Instruction Scheduling for Complex Pipelines

M. Anton Ertl   Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{anton,andi}@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 {4459,4462}

Abstract. We designed heuristics for applying the list scheduling algorithm to processors with complex pipelines. On these processors the pipeline can stall due to resource contention (structural hazards) in addition to the usual data hazards. Conventional heuristics consider only data hazards. Our heuristics reduce structural hazards, too. Code with much instruction-level parallelism is optimized to avoid structural hazards, sequential code is scheduled for reducing data hazards. Embedded in a postpass strategy our scheduler removes 60%–100% of the removable stalls from conventionally scheduled code.

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 can be executed. E.g., on the Motorola MC88100 one floating point multiplication can be started at every cycle, but the result is only available after six cycles. If the next instruction needs the result (*data hazard*), it has to wait and the pipeline stalls. The problem of arranging the instructions in a way that reduces the number of wait cycles is known as instruction scheduling or instruction reordering.

Stalls can also occur when two instructions want to use the same pipeline stage at the same time (*structural hazard*). E.g., on the MC88100 only one result at a time can be written back to the register file, but up to three execution units may want to write a result. This plays an important role in instruction scheduling: The main cause for suboptimal scheduling in the Harris C compiler for the MC88100 is contention for the writeback stage. Structural hazards arise in the new superscalar processors, too, because some of their functional units are not fully replicated.

Although instruction scheduling is now standard in RISC compilers, only few attempts have been made to address structural hazards [BHE91]. Therefore we designed a scheduler that reduces the number of structural hazards. As an example we used the MC88100 RISC processor [Mot90].

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Compiler Construction (CC'92), Springer LNCS 641, 1992, pages 207--218

2 List scheduling

Even a simple formulation of optimal instruction scheduling is an NP-complete search problem [HG83]. A search for the optimal solution, as in [EK91], can take exponential time. Therefore most instruction schedulers try to find good, but possibly suboptimal schedules using heuristic algorithms. A short overview of the field is given in [Kas90, chapter 8.5]

FORTRAN

```
do i=1,n
  y(iy) = y(iy)+a*x(ix)
  ix = ix+incx
  iy = iy+incy
```

unscheduled 88000 code

```
@L5:
  ld       r12,r14[r5] ;r12=x(ix)
  fmul.sss r12,r2,r12  ;r12=a*r12
  ld       r11,r3[r6]  ;r11=y(iy)
  fadd.sss r12,r12,r11 ;r12=r12+r11
  st       r12,r3[r6]  ;y(iy)=r12
  addu     r5,r5,r4    ;ix=ix+incx
  addu     r9,r9,1     ;i=i+1
  cmp      r10,r9,r8   ;i<=n?
  bb1.n    le,r10,@L5  ;do
  addu     r6,r6,r7    ;iy=iy+incy
```

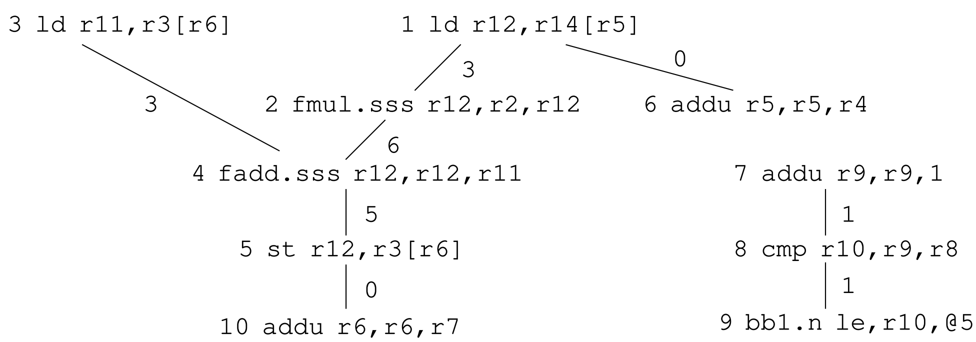

Fig. 1. SAXPY loop from the Linpack benchmark and its data dependence graph. Edge lengths $= 0$ indicate write-after-read dependences, edge lengths > 0 are instruction latencies.

The most common algorithm is list scheduling [LDSM80, GM86, War90, SKAH91]. It builds a data dependence graph for each basic block. Figure 1 shows an example graph. An edge from instruction a to instruction b indicates that a must be executed before b to preserve the correctness of the overall program. Dependence edges exist between reads and writes, writes and reads and between writes to the same register or memory location[1].

[1] The algorithm in [HG83] relaxes this restriction to allow swapping live ranges of the same register.

The data dependence graph is essentially the expression evaluation graph (drawn up-side-down), with some edges added due to dependencies between memory accesses and with write-after-read edges added due to allocations of values to the same register.

```
list_scheduling(graph, select)
  schedule←empty
  while graph ≠ empty
    leaders←{nodes in graph without parents}
    next_inst←select(leaders)
    append next_inst to schedule
    remove next_inst from graph
```

Fig. 2. The list scheduling algorithm

After building the dependence graph the algorithm selects one of the leaders (instructions without predecessors) and removes it from the graph. This step is repeated until the graph is empty. The order in which the instructions are removed is the new instruction order of the basic block (see figure 2).

The selection function determines the quality of the schedule. A typical selection function uses:

smallest earliest execution time (EET) The EET of an instruction is the cycle when the instruction can start executing, because it is no longer delayed by any hazards. Nothing can be gained by choosing an instruction with a higher EET, because (in the absence of structural hazards) a leading instruction cannot be delayed by instructions that are executed before its EET has arrived. Ties are broken by

maximum path length The path length is the sum of the latencies along the longest path to the end of the basic block[2]. This heuristic exposes delay slots early, while there are other instructions to fill them.

```
ep(leaders)
  ready_leaders←{leaders with minimal EET}
  return one of the ready_leaders with maximal path length
```

Fig. 3. The selection function *ep*

We call this selection function *ep* (see Figure 3). As an example, consider again the SAXPY loop of figure 1. In the beginning the graph has three leaders, the instructions 1 (`ld`), 3 (`ld`) and 7 (`addu`), all with EET=0. Instruction 1 is selected,

[2] The path length is defined differently in [SKAH91]

because it has the largest path length to the end (15 cycles). After its removal from the graph the instructions 3, 7, 2 (`fmul`) and 6 (`addu`) are the leaders. The `fmul`'s EET=3, because it depends upon the result of instruction 1, the other instructions' EET=1. The final schedule is shown in figure 4[3].

```
@L5:
  ld       r12,r14[r5] ;r12=x(ix)
  ld       r11,r3[r6]  ;r11=y(iy)
  addu     r9,r9,1     ;i=i+1
  fmul.sss r12,r2,r12  ;r12=a*r12
  cmp      r10,r9,r8   ;i<=n?
  addu     r5,r5,r4    ;ix=ix+incx
  fadd.sss r12,r12,r11 ;r12=r12+r11
  st       r12,r3[r6]  ;y(iy)=r12
  bb1.n    le,r10,@L5  ;do
  addu     r6,r6,r7    ;iy=iy+incy
```

Fig. 4. SAXPY loop scheduled using *ep*

3 Complex pipelines

The common heuristics used in schedulers are designed for simple, straight pipelines, where the only bottleneck is the entry into the pipeline. In real processors like the Motorola 88100 (see figure 5) additional problems occur, which must be considered by the scheduler.

Structural hazards, i.e. the situation when several instructions want to use the same pipeline stage, can arise in two ways:

- The stage has several inputs (subpipe merge). If two instructions arrive at a pipeline merge point at the same time, one of them has to wait. An example is the writeback stage in the 88100, which is granted using a hardware priority scheme. Another example are not fully replicated functional units in superscalar processors.
- One instruction is already in the stage when another instruction arrives. This happens when an instruction uses a stage for more than one cycle or when an instruction stalls in a stage. An example is the `Add2` stage in the 88100, which is used for several cycles by the divide instructions.

Structural hazards are becoming more wide-spread with the advent of superscalar processors: The 88110 [DA92] (a dual-issue processor) has only two writeback

[3] The EET is computed without considering structural hazards. This results in a better schedule in this case.

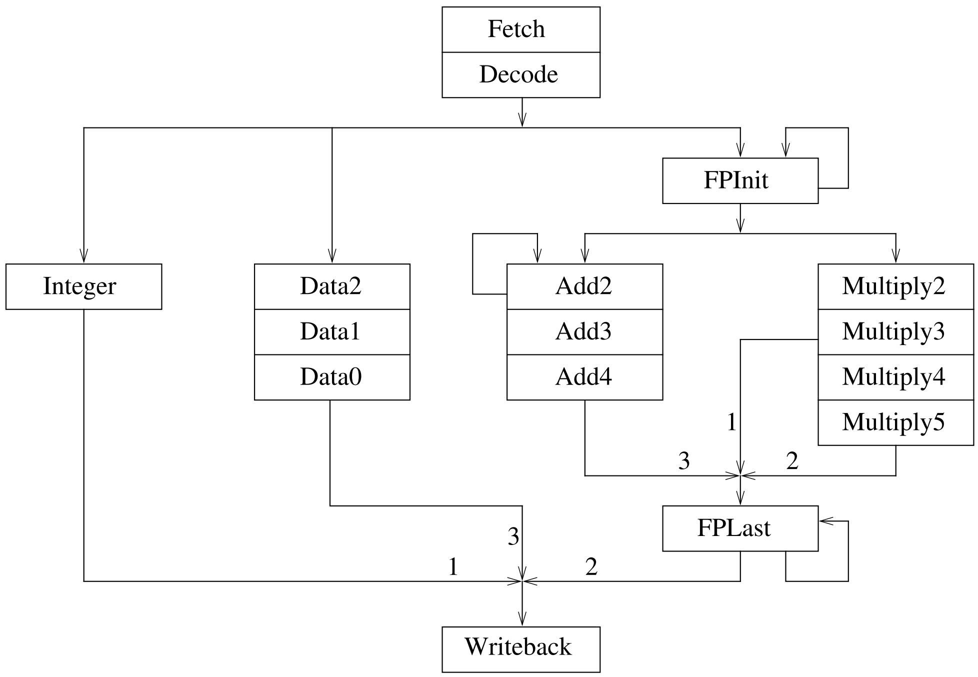

Fig. 5. Pipeline stages in the Motorola 88100. Numbers indicate priorities.

busses, but up to six values to write back. Most of its functional units are not replicated, resulting in instruction class conflicts. Similarly, the SuperSPARC [BK92] (also known as Viking, triple-issue) has only two integer writeback slots.

As an example for structural hazards, let's take another look at the SAXPY loop, as scheduled using *ep* (see figure 4). The `fmul` is scheduled to use the result of the first `ld` as soon as it is available. Unfortunately the writeback of the `ld` collides with the writeback of the `addu`. The writeback slot is granted to the `addu` and the `fmul` (and the rest of the loop) stalls one cycle to wait for the result of the `ld`.

The Marion System [BHE91] deals with structural hazards by using resource vectors. The method used in microcode compaction is similar [LDSM80]. If an instruction conflicts with the already scheduled instructions, its EET is increased. This scheme works satisfactorily when conflicts are rare. But if the stage the instructions compete for is a bottleneck, a more aggressive heuristic is needed, that keeps the stage busy. Every cycle lost in a bottleneck stage ends up as a stall cycle.

4 A Better Selection Function

The argument supporting the selection function *ep* does no longer hold, if structural hazards are considered. An instruction, that executes, before the EET of an other instruction has arrived, can delay the other instruction through structural hazards.

4.1 Switching Between Heuristics

In mainly sequential code data hazards are likely to occur. Structural hazards are easy to avoid, because the pipeline is often idle. On the other hand, in code with much instruction-level parallelism the execution speed is limited by pipeline contention. Structural hazards are common. Data hazards are easy to avoid, because there are many independent instructions that can be scheduled into delay slots.

```
seq_par(leaders)
  if parallelism<threshold
    return seq(leaders)
  else
    return par(leaders)
```

Fig. 6. The main selection function *seq_par*

Therefore our selection function *seq_par* switches between a selection function for sequential code and one for parallel code. Both selection functions are described below. If there are no structural hazards, they produce the same results as *ep*. The choice is based on the parallelism of the basic block. We define the parallelism as:

$$parallelism = \frac{cycles\ needed\ by\ the\ most\text{-}used\ stage}{critical\ path\ length}$$

This ratio is maintained as instructions are scheduled, thereby adjusting the strategy to the situation. We empirically determined the best switching threshold to be 1.1 (on the 88100 using the selection functions described below). The switching function is shown in figure 6.

The initial parallelism of the SAXPY loop is 0.625. During the scheduling *seq* is used most of the time, only at the end the parallelism reaches 2 (when instructions 9 and 10 are leaders).

A related idea is used in Integrated Prepass Scheduling [GH88, BEH91], which tries to reconcile instruction scheduling with register allocation. It switches between scheduling for pipelining and scheduling for register allocation based on the number of used and available registers.

4.2 The Sequential Heuristic Seq

In sequential code the instruction with the longest path length must be executed as soon as possible. On a machine with structural hazards scheduling an early instruction can delay a later instruction. Since we want to execute the instruction with the longest path length as soon as possible, we select only this instruction or an instruction that does not delay its execution. Among those we choose with a secondary selection function. *Seq* is shown in figure 7.

We tested several reasonable secondary selection functions and found that they did not make any difference. In the measurements we *seq* itself as secondary function.

```
seq(leaders)
  critical_leaders←{leaders with maximal path length}
  ready_critical← one of critical_leaders with minimal EET
  earlier←{leaders that do not increase the EET of ready_critical when one of
   them is scheduled before ready_critical}
  if earlier= ∅
    return ready_critical
  else
    return secondary(earlier)
```

Fig. 7. The sequential selection function *seq*

Let's see how *seq* handles the SAXPY loop. After scheduling the first `ld` the `fmul` becomes the ready_critical instruction, because it has maximal path length. The second `ld` still fits in front of it, but the `addu` does not—it would delay the `fmul` as we have already seen. So the `fmul` is scheduled immediately. The `addu` easily fits in one of the later delay slots. The resulting schedule is optimal (see figure 8).

```
@L5:
  ld       r12,r14[r5] ;r12=x(ix)
  ld       r11,r3[r6]  ;r11=y(iy)
  fmul.sss r12,r2,r12  ;r12=a*r12
  addu     r9,r9,1     ;i=i+1
  cmp      r10,r9,r8   ;i<=n?
  addu     r5,r5,r4    ;ix=ix+incx
  fadd.sss r12,r12,r11 ;r12=r12+r11
  st       r12,r3[r6]  ;y(iy)=r12
  bb1.n    le,r10,@L5  ;do
  addu     r6,r6,r7    ;iy=iy+incy
```

Fig. 8. SAXPY loop scheduled using *seq* or *seq_par*

4.3 The Parallel Heuristic Par

In parallel code the scheduler has two goals: to avoid structural hazards and to keep the bottleneck stages busy. Therefore it should select instructions that use the bottleneck stages and do not cause structural hazards.

While this is a good solution for problems like nonpipelined functional units and nonreplicated functional units in superscalar processors, it is not the whole story. Depending on the pipeline structure, such a heuristic can lead to the suppression of some instruction classes, causing unbalanced and bad scheduling. Applied to the 88100 writeback problem, it suppresses non-integer instructions, because they do not use the writeback stage as early as integer instructions.

Therefore our selection function delays non-integer instructions only if it knows that non-integer instructions will be preferred soon. I.e., if an integer instruction in the next cycle after the current one would cause a writeback collision.

The parallel selection function uses the EET as the primary criterion, the instruction class heuristic described above as the second, and path length as the least significant criterion. The EET is first, because in parallel code we cannot afford to miss a cycle. Path length is last, because it is not so important in parallel code. The switching scheme protects from the problems that this selection function may exhibit in sequential code.

```
par(leaders)
  ready_leaders←{leaders with minimal EET}
  non_conflicting←{ready_leaders that do not conflict with the scheduled
   instructions}
  if non_conflicting= ∅ non_conflicting←ready_leaders
  if an instruction that uses the bottleneck stage early causes a collision when
   executed in the next cycle
    early_users←{non_conflicting that use the bottleneck stage early}
    if early_users= ∅ early_users←non_conflicting
  else
    early_users←non_conflicting
  return one of the early_users with maximal path length
```

Fig. 9. The parallel selection function *par*

Par is shown in figure 9. As we present it, this heuristic is quite specific for the writeback problem, but it can be adapted to other situations easily. Just change the instruction class heuristic appropriately.

How well does *par* do on the (sequential) SAXPY loop? The first instruction chosen is again the first `ld`. The instruction class heuristic then selects the `addu` and saves the second `ld` for the third cycle to avoid *ep*'s writeback collision. Then the scheduler is faced with integer instructions and the `fmul`. Since the instruction class heuristic has higher precedence than path length, an integer instruction is chosen (to save the `fmul` for the next cycle, when an integer instruction would cause a writeback collision with the second `ld`). *Par*'s schedule (see figure 10) is as bad as *ep*'s, which demonstrates the need to switch to *seq* when scheduling sequential code.

5 Results

We implemented these ideas in a prototype scheduler for the Motorola 88100. For simplicity it schedules the assembly language output of compilers (also known as postpass strategy) and it does not perform alias analysis, i.e. all memory references are treated as possibly equal.

To determine the efficiency of our heuristics we scheduled four benchmarks and compared the results with the optimal schedules produced by the scheduler

```
@L5:
  ld       r12,r14[r5] ;r12=x(ix)
  addu     r9,r9,1     ;i=i+1
  ld       r11,r3[r6]  ;r11=y(iy)
  cmp      r10,r9,r8   ;i<=n?
  fmul.sss r12,r2,r12  ;r12=a*r12
  addu     r5,r5,r4    ;ix=ix+incx
  fadd.sss r12,r12,r11 ;r12=r12+r11
  st       r12,r3[r6]  ;y(iy)=r12
  bb1.n    le,r10,@L5  ;do
  addu     r6,r6,r7    ;iy=iy+incy
```

Fig. 10. SAXPY loop scheduled using *par*

in [EK91]. The benchmark programs are: an abstract Prolog machine interpreter (VAM), the dhrystone synthetic integer benchmark (dhry), a fast Fourier transformation routine (fft), and the SAXPY loop with a bit of scaffolding (SAXPY). These benchmarks were compiled with the Harris C compiler, which already includes an instruction scheduler and therefore provides a good baseline for comparison.

We tested the following heuristics: EET, break ties with path length (*ep*); our sequential selection function (*seq*), our parallel selection function (*par*); and switching between *seq* and *par* (*seq_par*). *opt* denotes the optimal schedule and *orig* the original Harris-scheduled code.

Prog.	*orig*	*ep*	*seq*	*par*	*seq_par*	*opt*	global%
SAXPY	44/0%	45/-50%	42/100%	43/ 50%	42/100%	42/100%	4.8%
fft	641/0%	641/ 0%	641/ 0%	640/100%	640/100%	640/100%	0.2%
dhry	519/0%	520/-13%	514/ 63%	518/ 13%	514/ 63%	511/100%	1.0%
VAM	8335/0%	8328/ 4%	8273/ 39%	8283/ 33%	8239/ 60%	8174/100%	1.2%

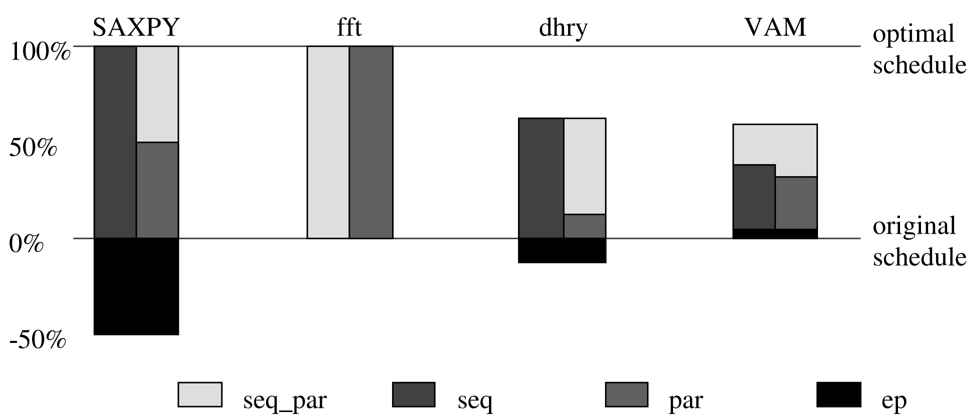

Fig. 11. Static execution cycles/improvement on *orig* as percentage of possible improvement

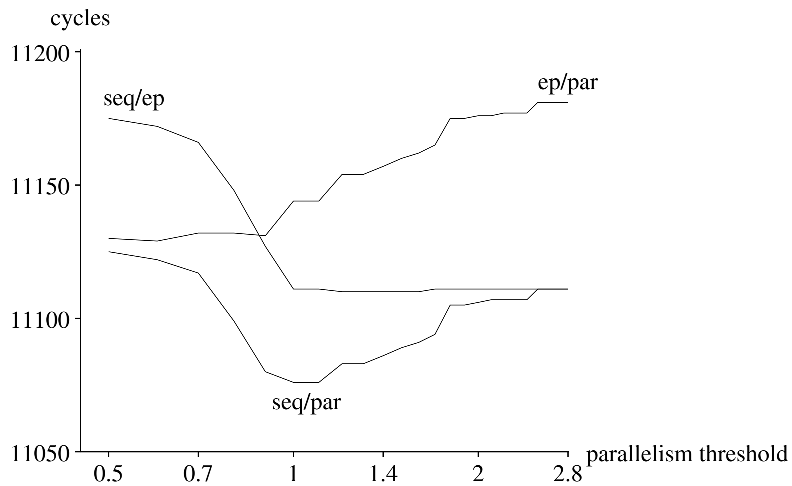

Fig. 12. Performance at varying *parallelism* thresholds

Figure 11 shows static measurements of the benchmarks (cumulated execution times of all basic blocks) in tabular and graphic form. The last column shows the improvement of seq_par relative to the total (static) cycles. Figure 12 shows the behaviour of various switching combinations of *seq*, *par* and *ep* at various switching thresholds.

Both *seq* and *par* are nearly always as good as or better than *ep*, even at their weak points (parallel and sequential code respectively). Since a postpass strategy can exploit less parallelism, we expect *par* to perform even better in a prepass strategy. Both of them are better than the other at their respective strong points, so *seq_par* combines them nicely to remove between 60% and 100% of the avoidable stalls in the original code. This corresponds to an absolute gain of up to 5%. Execution time measurement of a few programs also show an improvement of 1%–5%.

The measurements of the scheduling time have to be taken with a grain of salt, since we did not build the scheduler for efficiency. Replacing *ep* with *seq_par* increases scheduling time by 6%. Since scheduling takes only a small amount of compile time, we consider *seq_par* to be cost-effective.

6 Further work

The ideas presented here should be integrated into a code generation system like Marion. The most difficult problem is finding a nice way to generate or specify the instruction class heuristic of *par* in a general way.

If some instructions (like divide instructions) occupy a stage for a long time (for a typical basic block length or longer), it may be useful to preschedule them by adding edges to the dependence graph between these instructions and others using the same stage.

Currently the parallelism-dependent behaviour is controlled by parameters for the whole remaining basic block. In long basic blocks the local parallelism often differs from the value *parallelism*. Therefore parameters that reflect the local situation are useful.

The parallel/sequential distinction can be used for integrating register allocation and instruction scheduling: In parallel code it is cheap to schedule for register allocation, whereas in sequential code spilling registers is cheap: The spill code fills delay slots.

7 Conclusion

We have described heuristics for dealing with structural hazards in a list scheduling framework. The heuristic selection function *seq* is designed for scheduling code with low instruction-level parallelism. It gives absolute priority to the instruction with the longest path length to the end of the basic block. Other instructions are scheduled earlier only if they do not interfere. The heuristic *par* is designed for scheduling parallel code. It uses an instruction class selection heuristic (as secondary heuristic) to avoid structural hazards. Our scheduler switches between these two heuristics for each new instruction to schedule depending on the parallelism of the remaining basic block. This technique removes 60%–100% of the removable stalls or up to 5% of the cycles from already conventionally scheduled code.

Acknowledgements

We wish to acknowledge the efforts of several others who contributed to the work in this paper. The anonymous referees, Paul Beusterien, Manfred Brockhaus, Andreas Falkner, Horst Hogenkamp, Christoph Keßler and Ulrich Neumerkel commented on earlier versions; Paul Beusterien of Harris 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 (ASPLOS)*, pages 122–131, 1991.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240, Toronto, 1991.

[BK92] Greg Blanck and Steve Krueger. The SuperSPARC microprocessor. In *COMPCON: Digest of Papers*, pages 136–141, 1992.

[DA92] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[EK91] M. Anton Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming (PLILP)*, pages 75–86, Passau, 1991. Springer LNCS 528.

[GH88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pages 442–452, 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.

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

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

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

[SKAH91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *MICRO-24*, 24^{th} *Annual Intl. Symp. on Microarchitecture*, pages 93–102, 1991.

[War90] Henry S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, January 1990.

This article was processed using the LaTeX macro package with LLNCS style