Local Instruction Scheduling

Quinn Jacobson and Eric Rotenberg
{quinnj,ericro}@cs.wisc.edu

1.0 Introduction

Sequential program semantics gives the appearance of machine instructions executing one at a time. While this is a convenient programming model, it artificially limits performance. Parallelism exists between machine instructions; processors can exploit this parallelism while preserving sequential semantics.

Processors take advantage of instruction-level parallelism in two ways. First, the various stages of instruction execution can be overlapped through *pipelining*. Second, processors can exploit even more parallelism through *multiple instruction issue*. In this project, we only consider instruction scheduling for improving simple pipeline performance.

With simple pipelining, instructions begin and complete execution one at a time, but a new instruction can be initiated before the preceding instructions complete. The pipeline example in Figure 1 shows multiple independent instructions in different stages of execution. By overlapping processing in this way, instruction throughput becomes somewhat independent of the time it takes to fully process an instruction.

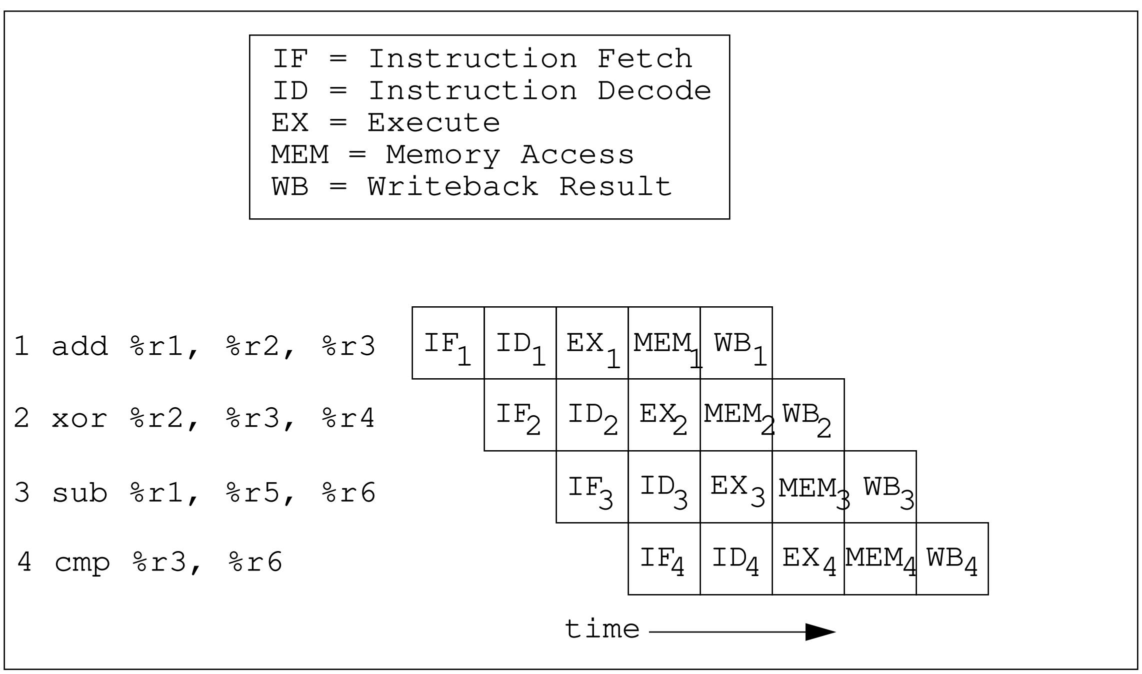

FIGURE 1. Timing diagram for a simple 5-stage pipeline with result forwarding to the EX stage. The example instruction stream shows no data or control dependence stalls.

Ideally, we would like pipelining to yield a throughput of 1 instruction per cycle. Unfortunately, this is hardly the case due to imperfect memory systems (instruction and data cache misses) and control and data dependences. Many data dependences can be preserved without stalling the pipeline through data forwarding. However, multi-cycle operations such as loads and floating point operations may incur stall cycles if their dependent instructions are not separated by enough instructions in the pipeline. Control transfer instructions also cause wasted pipeline slots because it generally takes a cycle or two to redirect instruction fetching. Refer to Figure 2 for example data and control dependence stalls.

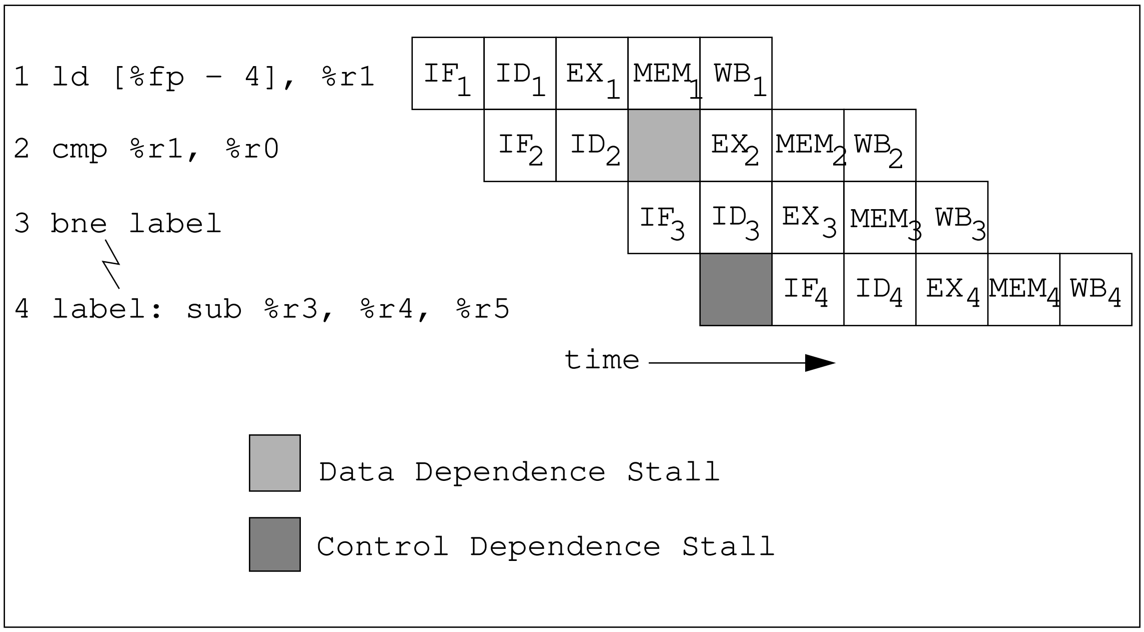

FIGURE 2. Stalls due to a load-use following a load and a taken branch instruction.

Hardware mechanisms exist which mitigate the effects of data and control dependences. *Dynamic scheduling* allows instructions to issue out of order; thus instructions waiting for data operands need not stall subsequent independent instructions. *Dynamic branch prediction* (branch history table and branch target buffer) is quite successful in presenting a continuous instruction stream to the decoder in the presence of taken branches.

While dynamic approaches can achieve good performance, it is worthwhile to consider static scheduling as well. First, if static scheduling is successful a significant fraction of the time, it justifies simpler in-order issue machines. Second, static scheduling can improve the performance of dynamic, out-of-order issue processors by virtue of increasing the effective window size.

In this paper, we investigate static instruction scheduling as a means of improving simple pipeline performance. For complexity reasons, we limit our study to local instruction scheduling--i.e. scheduling within a basic block.

1.1 Scheduling Branch Delay Slots

Branches and other control transfer instructions (CTI) may take a cycle or more to redirect instruction fetching, during which time the pipeline is being fed useless instructions. It is for this reason that *delayed branches* exist in some architectures; the point of control transfer is delayed a cycle so that a useful instruction may be scheduled immediately following the CTI, by the compiler. In this project we will attempt to move a useful instruction into every CTI delay slot.

1.2 Scheduling Instructions between a Load and its Use

Certain instructions, in particular loads and floating point operations, have multi-cycle latency; therefore pipeline stalls result if these instructions are followed immediately by data-dependent instructions (assuming no dynamic scheduling in the processor). In this project we will consider scheduling load-use instructions far enough away from the defining loads to avoid stalls.

2.0 Instruction Scheduling

This section describes the algorithms used to perform scheduling optimizations for conditional branches and other CTI instructions, calls, and loads.

2.1 Postpass Scheduling

When choosing a compiler optimization, one must decide at what point during compilation to perform the optimization. We chose to do postpass scheduling, i.e. after Sparc code generation. Intermediate SUIF code and the final code are sufficiently different that moving SUIF instructions does not make sense. For example, a single SUIF instruction may generate multiple Sparc instructions; scheduling opportunities will be different accordingly.

Pre-pass scheduling does have one advantage, however. Because register allocation has not been performed yet, all destinations are bound to unique pseudoregisters, eliminating output and anti dependences. Only true data dependences exist, exposing more opportunity for code motion.

2.2 Preparing for Code Motion

Several steps are taken to prepare for code scheduling:

- Instead of sending code output to *stdout*, a new instruction list is created for each procedure.

- Each node of the instruction list contains the following: Sparc opcode, operand and destination information (general enough to represent all forms of operands: labels, addresses, registers, immediates, etc.), and two bit vectors representing processor state used and modified by the instruction. The two bit vectors are described in the next bullet.
- To ease the job of the code scheduler, a second pass through the new instruction list is made to generate **src** and **dst** vectors. The vectors have bits for each processor register:
    1. [0:31] are for integer registers,
    2. [32:47] are for floating point registers (there are only 16 since our code generator always works in double precision, with appropriate conversions),
    3. [48] represents both integer and floating point condition code registers (we justify 1 bit for both icc and fcc by realizing that a basic block will most likely set only one or the other),
    4. [49] represents memory; *no attempt is made to do memory disambiguation*,
    5. [50] is for the Y register.

  Bits of the **src** and **dst** vectors are set based on what state the instruction depends on (**src**) and what state the instruction changes (**dst**). Having this information in hand results in an elegant code scheduler.

  Some special cases deserve mention. The **dst** vector of calls have bits set for everything assumed to be volatile across calls: volatile global and output registers, floating point registers, memory, condition codes, and the Y register. Loads and stores have the memory bit set in **src** or **dst**, respectively.

2.3 Scheduling Algorithm

The scheduler scans the instruction list backward from the last instruction. It searches for 3 classes of instructions:

- control transfer instructions that terminate a basic block; this includes conditional branches, branch always, jumps, and returns,
- calls,
- loads.

If the control transfer instruction or call is followed by a NOP, the delay slot is considered for scheduling. Loads are considered only if the load use directly follows the load. Once a delay slot needing scheduling is found, the scheduler searches for an instruction to place in the slot. There are three possible approaches:

1. Only consider instructions within the same basic block as the delay slot.
2. Consider *safe* instructions from other basic blocks; that is, consider instructions that, if moved, will not affect program correctness despite moving across control flow edges.
3. Consider instructions from arbitrary basic blocks; that is, consider instructions that, if moved, may violate original semantics.

Our scheduler use only the first approach, because it is the least complex. Using this approach requires only checking data dependences between instructions in a basic block. The second approach requires analysis of data dependences between many more instructions, as well as introducing control flow analysis. The third approach is even more complex than the second--it involves speculation, and therefore requires inserting recovery code.

There is actually one special case of speculation that is worth doing. Conditional branches in the Sparc architecture have the *annul bit*, which enables speculating branches in the taken direction. The annul bit makes moving instructions from the target basic block of a branch easier. Our code scheduler uses this feature, which is described in more detail in Section 2.3.3.

We now describe the general approach for finding a candidate instruction for filling a delay slot. Specifics unique to load, call, and branch scheduling are left for following subsections.

The basic idea is that if an instruction is to be moved past any instructions within a basic block, no *data hazards* can be violated. The "architecture" definition of a data hazard is a *possible* data dependency between two instructions. Data dependencies include true and false dependencies, both registers and memory. The scheduler must determine if any RAW (read-after-write, or true data dependency), WAR (write-after-read, or anti-dependency), or WAW (write-after-write, or output dependency) hazards exist between the candidate instruction and any of the instructions between the candidate and the delay slot.

Hazard detection is achieved elegantly by use of the **src** and **dst** bit vectors included with every instruction. As the scheduler scans for a candidate instruction, the **src** and **dst** vectors of each instruction are added to *cumulative* src and dst vectors. An instruction is a candidate for motion if:

1. **src** does not intersect the cumulative dst vector, and
2. **dst** does not intersect the cumulative src vector, and
3. **dst** does not intersect the cumulative dst vector, and
4. this instruction has not been moved before.

The first criterion is a RAW hazard check if moving the candidate up or a WAR hazard check if moving it down. The second criterion is a WAR hazard check if moving the candidate up or a RAW hazard check if moving it down. The third criterion is a WAW hazard check in any case. Note that memory dependencies are handled trivially by virtue of the memory bit in the **src** and **dst** vectors; i.e. it looks like another register. This is the very conservative approach to moving loads and stores.

The fourth criterion makes sure we don't move a candidate more than once. Without this rule, loads and calls could steal delay slot instructions from the terminating branch or other loads and calls. This is because loads and calls search both down and up the basic block; searching down means previously scheduled slots are scanned. Stealing from a previously scheduled slot is bad for two reasons: (1) it reduces the benefit of scheduling and (2) for CTI slots, it results in a bad program since the NOP is not reinserted after stealing from the CTI delay slot.

The following subsections describe specifics unique to loads, calls, and CTI.

2.3.1 Loads

Loads followed immediately by a load use are considered for delay slot scheduling. Only instructions in the same basic block are considered for filling the load delay slot. The following instructions are *not* candidates for moving: branches, jumps, returns, calls, and nops. Note that loads can be moved to load delay slots. Stores following the load may also move. However, because of the conservative memory hazard checking, stores preceding the load will never move past the load. Of course, branches, jumps, and returns are not moved because they change control flow; calls are not considered because the call delay slot would have to be moved with it.

The scheduler begins its search after the load, stopping at the terminating branch or label, or after finding a candidate for moving. If no candidate was found, the scheduler then searches for a candidate above the load, stopping at the basic block's label or upon finding a candidate.

If a candidate is found, it is removed from the original location and reinserted after the load.

2.3.2 Calls

Calls followed immediately by a NOP are considered for call delay slot scheduling. Only instructions in the same basic block are considered for filling the call delay slot. The following instructions are *not* candidates for moving: branches, jumps, returns, calls, and nops. Branches, jumps, returns, and calls are not moved for two reasons: (1) these instructions change control flow (2) it is bad to have DCTI couples (delayed control transfer couples). Note that loads and stores preceding the call may be moved to its delay slot, but not loads and stores following the call.

The scheduler begins its search after the call's delay slot, stopping at the terminating branch or label, or after finding a candidate for moving. If no candidate was found, the scheduler then searches for a candidate above the call, stopping at the basic block's label or upon finding a candidate.

If a candidate is found, it is removed from the original location and reinserted after the call. The original NOP filler is removed.

2.3.3 Branches and othe CTI

Branches and jumps followed immediately by a NOP are considered for CTI delay slot scheduling. Note that returns are also considered, but in practice will not be since their delay slots always contain a restore instruction. The following instructions are *not* candidates for moving to a CTI delay slot: branches, jumps, returns, calls, and nops. Branches, jumps, returns, and calls are not moved for two reasons: (1) these instructions change control flow (2) it is bad to have DCTI couples (delayed control transfer couples).

Our scheduler is a bit more aggressive scheduling conditional branch delay slots than other delay slots. It first looks for a candidate instruction in the same basic block as the branch. If one is not found, the scheduler looks at the *very first instruction of the target basic block*. The target instruction is a candidate for filling the delay slot if it is not a branch, jump, return, call, or nop.

If an instruction is found within the same basic block, it is removed from its current position, and reinserted in place of the NOP in the delay slot. If instead we found a candidate instruction at the target:

1. The target instruction is *replicated*; the replicated instruction replaces the NOP in the branch delay slot.
2. A new label instruction is created; this label is inserted directly after the target instruction.
3. The conditional branch is changed to branch to the new label.
4. The annul bit of the conditional branch is set.

Notice that we are performing static speculation. We hope that the branch is taken more often than not taken. If the branch is taken, the instruction in the delay slot is executed just as if it were at the target of the branch. If the branch is not taken, then we must not execute the delay slot; i.e. the speculation is incorrect, and the annul bit provides a convenient recovery mechanism (convenient for the compiler, that is!).

Also note that the target instruction must, in general, be replicated rather than simply moved. This is because the target basic block may be entered from a different source. The source must see the first instruction at the target.

In the case of moving from the target basic block, we could be even more aggressive. Right now we only check the very first instruction for a candidate. We could be more aggressive and search further down in the target block, but this would then require moving the candidate instruction to the top of the target block and then proceeding as before. Hazards would likely limit the benefit of this approach, and the complexity does not seem justified.

We do not consider moving instructions from the fall-thru basic block because this is speculating down the not taken path, which cannot be recovered from. The annul bit is for recovery from speculating down the taken path.

Moving the target instruction is not done for jumps and returns. This is because there are any number of target basic blocks. Even if there were an annul bit equivalent, it is hard to recover without complicated hardware support. The branch-always instruction is treated the same as a conditional branch, but the annul bit is not set (skip step 4 above).

3.0 Evaluation

3.1 Static Results

In this section we look at the number of delay slots in a program and the scheduler's ability to fill them. These results do not directly relate to actual program performance.

We present static results for three large programs: tsd, yacc, and compress. Compress is a compression program taken from the SPEC92 benchmark suite. Tsd is a user-level thread library. Yacc is a program taken from the LCC test suite; it is a yacc-generated program.

3.1.1 Load Delay Slots

We observed that only slightly better than half the load delay slots were being filled. We were concerned that false dependencies from temporary register reuse may be limiting the ability of the scheduler to fill these delay slots. This is only an issue for loads that have a temporary register as a destination. The getreg()/freereg() functions for temporary registers originally used a stack policy.

We changed getreg() to use a slightly smarter policy. The "smart" getreg() function implements a stack based policy, except registers for loads are taken from the bottom of the stack. Freereg() was changed so that registers freed from loads are placed at the middle of the stack. This means that if a load writes into a temporary register, nearby instructions are less likely to use the same temporary register. This should increase the chances of finding an independent instruction to fill in the delay slot.

The changes to getreg() attempt to reduce false sharing, due to register reuse, with instructions above the load. This is accomplished by giving it a register that has hopefully not been used recently.

The changes to freereg() attempt to reduce false sharing, due to register reuse, with instructions immediately below the load instruction. This is accomplished by hopefully delaying the time until a register freed from the load-use is reallocated.

TABLE 1. Load Delay slots

Benchmark	Load Delay Slots	Delay Slots Filled	Percentage Filled
tsd	645	382	59%
yacc	158	96	61%
compress	225	113	50%

Table 1 on page 8 shows how well the compiler scheduled load delay slots. The compiler does a fairly good job. The smart getreg()/freereg() help significantly in allowing the compiler to schedule load delays.

3.1.2 Call Delay Slots

TABLE 2. Call Delay Slots

Benchmark	Call Delay Slots	Delay Slots Filled	Percentage Filled
tsd	533	476	89%
yacc	28	23	82%
compress	140	125	89%

The compiler is very successful at filling call delay slots, as shown in Table 2 on page 9. The delay slot is usually filled with an instruction that loads or moves a parameter into an out register for the call.

3.1.3 Control Transfer Instruction (CTI) Delay Slots

TABLE 3. CTI Delay Slots

Bnchmark	CTI Delay Slots	Delay Slots Filled from Same Basic Block	Delay Slots Filled from Target Basic Block	Percentage Filled
tsd	590	195	357	94%
yacc	177	59	90	84%
compress	315	127	170	94%

Table 3 on page 9 shows how well the compiler did scheduling control flow instruction (CTI) delay slots. Less than half the time, for our benchmarks, was the scheduler able to find an eligible instruction from above the branch in the same basic block. Using the annul bit and filling the delay slot with an instruction from the target significantly increases the number of delay slots that can be filled.

Filling the delay slot with an instruction from the target is a win only if the branch is taken. If the branch is not taken, then the hardware behaves as if the instruction in the delay slot is a NOP.

3.2 Dynamic Results

As for dynamic results, dynamic instruction count is the only measure we could easily obtain. This is not a particularly good measure of the effects of our scheduling. A better measurement would be cycle count.

The dynamic instruction count does not reflect cycles saved by scheduling load delay slots. Scheduling load delay slots is a major part of our optimization. The effect of scheduling load delay slots can only be extrapolated from the reduction in total instructions seen by the CTI and call delay slot scheduling.

The dynamic instruction count also does not count the lost cycle when the delay slot of a branch is annulled because the branch is not taken. We unfortunately do not have statistics on the number of cycles lost due to this.

TABLE 4. Dynamic Instruction Count

Benchmark	No Scheduling	Scheduling Only from Same Basic Block	Scheduling with Annul	Reduction in Instruction Count
tsd	33,124,963	32,320,265	30,297,741	9%

Table 4 on page 10 shows dynamic instruction count for tsd. Note again that the 9% number does not reflect anything concerning load schedule effectiveness, and it is an optimistic bound with respect to cycles saved for annulled branches.

Unfortunately, this is the only data we have for dynamic measurements. The other two benchmarks execute correctly, but *qpt2* cannot handle them. We attempted all of the other SPEC92 integer benchmarks, but ran into difficulties either with SUIF, our code generator, or *qpt2*.

4.0 Conclusions

We successfully implemented a postpass instruction scheduler for filling delay slots. The static results show that there is much opportunity for scheduling delay slots. We were pleasantly surprised both by the number of opportunities and the success rate achieved. Here are some important points we observed:

- calls have a common case which allow for easy scheduling--this is a move to an out register,
- branches are most difficult to schedule unless the scheduler considers the target basic block for candidates; this is facilitated in the Sparc architecture by the annul bit,
- a load can be separated from an immediate use about half the time; scheduling is made more effective by using a smarter temporary register allocator.

We have presented evidence that scheduling can significantly reduce execution time. We feel that more thorough evaluation, using cycle count as a measure, would further support the importance of code scheduling.