected

[4] A strongly connected region of any directed graph is a collection of nodes and edges in which there is at least one path between any pair of nodes. TOBEY also includes the entire control flow graph as an outermost containing region.

region of the flow graph, we may also assume that there is no backward control flow (that is, we assume that the region has a single entry point called a *header* and that any edges leading to the header are not included in the region). This simplifies the control dependence graph to a special form known as the *forward control dependence graph* [19]. This form of CDG is a directed acyclic graph in which a directed edge from B_1 to B_2 represents a conditional flow of control at the exit from block B_1, which proceeds to block B_2 if the condition evaluates to a given value (for example, true or false) labeling the edge. We insert additional edges in the graph to connect blocks with the same set of control dependences (that is, they are executed under the same conditions). We call these blocks *equivalent*.

We need to review the concepts of *dominance* and *postdominance* to further understand the constraints of control flow on global code motion in instruction scheduling. Block B_1 *dominates* block B_2 if every thread of control from the entry of the procedure to B_2 includes B_1. Block B_2 *postdominates* block B_1 if every thread of control from B_1 to the exit of the procedure includes B_2. Note that we use a modified definition of postdomination in which back edges are considered equivalent to loop exits. Two trees are built, one to represent a relation of immediate dominance and the other, postdominance. Note that if B_1 dominates B_2 *and* B_2 postdominates B_1, B_1 and B_2 are equivalent, as defined above.

There are three primary classes of code motion. If an instruction is moved from block B_2 to block B_1, the movement is called *useful* if B_1 and B_2 are equivalent. The movement is called *speculative* if B_2 does not postdominate B_1. The movement requires *duplication* if B_1 does not dominate B_2. A movement may both be speculative and require duplication.

Figure 4 shows a small program written in C along with a typical sequence of instructions (written using pseudo-assembly language) that might be generated for the PowerPC 601. The generated code includes markers to indicate the beginning of new basic blocks. Movement of an instruction from B_4 to B_1 is classified as useful. Movement from B_2 to B_1 is speculative but does not require duplication. Movement from B_4 to B_3 requires duplication (in B_2) but is not speculative. Movement from B_5 to B_3 is speculative and requires duplication (in B_2).

- *Speculative code motion*

We need to take some care in handling code motion from a conditionally executed part of a program to a part of the program that does not depend on the same conditions. Since code motion occurs in the upward direction (to control flow predecessors) only, speculative code motion is strictly concerned with the movement of instructions from a block B_2 into some block B_1 where B_2 does not postdominate B_1. There are two problems to address here: legality and profitability.

The code motion is *legal* if the instruction does not destroy any upward-exposed variables[5] in other successor blocks to B_1 and does not cause any observable side effects. In general, we cannot speculatively move an instruction that may cause an exception or alter memory. On the RS/6000 and PowerPC machines, this restriction excludes the movement of floating-point instructions, stores, procedure calls, and some loads. We will later see that some effort in the analysis of memory addressing will allow us to move a large class of load instructions.

Movement of an instruction is *profitable* if the instruction would likely be executed anyway or if the instruction costs nothing to execute (for example, if it is scheduled in an otherwise idle processor slot). To assess the relative likelihood of execution, we label the edges in the control dependence graph with probabilities which represent an estimate of the chance of following the conditional path indicated in the graph. The probability of executing an instruction in B_2 relative to the execution of B_1 is the product of the probabilities labeling the edges on the path from B_1 to B_2 in the CDG. Generally, we move an instruction that has a relative probability of 50% or more of being executed. In the absence of actual knowledge of the likely direction of conditional branches, we use static predictions based on the context of the branch. For example, if the branch is the closing of a loop, we assume that the branch is almost always taken (that is, we assume that loops are usually iterated many times before exiting). This framework for assessing the profitability of code motion is also capable of using information that might be available from a basic block execution profile.

- *Speculative loads*

The most important class of instructions to move across basic blocks are those that load values from memory. Since most computation is done on register operands and the most common way to get values into registers is by using a load instruction, the movement of a load instruction usually allows the movement of many more instructions which are dependent upon the value loaded. Unfortunately, it is difficult to decide whether the movement of a load is legal. Speculative movement of a load is considered illegal if it violates memory-ordering constraints (that is, if the memory location is *volatile*) or causes an exception that would not have happened if the instruction had not been moved. The first of these conditions is easy to check; the second is much more

[5] A variable is upward-exposed in a basic block if there is a use of the variable before any definition in the block (that is, it depends on a value defined in a predecessor block).

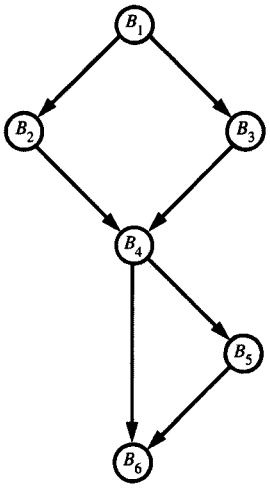
C program:

```
if (a < 10)

 b = b + 1;


else
  c = c + 1;

 d = d + 1;

if (b > 20)

 e = e + 1;
```

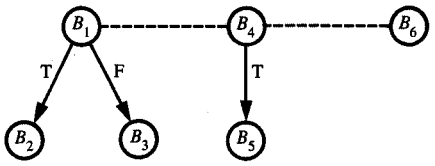
Generated code:

```
*begin 1
    lwz     r0=a
    cmpwi   cr0=r0,1
    bge     L1,cr0
*begin 2
    lwz     r1=b
    addi    r1=r1,1
    stw     b=r1
    b       L2
*begin 3
l1: lwz     r2=c
    addi    r2=r2,1
    stw     c=r2
*begin 4
l2: lwz     r3=d
    addi    r3=r3,1
    stw     d=r3
    lwz     r4=b
    cmpwi   cr1=r4,20
    ble     L3,cr1
*begin 5
    lwz     r5=e
    addi    r5=r5,1
    stw     e=r5
*begin 6
l3:
```

Control flow graph (CFG):

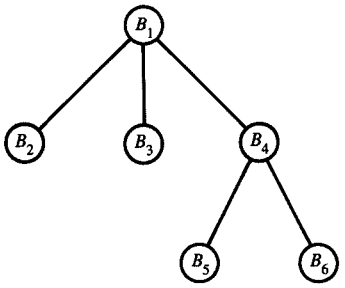


Control dependence graph (CDG):



-------- Equivalence edge

Dominator tree:



Postdominator tree:

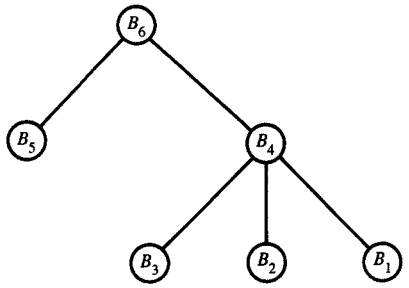


Figure 4

C program with example RS/6000 code and the associated flow and dominance graphs.

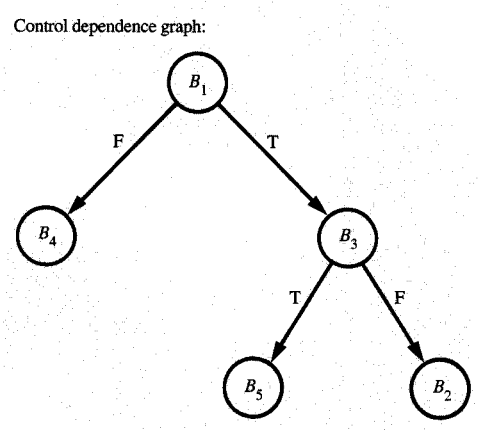


Figure 5

Example control dependence graph showing escape blocks.

difficult to determine. Bernstein et al. [20] describe a collection of proof techniques that may determine the legality of a speculative load.

Determining that a load does not cause an exception is dependent upon the operating system and machine on which the program runs. We later describe the set of tests used for the RS/6000 machines running the AIX 3.2 operating system.

The compiler can determine that many loads are incapable of generating an exception regardless of their context. We call these loads *globally safe*. The scheduler recognizes these loads by their use of a special base addressing register. For example, loads using the stack frame pointer or TOC (table of contents) base register are usually considered safe. Also, those loads using base registers loaded from the TOC and with displacements that are smaller than the allocated memory for the external being referenced are usually considered safe.

When a load cannot be determined to be globally safe, the load is usually an array reference or an indirect reference through a pointer variable. In this case, the safety of the load may be dependent on the context of the reference.

The first set of context-sensitive tests is performed on the *target* block and its dominators. The compiler searches the target and dominators for other loads that are *similar* to the candidate load or that define its base register to an address known to be safe. Given two loads L_1 and L_2, we consider L_1 to be *similar* to L_2 if L_1 and L_2 share addressing registers and the displacement on L_1 "covers" the displacement on L_2. There are varying degrees of strictness in the similarity test. The strictest version of the test demands that the safety of load L_1 implies the safety of load L_2 (that is, the memory referenced by L_2 is a subset of the memory referenced by L_1). More permissive versions of the test are used at higher optimization levels (−O3) that allow for a larger tolerance in the difference in displacements on L_1 and L_2. In addition to similar loads, the target block is searched for definitions of the base register used on the candidate load that are known to be safe locations. For example, loads from the TOC are usually considered safe definitions.

The second set of context-sensitive tests is performed on the conditional branches controlling the execution of the candidate load and the set of *escape blocks* along the path from B_1 (the target block) to B_2 (the source block) in the CDG. An escape block is a block that is control-dependent on some block on the path from B_1 to B_2 along a condition other than the one controlling the execution of B_2. For example, in **Figure 5**, if B_1 is a target block and B_2 is a source block, then B_4 and B_5 are escape blocks. First, the escape blocks are scanned for the presence of any similar loads. If there are none, the conditional branches closing the blocks in the CDG between B_1 and B_2 are scanned for conformance to a special form that compares the base register of the candidate load with zero. If the "equal to zero" condition controls the escape path, the load is proven to be safe to move above that condition, because if control were to pass to the escape block, the load would have been using a base register of value zero. We are guaranteed not to get an exception in this case because AIX 3.2 allows the first page of memory to be readable (therefore, we also require the load displacement to be in the range [0, 4095]). This context-sensitive test is most commonly applied in programs which follow pointer-based data structures such as trees or linked lists that represent null pointers with a value of zero. In **Figure 6**, showing a program fragment, the load of r4 may be moved above the conditional branch. Once the load is moved above the conditional branch, we may also move the addi instruction (since r6 is not live on the escape path of the branch), but we cannot move the store instruction speculatively, since it may alter memory which might not otherwise have been altered.

- *Software pipelining*

Recent efforts in the scheduling of instructions for wide-issue and deeply pipelined computers have turned to loop-level transformations designed to offer greater opportunity for the resolution of delay slots and the parallel execution of instructions. The most widely used technique is called *software pipelining* [12–16]. The technique rearranges

loops so that an iteration can be started before a previous iteration completes. Optimal software pipelining is a difficult problem in general[6], but it has a variety of approximate solutions that work well in practice. The global scheduling algorithm presented in [17] does not include a software pipelining technique but has been subsequently enhanced to include an approximate software pipelining algorithm [22]. This algorithm is implemented in the TOBEY compiler. Ebcioğlu and Nakatani have demonstrated a similar technique [23].

The algorithm is a simple extension to the global scheduling framework described above. Most inner loops that are candidates to be globally scheduled are first unrolled once. The loop is then processed normally, except that only the original loop blocks (the *real* iteration) are scheduled. The loop blocks from the unrolled or *virtual* iteration are used only as sources of instructions for the scheduling of the original loop body. Movement of an instruction from the virtual iteration to the real iteration is a special code motion that requires the "twin" instruction in the real iteration to be moved to a prologue of the loop. When scheduling is complete, some instructions may have been moved to a loop prologue because of pipelined code motion. We construct a complementary epilogue section of code following the loop that contains all instructions of the virtual iteration which were not scheduled into the real iteration. The algorithm as stated handles the overlapping of only two iterations of the loop. A simple extension allows us to overlap an arbitrary number of iterations while constructing the appropriate prologue and epilogue sections of the loop. **Figure 7** shows an example of a loop compiled for the PowerPC 601 before, during, and after software pipelining. Notice that the resulting loop avoids delays between the load and use of registers r7 and r8 in the loop.

Some machines include dynamic scheduling hardware in the floating-point unit, such as register renaming and a store queue which realizes many of the benefits of static pipelining. Software pipelining has a smaller observable effect on loops which can take advantage of these special facilities.

- *Duplication of code*

The movement of instructions which require duplication presents a special problem for the scheduler. For example, consider the movement of an instruction from B_4 to B_3 in Figure 4. Since B_3 is not on every path of execution that might reach B_4, the scheduler must create copies of the instruction to ensure that the program produces the same results. The scheduler computes a set of basic blocks including B_3 that collectively dominates B_4 and creates copies of the moved instruction in each block in the set.

[6] Schwiegelshohn et al. show that optimal pipelining of loops with conditional jumps on a machine with infinite resources is undecidable [21]. The decidability of optimal pipelining for a machine with finite resources is an open problem.

Before scheduling:

```
    lwz     r5,0(r3)
    cmpwi   cr1,r5,0
    beq     cr1,l1
    lwz     r4,0(r5)
    addi    r6,r4,1
    stw     r4,0(r5)
l1: lwz     r6,8(r3)
    lwz     r4,16(r3)
```

After scheduling:

```
    lwz     r5,0(r3)
    lwz     r4,0(r5)
    cmpwi   cr1,r5,0
    addi    r6,r4,1
    beq     cr1,l1
    stw     r4,0(r5)
l1: lwz     r6,8(r3)
    lwz     r4,16(r3)
```

Figure 6

Example of moving a load above a compare with zero.

The set does not include more than one block which lies on the same path to B_4, and the instruction is thus executed only once on any path from the entry node.[7] The scheduler also tries to minimize the number of duplicates that must be generated. The dominating set generated under these requirements is called a *minimal independent separating set* (MISS). An algorithm for computing the set is described in [25].

The algorithm assumes that there are no *join–split* control flow edges in the program. A control flow edge is called a *split edge* if the predecessor block has more than one successor and is called a *join edge* if the successor block has more than one predecessor. A join–split edge is both a split edge and a join edge. To ensure that the program does not contain a join–split edge, the control flow graph is modified before global scheduling by replacing join–split edges with a split edge leading to a dummy basic block which leads to a join edge. Given two basic blocks B_1 and B_2, the following algorithm computes the MISS $\mathcal{N}(B_1, B_2)$. The algorithm is guaranteed not to fail if the control flow graph contains no join–split edges.

1. Mark all blocks that are reachable from B_1.
2. $\mathcal{M}$, $\mathcal{M}' = \{B_2\}$.
3. Remove a marked block v from $\mathcal{M}$.
4. Let P be the set of immediate predecessors of v. Let $P' = P - \mathcal{M}'$. Let $\mathcal{M} = \mathcal{M} \cup P'$ and $\mathcal{M}' = \mathcal{M}' \cup P'$.
5. Mark all blocks reachable from members of P'.
6. If B_1 is marked, then fail.
7. If $\mathcal{M}$ contains marked blocks, then goto step 3.
8. $\mathcal{N}(B_1, B_2) = \mathcal{M}$.

[7] The requirement that no two nodes lie on the same path ensures that no execution path is made slower because of duplication. Moon and Ebcioğlu demonstrate a technique which relaxes this requirement for less frequently executed paths [24]. This may result in better optimization of more frequently executed paths.

Before:

```
    lwz   r3,0(r6)
11: lwzu  r7,8(r4)
    add   r3,r3,r7
    lwzu  r8,8(r5)
    subf  r3,r8,r3
    bdnz  11
```

During:

```
    lwz   r3,0(r6)
11: lwzu  r7,4(r4)
    add   r3,r3,r7
    lwzu  r8,4(r5)
    subf  r3,r8,r3
-------------------
    lwzu  r7,4(r4)
    add   r3,r3,r7
    lwzu  r8,4(r5)
    subf  r3,r8,r3
    bdnz  11
```

After:

```
    lwz   r3,0(r6)
    lwzu  r7,4(r4)
    lwzu  r8,4(r5)
    bdz   12
11: add   r3,r3,r7
    lwzu  r7,4(r4)
    subf  r3,r8,r3
    lwzu  r8,4(r5)
    bdnz  11
12: add   r3,r3,r7
    subf  r3,r8,r3
```

Figure 7

Software pipelining of a simple loop.

• *Elimination of false dependences*

Data dependences are the fundamental constraints on the movement of code by instruction scheduling. True dependence represents data flow and usually cannot be avoided. Antidependence and output dependence are artifacts of variable naming and register allocation. We call these *false dependences*, since they can both be avoided by using a different selection of variable names or register numbers.

The global scheduler identifies situations where the presence of an antidependence interferes with the generation of well-scheduled code. In such situations, it creates a new register name for the written register in an antidependence pair and then introduces a register copy to the old register name. This renaming allows the antidependence edge in the PDG to be broken, and therefore allows the incident nodes to be reordered as necessary. Once the scheduling is complete, we are left with the problem of removing the register copy instructions. This is done by unrolling the containing loop once and then propagating the renamed register into all uses in the unrolled iteration, eliminating the need for the register copy instruction. **Figure 8** illustrates this process for a simple program segment. A similar dynamic renaming technique is used in the global scheduling algorithm presented in [24].

Some true dependences can also be collapsed by the scheduler. In cases where two instructions which have immediate forms on the target machine are related by a true dependence, it may be possible to rewrite the dependent instruction to reflect the effect of the other instruction. For example, consider the instruction `add r3,r3,1`, followed by `cmpwi cr1, r3, 1`. The compare depends on the incremented value of `r3`, but may be moved above the add instruction if it is rewritten as `cmpwi cr1,r3,0`.

Some machines in the PowerPC family lack dynamic register-renaming capabilities. This provides further incentive to statically rename registers involved in false dependences. Since many antidependences are created by the register allocator in an attempt to optimize register reuse, a special renaming optimization transforms the program after register assignment. This optimization understands when registers should be renamed for the target machine and which registers are available for use at any point in the program. The renaming transformation is done before postpass instruction scheduling to explicitly break false dependence edges.

Postpass instruction scheduling

The instruction scheduling algorithms used in the compiler are general enough to create good schedules for most programs, but experience shows that some specialized optimizations are necessary to obtain the best performance for certain code patterns. One weakness of the general scheduling algorithm is the inability to rearrange

Before:

```
    lwz  r3,0(r4)
l1: lwz  r6,0(r7)
    addi r5,r3,1
    addi r3,r6,1
    stwu r5,4(r4)
    stwu r3,4(r7)
    bdnz l1
```

During:

```
    lwz  r3,0(r4)
l1: lwz  r6,0(r7)
    addi r8,r6,1
    addi r5,r3,1
    COPY r3=r8
    stwu r3,4(r7)
    stwu r5,4(r4)
    bdnz l1
```

After:

```
    lwz  r3,0(r4)
l1: lwz  r6,0(r7)
    addi r8,r6,1
    addi r5,r3,1
    stwu r8,4(r7)
    stwu r5,4(r4)
    lwz  r6,0(r7)
    addi r3,r6,1
    addi r5,r8,1
    stwu r3,4(r7)
    stwu r5,4(r4)
    bdnz l1
```

Figure 8

Elimination of an antidependence.

branches to avoid delays in the speculative dispatch of instructions.

- *Branch swapping*

The RS/6000 machines execute branch instructions in parallel with the execution of fixed- and floating-point instructions. Optimal performance can be achieved only when the stream of fixed- and floating-point computation remains uninterrupted. If the branch unit encounters a conditional branch which depends on either an unavailable condition (that is, the condition computation has not yet completed) or a branch whose target address is similarly unavailable, we call such branches *unresolved*. In these situations, the branch unit cannot determine which instructions are to be executed next, but speculatively dispatches instructions to the fixed- and floating-point units from the fall-through path of the branch. If there are enough fixed- or floating-point instructions on the fall-through path, unresolved branches suffer no penalty if they are not taken. If, however, the branch unit encounters another branch (which may or may not be resolved), speculative dispatch of instructions is stopped and waits until the first branch is resolved.

The compiler often has opportunities to rearrange the instructions in a program to avoid the branch unit hold-off caused by encountering a resolved branch while speculatively dispatching instructions on the fall-through path of an unresolved branch. An optimization called *branch swapping* identifies situations where an unresolved branch may contain a resolved branch on the fall-through path. In these situations, it attempts to swap the branches while duplicating any intervening code. A typical branch swapping opportunity is a conditional exit from a counted loop. In this case, we usually have an unresolved conditional branch that contains a branch-on-count-register instruction (always resolved) in the fall-through path. **Figure 9** shows a simple example of branch swapping.

- *Branch reversal*

Conditional branches which are not taken usually suffer no penalty on the POWER and POWER2 machines. It is possible to put the most likely target of a branch on the fall-through path in order to avoid the delay of a taken branch. The problem here is that it is not always clear what direction a branch is most likely to take. The compiler currently uses a set of heuristics to guess which direction the branch will most likely take. Sometimes the heuristics are fairly certain (for example, the branch closing a loop is likely to be taken); in other cases, the heuristic is weaker (for example, we assume that if a branch leads to a call in one successor path but not the other, the path without the call is more likely to be taken). More certainty can be achieved by using the information contained in a basic block execution profile.

Given information about the relative probability of branch directions, the *branch reversal* optimization may

FORTRAN program:

```
      COMMON A,B,C
      REAL A(100),B(100),C(100)

      DO 10 I=1,N
        B(I) = -B(I)

        IF (B(I) .LT. 1.0) GOTO 20

        A(I) = B(I) * C(I)


10    CONTINUE
20    STOP
```

Generated code:

```
      lfs   fp2=1.0
11:   lfs   fp0,4(r3)    # load B(I)
      fneg  fp0,fp0
      stfsu fp0,4(r3)    # store B(I)
      fcmpu cr1,fp2,fp0
      bgt   cr1,12
      lfs   fp1,400(r3)  # load C(I)
      fmuls fp0,fp0,fp1
      stfs  fp0,-400(r3) # store A(I)
      bdnz  11
12:   ⟨stop⟩
```

Generated code after branch swapping:

```
      lfs   fp2=1.0
      b     13
11:   bgt   cr1,12
      lfs   fp1,400(r3)   # load C(I)
      fmuls fp0,fp0,fp1
      stfs  fp0,-400(r3)  # store A(I)
13:   lfs   fp0,4(r3)     # load B(I)
      fneg  fp0,fp0
      stfsu fp0,4(r3)     # store B(I)
      fcmpu cr1,fp2,fp0
      bdnz  11
      bgt   cr1,12
      lfs   fp1,400(r3)   # load C(I)
      fmuls fp0,fp0,fp1
      stfs  fp0,-400(r3)  # store A(I)
12:   ⟨stop⟩
```

Figure 9

Branch swapping for a conditional loop exit.

decide to reverse the direction of a branch, bringing the instructions on the taken path into the fall-through position and moving instructions on the fall-through path to the taken position. This often has the effect of moving infrequently executed code in a loop out of line, so that the fall-through paths of conditional branches lead directly to the closing of the loop.

- *Gluing*

An additional opportunity to avoid branch unit hold-offs is the situation in which an unconditional branch lies in the fall-through path of an unresolved conditional branch. To avoid the hold-off upon encountering the unconditional branch, *gluing* copies code from the target of the unconditional branch to the fall-through of the unresolved

Before branch reversal:

```
11: lwz   r3,4(r4)
    cmpwi cr1,r3,1
    bne   cr1,12
    bl    some_function
12: addi  r3,r3,1
    stwzu r3,4(r4)
    bdnz  11
```

After branch reversal and gluing:

```
11: lwz   r3,4(r4)
    cmpwi cr1,r3,1
    beq   cr1,13
12: addi  r3,r3,1
    stwzu r3,4(r4)
    bdnz  11
    b     14
13: bl    some_function
    b     12
14:
```

Figure 10

Example of branch reversal and gluing.

branch, and then replaces the unconditional branch with one targeted to the end of the code that was copied. Gluing also serves as the code-copying mechanism for branch reversal. **Figure 10** illustrates the reversal of a branch and the subsequent gluing of instructions on the taken path into the fall-through position. Notice that if the branch is mostly taken in the original code, it is now mostly not taken in the transformed code, making the resulting loop faster.

Performance results

All of the instruction scheduling techniques described so far have been implemented as part of the latest production versions of the CSet++ and XL FORTRAN/6000 compilers. **Tables 1** and **2** show the results of running the *SPECint92* and *SPECfp92* benchmark programs on the POWER2 (RS/6000 Model 590) and PowerPC 601 (RS/6000 Model 250) machines. The various levels of scheduling measured are described in **Table 3.** The *SPECint92* and *SPECfp92* benchmark programs are described in **Tables 4** and **5**, respectively. The measurements should not be taken as official performance measurements, since they were not performed using the carefully selected set of options and quiet execution environment required for regular published results. The measurements compare the benchmark scores (expressed as *SPECratios*) using various phases of instruction scheduling against a baseline score which represents no instruction scheduling. The percentage of execution time reduction is relative to the next lowest level of instruction scheduling (B relative to A, C relative to B, and so on). All measurements include the maximum optimization available in the TOBEY compiler and also include options to specifically target the machines being measured.

The measurements illustrate the effect of the various types of instruction scheduling on the performance of the two machines. Notice first that the effect is more significant on the POWER2, which has a larger degree of instruction-level parallelism (two integer and two floating-point units) than the PowerPC 601 machine. The net effect of all instruction scheduling on the POWER2 is 19.4% on *SPECint92* and 37.8% on *SPECfp92*, whereas the respective improvements on the PowerPC 601 are 7.9% and 16.5%. Another interesting measurement is the small or nonexistent improvement due to loop-based optimizations such as unrolling and software pipelining on the *SPECint92* programs. The performance of these programs does not depend as heavily as that of the *SPECfp92* programs on the execution time of loops and is not expected to benefit much by these optimizations. Notice also that the effect of local instruction scheduling is more significant for the *SPECfp92* programs and the effect of branch optimizations is more significant for the *SPECint92* programs. These effects are an indication of the relative size of basic blocks in the two benchmark suites (*SPECfp92* blocks are larger) and the relative frequency of branch instructions (*SPECint92* has more frequent branching).

Table 1 SPECint92 performance results.

	A None	*B Local pass 1* (%)	*C Local passes 1 & 2* (%)	*D C + branch optimization* (%)	*E D + global scheduling* (%)	*F E + software pipelining* (%)	*G F + loop unrolling* (%)
POWER2	73.18	4.4	1.0	8.4	2.9	0.0	1.5
PowerPC 601	46.58	2.7	2.0	2.4	0.0	0.0	0.6

Table 2 SPECfp92 performance results.

	A None	*B Local pass 1* (%)	*C Local passes 1 & 2* (%)	*D C + branch optimization* (%)	*E D + global scheduling* (%)	*F E + software pipelining* (%)	*G F + loop unrolling* (%)
POWER2	139.04	9.1	1.5	3.2	8.3	6.2	4.9
PowerPC 601	54.35	8.4	0.5	0.1	3.0	2.9	0.8

Table 3 Levels of instruction scheduling measured.

A	Baseline: no instruction scheduling.
B	The first pass of local scheduling (before register allocation) only.
C	Both passes of local scheduling.
D	Local scheduling and branch optimizations (branch swapping, reversal, and gluing).
E	Local scheduling, branch optimizations, and global scheduling.
F	Local scheduling, branch optimizations, global scheduling, and software pipelining.
G	Local scheduling, branch optimizations, global scheduling, software pipelining, and inner loop unrolling.

Future directions

Higher degrees of instruction-level parallelism in new processors will require the TOBEY instruction scheduler to transform code more aggressively to keep the many execution units busy and will demand higher performance in the presence of conditional branches. This anticipated advance in the organization of superscalar machines changes the problem of instruction scheduling to be more similar to the problem of generating code for certain VLIW (very long instruction word) machines [26]. Aggressive scheduling techniques for these processors have been an active research topic [6, 8, 10, 24]. Our future work will rely increasingly on these techniques.

- *Advanced scheduling algorithms*

The TOBEY compiler is beginning to incorporate techniques developed as part of the IBM VLIW machine project at the Thomas J. Watson Research Center [10, 23, 24, 27–31]. The compiler developed for this project includes a global scheduling algorithm, an approximate software pipelining algorithm, loop unrolling, and register renaming. All of these techniques have been prototyped in a development version of the TOBEY compiler.

Before the application of any of the instruction scheduling techniques, inner loops are unrolled where heuristics suggest a benefit. Other parts of the TOBEY compiler attempt both inner and outer loop unrolling, but this unrolling phase is specifically concerned with exposing more independent instructions to the scheduler. The heuristics attempt to determine the optimal unrolling factor for a loop based on the number of execution units of each type available on the machine and the set of data dependence relations and associated pipeline delays applicable to the instructions in the loop. Where possible, registers in unrolled iterations are given names distinct from the corresponding registers in the original iteration in order to avoid introducing antidependence relations between iterations.

In the prototype implementation, the TOBEY global scheduling phase is replaced by the VLIW global scheduling and enhanced pipeline scheduling algorithms [23, 24]. For each basic block, VLIW scheduling creates a set of instructions which are available to move forward. Scheduling chooses the best instruction from the set of instructions that can move to a point in the program and moves the instances of that instruction forward. Scheduling also makes bookkeeping copies (similar to the duplication used in the TOBEY scheduler) for edges that join the path of the selected instructions' upward motion (but are not on these paths) and updates the set of

available instructions associated with basic blocks only on the paths that were traversed by the moved instructions. This algorithm provides a general mechanism for the reordering of instructions in a program across arbitrary control flow while preserving the semantics of the original program. The key advantages of this technique over the existing TOBEY scheduler are the larger scope of analysis and the ability to handle the movement of conditional branch instructions between basic blocks. One of the key disadvantages is the greatly increased compilation time required.

- *Memory disambiguation*

To extract large amounts of parallelism from sequential code, the instructions must be somewhat independent. Instructions which reference memory often have an unknown dependence relationship because the effective addresses referenced may not be known at compile time. The compiler is able to determine that certain variable references cannot refer to the same memory location, but elements of the same array and indirect pointer references are usually *aliased* to one another by the compiler. Some of these aliased references do not in fact interfere, and the scheduler can do a better job if they can be proven not to interfere at compile time.

When memory is referenced through the indexing of the same array, indices can be compared and determined not to reference the same array element. The problem of determining that two array references are independent has many practical solutions originally designed as part of vectorizing compilers [32–35]. The problem is somewhat more difficult when encountered by instruction scheduling, since the subscript expressions of multidimensional arrays have usually been linearized. A framework for building symbolic indexing expressions along with a suite of dependence tests has been prototyped in the TOBEY compiler. The dependence tests are successful in proving that many array references are distinct, but fail to disambiguate most pointer dereferences. In order to generate well-scheduled code for programs which have uncertain pointer-induced aliases, the scheduler includes a prototype implementation of run-time disambiguation. This technique creates a copy of certain loops and optimizes one of the copies assuming that certain reference pairs are distinct and the other copy assuming that they are aliased. A test is introduced to choose which loop to execute depending on the actual pointer values. The first disambiguation technique (array-based dependence analysis) is uniformly successful in improving execution time but has a large compile-time cost. The run-time disambiguation technique is successful in some cases but requires a careful analysis to determine when the cost of a run-time test is profitable.

Table 4 SPECint92 benchmark programs.

espresso	Generates and optimizes programmable logic arrays.
li	Uses a LISP interpreter to solve the nine-queens problem, using a recursive backtracking algorithm.
eqntott	Translates a logical representation of a Boolean equation to a truth table.
compress	Reduces the size of input files by using Lempel–Ziv coding.
sc	Calculates budgets, SPEC metrics, and amortization schedules in a spreadsheet based on the UNIX® cursor-controlled package "curses."
gcc	Translates preprocessed C source files into optimized Sun-3 assembly language output.

Table 5 SPECfp92 benchmark programs.

spice2g6	Simulates analog circuits.
doduc	Performs Monte Carlo simulation of the time evolution of a thermo-hydraulic model for a nuclear reactor's component.
mdljdp2	Solves motion equations for a model of 500 atoms interacting through the idealized Lennard–Jones potential (double precision).
mdljsp2	Same as mdljdp2 but single precision.
wave5	Solves particle and Maxwell's equations on a Cartesian mesh.
tomcatv	Generates two-dimensional, boundary-fitted coordinate systems around general geometric domains.
ora	Traces rays through an optical surface containing spherical and planar surfaces.
alvinn	Trains a neural network using back-propagation.
ear	Simulates the human ear by converting a sound file to a cochleagram using fast Fourier transforms and other math library functions.
swm256	Solves the system of shallow-water equations using finite difference approximations.
su2cor	Calculates masses of elementary particles in the framework of the quark gluon theory.
hydro2d	Uses hydrodynamical Navier–Stokes equations to calculate galactical jets.
nasa7	Executes seven program kernels representative of operations used in NASA applications.
fpppp	Calculates multi-electron integral derivatives.

- *Profile-directed optimizations*

Many aggressive algorithms for extracting parallelism from sequential code assume some knowledge of the paths of execution that are most often taken. In fact, some

techniques go so far as to optimize the most frequent paths at the expense of the less frequently taken paths. If we are to use such aggressive techniques, we must determine the most frequently taken paths on the basis of actual program executions. It is necessary to evaluate the dynamic behavior of the program while processing typical data sets and then feed this information back into the compiler. The level of detail required by the compiler to make decisions about the likelihood of certain execution paths is beyond most standard programs which produce execution profiles. Therefore, we are investigating various techniques for obtaining execution profile information at the basic block level. In addition, we are investigating enhancements to the compiler which might take advantage of this information and solutions to the problem of keeping the information correct while performing the normal set of optimizing program transformations.

Acknowledgments

The development of most of the instruction scheduling technology described here was undertaken as a joint effort among the IBM Haifa Scientific Center, the IBM Thomas J. Watson Research Center, and various members of the TOBEY compiler development team in the IBM Toronto Laboratory. In particular, I would like to acknowledge the significant contributions of David Bernstein, Doron Cohen, Yuval Lavon, Vladimir Rainish, Dror Maydan, Steve Hoxey, Joanne Minish, Andrew MacLeod, David Gillies, Brian Hall, Roch Archambault, Kemal Ebcioğlu, Gabby Silberman, Isaac Ziv, and Hank Warren. Since understanding instruction scheduling requires an intimate knowledge of the target machines, I would like to thank Bill Hay and Steve White for many fruitful discussions on the implementation of the RS/6000 family of machines.

References

1. G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, "Register Allocation via Coloring," *Computer Lang.* **6,** 47–57 (January 1981).
2. J. A. Fisher, "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Process Scheduling," Ph.D. thesis, New York University, New York, 1979.
3. J. Hennessy and T. Gross, "Postpass Code Optimization of Pipelined Constraints," *ACM Trans. Programming Lang. & Syst.* **5,** No. 3, 422–448 (July 1983).
4. D. Bernstein, "An Improved Approximation Algorithm for Scheduling Pipelined Machines," *Technical Report RC-14531*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1989.
5. H. S. Warren, "Instruction Scheduling for the RS/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 85–92 (January 1990).
6. J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers* **C-30,** No. 7, 478–490 (July 1981).
7. A. Nicolau, "Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers," Ph.D. thesis, Yale University, New Haven, CT, 1984.
8. J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures," *Technical Report YALE/DCS/RR-364*, Yale University, New Haven, CT, 1984.
9. K. Ebcioğlu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," *Technical Report RC-13214*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1987.
10. K. Ebcioğlu and A. Nicolau, "A Global Resource-Constrained Parallelization Technique," *Proceedings of the ACM SIGARCH International Conference on Supercomputing*, Crete, Greece, June 1989, pp. 154–163.
11. S.-M. Moon and K. Ebcioğlu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Technical Report RC-17962*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1992.
12. B. R. Rau and C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proceedings of the 14th Annual Workshop on Microprogramming*, 1981, pp. 183–198.
13. R. F. Touzeau, "A FORTRAN Compiler for the FPS-164 Scientific Computer," *ACM SIGPLAN Notices* **19,** No. 6, 48–57 (June 1984).
14. R. Cohn, T. Gross, M. Lam, and P. S. Tseng, "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, ACM, April 1989, pp. 2–14.
15. A. Aiken and A. Nicolau, "Optimal Loop Parallelization," *ACM SIGPLAN Notices* **23,** No. 7, 308–317 (July 1988).
16. M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *ACM SIGPLAN Notices* **23,** No. 7, 318–328 (June 1988).
17. D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines," *ACM SIGPLAN Notices* **26,** No. 6, 241–255 (June 1991).
18. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Programming Lang. & Syst.* **9,** No. 3, 319–349 (July 1987).
19. R. Cytron, M. Hind, and W. Hsieh, "Automatic Generation of DAG Parallelism," *ACM SIGPLAN Notices* **24,** No. 7, 54–68 (July 1989).
20. D. Bernstein, M. Rodeh, and M. Sajiv, "Proving Safety of Speculative Load Instructions at Compile Time," *Proceedings of the 4th European Symposium on Programming*, Rennes, France, February 1992, Springer-Verlag, pp. 56–72.
21. U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu, "On Optimal Loop Parallelization," *Technical Report RC-14595*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1989.
22. D. Bernstein and Y. Lavon, "A Software Pipelining Algorithm Based on Global Instruction Scheduling," *Technical Report TR-88.338*, IBM Haifa Scientific Center, Haifa, Israel, 1992.

23. K. Ebcioğlu and T. Nakatani, "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture," *Languages and Compilers for Parallel Computing*, The MIT Press, Cambridge, MA, 1990, pp. 213–229.
24. S.-M. Moon and K. Ebcioğlu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Proceedings of the 25th Annual Workshop on Microprogramming and Microarchitecture*, Portland, OR, November 1992, ACM and IEEE, pp. 55–71.
25. D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling," *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture*, ACM and IEEE, November 1991, pp. 103–113.
26. J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, 1983, IEEE, pp. 140–150.
27. K. Ebcioğlu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," *Proceedings of the 20th Annual Workshop on Microprogramming*, Colorado Springs, CO, December 1987, ACM and IEEE.
28. K. Ebcioğlu, "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software," *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, Pisa, Italy, April 1988, pp. 3–21.
29. T. Nakatani and K. Ebcioğlu, "Combining as a Compilation Technique for VLIW Architectures," *Proceedings of the 22nd Annual Workshop on Microprogramming*, Dublin, Ireland, 1989, ACM and IEEE, pp. 43–55.
30. K. Ebcioğlu and R. Groves, "Some Global Compiler Optimizations and Architectural Features for Improving Performance of Superscalars," *Technical Report RC-16145*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, October 1990.
31. G. M. Silberman and K. Ebcioğlu, "An Architectural Framework for Migration from CISC to Higher Performance Platforms," *Proceedings of the 1992 International Conference on Supercomputing*, Washington, DC, July 1992, ACM SIGARCH, pp. 198–215.
32. U. Banerjee, "Data Dependence in Ordinary Programs," Master's thesis, University of Illinois at Urbana-Champaign, 1976.
33. M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Research Monographs in Parallel and Distributed Computing, The MIT Press, Cambridge, MA, 1989.
34. D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and Exact Data Dependence Analysis," *ACM SIGPLAN Notices* **26**, No. 6, 1–14 (June 1991).
35. G. Goff, K. Kennedy, and C.-W. Tseng, "Practical Dependence Testing," *ACM SIGPLAN Notices* **26**, No. 6, 15–29 (June 1991).

Received November 3, 1993; accepted for publication April 13, 1994

Robert J. Blainey *IBM Software Solutions, 844 Don Mills Road, Toronto, Ontario, Canada M3C 1V7 (BLAINEY at TOROLAB, blainey@vnet.ibm.com).* Mr. Blainey is a Staff Development Analyst at the Toronto Software Solutions Laboratory. He graduated from the University of Toronto in 1988 with a B.S. degree in computer science. In 1989 he joined IBM at the Toronto Laboratory and has since been involved with the development of the XL FORTRAN/6000, XL C/6000, XL Pascal/6000, and CSet++ compilers. More recently, he has specialized in the instruction scheduling component of the common optimizing back end for these compilers. Mr. Blainey's interests include new techniques for instruction scheduling and the exploitation of interprocedural program analysis for compiler optimization.