Optimal Global Instruction Scheduling with Unlimited Resources

M. Anton Ertl, David Gregg, and Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, 1040 Wien, Austria
{anton,dave,andi}@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 18515
Fax.: (+43-1) 58801 18598

Abstract. We present a method for optimal whole-procedure instruction scheduling for machines with unlimited resources: The program and its dependences are transformed into a linear programming problem, which can then be solved using an off-the-shelf linear problem solver. This scheduler is an intermediate step towards a more realistic global instruction scheduler, but it has also an immediate use: We use it to evaluate the significance of the restrictions imposed by static scheduling and for determining an upper bound for the performance of more realistic global instruction schedulers. We have applied the scheduler to several benchmarks and compared it to a dynamic scheduler with unlimited resources. For some benchmarks, they perform equally well; for others, dynamic scheduling performs much better; on closer inspection it appears that the causes for this performance difference can be reduced by performing well-known transformations before scheduling (in particular, loop transformations).

1 Introduction

Computer designers and computer architects have been striving to improve uniprocessor performance since the invention of computers. In this quest for higher performance the latest challenge has been to increase the number of instructions that can be executed in parallel (*instruction-level parallelism*), which ideally would make it possible to execute the same number of instructions in a shorter time. Deeply pipelined, superscalar and VLIW (very large instruction word) (micro)architectures already offer the hardware resources necessary for instruction-level parallelism.

However, in real programs instructions depend on the results of other instructions (or depend on them in some other way), and must be executed after them (i.e., not in parellel). In order to utilize the hardware resources (in particular, parallel functional units) effectively, the instructions have to be rearranged (scheduled). This can be done by the hardware (out-of-order execution) or by the compiler.

The main techniques used for compiler-based instruction scheduling are trace or superblock scheduling, which works on loop-less code, and software pipelining, which works on loops. Software pipelining does very well on loops with high trip counts, but not so well for loops with low trip counts [STK98], that are common in non-numeric programs [Lar91]. Many papers have proposed ways to extend the scope of these techniques, but to the best of our knowledge no usable technique for whole-procedure scheduling has been presented yet.

Our goal is to schedule a whole procedure (with arbitrary control structures) at a time, thus eliminating ramp-up and ramp-down effects on trace or loop boundaries (procedure inlining can reduce the ramp-up/down effects for procedure boundaries).

In Section 3 we present an optimal solution for this problem for machines with unlimited resources. This machine model is not realistic, but we believe that our solution is an intermediate step towards a realistic whole-procedure instruction scheduler; e.g., it can be used for guiding the heuristics of more realistic schedulers.

Our solution also has an immediate application: We can use it to determine the significance of the restrictions imposed by compiler-based scheduling. A fundamental difference between compiler-based (static) and hardware-based (dynamic) scheduling is that dynamic scheduling can schedule an instruction differently every time it is executed, depending on circumstances such as the actual control-flow, whereas with static scheduling the schedule is fixed at compile-time. We compare our static scheduler to an optimal dynamic scheduler with unlimited resources in Section 4.

2 Related Work

There exists a vast body of work on instruction scheduling. The simplest form is basic block instruction scheduling [LDSM80]; however, the restriction on basic blocks limits the parallelism to about two parallel instructions on average [JW89].

This is symptomatic of a general problem in instruction scheduling: if scheduling is limited to a certain control-flow region, on entry to a region a ramp-up in parallelism occurs, and similarly, on exit a ramp-down occurs, reducing the parallelism.

The two main directions for instruction scheduling beyond basic blocks have been trace scheduling and software pipelining.

Trace scheduling [Fis81, Ell85, LFK$^+$93] (and its variant, superblock scheduling [H$^+$93]) schedule cycle-free paths through the control-flow graph. In loops this results in ramp-up and ramp-down effects on every iteration; loop unrolling is performed to reduce this problem.

Software pipelining is used for scheduling loops, originally just simple loops (i.e., loops containing only a single basic block). The most successful method seems to be modulo scheduling [RG81, Rau94], which produces near-optimal results. The main problems with software pipelining are its limited applicability and its performance on loops with low trip counts [STK98].

Many attempts have been made to extend the applicability of software pipelining to loops containing control structures [Lam88, WHSB92, ME97]. However, these extended versions are not as close to optimal for complex loops as modulo scheduling is for simple loops, so there is still room for improvement.

There are also a few approaches for scheduling arbitrary control-flow graphs: Percolation scheduling [Nic85] is a general framework for global instruction scheduling, based on local scheduling transformations. The main problem with this approach is finding good guidance rules for applying these transformations. Aiken and Nicolau [AN91] present a global scheduling algorithm that can be applied to general control-flow graphs. It requires a (heuristic) selection function to guide the process. To the best of our knowledge, no useful selection function has been presented. The present work differs from these works in that it produces an optimal schedule (i.e., no need to find guidance heuristics), but assumes an unrealistic machine model.

The problem of optimal software pipelining of simple loops with optimal register allocation has been cast into integer linear programming problems [GAG94, EDA95]. Our work differs from these works in attacking a completely different problem: optimal scheduling of whole procedures without dealing with register allocation or resource constraints. Moreover, our work casts the problem into a linear programming problem (instead of an *integer* linear programming problem). So, our work has as much similarity with these works as two papers have that use pseudocode to present algorithms for different problems.

3 Optimal Global Scheduling

3.1 Machine Model

The machine model used in this article is a statically scheduled superscalar with an unlimited set of resources (instruction issue bandwidth, functional units, etc.). This model is equivalent to a VLIW, except that the compiler does not use tricks like writing to a register that a later instruction will read before the write is in effect.

We assume a machine model that can execute several branches per cycle. If we were restricted to one branch per cycle, instruction-level parallelism would be limited by branch bandwidth. This model is equivalent to Ebcioğlu's tree instructions [EA97]. Of course, our machine model has a mechanism allowing the speculative execution of exception-generating instructions, e.g., delayed exceptions [EK94].

We assume that there is an unpartitioned integer and an unpartitioned FP register set. We assume a fixed latency for all instructions. While it is possible to handle variable latencies (e.g., loads with a mixture of cache hits and cache misses) in our scheduler (by assuming a certain probability for each possible latency), we leave this to future work.

3.2 Important concepts

The goal of scheduling is to minimize the execution time:

$$\sum_{b \in basic\_blocks} \mathit{frequency}(b)\mathit{length}(b)$$

Scheduling does not change the basic block execution frequencies, it can only change the length of basic blocks. Therefore, the control-flow graph is an important data structure for optimal global scheduling.

Each basic block has a *basic block length.* In this paper it is only used for determining the lengths of paths containing the basic block. Basic block lengths can be negative; this just indicates that a path through this basic block is shorter than a path that bypasses the basic block. A basic block also has a *start time*; for a given path the start time of (an instance of) the basic block is the sum of the basic block lengths on the path up to (the instance of) the basic block.

A program also contains instructions. The *home basic block* of an instruction is the basic block in which the instruction resides in the original program. Each instruction has an *issue time* relative to the start time of its home basic block. The issue time can be negative or larger than the basic block length. The instruction may have several instances along different control-flow-paths in the final result, but all are scheduled at the same time relative to the start of the home basic block of the instruction.

There is one exception from this total independence of basic block lengths and instruction issue times: The issue time of a conditional or indirect branch is always the last cycle of its home basic block.

```
 #   1    int strlen(char *s) {
strlen:                                             # bb2
 #   2    char *t = s;
        move    t,s          # t=s                  # inst0
 #   3    while (*s != '\0')
        lb      t0,0(s)      # t0=*s                # inst1
        beqz    t0,end       # while (t0 != '\0')   # inst2
loop:                                               # bb3
 #  4     s++;
        addu    s,s,1        # s++                  # inst3
        lb      t0,0(s)      # t0=*s                # inst4
        bnez    t0,loop      # while (t0 != '\0')   # inst5
end:                                                # bb4
 #   5    return s-t;
        subu    v0,s,t       # return_value = s-t   # inst6
        j       ra           # return               # inst7
```

Fig. 1. MIPS R3000 assembly language source of strlen, with instruction and basic block numbers

As a running example throughout this article, we will use `strlen()`. Figure 1 shows the function with the basic blocks and instructions numbered for future purposes.

3.3 Optimality

The algorithm we present is optimal (i.e., minimizes the execution time for a given execution profile), given the following restrictions:

- It assumes given dependences, and does not apply dependence-reducing transformations by itself. Such transformations have to be applied beforehand (or ignored, and the compiler must fix the code afterwards).
- It assumes given control flow, and does not apply transformations like loop unrolling. Therefore every loop iteration takes at least one cycle. However, it can change basic block lengths, and it can even move branches and control-flow joins up or down a fixed amount of cycles (even across branches and control-flow joins). So, our algorithm can perform a bit of loop peeling, but more complex loop transformations have to be applied before scheduling.
- It schedules all instances of an instructions always at the same issue time relative to its home basic block. Thus it foregoes the opportunity to schedule instances of an instruction differently, depending on the context. This restriction may look more severe than it actually is: At least if there is only one instance of the home basic block in the schedule, scheduling the instances of the instructions differently cannot improve the schedule: It just introduces slack between some instances of the instruction and the instructions that depend on this instruction (if the replication happens by moving the instruction up from its home block) or instructions on which this instruction depends (if the replication happens by moving down). We are still investigating the case with multiple instances of the home basic block.

The restrictions on given dependences and given control flow introduces a phase ordering problem for some dependence-removing transformations, in particular, for run-time disambiguation [Nic89], a transformation that removes memory dependences: Normally we would apply the transformation by scheduling as if the dependence was not there, and then applying the transformation where necessary. However, this transformation introduces new branches, so we would have to profile and schedule again.

3.4 Control-flow graph refinements

Before scheduling, we introduce basic blocks (*landing pads*) at control flow edges from basic blocks with several children to basic blocks with several parents. This transformation is optional, but beneficial: It makes it possible to change the length (in cycles) of control-flow paths through such edges without changing other path lengths. E.g., in our example (Fig. 2) the landing pad 7 makes it possible to change the length of the path 2-4 without changing the length of the paths 2-3-4, 2-3-3-4 etc. Many code-moving optimizations (e.g., partial redundancy elimination) also benefit from or require this transformation: It makes it possible to move an instruction from any basic block without introducing it into a different path. The newly introduced basic blocks are therefore called *landing pads*.

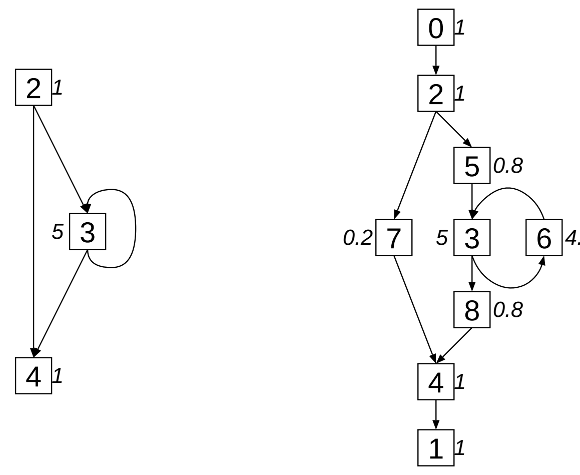

Fig. 2. The control-flow graph of strlen, without (left) and with (right) landing pads and top and bottom nodes; the numbers in the boxes are basic block numbers, the numbers outside are the execution frequencies.

To simplify processing, we also add a top and a bottom node: The top node is the control-flow predecessor of all entry points to the procedure and all other basic blocks that may be jumped to from unknown places. The bottom node is the successor of all procedure exits and jumps with possibly unknown target (indirect jumps). Of course, for, e.g., C switch statements the scheduler should know all targets of the indirect jump. We could also treat calls as exit points and the instructions after them as entry points, but this would make it impossible to schedule across calls.

We may want to execute a branch speculatively (typically, if its condition is known earlier and if it has a lower prediction accuracy than the branch originally before it). This is represented in our control-flow graph by having negative basic block lengths, so we don't have to transform the control-flow graph for this.

3.5 Linear Programming Problem

Without resource constraints, global scheduling (by changing basic block lengths) becomes a linear programming problem. The object function is to minimize the execution time:

$$\sum_{b \in basic\_blocks} frequency(b) length(b)$$

The basic block execution frequencies are determined by profiling or by estimates and are constant for our optimization problem.

There are also a number of side conditions, described below. We use the following variables in the problem (see Section 3.2 for an explanation of the concepts):

length(b) The length of the basic block b in cycles.[1]
time(i) The issue time of an instruction i.

The side conditions are:

- An instruction a that depends on instruction b must be scheduled at least as many cycles after b as the latency is between b and a, for every path between b and a. Formally,

  $$\bigwedge_{a \in insts} \bigwedge_{b \in preds(a)} \bigwedge_{p \in paths(b,a)} time(b) + latency(b,a) \leq time(a) + \sum_{bb \in p} length(bb)$$

  $paths(b, a)$ includes the home basic block of b, but not the home of a (if the dependence does not cross a basic block boundary, the path is empty). It also does not include control-flow graph cycles, because running through a cycle can only increase the path length (every cycle has at least the path length 1, see below), resulting in a less severe (and therefore redundant) constraint than the corresponding constraint without the cycle.
- Conditional branches (and other control-flow splits) have to be scheduled at the end of their home basic blocks:

  $$\bigwedge_{b \in branches} time(b) = length(home(b)) - 1$$

- All instructions have to be scheduled after the start of the top basic block and before the end of the bottom basic block:

  $$\bigwedge_{a \in instructions} \bigwedge_{p \in paths(\top,a)} time(a) + \sum_{bb \in p} length(bb) \geq 0$$

  $$\bigwedge_{a \in instructions} \bigwedge_{p \in paths(a,\bot)} time(a) + 1 \leq \sum_{bb \in p \cup \{\bot\}} length(bb)$$

[1] The tool we use for for solving linear problems (`lp_solve`) requires that all variables be positive. We represent potentially negative variables by subtracting a sufficiently large constant offset.

– Cycles in the control-flow graph should at least take one cycle (if this is too long, the compiler can unroll the loop before scheduling):

$$\bigwedge_{b \in basic\_blocks} \bigwedge_{p \in paths(b,b)} \sum_{bb \in p} length(bb) \geq 1$$

This is necessary for the termination of the following scheduling phase (see Section 3.6). It is not really a restriction, because recurrences normally[2] enforce this condition anyway.

For the strlen example, we get the following constraints:

$$\begin{aligned}
inst0 + bb0 &\geq 0\\
bb1 + bb2 + bb5 + bb3 + bb8 + bb4 &\geq inst0 + 1\\
bb1 + bb2 + bb7 + bb4 &\geq inst0 + 1\\
inst1 + bb0 &\geq 0\\
bb1 + bb2 + bb5 + bb3 + bb8 + bb4 &\geq inst1 + 1\\
bb1 + bb2 + bb7 + bb4 &\geq inst1 + 1\\
inst2 - inst1 &\geq 2\\
inst2 + bb0 &\geq 0\\
bb1 + bb2 + bb5 + bb3 + bb8 + bb4 &\geq inst2 + 1\\
bb1 + bb2 + bb7 + bb4 &\geq inst2 + 1\\
inst2 - bb2 &= -1\\
inst3 - inst3 + bb3 + bb6 &\geq 1\\
inst3 + bb0 + bb5 + bb2 &\geq 0\\
bb1 + bb3 + bb8 + bb4 &\geq inst3 + 1\\
inst4 - inst3 &\geq 1\\
inst4 + bb0 + bb5 + bb2 &\geq 0\\
bb1 + bb3 + bb8 + bb4 &\geq inst4 + 1\\
inst5 - inst4 &\geq 2\\
inst5 + bb0 + bb5 + bb2 &\geq 0\\
bb1 + bb3 + bb8 + bb4 &\geq inst5 + 1\\
inst5 - bb3 &= -1\\
inst6 - inst0 + bb2 + bb7 &\geq 1\\
inst6 - inst0 + bb2 + bb8 + bb3 + bb5 &\geq 1\\
inst6 - inst3 + bb2 + bb8 &\geq 1\\
inst6 + bb0 + bb7 + bb2 &\geq 0\\
inst6 + bb0 + bb8 + bb3 + bb5 + bb2 &\geq 0
\end{aligned}$$

[2] The only exception that comes to mind are busy waiting loops, where progress in time is also welcome.

$$\begin{aligned}
bb1 + bb4 &\geq inst6 + 1 \\
inst7 + bb0 + bb7 + bb2 &\geq 0 \\
inst7 + bb0 + bb8 + bb3 + bb5 + bb2 &\geq 0 \\
bb1 + bb4 &\geq inst7 + 1 \\
inst7 - bb4 &= -1 \\
bb3 + bb6 &\geq 1
\end{aligned}$$

The constraints here are arranged somewhat differently from the general versions. The *inst* variables are the issue times of the instructions, and the *bb* variables are the lengths of the basic blocks. We will assume execution frequencies for the basic blocks as shown in Fig. 2, resulting in the following object function that has to be minimized:

$$bb0 + bb1 + bb2 + 5bb3 + bb4 + 0.8bb5 + 4.2bb6 + 0.2bb7 + 0.8bb8$$

We can solve such problems using a linear problem solver, resulting in values for the variables. For our example problem, an optimal solution is shown in Fig. 3. It has a cost of 7, meaning that the average call of strlen takes seven cycles.

variable	value	variable	value
bb0	0	inst0	0
bb1	2	inst1	0
bb2	3	inst2	2
bb3	0	inst3	-4
bb4	0	inst4	-3
bb5	1	inst5	-1
bb6	1	inst6	1
bb7	-2	inst7	-1
bb8	-2		

Fig. 3. Solution of the linear programming problem for strlen

There is always an optimal integral solution to this problem, because all the coefficients for the variables in the constraints are 1 or -1 [PS82]. This problem is therefore a linear programming problem (polynomial), not an integer linear programming problem (NP-complete).

The introduction of constraints for all paths between two points may lead to an exponential number of constraints for some control flow graphs. To avoid this problem, we introduce summary variables for some subgraphs of the control-flow graph. We do not discuss this here because of the length constraints on the paper.

3.6 Producing a Schedule

In order to turn this solution into a schedule, we first have to apply transformations that eliminate negative basic block lengths while maintaining optimality (i.e., without changing control-flow path lengths). If a basic block has a negative length, we first try to push the negative length up to basic blocks it post-dominates (see Fig. 4). I.e., we do not push the negative length up across branches. If this does not suffice, we pull the basic block lengths of post-dominating basic blocks up (again using the transformation shown in Fig. 4 and again not crossing branches). If this still does not suffice, the control-flow joins directly after the negative-length basic block are pushed down below the next branch, then the instance of the branch now dominated by the negative-length basic block is swapped with the branch above it (Fig. 5). This gets the branches in the right order, but still leaves us with basic blocks of negative length; however, they are now below the branch and we can continue by pulling up the basic block lengths of post-dominating basic blocks etc. Does this process terminate, if there are cycles in the control-flow graph? Yes: Cycles have at least length one; the negative basic block length cycles around for a while, but at every iteration it becomes larger, until the length reaches zero.

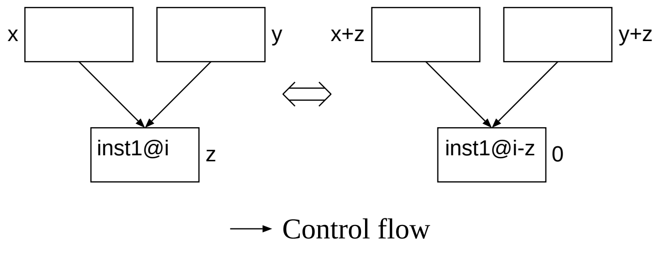

Fig. 4. Pushing the basic block length up

These transformations may replicate parts of the control-flow graph, but the distances between the branches and the lengths of control-flow paths are not changed; if one branch was -5 cycles before another branch, afterwards all its instances are 5 cycles after (all instances of) the other branch. The basic block starts have shifted, however. After the transformations, no basic blocks with negative length are left. Figure 6 shows the control-flow graph of strlen before and after the transformation; in this case, pulling the basic block lengths of post-dominating basic blocks up was sufficient to eliminate negative basic block lengths.

Now that the control-flow graph is in canonical form, we can schedule the rest of the instructions (the branches are already scheduled).

First, we need to translate the issue times of the instructions with respect to the original control flow graph into issue times with respect to the canonical

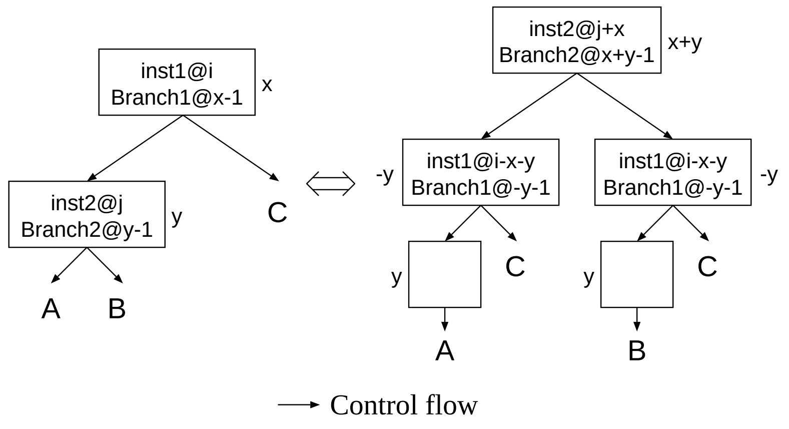

Fig. 5. Getting two branches in the right order (y < 0)

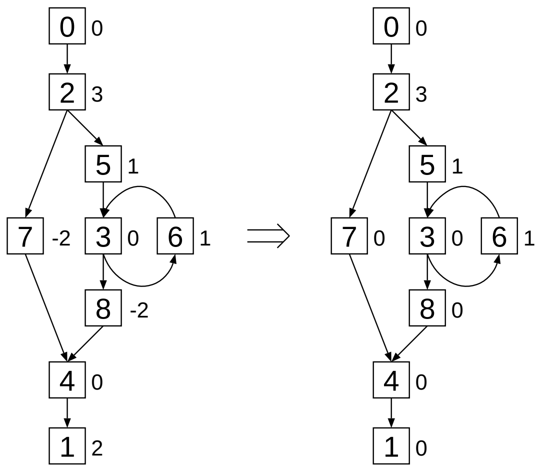

Fig. 6. The control-flow graph of strlen with optimal basic block lengths, before and after removing negative basic block lengths

form. This is best done by performing changes to the instruction issue times corresponding to the changes in the control-flow graph: Basically, whenever some length is added to a basic block before the home basic block of an instruction, the same value is subtracted from the issue time of the instruction. Figure 4 and 5 show how this rule applies for the transformations used; `inst@i` means that the instruction `inst` has issue time `i`.

Then we can move the instructions from their home basic block to basic blocks where their issue time is within the basic block: If the instruction's issue time is negative, then it is moved up to all control flow predecessors using the

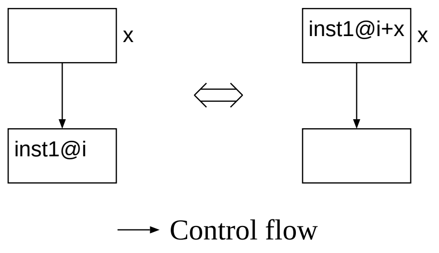

Fig. 7. Moving an instruction up or down

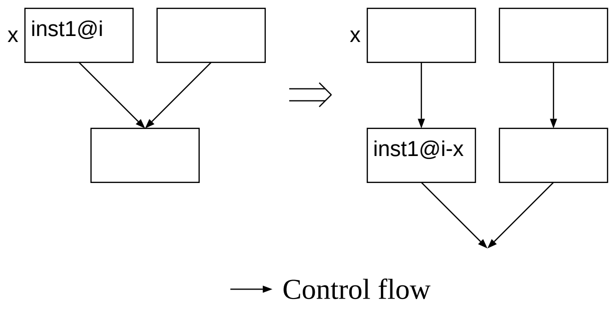

Fig. 8. Basic block replication when moving an instruction down across a control-flow join

transformation shown in Fig. 7; if one of the resulting instances still has a negative issue time, it is moved up further. If the instruction's issue time is greater than or equal to the basic block length, it is moved down to all control-flow successors using the transformation shown in Fig. 7 in the right-to-left direction; again, this transformation is repeated for the instances where it is necessary. An instruction must not be moved down across a control-flow join; therefore, whenever it is necessary to move an instruction down to a basic block that has more than one control-flow predecessor, this basic block is replicated (see Fig. 8).

For our example, this results in the schedule shown in Fig. 9. In this case, it is not necessary to move a join down, so the control-flow graph does not change. There are five instances of instruction 3, created by crossing the control-flow join before the basic block 3 four times; all of these instances are necessary, as they belong to different iterations of the loop. It is easy to see that one loop iteration has been peeled off unnecessarily (in the basic block immediately preceding the loop); the peeling does not change the object function (the average execution time, ignoring cache effects)), so the amount of peeling that is performed is arbitrary and depends on the linear problem solver.

Because we ignored antidependences during scheduling, we now have to perform register renaming (or repairing) and maybe some more code replication to ensure correctness. We also have to introduce run-time-disambiguation and

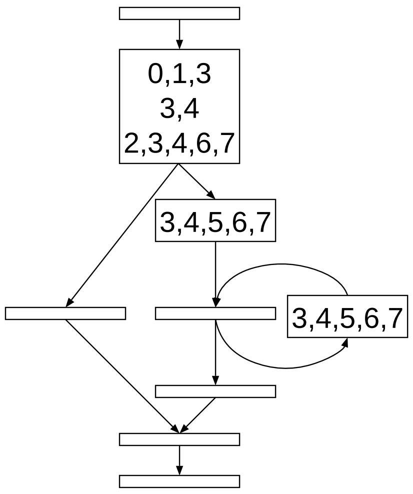

Fig. 9. The control-flow graph of strlen, after scheduling the instructions

delayed exception code if we ignored the corresponding dependences. In our example, we will use register renaming; to keep the example simple, we will not do delayed exceptions, but assume that the code does not trap. Figure 10 shows the final code. We unrolled the loop four times for register renaming purposes. The control-flow instructions are moved to the end of each group.. The sense of some branches has been reversed to get the instructions of one group together.

```
       move t,s;     lb t0,0(s);  addu s1,s,1;
       addu s2,s1,1; lb t1,0(s1);
       addu s3,s2,1; lb t2,0(s2); subu v0,s,t;  bnez t0,cont1; j ra
cont1: addu s4,s3,1; lb t3,0(s3); subu v0,s1,t; bnez t1,cont2; j ra
cont2: addu s1,s4,1; lb t4,0(s4); subu v0,s2,t; bnez t2,cont3; j ra
cont3: addu s2,s1,1; lb t1,0(s1); subu v0,s3,t; bnez t3,cont4; j ra
cont4: addu s3,s2,1; lb t2,0(s2); subu v0,s4,t; bnez t4,cont5; j ra
cont5: addu s4,s3,1; lb t3,0(s3); subu v0,s1,t; bnez t1,cont2; j ra
```

Fig. 10. strlen, scheduled optimally for a machine without resource constraints

4 Comparison with dynamic scheduling

This section compares our static scheduler against a dynamic scheduler; both are optimal and use unlimited resources. Note that under these assumptions the static scheduler cannot beat the dynamic scheduler, because the static scheduler has to satisfy the same dependences, but has a few additional restrictions. The main purpose of this section is to gain some insight how much of a problems the additional restrictions of static scheduling are. Note that for more realistic assumptions a static scheduler can beat a dynamic scheduler.

We implemented an optimal dynamic scheduler based on a trace-driven simulation, and a prototype of an optimal static scheduler as described in Section 3. Both were implemented as tools built with Atom [SE94]; the input for the tools is executable code for the Alpha architecture. For solving the linear optimization problem in the static scheduler we use `lp_solve`; our prototype static scheduler does not yet do the tasks discussed in Section 3.6, but since we do not have a machine with unlimited resources, we do not need the final code anyway.

We did not implement transformations for removing dependences, but the schedulers can optionally ignore certain dependences. In particular, we ignore for anti and output dependences, because they can be removed without effect on the schedule on a processor with unlimited resources.

Other dependence types pose more problems: We tried to avoid benchmarks that have memory dependences, because of the phase ordering problems discussed in Section 3.3. For those benchmarks that have memory dependences, in the tradition of other limits studies we used unrealistically good static disambiguation (based on the dependences that actually do occur) to compete with the dynamic scheduler that counts only dynamic flow dependences through memory.

We benchmarked leaf procedures to eliminate the differences between static and synamic scheduling at procedure boundaries.

We used the following C functions as benchmarks: strlen, binsearch, sosearch (a move-to-front linear search), compute_crit_path (compute the critical path of a data dependence graph), check_mi_conflict (conflict check in a class hierarchy), test_encode (checking a class hierarchy encoding), abalone (position evaluation for a game).

These procedures all have some characteristics that make them problematic for conventional software pipelining or trace scheduling algorithms: low trip counts, loops containing other control structures, data-dependent loops or low branch prediction accuracy.

We compared the results of our scheduler to the results for a dynamic scheduler with unlimited resources (i.e., an upper limit for the performance with the given dependences). We used a trace-driven simulator to determine the run-time of the dynamically scheduled code. For our experiments we considered only data flow dependences (because we know how to remove the others, given unlimited resources). Note that static scheduling can never outperform dynamic scheduling under the conditions of our test. Figure 11 shows the results.

We see several patterns: strlen, binsearch and sosearch give similar results for static scheduling as for dynamic scheduling; simple recurrences limit their parallelism. For compute_crit_path we explain the lower performance for static scheduling as an effect of the worse disambiguation compared to dynamic scheduling.

function name	basic block count	useful instructions executed	cycles statically scheduled	IPC	cycles dynamically scheduled	IPC
strlen	3	490	102	4.8	99	4.9
binsearch	8	206	79	2.6	71	2.9
sosearch	10	14025	2007	7.0	2005	7.0
compute_crit_path	18	692	45	15.4	17	40.7
check_mi_conflict	24	9800432	677445	14.5	709	13823.0
test_encode	24	792227	151891	5.2	230	3444.0
abalone	75	46568567	860670	54.1	439898	105.9

Fig. 11. Results of scheduling with unlimited resources

Check_mi_conflict and test_encode both contain nested loops that apparently can be very well parallelized (with hardly any memory dependences, this is not surprising), but not by our static scheduler; performing loop optimizations before scheduling should reduce this problem.

For abalone we are still investigating the difference between the static and the dynamic scheduling results.

5 Conclusion

Present methods for compiler-based instruction scheduling are restricted to traces or relatively simple loops. We have presented a first step towards a general global scheduler: a global scheduler for a machine with unlimited resources. Moreover, this scheduler produces optimal schedules.

It works by transforming the scheduling problem into a linear programming problem, solving that problem, and using the solution to rearrange the control-flow graph and to schedule the instructions in the new control-flow graph.

We compared this scheduler with an optimal dynamic scheduler (in the form of a trace-driven simulator). As a result, we see that recurrences are the limiting factor for both schedulers on some benchmarks, while the lack of transformations like loop optimizations limits static scheduling in comparison with dynamic scheduling for other benchmarks.

Acknowledgements

Alexander Mehlmann explained some linear programming issues to us. Discussions with Sylvain Lelait helped improve the paper. The referees of CC'98 and Micro-31 provided valuable feedback on earlier versions of this paper.

References

[AN91] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In Alexandru Nicolau, David Gelernter, Thomas Gross, and

David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Programming, pages 274–290. Pitman, London, 1991.

[EA97] Kemal Ebcioğlu and Erik Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In 24th *Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[EDA95] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 31–40, July 1995.

[EK94] M. Anton Ertl and Andreas Krall. Delayed exceptions — speculative execution of trapping instructions. In *Compiler Construction (CC '94)*, pages 158–171, Edinburgh, April 1994. Springer LNCS 786.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *International Symposium on Microarchitecture (MICRO-27)*, pages 85–94, 1994.

[H$^+$93] Wen-mei W. Hwu et al. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1/2):229–248, 1993. Reprinted in [RF93].

[JW89] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–282, 1989.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[Lar91] J. R. Larus. Parallelism in numeric and symbolic programs. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Programming, pages 331–349. Pitman, London, 1991.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Pattrick W. Mallet. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[LFK$^+$93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Liechtenstein, Robert P. Nix, John S. O'Donnel, and John C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1/2):51–142, 1993. Reprinted in [RF93].

[ME97] S. Moon and K. Ebcioglu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[Nic85] Alexandru Nicolau. Uniform parallelism exploitation in ordinary programs. In *1985 International Conference on Parallel Processing*, pages 614–618, 1985.

[Nic89] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining. In *International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, 1994.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher, editors. *Instruction-level parallelism.* Kluwer Academic Publishers, 1993. Reprint of The Journal of Supercomputing, 7(1/2).

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techgniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Microprogramming Workshop (MICRO-14)*, pages 183–198, 1981.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM – A system for building customized program analysis tools. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, 1994.

[STK98] Esther Stümpel, Michael Thies, and Uwe Kastens. VLIW compilation techniques for superscalar architectures. In Kai Koskimies, editor, *Compiler Construction (CC'98)*, pages 234–248, Lisbon, 1998. Springer LNCS 1383.

[WHSB92] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 170–179, 1992.