Optimal Instruction Scheduling using Constraint Logic Programming

M. Anton Ertl
DMS Decision Management Systems Ges.m.b.H.
Wallnerstraße 2, A-1014 Wien
ertl@vip.at

Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
andi@mips.complang.tuwien.ac.at

Abstract

Instruction scheduling is essential for the efficient operation of today's and tomorrow's processors. It can be stated easily and declaratively as a logic program. Consistency techniques embedded in logic programming enable the efficient solution of this problem.

This paper describes an instruction scheduling program for the Motorola 88100 RISC processor, which minimizes the number of pipeline stalls. The scheduler is written in the constraint logic programming language ARISTO and uses a declarative model of the processor to generate an optimal schedule. The model uses lists of domain variables to represent the pipeline stages and describes the dependencies between instructions by constraints in order to ensure correct scheduling. Although optimal instruction scheduling is NP-complete, the scheduler can be applied to real programs because of the speed gained through consistency techniques.

1 Introduction

Current RISC processors achieve their high performance by exploiting parallelism through pipelining and multiple execution units. As a consequence, the results of previous instructions are sometimes not available when the next instruction is executed. If the next instruction needs the result, it has to wait. The problem of arranging the instructions in a way that reduces the number of wait cycles is known as instruction scheduling or instruction reordering. Microcode compaction is a related problem.

Instruction scheduling can have a drastic impact on performance: On the Motorola 88100 one floating point multiplication can be started at every cycle, but the result is only

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Programming Language Implementation and Logic Programming (PLILP) 1991, Springer LNCS 528, pages 75-86

available after six cycles. Even a simple formulation of optimal instruction scheduling is an NP-complete search problem [HG83]. Scheduling is further complicated by the interactions between the execution units. E.g., on the Motorola 88100 only one result at a time can be written back to the register file. Since up to three execution units may want to write a result, the scheduler must also consider the priority scheme implemented in the hardware. Scheduling is even more important for the superscalar and VLIW processors now being developed which can execute multiple instructions per cycle.

The existing algorithms make use of an explicit dependency graph. The scheduler determines the path length, heuristically selects one of the instructions having no predecessor, appends it to the instruction sequence, and removes it from the graph. The usual heuristic procedure chooses the instruction with the longest path length. Hu [Hu61] developed an early algorithm for a similar problem. Hennessy and Gross [HG83] present an algorithm with $O(n^4)$ worst-case complexity for a simple instruction scheduling problem: The results of the instructions are available after a fixed amount of time. Gibbons and Muchnick [GM86] describe an algorithm with $O(n^2)$ worst-case and observed linear complexity, that produces slightly worse schedules. [GH88] and [BEH91] integrate instruction scheduling and register allocation. These algorithms work on basic blocks, whereas Fisher [Fis81] introduces trace scheduling for global microcode compaction. The same technique is used for VLIW machines [CNO+88]. Another technique to achieve better scheduling by transcending basic block boundaries is software pipelining [Lam88], which can also be combined with loop unrolling [LKB91]. A short overview of the field is given in [Kas90, chapter 8.5].

The use of consistency techniques combined with tree searching for solving combinatorial problems has been an Artificial Intelligence research topic for a long time [Wal72, Nud83]. Problems are represented as networks of constraints on variables. The domains of the variables are represented explicitly. Constraints are used actively to remove values from the domains which cannot appear in a solution. In this way the search tree is pruned *a priori*.

Van Hentenryck [VH89, VHD87] integrated consistency techniques in logic programming. He added a new data type, the domain variable. It behaves like an ordinary logic variable except that it can be instantiated only with values from its domain. Constraints are used to reduce these domains and thus the search tree.

For example, given the variables `X` with the domain $\{1, 2, \ldots 6\}$ and `Y` with the domain $\{4, 5, \ldots 9\}$, the constraint `X #> Y` (Declaratively `#>` means the same as `>`) immediately reduces the domains to $\{5, 6\}$ and $\{4, 5\}$ respectively. The constraint keeps watching the variables and becomes active again if the domains are further reduced.

Domain variables and constraints combine nicely with the search capabilities inherent in logic programming languages and form an efficient approach to solving combinatorial problems. The resulting language has been used to efficiently solve a wide range of toy and real-world problems, among them scheduling problems [VH89, DSVH90, Ert90]. Programs for these purposes look like generate&test-programs with the tests coming first.

The experiences gained with this method suggest its application for instruction scheduling. Furthermore the use of constraint logic programming eases the integration of other code generation and optimization techniques, e.g. [Gan89].

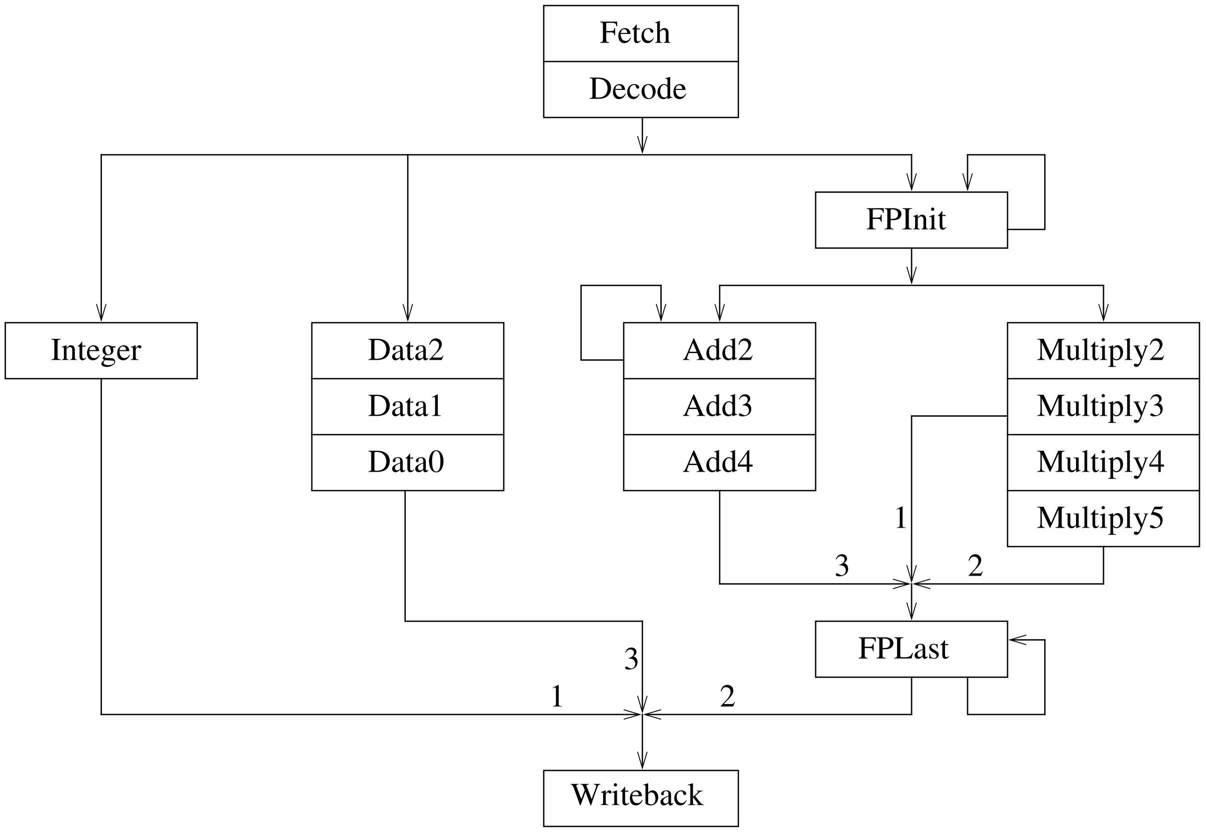


Figure 1: Pipeline stages in the Motorola 88100. Numbers indicate priorities.

2 The Problem

A modern microprocessor consists of several resources, which can be used by only one instruction at a time. These resources include pipeline stages, buses and register file ports. The goal of an instruction scheduler is to achieve high throughput by maximizing resource utilization.

As an example, take a look at the Motorola 88100 processor [Mot90] (see figure 1): It has

- a two-stage instruction fetch and decode unit,
- a one-stage integer execution unit,
- a three-stage data unit for accessing data memory,
- a five-stage floating-point add pipeline and a six-stage floating-point multiply unit, which share the initial and final stages,
- three 32-bit buses, and
- one write port to the register file.

Scoreboarding (an ‘in use’-bit for every register) is used to ensure that instructions that access a register are not executed before the register is up to date.

When several instructions compete for the same resource, an arbitration scheme selects one instruction for processing and the others are stalled. This can cause stalls in earlier stages of the pipeline up to the instruction fetch stage.

On the Motorola 88100 the following conflicts can occur:

- An instruction needing a register whose scoreboard bit is set is held in the decode stage until the bit is cleared by a writeback into the register.
- The integer unit, FPLast and the data unit compete for writeback slots.
- Multiply3 (integer multiply), Multiply5 (floating point multiply) and Add4 compete for FPLast.
- Several instructions need some pipeline stage(s) for more than one cycle, most notably instructions with double word source operands, which need two cycles for fetching data through the buses and stall the decode stage for one cycle.

This behaviour must be considered by the instruction scheduler. To preserve correctness, the reordering must obey several constraints: Only reads from a register can be swapped. Writes into a register can be swapped neither with reads nor with writes to this register. The same reasoning applies for accesses to memory locations. Since memory accesses can be aliases, the scheduler has to treat the whole memory like a single register, unless it can prove that the accesses are not for the same location.

3 The Solution

We have written an instruction scheduler for the Motorola 88100 in ARISTO. ARISTO is an industry-level constraint logic programming language, that employs consistency techniques for the solution of combinatorial problems [Ert90]. The scheduler is implemented as a logic program that models the execution structure of the processor.

The scheduling program consists of three parts. The first part reads the assembly language source and splits it into basic blocks[1]. Moreover, simple peephole optimization is performed. The second part works on basic blocks. It collects the constraints (test phase) and searches for an optimal solution (generate phase). This part is displayed in figure 2; The subgoals are explained in the subsequent sections. The final part uses the resulting ordering information and outputs the reordered instructions as assembly language source.

The basic data structure used by the scheduler is the domain variable. For every instruction i in the basic block there is a domain variable $\mathtt{D}_i$ representing its decode cycle. These variables are later instantiated by the generate part. In the same way, for every instruction and every pipeline stage it uses, there is a variable representing the cycle during which it resides in the stage. For some stages, where an instruction can stall, there are variable pairs `Start .. End`. The scheduler works by assigning cycles to the domain variables and thus to the instructions.

[1]It is assumed that jump destinations either are defined labels or follow subroutine calls.

```
schedule_block(Instructions, Decode):-
    clear_scoreboard(Scoreboard),
    collect(Instructions, Scoreboard, Decode, FPU, Data, Writeback),
    global_constraints(Decode, FPU, Data, Writeback),
    minimize(Decode, FPU, Data, Writeback).
```

Figure 2: The basic block scheduling predicate takes a list of instructions and returns a list of optimally ordered decode cycles for these instructions.

```
collect(['fadd.sss'(Rd, Rs1, Rs2)|Instructions],
        Scoreboard,
        [D|Decode],
        fpu([S1..E1|FP1], [S2..E2|Add2], [S3..E3|Add3], [S4..E4|Add4],
            Mul2, Mul3, Mul4, Mul5, [S5..E5|FPLast]),
        Data, [W|WriteBack]):-
    read(Rs1, D, Scoreboard),
    read(Rs2, D, Scoreboard),
    write(Rd, W, D, Scoreboard, NewScoreboard),
    S1#=D+1, S2#=E1+1, S3#=E2+1,                  %pipeline structure
    S4#=E3+1, S5#=E4+1, W#=E5+1,
    collect(Instructions, NewScoreboard, Decode,
            fpu(FP1, Add2, Add3, Add4, Mul2, Mul3, Mul4, Mul5, FPLast),
            Data, WriteBack).
```

Figure 3: The instruction description for a single-precision floating-point addition

3.1 Collecting the constraints

The constraints stated in the second part model the execution structure of the processor and the dependencies between the instructions.[2]

The constraint collector sequentially processes the instructions of the basic block, generates inequality constraints to enforce correct ordering and collects the domain variables for each pipeline stage into a list. These lists are then used in global constraints like "Only one instruction can be in this stage at a time".

The predicate `collect/6` (see figure 3) takes a list of instructions and the current state of the scoreboard and outputs the list of decode cycle variables for these instructions, a structure `fpu(FP1,... , FPLast)` containing lists of domain variables for every FPU pipeline stage, a similar structure for the data unit (`Data`), and the list of variables for the writeback stage (`WriteBack`). In this program the scoreboard is not represented by a bit for every register, but a pair of variables for every register r. One variable ($\texttt{LastWrite}_r$) represents the time when the scoreboard bit for the register was cleared by the completion of the last write. The other variable ($\texttt{NextCheckWrite}_r$) stands for the time when the scoreboard bit will be set by the next write. `clear_scoreboard/1`

[2] We assume 0 wait cycles for memory accesses.

initializes the scoreboard structure.

The `collect/6` predicates consists of instruction descriptions like the one in figure 3. The `read/3` predicate generates the constraints that force the instruction to be executed after the last write to the register and before the next write to the register, e.g.

$\texttt{LastWrite}_r$ `#=< D+1, D+1 #=<` $\texttt{NextCheckWrite}_r$

where `D` is the decode cycle of the instruction. Similarly, `write/5` generates

$\texttt{LastWrite}_r$ `#=< D+1, W #=<` $\texttt{NewNextCheckWrite}_r$

and unifies `D+1` and `W` with the corresponding variables in `Scoreboard` and `NewScoreboard`. These constraints are an implicit representation of the dependency graph used by existing algorithms.

`collect/6` also produces pipeline structure constraints that describe the relations of the stages (and the variables) within one instruction. E.g. `S2#=E1+1` means that the instruction enters the Add2 stage one cycle after it leaves FP1.

The other constraints are produced by the predicate `global_constraints/5`.

Only one instruction may be in a stage at a time. This fact is represented by an `alldifferent/1` constraint for each of the variable lists, that are collected for the pipeline stages. `alldifferent(L)` ensures that the variables in `L` get different values. For lists of start/end pairs an `alldisjoint/1` constraint is used. It ensures that two start/end pairs do not cover the same range.

Branches may only appear as last or second-to-last (delayed branch) instruction. This is enforced by a constraint of the form $\texttt{D}_{branch}$ `#>=` $\texttt{D}_i$`-1` for every instruction i in the basic block. This (and the `alldifferent/1` constraint on the decode cycles) ensures that only one instruction can be executed after the branch. Nondelayed branches are given a two cycle cost to make delayed branches preferable.

3.2 Searching for an optimal solution

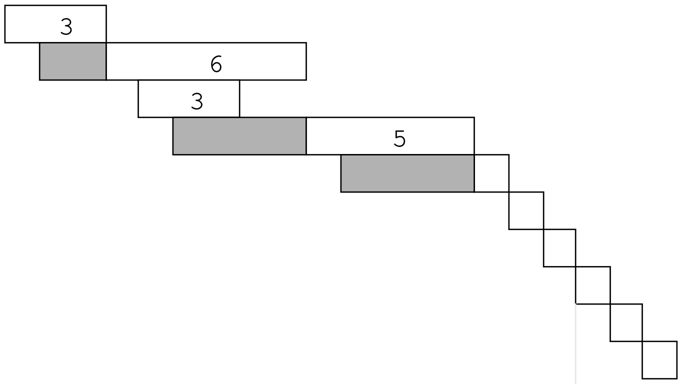
`minimize/4` searches for a solution that respects the constraints and minimizes the basic block execution time.

A solution can be found by instantiating all domain variables with values from their respective domains. Instantiating a variable may cause the execution of constraints, and therefore, failure. On backtracking, another value from the domain has to be chosen. This procedure is called labeling. For a more detailed discussion of the basic generator see [VH89]. The heuristic we used for choosing the next variable to be instantiated is: Choose the one with the smallest domain and the largest number of constraints. No specific heuristic is used to determine what value of the domain is chosen first.

In order to compute the optimal solution, the time when the last instruction leaves the decode stage is minimized. This is done by restricting all decode cycles to be less than the variable `MaxCycle` by the constraint $\texttt{D}_i$`#=<MaxCycle`. First, `MaxCycle` is instantiated with the number of decode cycles of the basic block, a lower bound. Then, a labeling is tried. If it fails, successively higher values are given to `MaxCycle`. The first solution found is optimal.

```
for (i=1; i<=n; i++) {
  dy[iy] += da*dx[ix];
  ix += incx;
  iy += incy;
}
```

```
    @L5:
1     ld       r12,r14[r5]
2     fmul.sss r12,r2,r12
3     ld       r11,r3[r6]
4     fadd.sss r12,r12,r11
5     st       r12,r3[r6]
6     addu     r5,r5,r4
7     addu     r9,r9,1
8     cmp      r10,r9,r8
9     bb1.n    le,r10,@L5
10    addu     r6,r6,r7
```




```
    @L5:
1     ld       r12,r14[r5]
3     ld       r11,r3[r6]
2     fmul.sss r12,r2,r12
6     addu     r5,r5,r4
7     addu     r9,r9,1
8     cmp      r10,r9,r8
4     fadd.sss r12,r12,r11
5     st       r12,r3[r6]
9     bb1.n    le,r10,@L5
10    addu     r6,r6,r7
```

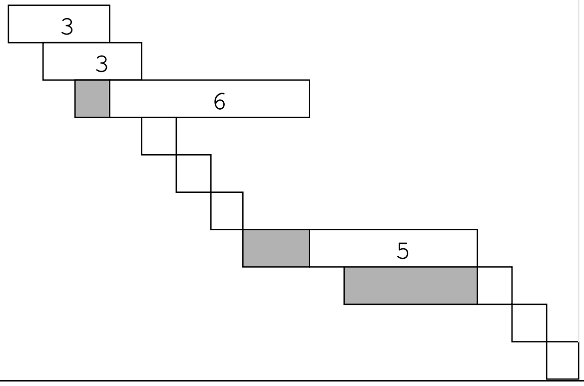


Figure 4: An ANSI C version of a Linpack loop, GNU C 1.37 output before and after scheduling (`bb1.n` is a delayed branch, `.sss` means single precision, the numbers in the boxes indicate instruction latencies)

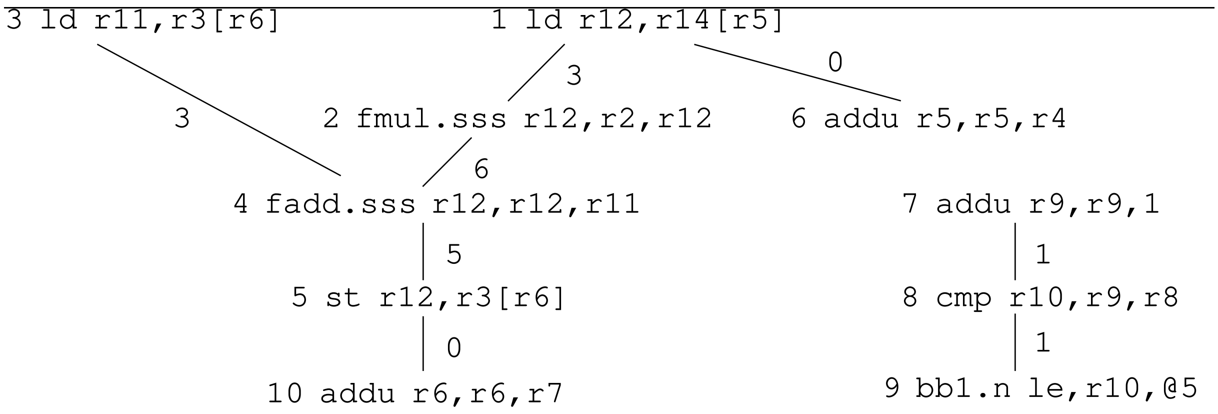


Figure 5: Dependency graph of the program in figure 4; Edge lengths > 0 are instruction latencies.

4 Example

We translated the Linpack loop given in [Mot90][3] into ANSI C and compiled it[4] (see figure 4). The `collect/6` predicate produces the following dependency constraints for this code. D_i represents the decode stage for instruction i, W_i represents the writeback cycle (redundant constraints are not shown).

D_2+1#>=W_1, D_4+1#>=W_2, D_4+1#>=W_3, D_5+1#>=W_4, D_6#>D_1,
D_8+1#>=W_7, D_9+1#>=W_8, D_{10}#>D_3, D_{10}#>D_5

This corresponds to the dependency graph in figure 5.

The pipeline structure constraints produced by `collect/6` for the first load instruction are:

D_1+1#=$S2_1$, $E2_1$+1#=$S1_1$, $E1_1$+1#=$S0_1$, $E0_1$+1#=W_1

where $S2_1$..$E2_1$, $S1_1$..$E1_1$ and $S0_1$..$E0_1$ represent the timespans when the instruction resides in the Data2, Data1 and Data0 stage respectively.

The global constraints produced look like this:

alldifferent([D_1,D_2,D_3,D_4,D_5,D_6,D_7,D_8,D_9,D_{10}]), %decode+1
alldifferent([W_1,W_2,W_3,W_4,W_5,W_6,W_7,W_8,W_{10}]), %writeback
alldisjoint([$S1_2$..$E1_2$,$S1_4$..$E1_4$]), %FPInit
... %other stages

Finally, the D_9#>=D_i-1 constraints for branch placement are produced.

The scheduler restricts the decode cycle of the last instruction by D_i#=<MaxCycle constraints.

At this point the constraints have reduced the domains of some variables: they have removed all values < 4 from the domain of D_2 (`fmul`),... and all values < 16 from D_{10} (`addu r6,r6,r7`) and therefore all values < 16 from `MaxCycle`, too.

The scheduler then instantiates `MaxCycle` to 16. By reducing their domains to a single value the constraints instantiate D_5 (`st`) and D_9 (`bb1`) to 15. Since this is incompatible with `alldifferent/1`, this attempt fails.

The next attempt instantiates `MaxCycle` to 17. The labeling then selects D_9 (`bb1`)

[3] The assembly language code given there is scheduled incorrectly.
[4] GNU C 1.37 (`gcc -ansi -O`)

Program	lines C	lines Assembly	measured speedup	scheduling time
example	19	84	1.17	1.6s
fft	101	288	1.17	7.9s
dhrystone	779	835	1.03	13.7s
WAM	2073	3481	1.06	69.1s
VAM	2647	4436	1.05	91.3s

Table 1: The test programs

for instantiation and tries to instantiate it with 15. This fails, because the `st` and `addu r6,r6,r7` would have to share cycle 16. Therefore `D`$_9$ is instantiated to 16; This causes the variables of the instructions on the critical path to be instantiated. Then the remaining instructions are scheduled without backtracking by labeling their variables.

Note that no integer instruction is scheduled for the third cycle. It would cause a collision with the writeback of the first `ld` and thereby would delay the execution of `fadd`. Conventional schedulers like the one in the Harris C compiler do not consider this.

5 Results

The instruction scheduler is written declaratively and shares all advantages of logic programming, among them ease and flexibility of programming and short development time, because constraint propagation and tree search programming are abstracted away from the programmer.

The schedules produced are optimal in the sense that there is no basic block that: contains the given instructions, respects the dependencies, and executes in shorter time, when all registers are initially available and there are no memory waits. Hennessy and Gross [HG83] report that their scheduling algorithm removes 85% of the removable stalls on the simpler MIPS processor.

Our main goal was to create a working example, so we did not try to make the program retargetable. However, developing the machine dependent parts for a machine like the Motorola 88100 takes about one person-day, so the retargetability of the scheduler is about as good as that of specialized tools.

We used the scheduler on a few programs: the Linpack loop of section 4, a fast fourier transformation routine, Dhrystone 1.1, and two Prolog abstract machine emulators (WAM and VAM [KN90]) running naïve reverse. All of these programs were compiled with `gcc-1.37 -O`, scheduled and run on a Data General AViiON 5000 (20 MHz 88100) with 16 MB RAM under DG/UX 4.32. Table 1 gives some information on these programs. The scheduling time includes I/O and instrumentation (mainly computing the old basic block length). We gathered some statistical data on these programs. Figure 6 shows the achieved speedup.

Since most basic blocks produced by GNU C are very short, many of them cannot be improved (81%). For the rest, speedups of up to 1.75 were achieved. The overall static

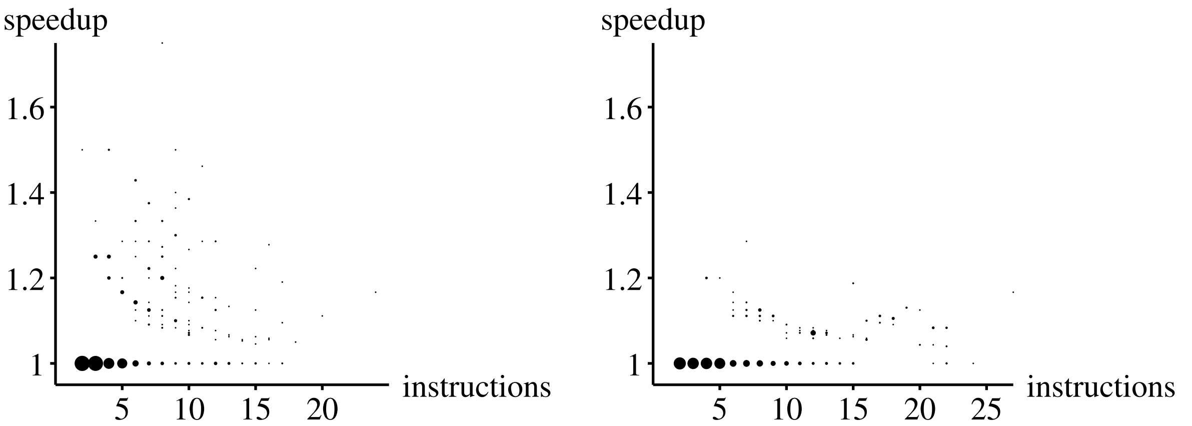


Figure 6: Speedups through optimal scheduling of code produced by GNU and Harris C compilers vs. basic block length. The area of the dots is proportional to the number of basic blocks.

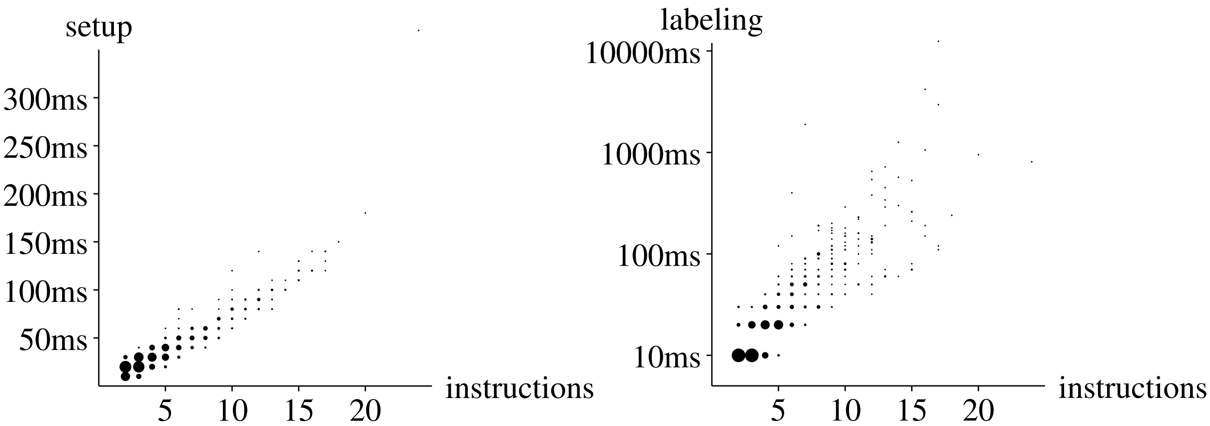


Figure 7: Timing behaviour of the scheduler

speedup (the ratio of the cumulated old and new basic block durations) is 1.04. A higher proportion of floating-point code would result in a higher speedup.

The Harris compiler includes an instruction scheduler and generally generates longer basic blocks. 8.5% of the basic blocks it produced can be improved by scheduling, the overall static speedup is 1.02. The main cause for suboptimal scheduling in the Harris compiler is writeback collisions.

The running time of the scheduler is acceptable, but should be improved. The current, untuned version takes about twice as long as compilation with `gcc`. Figure 7 shows the timing behaviour of the scheduler on the GNU C output. Constraint setup time and the number of variables and constraints are linear with the number of instructions, with floating-point code taking about twice as long as integer code. Labeling takes a short time for most basic blocks, but in some cases the NP-completeness of the problem results in exponential behaviour. This happens mainly in longer blocks with few dependencies. In seven cases, the scheduler ran into timeout before finding a solution. In such cases the basic block was divided and the pieces were scheduled. The figures 6 and 7 show statistics on the resulting basic blocks.

6 Further Work

Although long basic blocks were rare in the example programs, the scheduler should handle them in a better way, since it is counterproductive to divide basic blocks produced by an unrolling compiler. This could be achieved by using more restrictive constraints, which can be produced by a better analysis and by using a more problem-specific labeling procedure. Van Hentenryck has solved scheduling problems with 300 tasks [VH89].

Currently the scheduler works on just one basic block at a time. This local view can lead to avoidable pipeline stalls at basic block boundaries. Therefore the scheduler should also consider the adjacent blocks.

Our experience with compiler generated assembly language code has shown that the consideration of scheduling at earlier stages of the compilation process is essential for fast code. In general an optimizing compiler tries to minimize the usage of registers in a basic block. This increases the dependencies between instructions and prevents a good schedule. The declarative nature of the scheduler should make the integration of other parts of a compiler back end, e.g. register allocation, easy.

Acknowledgements

We wish to acknowledge the efforts of several others who contributed to the work in this paper. The anonymous referees, Manfred Brockhaus, Andreas Dieberger, Andreas Falkner, Thomas Graf, Max Hansch, Ulrich Neumerkel and Wolfgang Slany commented on earlier drafts; Paul Beusterien of Harris Corp. compiled our benchmarks with the Harris compiler; Martin Laubach supplied the fast fourier transformation routine.

References

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Architectural Support for Programming Languages and Operating Systems*, pages 122–131, 1991.

[CNO+88] Robert P. Colwell, Robert P. Nix, John J. O'Donnel, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):318–328, August 1988.

[DSVH90] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving large combinatorial problems in logic programming. *The Journal of Logic Programming*, (8):75–93, 1990.

[Ert90] M. Anton Ertl. Coroutining und Constraints in der Logik-Programmierung. Master's thesis, Technische Universität Wien, 1990.

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

[Gan89] Mahadevan Ganapathi. Prolog based retargetable code generation. *Computer Languages*, 14(3):193–204, 1989.

[GH88] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, 1988.

[GM86] Phillip B. Gibbons and Steve S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, 1986.

[HG83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[Hu61] T. C. Hu. Paralell sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[Kas90] Uwe Kastens. *Übersetzerbau.* R. Oldenbourg Verlag, München, 1990.

[KN90] Andreas Krall and Ulrich Neumerkel. The Vienna Abstract Machine. In P. Deransart and J. Małuzyński, editors, *Programming Language Implementation and Logic Programming (PLILP'90)*, pages 121–136. Springer LNCS 456, 1990.

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

[LKB91] Roland L. Lee, Alex Y. Kwok, and Fayé A. Briggs. The floating-point performance of a superscalar SPARC processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 28–37, 1991.

[Mot90] Motorola, Inc. *MC88100 RISC Microprocessor User's Manual*, second edition, 1990.

[Nud83] Bernard Nudel. Consistent labeling problems and their algorithms: Expected complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. The MIT Press, Cambridge, Massachusetts, 1989.

[VHD87] Pascal Van Hentenryck and Mehmet Dincbas. Forward checking in logic programming. In *Logic Programming: Proceedings of the Fourth International Conference*, pages 229–256, 1987.

[Wal72] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, 1972.