

Reorganizing Global Schedules for Register Allocation

Gang Chen
GTE Laboratories, Inc.
Waltham, MA 02451
gchen@gte.com

Michael D. Smith
Harvard University
Cambridge, MA 02138
smith@eecs.harvard.edu

Abstract

Instruction scheduling is an important compiler technique for exploiting more instruction-level parallelism (ILP) in high-performance microprocessors, and in this paper, we study how to perform global instruction scheduling before register allocation (*prepass scheduling*) for control-intensive non-numerical applications. As the number of parallel function units in modern microprocessors has increased, compiler writers have increased the size of prepass scheduling regions and the number of speculative operations scheduled. However, if prepass scheduling is not carefully managed, it is easy to create places in a prepass schedule that require more register resources than are architecturally available. We propose an approach that maintains the effectiveness of prepass scheduling in exploiting ILP even with the constraint of a limited number of architectural registers. In particular, we show that the pairing of a greedy prepass scheduler with a code reorganizer performs significantly better than the common approach of using a single register-pressure-sensitive scheduler. Our experimental results show that our non-backtracking, code-reorganizing algorithm can eliminate most of the excessive register pressure created by greedy ILP scheduling, and thus improve the overall performance of the globally scheduled regions.

Keywords: Instruction-level parallelism, superblock scheduling, register allocation.

1 Introduction

Instruction scheduling is an important compiler technique for exploiting instruction-level parallelism (ILP) in modern, high-performance microprocessors. The size of a scheduling region continues to increase (from single basic blocks, to linear sequences of blocks, to acyclic regions of blocks, and to cyclic regions of blocks) in order to expose more instruction-level parallelism and to exploit the ever-increasing number of parallel function units.

While instruction scheduling is important for ILP, finding the optimal schedule for a linear sequence of instructions

under resource constraints is computationally intractable [4]. The complexity of the problem grows when scheduling is extended to arbitrary acyclic (or cyclic) control-flow regions. If we also attempt to allocate registers (which is a hard problem by itself) during scheduling, the problem becomes even more complicated. Because of this, there are very few published approaches that perform instruction scheduling and register allocation together [3,11].

Performing instruction scheduling and register allocation separately leads to the well-known phase-ordering problem. If instruction scheduling is performed after register allocation (*postpass scheduling*), the register allocator introduces extra anti-dependences and output dependences that can overconstrain the later scheduling pass. Hwu et al. [9] have shown that instruction scheduling before register allocation (*prepass scheduling*) can produce shorter instruction schedules on aggressive ILP architectures than postpass scheduling. However, greedy prepass scheduling may unnecessarily increase the distance between the write to a register and the last read of the register. The stretching of register lifetimes tends to create a larger number of simultaneously live registers. This increases the probability that the register allocator will insert spill code. The addition of spill code can lengthen the carefully constructed schedules.

Most previous work has investigated various scheduling heuristics that directly or indirectly control register pressure. One well-understood and repeatedly-studied approach is integrated prepass scheduling (IPS) suggested by Goodman and Hsu [6]. IPS keeps track of the number of live variables in registers during scheduling, and tries to control this number by varying the scheduling heuristic. If the number of live variables in registers is less than the register limit, IPS schedules instructions that increase the amount of ILP. Otherwise, IPS attempts to schedule instructions that free registers (reduce register pressure). In order to schedule instructions that free registers, IPS may select instructions that cause interlocks, which sacrifices ILP for less register pressure. Also it is quite possible that there is no ready instruction that frees registers. IPS will proceed by exceeding the register limit, which forces the register allocator to insert spill code. Figure 1 gives such an example.

However, we have observed that the accumulated register pressure is often due to an overly-aggressive scheduler—one that has initiated an excessive amount of ILP, not because it is necessary to minimize the length of the instruction schedule, but simply because it can. In these cases, we can achieve an equally-short instruction schedule by scheduling some of the instructions later. In Figure 1, the register

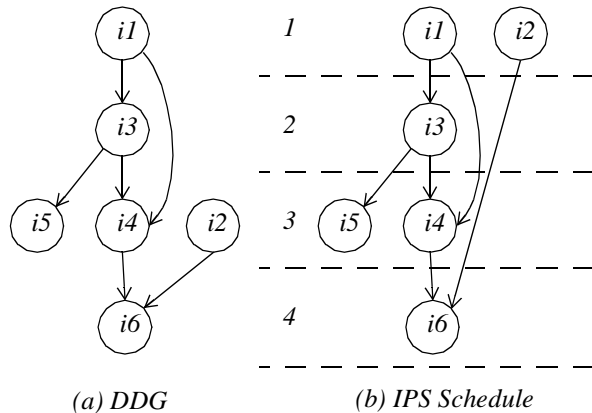


Figure 1. Example data dependence graph (DDG) and the resulting IPS schedule. We assume a machine with 3 universal function units, unit latencies, and 2 available registers. Instruction $i1$ is scheduled first because it is on the critical path. Instruction $i2$ is greedily scheduled into Cycle 1 because it is the only other ready instruction. When scheduling Cycle 2, we notice that we have run out of registers and the only ready instruction ($i3$) does not free a register. IPS goes ahead and schedules Instruction $i3$, creating 3 simultaneously live values between Cycle 2 and 3 and later causing the register allocator to insert spill code.

pressure problem is due to the early scheduling of Instruction $i2$. We can reduce the register pressure, remove the need for any spill code, and maintain the schedule length by delaying the scheduling of Instruction $i2$ until Cycle 3.

A common suggestion at this point is to introduce backtracking into the prepass scheduler, but researchers are leery of backtracking because it is hard to know how to limit the backtracking. In general, it is difficult to identify the scheduling actions that are overly aggressive *before* the final schedule is seen. This observation caused us to question the traditional approach of using a single scheduling pass before register allocation.

In this paper, we suggest a two-phase approach to prepass scheduling: first schedule the code to minimize path lengths and then reorganize the schedules to reduce register pressure. Instead of trying to control register pressure during scheduling, we allow the scheduler to do what it does best—exploit ILP. Then in a single following phase, we reduce the register pressure by reorganizing the schedule in a way that also maintains the overall schedule length when possible. This approach achieves the desired effect of undoing over-aggressive scheduling by iterating through the instruction sequence just one more time after scheduling and before register allocation. It also avoids any unnecessary use of interlocks and the complexity of a backtracking scheduler.

We evaluate our schedule-reorganize-allocate approach for control-intensive, non-numerical programs running on next-generation microarchitectures with lots of ILP. Most previous studies [2,6,13] have investigated the interaction between instruction scheduling and register allocation on scientific programs with large basic blocks and on machines

with little ILP. Though we focus on register pressure, we are aware that an overly aggressive global scheduling algorithm can also over-allocate other resources. For example, the scheduling of highly speculative instructions wastes instruction fetch bandwidth when the speculation is frequently incorrect. We believe that our technique is applicable to situations where the performance penalty of fetching highly speculative instructions outweighs the speculative benefits.

Section 2 introduces some basic aspects of instruction scheduling that help in understanding our proposal. Section 3 describes the register-pressure-sensitive scheduling approach proposed by Goodman and Hsu [6]. Though we do not use this heuristic during our prepass scheduling, we do incorporate a version of this heuristic in our reorganizing phase that runs between instruction scheduling and register allocation. Section 4 presents the details of our code reorganizing phase, and Section 5 presents our experimental results. Section 6 discusses more related work, and Section 7 concludes.

2 Basics of instruction scheduling

In this section, we discuss the differences between top-down (TD) and bottom-up (BU) cycle-based list scheduling, and the effect of these algorithms on register pressure. TD scheduling proceeds from the entry (or entries) of a control-flow region to its exit (or exits); BU proceeds from control-flow exits to entries. Though there are two different list-scheduling styles, cycle-based and operation-based scheduling, most compilers use cycle-based (or simply cycle) scheduling because it is conceptually simpler and easier to implement. In particular, it is straightforward to maintain and update pipeline and register-usage states during cycle scheduling.

2.1 Local scheduling

Within a single basic block, the algorithms for TD and BU scheduling are essentially identical. Figure 2 outlines the general algorithm for cycle scheduling without control-flow dependences. Conceptually, a TD scheduler tries to schedule instructions as close to the control-flow entry as possible, while a BU scheduler schedules them as close to the control-flow exit as possible. Without resource constraints, TD and BU scheduling achieve the same schedule length (the length of the critical path). Much of the previous work [6,9] has focused on how to choose the “best” scheduling candidate from the list of ready instructions.

2.2 Global scheduling

When scheduling is extended to multiple basic blocks, we must include control-dependence constraints as well as data-dependence constraints in our summary of the instruction ordering constraints. The addition of control-dependence constraints breaks the symmetry between TD and BU scheduling because of the difference between pulling a control-dependent instruction up across the control-flow branch point and pushing it down across the

```

schedule_region
  summarize instruction ordering constraints;
  initialize ready_list and hold_list;
  cur_cycle = 0;
  while (ready_list and hold_list not empty)
    pack_cycle;

pack_cycle
  create empty failed_list;
  // fill up current cycle
  while (free resources && ready_list not empty)
    choose "best" candidate C from ready_list;
    remove C from ready_list;
    if (C can schedule without resource conflicts)
      mark C as scheduled;
      update ready_list;
    else
      add C to failed_list;
  // prepare for next cycle
  move instr's from failed_list to ready_list;
  while (ready_list empty && hold_list not empty)
    cur_cycle++;
  promote instr's from hold_list to ready_list;

```

Figure 2. General outline of a cycle scheduler. For a TD approach, an instruction belongs in the ready_list if its data-dependent predecessors have all been scheduled and their latencies fulfilled. An instruction belongs in the hold_list if its data-dependent predecessors have all been scheduled but some of latencies are unfulfilled. A BU approach replaces the word predecessor with successor in the preceding definitions.

control-flow join point. Pulling an instruction up causes speculative execution; pushing it down requires predicated execution.

The goal of speculative execution is to eliminate the control dependence constraint on an instruction’s execution. This can be achieved if the execution of the instruction before¹ its control-dependent branch always maintains program semantics, independent of the execution of the branch. In some cases, we can freely speculate an instruction without changing it in any way. In other cases, we can only speculate using register renaming techniques and instruction set extensions such as non-excepting operations [14]. On the other hand, there is no way to push an instruction down across a join point, execute this instruction without regard to its control dependence constraint, and maintain correct program semantics. Predicated execution always requires explicit architectural support. For these reasons, many global scheduling algorithms [8,10,12] exploit ILP only by pulling instructions up. The natural way of pulling instructions up is TD cycle scheduling.

For example, it is straightforward to extend the TD cycle-

¹ To avoid confusion when discussing TD and BU scheduling, we will consistently use the words “before” and “after” when referring to the relative placement of instructions or blocks in the acyclic control flow of the scheduling region. In particular, an instruction i occurs before instruction j if i is encountered first when following control flow from the entry of the region.

scheduling algorithm in Figure 2 so that it implements superblock scheduling [10]. In particular, the following describes the specifics of our implementation of prepass superblock scheduling used in the experimental results. Because the global scheduler must consider control-dependence constraints, it adds an ordering constraint between a branch instruction and an instruction control dependent on that branch when necessary to prevent unsafe speculative execution [14]. To maximize the opportunities for speculative execution, our algorithm performs register renaming to eliminate illegal speculative executions. It also adds constraints to keep branch instructions in program order and to have them scheduled after all instructions preceding them in the program order.

Although all published superblock schedulers use a TD scheduling approach, it is possible to implement the superblock scheduler in a BU fashion. Because a superblock contains only a single entry point (there are no join points within a superblock), we can use the algorithm in Figure 2 and the same instruction ordering constraints as described above for TD superblock scheduling. To force instruction speculation, we give each branch instruction a high priority (the branch will be the “best” instruction when it becomes ready). In this way, the scheduler ends the scheduling of one basic block and starts the scheduling of the next block as quickly as possible. Once a branch is scheduled, all unscheduled instructions that were after the branch in the original instruction sequence will now be scheduled before that branch (they will be speculated).

Notice however that we have more control in TD than BU global scheduling. For example, in the TD scheduler, we can use an earliest-exit heuristic, i.e. instructions in block i have a higher priority than those in block j if i comes before j . This heuristic avoids lengthening the critical path to the earliest exit due to speculative operations, which is important when scheduling non-numerical programs that have relatively hard-to-predict control flow. On the other hand, a BU scheduler cannot decide to speculate an instruction in block j based on the resource constraints of block i , since i has not yet been scheduled. This lack of control will be more severe for treeregion [8] and DAG-based [12] scheduling, where we greedily speculate instructions up along multiple paths.

2.3 Register pressure

Even though TD cycle scheduling is the choice for many global instruction schedulers, experimental results have shown that greedy TD cycle scheduling increases register pressure excessively, forcing the register allocator to insert significantly more spill code. In the end, we may get performance worse than that achieved by postpass scheduling alone.

Excessive register pressure is due to the inherent greediness of cycle-based scheduling. A TD approach pulls instructions up as far as possible; a BU approach pushes them down as far as possible. When scheduling separates the write of a value from later reads by an excessive distance, we increase

the likelihood that the register allocator will encounter more simultaneously live values than fit into the available registers.

For local scheduling, both TD and BU schedulers could suffer from their greediness. For global scheduling however, the inequalities between upward and downward code motion (i.e., the constraints on downward code motion are much stronger) make the register pressure problem less severe for the BU approach. Since a BU scheduler cannot push instructions down very far, it does not have a chance to be very greedy in separating a value’s use from its definition. We will use these observations in Section 4 when presenting our new approach.

3 Register-pressure-sensitive scheduling

In this section, we describe a register-pressure-sensitive scheduling algorithm that uses the idea of integrated prepass scheduling (IPS) proposed by Goodman and Hsu [6]. They described the IPS idea for a TD basic-block scheduler. We refer to such IPS-style TD cycle schedulers as TD-IPS. We begin with a description of IPS-style scheduling for basic blocks and then describe how to extend this to superblocks. Whenever possible, we use Goodman and Hsu’s terms.

To perform IPS-style scheduling on *basic blocks*, the TD scheduler needs to keep track of the number of available registers (AVLREG) during scheduling. A register is reserved for a variable when a write to the variable is scheduled. The register for a variable is released when the last read of the variable is scheduled. We set the initial value of AVLREG to be the number of architectural registers minus the number of variables that are live at the top and referenced within the scheduling region. When AVLREG is above a threshold, the scheduler chooses the ready instruction using a candidate selection policy that minimizes the schedule length. Such a policy is called CSP (code scheduling for pipelined processor) for short. When AVLREG drops below the threshold, the scheduler uses a different candidate selection policy, i.e. it selects instructions that reduce register pressure even if an interlock is required. Such a policy is called CSR (code scheduling to minimize register usage) for short. When AVLREG is restored to an acceptable value, CSP resumes.

To perform IPS-style scheduling on *superblocks*, the TD scheduler must also count the references to live variables at each superblock exit. We can accomplish this by treating the variables read on the exit edges as being read by the branch instruction corresponding to that exit.

4 Bottom-up code reorganization

Considering the aforementioned pros and cons of TD and BU scheduling, we propose a bidirectional, two-phase approach for prepass global instruction scheduling. The first phase is a greedy TD scheduling that aggressively exploits ILP and compacts the overall schedule. The produced schedule indicates how well we can do and which instructions need to be speculated. In the second phase, we

reorganize the region using a BU scheduler.

The goal of the second phase is to push overhoisted instructions down (and thus reduce register pressure) while maintaining the schedule length determined during TD scheduling. As mentioned in Section 2, BU scheduling is better at maintaining register pressure than TD scheduling, but BU scheduling is not an effective approach for exploiting speculation. When the BU scheduler runs as the second phase, however, it can use the TD schedule to determine when to close scheduling for each basic block in the scheduling region. In this manner, we can reduce the excessive register pressure created by TD scheduling and still aggressively exploit ILP to minimize the length of the instruction schedule. Although in this paper we present the algorithms in the context of superblock scheduling, we have also applied this two-phase prepass scheduling approach to a DAG-based scheduler.

Section 4.1 introduces a straightforward BU code reorganizer and shows how it can reduce register pressure by pushing overhoisted instructions down within the space of the TD schedule. We refer to TD scheduling followed by BU reorganization as TD-BU. Section 4.2 then shows how we can control register pressure more effectively by incorporating the CSR approach in our BU reorganizer. We refer to this improved approach as TD-BU-CSR.

4.1 TD-BU

Our BU reorganizer is a cycle scheduler based on the algorithm in Figure 2, and it uses the same instruction ordering constraints that we used during the greedy TD scheduling phase. In addition, the BU reorganizer takes as input the sequential list of instructions produced by the TD scheduler, which we call *TD_sequence*. Overall, the BU reorganizer differs from the algorithm in Figure 2 in just two key ways: scheduling priority and resource checking.

Scheduling priority. The BU reorganizer uses the position of an instruction in the *TD_sequence* to determine its scheduling priority. The instruction with the highest priority is the one at the end of the *TD_sequence*. If instruction j is after instruction i in the *TD_sequence*, j has a higher priority than i .

If instruction j is unscheduled and all instructions after j in the *TD_sequence* have been scheduled, we refer to j as *next_unsched*. We refer to the point immediately after *next_unsched* in the *TD_sequence* as the *scheduled_frontier*.

In a sense, the BU code reorganizer walks up the *TD_sequence* looking for holes to fill with instructions that were scheduled before this point in the *TD_sequence*. More specifically, we look for the scenarios during walk-up that there are empty slots in the current BU cycle and *next_unsched* is not ready. We then search the *ready_list* for a scheduling candidate, and if we find one, we schedule it in the current cycle. This instruction was obviously not on the critical path since we were able to move it later in the *TD_sequence*.

Resource checking. In order to schedule an instruction, we certainly need to check if it has resource conflicts with already-scheduled instructions, as every scheduler does. More than that, if it is not *next_unsched*, we also check in advance whether it has resource conflicts with the intervening unscheduled instructions in the *TD_sequence*. Note that we know which instructions they are, and we can estimate their issue cycles according to the TD schedule. We check the unscheduled instructions from *next_unsched* up to the scheduling candidate. We know the timings of unscheduled instructions relative to the *scheduled_frontier* as determined by the TD scheduler. We use these relative timings to estimate issue cycles of the unscheduled instructions based on the issue cycle for the *scheduled_frontier* as determined by the BU reorganizer. For machines that are not fully pipelined, we can use this heuristic to avoid producing a schedule after BU reorganization that is significantly longer than the schedule produced by TD scheduling.

As a simplification, we also allow the compiler to bound the range of the resource check. Conceptually, the benefit gained by shortening a very long lifetime often overcomes any small schedule perturbations. Furthermore, we can perform these checks quickly by employing an efficient resource checking technique. In our implementation, we use the automaton-based resource checking technique described by Bala and Rubin [1].

4.2 TD-BU-CSR

The greedy BU reorganizer described above pushes instructions down opportunistically and in general can help to reduce register pressure. But it does not directly attempt to reduce register pressure to a value below a specified limit. As Figure 3 shows, the greedy BU reorganizer can push too many instructions down and introduce new spill points. By borrowing ideas from the CSR approach of IPS however, we can overcome the greediness of the BU reorganizer and direct it to strive for a particular target register limit.

In particular, our TD-BU-CSR approach tracks AVLREG only during bottom-up reorganization. We set the initial value of AVLREG in a manner appropriate for a BU traversal; it is the number of architectural registers minus the number of variables that are live at the bottom of the superblock and written within the superblock. Interestingly, it is simpler to track AVLREG during BU scheduling than TD scheduling. Assume that v is a program variable, and that node n_v is a node in the DDG that writes that variable. TD-IPS scheduling requires a count of read references to be associated with every node n_v . The count is initialized to the number of read references of n_v when we schedule n_v ; and it is decremented as we schedule each instruction that uses the value produced by n_v . BU-CSR on the other hand reserves only one bit for every node n_v . We assert the bit for n_v whenever we schedule an instruction that reads the value produced by n_v . The bit is cleared when we schedule n_v .

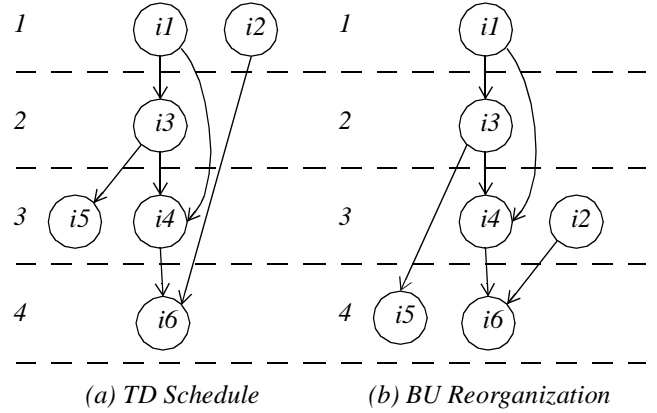


Figure 3. BU reorganization of the TD example from Figure 1. Reorganization helps by pushing Instruction i_2 down from Cycle 1 to 3; this reduces the register pressure between Cycles 2 and 3 to a value equal to the number of available registers. Unfortunately, it also pushes Instruction i_5 from Cycle 3 to 4, thereby increasing the register pressure between Cycles 3 and 4 to a value greater than the number of available registers.

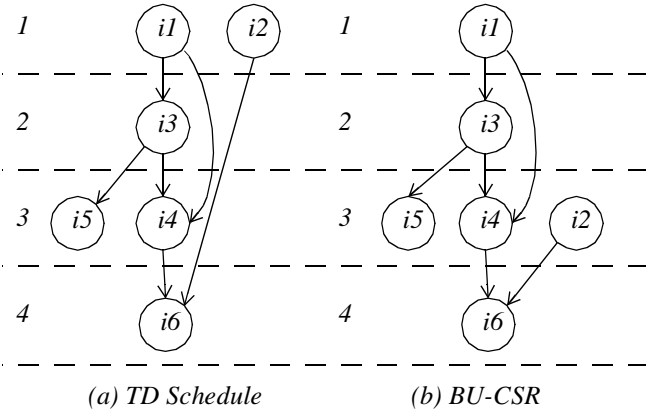


Figure 4. BU-CSR reorganization of the example in Figure 1. We start from the bottom of the TD schedule. We schedule Instruction i_6 first; it reserves the 2 destination registers of Instructions i_2 and i_4 . Then we switch to CSR since AVLREG is 0. Under CSR, we will not push Instruction i_5 into Cycle 4 since it would drive AVLREG below 0. Instead, we move up to Cycle 3 and schedule Instruction i_2 because it releases one register. (Instruction i_4 also releases one register but it reserves two others.) After Instruction i_2 is scheduled, AVLREG becomes 1, and we can then schedule Instructions i_4 and i_5 . After reorganization, we have generated a schedule that is as short as the greedy TD schedule and never exceeds the register limit.

For BU-CSR, when AVLREG is above a threshold, we select the candidate as described in Section 4.1. When AVLREG is below the threshold, we select instructions that reduce register pressure, and we do not check whether the scheduling candidate has resource conflicts with unscheduled instructions. When AVLREG is restored to an acceptable value, greedy BU reorganization resumes. Figure 4 shows the result of BU-CSR for the example in Figure 1.

5 Experimental evaluation

This section presents an empirical comparison of the scheduling schemes described in Sections 3 and 4.

5.1 Methodology

We use compiled simulation to collect our performance numbers. Compiled simulation uses the instructions and features of a real machine wherever possible, falling back on software to emulate features that are not available and to monitor performance. To perform compiled simulation, we first compile a version of the program that would run on our hypothetical machine. We analyze this program, building a database of information about the number of cycles and instructions associated with each basic block. Then we undo the register allocation and spill decisions (if necessary) and re-optimize for a real machine. This produces an executable that performs all of the operations that would have been performed on the hypothetical machine, but spills and reloads values more often due to the limitations of the real machine. We then instrument the resulting executable with the values from the database, so the program records cycle and instruction counts. Lastly, we add runtime support to emulate non-excepting instructions using the operating system’s trap handlers.

For all experiments in this paper, we simulate machines with an Alpha-like instruction set architecture. Our machine models support non-excepting variants of instructions, allowing speculation of unsafe operations like loads, but not stores and divides. We simulate two machine models: a 4-issue model and an 8-issue model. They use the same latency model for all the operations. The maximum numbers of issues per instruction category are given in the first and second rows of Table 1; the function unit latencies are given in the third row. For both models, we assume there are 32 integer registers and 32 floating-point registers.

We mainly use a number of SPECint benchmarks in our experiments. The SPECint benchmarks tend to have shorter basic blocks and more unpredictable control flow than the SPECfp benchmarks. For all of our experimental results, we train and test on separate inputs.

	integer		floating point		loads & stores
	mul	other	div	other	
4 - issue	4	4	2	2	2
8 - issue	8	8	4	4	4
latency in cycles	6	1	8/16	3	2

Table 1: Machine models—issue rules and latencies.

5.2 Compiler

For all of our experiments, we use a two-pass scheduling approach: prepass scheduling, global register allocation, then postpass scheduling. We vary only the prepass scheduler. The prepass scheduling region is a superblock [10]. We select superblocks using the mutual-most-likely heuristic [11], and before scheduling them, we enlarge (using loop unrolling, loop peeling, and branch target expansion techniques) and optimize (using redundancy elimination, move combining, and register renaming techniques) the superblocks [15]. Unless specifically stated otherwise, we peel loops that iterate at most 4 times, and we unroll loops up to a limit of 16 copies or 512 instructions.

We compare the four prepass scheduling techniques described in Sections 2–4: greedy TD scheduling (TD), IPS-style TD scheduling (TD-IPS), greedy TD scheduling plus greedy BU reorganization (TD-BU), and greedy TD scheduling plus BU reorganization with CSR (TD-BU-CSR). For both TD-IPS and TD-BU-CSR, we set the AVLREG threshold for entering CSR at 2 and for exiting CSR at 4.² We also measure the benefit of greedy TD scheduling with an unlimited number of registers; these results serve as an upper bound that we would like to reach if possible.

After prepass scheduling, we apply a graph-coloring register allocator with a round-robin policy controlling its list of free registers. Our register allocator is a faithful implementation of the coloring algorithm described by George and Appel [5], which also integrates register coalescing (copy propagation). After register allocation, we again schedule the superblocks using a greedy TD approach to hide the latency of spill code.

5.3 Results

Figures 5 and 6 present relative speedup. These are improvements in performance over postpass basic-block scheduling on a single-issue machine with otherwise identical latency and register models.

Figure 5 presents experimental results for the 4-issue model. We see that TD-BU performs uniformly better than TD-IPS. This means that BU code reorganization is overall more effective than TD-IPS in terms of reducing register pressure and producing shorter schedules. Greedy BU reorganization is so effective on some benchmarks, such as *wc*, *compress*, and *perl*, that the speedup for 32 registers is equivalent to that for unlimited registers. Results for these benchmarks are limited by the machine parallelism and not by the number of registers. Thus, the poor performance of TD and TD-IPS is mainly due to the excessive greediness of TD

² Our AVLREG thresholds are a bit conservative because we cannot precisely track the number of live variables that the register allocator will commit to registers. For example, our calculation ignores global variables that are live through the superblock but never used within it; the allocator may decide to allocate a subset of these globals to registers. The exiting bound is greater than the entering bound to add hysteresis to the system.

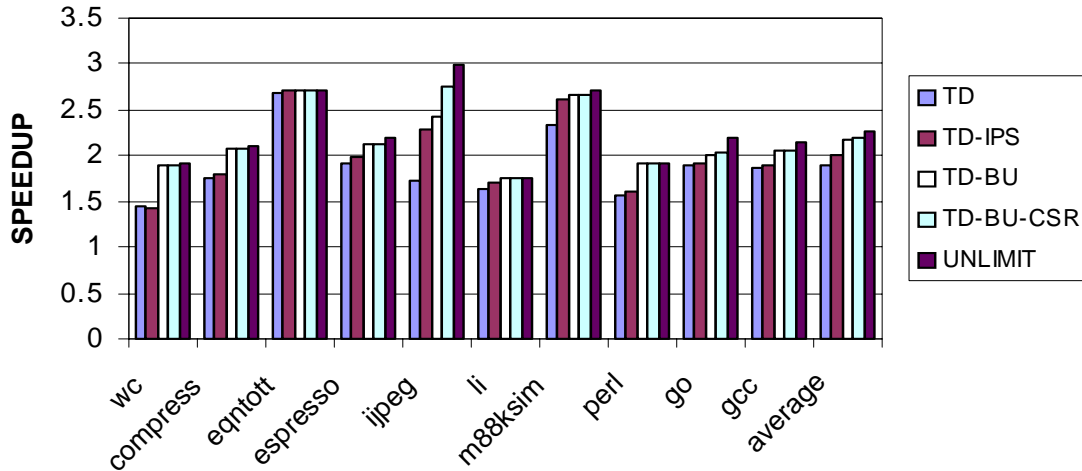


Figure 5. Speedup comparison on the 4-issue model. The average improvement of TD-BU-CSR over TD-IPS is 12%. The average difference between TD-BU-CSR and UNLIMIT is 3%.

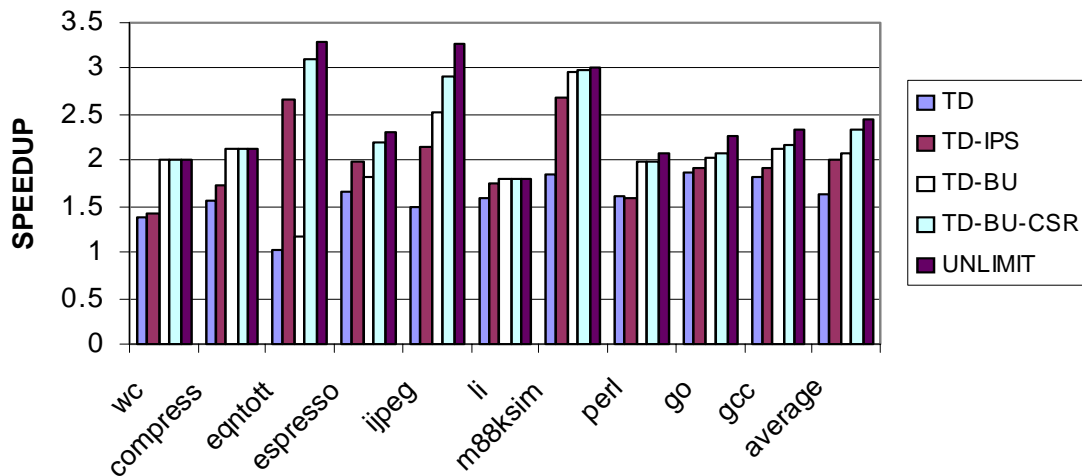


Figure 6. Speedup comparison on the 8-issue model. The average improvement of TD-BU-CSR over TD-IPS is 18%. The average difference between TD-BU-CSR and UNLIMIT is 4%.

scheduling causing the benchmarks to become register-bound. BU reorganization suppresses this greediness effectively. As shown in Table 2, there is no spill code inserted by register allocator after TD-BU. TD-BU-CSR in fact never invokes CSR during reorganization on these benchmarks; TD-BU and TD-BU-CSR therefore achieve identical speedups.

TD-BU-CSR does significantly outperform TD-BU on *jpeg* however. This is because *jpeg* is register-bound at several dynamically-frequent program points, and thus it is extremely important to balance the exploitation of ILP with increases in register pressure. When register pressure increases too much, we lose less performance from reducing ILP than inserting spill code. Table 2 shows one such procedure in *jpeg* where TD-BU-CSR reduces spill code

significantly more than TD-BU. This compromise also explains the gap that remains for *jpeg* between TD-BU-CSR and UNLIMIT.

A noticeable difference exists between TD-BU-CSR and UNLIMIT for the benchmarks *go* and *gcc* because these benchmarks contain large procedures with many variables. When there are only 32 integer registers, the register allocator inserts spill code even without prepass scheduling. We have inspected the code in these applications, and we believe that we could reduce the difference between TD-BU-CSR and UNLIMIT by improving our register allocator. In particular, we notice that some of the spill code was inserted into frequently-executed areas, even though it was possible to move it to less-frequently-executed areas.

	TD	TD-IPS	TD-BU	TD-BU-CSR
wc.main	72/47	58/36	0/0	0/0
compress.compress	82/70	51/45	2/2	2/2
eqntott.cmppt (4)	0/0	0/0	0/0	0/0
eqntott.cmppt (8)	49/47	0/0	40/38	0/0
jpeg. jpeg_idct_islow	306/230	66/63	100/101	16/16

Table 2: Number of static loads/stores inserted by register allocator in some selected procedures. For eqntott, we provide the numbers for both the 4-issue and the 8-issue model. The other numbers are for the 4-issue model only. These numbers give further perspective on the results shown in Figure 5 and Figure 6.

Figure 6 presents the results for the 8-issue model. Here the benefit of TD-BU-CSR becomes more noticeable. TD-BU-CSR outperforms both TD-IPS and TD-BU on *eqntott*, *espresso*, *go*, *gcc*, and *jpeg*. This is because prepass scheduling requires more ILP (and thus greater register pressure) to exploit fully the machine parallelism in the 8-issue model. Greedy BU reorganization alone cannot effectively limit the register pressure, and hence CSR is an important component of the reorganization phase. This contrast between 4-issue and 8-issue is strongly shown for *eqntott*. On the 4-issue model, greedy TD scheduling with 32 registers works well enough that the impact of IPS and BU reorganization is negligible. On the 8-issue model, greedy TD scheduling generates a compact schedule that exceeds the register pressure limit. The register allocator is forced to spill values and ruin the schedule. Table 2 illustrates this difference in terms of spill code generated. This makes greedy TD scheduling a poor choice for *eqntott* on the 8-issue model. Furthermore, TD-IPS is more effective than TD-BU for *eqntott* on the 8-issue model. TD-BU performs poorly because the frequently-executed portion of *eqntott* has a very dense schedule that leaves few chances for downward code motion that do not trade-off ILP for reduced register pressure.

6 Related work

Goodman and Hsu [6] proposed the IPS approach. They applied IPS on large basic blocks in the context of local register allocation. Bradlee et al. [2] studied the interaction between basic-block IPS and graph-coloring global register allocation. Norris and Pollock [13] incorporated the idea of IPS into Region Scheduling [7], and also studied its interaction with graph-coloring global register allocation. All these studies integrated the IPS idea directly into a single, prepass scheduler. We use the IPS idea in our BU reorganizer, which runs immediately after a greedy TD scheduler.

Hwu et al. [9] studied the importance of prepass superblock scheduling for control-intensive, non-numerical programs. They showed that prepass scheduling extracts significantly more parallelism than postpass scheduling alone for aggressive ILP microarchitectures. In that study, the prescheduler computes the scheduling priority of an instruction as the weighted sum of several heuristic functions, including factors such as *slackness* and *register_use* that help to control the greediness of TD scheduling. Unlike IPS, their approach does not attack the register pressure problem directly. Also the weights given to heuristic functions are architecture- and application-dependent.

Broadly stated, we have proposed a bidirectional scheduling scheme. In the Bulldog compiler, Ellis [3] implemented a bidirectional scheme, called bottom-up greedy (BUG), that traverses his DDG assigning operations to the individual clusters in his VLIW machine. Given an operation in the middle of the DDG that must be assigned to a unique cluster, only a bidirectional algorithm can easily ensure that the dependent successors and predecessors of that operation are clustered together with the operation. Even though the assignment of operations to clusters may be thought of as a piece of scheduling, Ellis used a separate (traditional) top-down, cycle scheduler to pack the operations into the VLIW instructions. We unaware of any other work in bidirectional scheduling.

The Bulldog compiler is novel also because it integrated register allocation with top-down cycle scheduling. The Multiflow compiler [11], a production successor of Bulldog, introduced a limited form of backtracking into this integrated algorithm to achieve better results.

7 Conclusion

We have developed a new approach to solving the phase-ordering problem associated with instruction scheduling and register allocation. Instead of trying to perform instruction scheduling and register allocation together or trying to backtrack during scheduling, we proposed a global code reorganization phase that runs after the greedy prepass scheduler and before the register allocator. This reorganizer controls register pressure while maintaining the effectiveness of the prepass scheduler.

In particular, we have implemented our two-phase approach as a TD traversal followed by BU traversal of the scheduling region. Though the specific traversal directions are those that are natural, the key aspect of our approach is a separation of the determination of scheduling length from the minimization of register pressure.

We show that this two-phase approach to prepass scheduling performs significantly better than the common approach of using a single register-pressure-sensitive scheduler. We demonstrated this for control-intensive, non-numerical applications running on next-generation, high-performance microprocessors. Though we specifically described how to implement our approach for superblock scheduling, this is a

general approach that we have also used in a global, DAG-based scheduler.

8 Acknowledgments

We would like to thank Cliff Young and Glenn Holloway for providing part of the compiler infrastructure used in this work. Gang Chen performed this work as part of his Ph.D. research at Harvard University. Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), a DARPA grant no. NDA904-97-C-0225, and research grants from AMD, Compaq, HP, IBM, and Intel.

9 References

- [1] V. Bala and N. Rubin, "Efficient Instruction Scheduling Using Finite State Automata," *Proc. 28th Annual Intl. Symp. on Microarchitecture*, Nov. 1995, pp. 46–56.
- [2] D. G. Bradlee, S. J. Eggers, and R. R. Henry, "Integrating Register Allocation and Instruction Scheduling for RISCs," *Proc. 1991 Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems*, SIGPLAN Notices 26(4), Sept. 1991, pp. 122–131.
- [3] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures," MIT Press, Cambridge, MA, 1986.
- [4] J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode compaction: Looking Backward and Looking Forward," *National Computer Conf.*, 1981, pp. 95–102.
- [5] L. George and A. Appel, "Iterated Register Coalescing," *ACM Transactions on Programming Languages and Systems*, 18(3):300-324, May 1996, pp. 300–324.
- [6] J. R. Goodman and W. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," *Proc. 1988 Intl. Conf. on Supercomputing*, July 1988, pp. 442–452.
- [7] R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Trans. on Software Engineering*, 16 (4), April 1990, pp. 421–431.
- [8] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treeregion Scheduling for Wide Issue Processors," *Proc. 4th Intl. Symp. on High-Performance Computer Architecture*, Feb. 1998, pp. 266–276.
- [9] W. Hwu, et al, "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," *IEEE Transactions on Computers*, VOL 44, NO. 3, March 1995, pp. 353–370.
- [10] W. Hwu, et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing* 7(1/2), Kluwer Academic Publishers, May 1993, pp. 229–248.
- [11] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing* 7(1/2), Kluwer Academic Publishers, May 1993, pp. 51–142.
- [12] S. Moon and K. Ebcioğlu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Proc. 25th Annual Intl. Symp. on Microarchitecture*, Dec. 1992, pp. 55–71.
- [13] C. Norris and L. Pollock, "An Experimental Study of Several Cooperative Register Allocation and Instruction Scheduling Strategies," *Proc. 28th Annual Intl. Symp. on Microarchitecture*, Nov. 1995, pp. 169–179.
- [14] Michael D. Smith. "Architectural Support for Compile-Time Speculation," *The Interaction of Compilation Technology and Computer Architecture*, edited by David Lilja and Peter Bird, Kluwer Academic Publishers, 1994, pages 13–49.
- [15] C. Young, "Path-based Compilation," Ph.D Thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, Jan. 1998.