Load Scheduling with Profile Information*

Götz Lindenmaier[1], Kathryn S. McKinley[2], and Olivier Temam[3]

[1] Fakultät für Informatik, Universität Karlsruhe
[2] Department of Computer Science, University of Massachusetts
[3] Laboratoire de recherche en informatique, Universite de Paris Sud

Abstract. Within the past five years, many manufactures have added hardware performance counters to their microprocessors to generate profile data cheaply. We show how to use Compaq's DCPI tool to determine load latencies which are at a fine, instruction granularity and use them as fodder for improving instruction scheduling. We validate our heuristic for using DCPI latency data to classify loads as hits and misses against simulation numbers. We map our classification into the Multiflow compiler's intermediate representation, and use a locality sensitive Balanced scheduling algorithm. Our experiments illustrate that our algorithm improves run times by 1% on average, but up to 10% on a Compaq Alpha.

1 Introduction

This paper explores how to use hardware performance counters to produce fine grain latency information to improve compiler scheduling. We use this information to hide latencies with any available instruction level parallelism (ILP). (ILP for an instruction is the number of other instructions available to hide its latency, and the ILP of a block or program is the average of the ILP of its instructions). We use DCPI, the performance counters on the Alpha, and Compaq's `dcpicalc` tool for translating DCPI's statistics into a usable form. DCPI provides a very low cost way to collect profiling information, especially as compared with simulation, but it is not as accurate. For instance, `dcpicalc` often cannot produce the reason for a load stall. We show nevertheless it is possible to attain fine grain latency information from performance counters. We use a heuristic to classify loads as hits and misses, and this classification matches simulation numbers well. We are the first to use performance counters at such a fine granularity to improve optimization decisions.

In the following, Section 2 presents related work. Section 3 describes DCPI, the information it provides, how we can use it, and how our heuristic compares with simulation numbers. Section 4 briefly describes the load sensitive scheduling algorithm we use, and how we map the information back to the IR of the compiler. Section 5 presents the results of our experiments on Alpha 21064 and 21164 that show execution time improvements are possible, but our average improvements are less than 1%. Our approach is promising, but it needs new scheduling algorithms that take into account variable latencies and issue width to be fully realized.

* This work is supported by EU Project 28198; NSF grants EIA-9726401, CDA-9502639, and a CAREER Award CCR-9624209; Darpa grant 5-21425; Compaq and by LTR Esprit project 24942 MHAOTEU. Any opinions, findings, or conclusions expressed are the authors' and not necessarily the sponsors'.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 223–233, 2000.

2 Related Work

The related work for this paper falls into three categories: performance counters and their use, latency tolerance, and scheduling. Our contribution is to show how to use performance counters at a fine granularity, rather than aggregate information, and how to tolerate latency by improving scheduling decisions.

We use the hardware performance counters and monitoring on a 4-way issue Alpha [3, 5]. Similar hardware now exists on the Itanium, Intel PentiumPro, Sun Sparc, SGI R10K and in Shrimp, a shared-memory parallel machine [3, 9]. The main advantages of using performance counters instead of software simulation or profiling are time and automation. Performance counters yield information at a cost of approximately 1-2% of execution time, and do not require users to compile with and without profiling. Continuous profiling enables recompilation after program execution to be completely hidden from the user with later, free cycles (our system does not automate this feature). Previous work using performance counters as a source of profile information have used aggregate information, such as the miss rate of a subroutine or basic block [1] and critical path profiles to sharpen constant propagation [2]. Our work is unique in that it uses information at the instruction level, and integrates it into a scheduler.

Previous work on using instruction level parallelism (ILP) to hide latencies for non-blocking caches has two major differences from this work [4, 6, 8, 10, 12]. First, previous work uses static locality analysis which works very well for regular array accesses. Secondly, these schedulers only differentiates between a hit or a miss. Since we use performance counters, we can improve the schedules of pointer based codes that compilers have difficulty analyzing. In addition, we obtain and use variable latencies which further differentiates misses and enables us to concentrate ILP on the misses with the longest observed average latencies.

3 DCPI

This section describes DCPI, `dcpicalc` (a Compaq tool that translates DCPI output to a useful form), and compares DCPI results to simulation. DCPI is a runtime monitoring tool that cheaply collects information by sampling hardware counters [3]. It saves the collected data efficiently in a database, and runs continuously with the operating system. Since DCPI uses sampling, it delivers profile information for the most frequently executed instructions, which are, of course, the most interesting with respect to optimization.

3.1 Information Supplied by DCPI

During monitoring, the DCPI hardware counter tracks the occurrence of a specified event. When the counter overflows, it triggers an interrupt. The interrupt handler saves the program counter for the instruction at the head of the issue queue. `Dcpicalc` interprets the sampled data off-line to provide detailed information about, for example, how often, long, and why instructions stall. If DCPI samples an instruction often, the instruction spends a lot of time at the head of the issue queue, which means that it

suffers long or frequent stalls. `Dcpicalc` combines a static analysis of the binary with the dynamic DCPI information to determine the reason(s) for some stalls. It assumes all possible reasons for stalls it cannot analyze.

```
     instruction          static  dynamic
                          stalls  stalls

1  ldl   r24, -32720(gp)   1      1.0cy
2  lda   r2,  -32592(gp)   0
        i
3  ldl   r26, -32668(gp)   1      1.5cy
4  lda   gp,  0(sp)        0
     b
     b
       d
       d
       d
5  cmplt zero, r26, r26    2      3.5cy
6  ldl   r25, -32676(gp)   0
     b
     b
       i    ...20.0cy
       i
7  cmplt r26, r25, r25     2     20.0cy
     b
8  bis   r24, r25, r25     1      1.0cy
     a
9  beq   r25, 0x800f00     1      1.0cy
```

Fig. 1. Example for the calculation of locality data.

The reasons for a stall are `a`, `b`, `i`, and `d`; `a` and `b` indicate stalls due to an unresolved data dependence on the first or second operand respectively; `i` indicates an instruction cache miss; and `d` indicates a data cache miss. Figure 1 shows an example basic block from `compress`, a SPEC'95 benchmark executed on an Alpha 21164 annotated by `dcpicalc`. Five instructions stall; each line without an instruction indicates a half cycle stall before the next instruction can issue.[1] `Dcpicalc` determines reasons and lengths of `a` and `b` from the static known machine implementation, and `i` and `d` from the dynamic information. For example, instruction 3 stalls one cycle because it waits for the integer pipeline to become available and on average an additional half cycle due to an instruction cache miss. The average stall is very short, which also implies it stalls infrequently. Instruction 7 stalls due to an instruction cache miss, on average 20 cycles.

3.2 Deriving Locality Information

This section shows how to translate the average load latencies from `dcpicalc` into hits and misses for use in our scheduler. We derive the following six values about loads from the `dcpicalc` output. Some are determined (d)ynamically, others are based on (s)tatic features of the program.

– *MissMarked* (d): `dcpicalc` detects a cache miss for this load, i.e., the instruction that uses this load stalls and is marked with `d`.

[1] Because more than two instructions rarely issue in parallel, the output format ignores this case.

- *Stall* (d): the length of a *MissMarked* stall.
- *StatDist* (s): The distance in static cycles between the load and the depending instruction.
- *DynDist* (d): The distance in dynamic cycles between the load and the depending instruction.
- *TwoLoads* (s): The instruction using the data produced by a load marked with *TwoLoads* depends on two loads and it is not clear which one caused the stall.
- *OtherDynStalls* (d): The number of other dynamic stalls between the load and the depending instruction.

For example, instruction 1 in Figure 1 has *MissMarked* = *false*, *Stall* = 0, *StatDist* = 6, *DynDist* = 26.0, *TwoLoads* = *false*, and *OtherDynStalls* = 3. Using these numbers, we reason about the probability of a load hitting or missing, and its average dynamic latency as follows. If a load is *MissMarked*, it obviously misses in the cache on some executions. But *MissMarked* gives no information about how often it misses. *Stall* is long if either the cache misses of this load are long, or if they are frequent. Thus, if a load is *MissMarked* and *Stall* and *StatDist* are large, the probability of a miss is high.

Even when a load misses, it may not stall (*MissMarked* = *false*, *Stall* = *0*) because static or dynamic events may hide its latency. If *StatDist* is larger than the latency of the cache, it will not stall. If *StatDist* does not hide a cache latency, a dynamic stall could, in which case *DynDist* or *OtherDynStalls* are high. The DCPI information is easier to evaluate if *StatDist* is small and thus dynamic latencies are exposed, i.e., the loads are scheduled right before a dependent instruction. We generate the initial binaries assuming a load latency of 1, to expose stalls by cache misses.

The Balanced scheduler differentiates *hits* and *misses*, and tries to put available ILP after *misses* to hide a given fixed miss latency. Although we have the actual expected, dynamic latency, the scheduler we modified cannot use it. Since the scheduler assumes misses by default, we classify a load as a *hit* as follows:

$$\neg MissMarked \ \wedge \ (StatDist < 10) \ \wedge \ (Stall = 0 \ \vee \ DynDist < 20)$$

and call it the *strict* heuristic because it conservatively classifies a load as a *hit* only if it did not cause a stall due to a cache miss, and for which it is unlikely that the latency of a cache miss is hidden by the current schedule. It also classifies infrequently missing loads as *misses*. We examined several other heuristics, but none perform as well as the *strict* heuristic. For example, we call the following the *generous* heuristic:

$$(StatDist < 5 \ \wedge \ Stall < 10).$$

As we show below, it correctly classifies more loads as *hits* than the strict heuristic, but it also missclassifies many missing loads as *hits*.

3.3 Validation of the Locality Information

We validated the heuristics by comparing their performance to that of a simulation using a version of ATOM [13] that we modified to compute precise hit rates in the three cache levels of the Alpha 21164. The 21164 has a split first level instruction and data cache. The data cache is a 8 KB direct mapped cache, a unified 96 KB three-way associative

second level on chip cache, and a 4MB third level off chip cache. The latencies of the 21164's first and second cache are 2 and 8 cycles, respectively. (The first level data cache of the 21064 which we use later in the paper has a latency of 3 cycles and is also 8 KB, and the second level cache has a latency of at least 10 cycles.)

Figure 2 summarizes all analyzed loads in eleven SPEC'95[2] and the Livermore benchmarks. The first chart in Figure 2 gives the raw number of loads that hit in the first level cache according to the simulator as a function of how often they hit; each bar represents the number of loads that hit *x%* to *x+10%* in the first level cache. We further divide the 90-100% column into two columns: 90-95% and 95-100% in both figures. Clearly, most loads hit 95-100% of the time.

The second chart in Figure 2 compares how well our heuristics find hits as compared to the simulator. The x-axis is the same as Figure 2. Each bar is the fraction of these loads that the heuristics actually classifies as a hit. Ideally, the heuristics would classify as *hits* all of the loads that hit more than 80%, and none that hit less than 50% of the time. However, since the conservative assumption for our scheduler is miss, we need a heuristic that does not classify loads that mostly miss as *hits*. The generous heuristic finds too many *hits* in loads that usually miss. The strict heuristic instead errs in the conservative direction: it classifies as *hits* only about 40% of the loads that in simulation hit 95% of the time, but it is mistaken less than 5% of the time for those loads that hit less than 50% of the time. In absolute terms these loads are less than 1% of all loads.

4 Scheduling with Runtime Data

In this section, we show how to drive load sensitive scheduling with runtime data. Scheduling can hide the latency of a missing load by placing other useful, independent operations in its delay slots (*behind* it) in the schedule. In most programs, there is not enough ILP to assume all loads are misses in the cache and with the issue width of current processors increasing this problem is exacerbated. With locality information, the scheduler can instead concentrate available ILP behind the missing loads. The ideal

[2] applu, apsi, fpppp, hydro2d, mgrid, su2cor, swim, tomcatv, turb3d, wave5, and compress95.

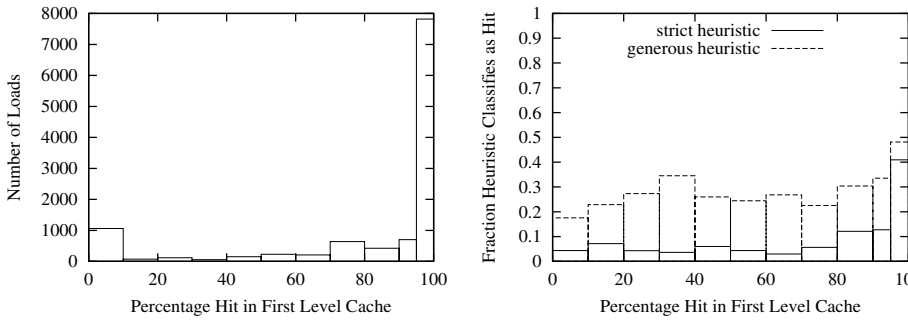

Fig. 2. Simulated number of loads and comparison of heuristics to simulation.

scheduler could differentiate between the expected latency of a miss, placing the most ILP behind the misses with the longest latencies.

4.1 Balanced Scheduling

We use the Multiflow compiler [7, 11] with the Balanced Scheduling algorithm [8, 10], and additional optimizations, e.g., unrolling, to generate ILP and *traces* of instructions that combine basic blocks. Below we first briefly describe Balanced scheduling and then we present our modifications.

Balanced scheduling first creates an acyclic scheduling data dependency graph (DAG) which represents the dependences between instructions. By default it assumes all loads are misses. It then assigns each node (instruction) a weight which is a function of the static latency of the instruction and the available ILP (i.e., how many other instructions may issue in parallel with it).[3] For each instruction `i`, the scheduler finds all others that `i` may come after in the schedule; `i` is thus available as ILP to these other instructions. The scheduler then increases the weight of each instruction after which `i` can execute and hide latency. The usual list scheduling algorithm which tries to cover all the weights then uses this new DAG [8], where the weights reflect a combination of the latency of the instruction and the number of instructions available to schedule with it.

Furthermore, the Balanced scheduler deals with variable load latencies as follows. It makes two passes. The first pass assigns ILP to hide the static latency of all non-load instructions. (For example, the latency of the floating point multiply is known statically and occurs on every execution.) If an instruction has sufficient weight to cover its static latency, the scheduler does not give it any additional weight. In a second pass, the scheduler considers the loads, assigning them any remaining ILP. This structure guarantees that the scheduler first spreads ILP weight to instructions with known static latencies that occur every time the instruction executes. It then distributes ILP weight equally to load instructions which might have additional dynamic latencies due to cache misses. The scheduler thus balances ILP weight across all loads, treating loads uniformly based on the assumption that they all have the same probability of missing in the cache.

4.2 Balanced Scheduling with Locality Data

The Balanced scheduler can further distinguish loads as *hits* or a *misses*, and distribute ILP only to missing loads. The scheduler gives ILP weight only to *misses* after covering all static cycles of non-loads. If ILP is available, the *misses* will receive more weight than before because without the *hits*, there are fewer candidates to receive ILP weight. Ideally, each miss could be assigned weight based on its average expected dynamic latency, but to effect this change would require a completely new implementation.

[3] The weight of the instruction is *not* a latency.

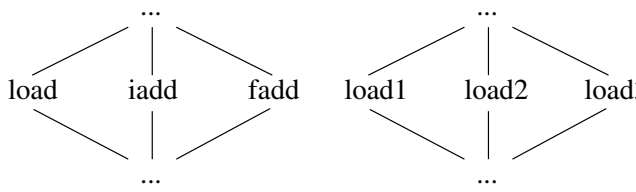

Fig. 3. Example: Balanced scheduling for multi issue processors.

4.3 Communicating Locality Classifications to the Scheduler

In this section, we describe how to translate our classification of *hits* and *misses* which are relative to the assembler code into the Multiflow's higher-level intermediate representation (IR). Using the internal representation before the scheduling pass, we add a unique tag to each load. After scheduling, when the compiler writes out the assembly code, it also writes the tags and load line number to a file. The locality analysis integrates these tags with the runtime information. When we recompile the program, the compiler uses the tags to map the locality information to the internal representation.

The locality analysis compares the Multiflow assembler and the executed assembler to find corresponding basic blocks. The assembler code output by the Multiflow is not complete, e.g., branch instructions and nops are missing. Some blocks have no locality data and some cannot be matched. These flaws result in no locality information for about 25% of all blocks. When we do not have or cannot map locality information, we classify loads as *misses* following the Balanced scheduler's conservative policy.

4.4 Limitations of Experiments

A systematic problem is that register assignment is performed after scheduling, which is true in many systems. We cannot use the locality data for spilled loads or any other loads that are inserted after scheduling because these loads do not exist in the scheduler's internal representation, and different spills are of course required for different schedules. Unfortunately these loads are a considerable fraction of all loads. The fraction of spilled loads for our benchmarks appear in the second and sixth columns of Table 1. `Apsi` spills 44.9% of all loads, and `turb3d` spills 47.7%. A scheduler that runs after register assignment would avoid this problem, but introduces the problem that register assignment reduces the available ILP.

The implementation of the Balanced scheduling algorithm we use is tuned for a single issue machine. If an instruction can be placed behind an other one the other's weight is increased, without considering whether a cycle can be hidden at all; i.e., the instruction could be issued in parallel with a second one placed behind that other instruction. Figure 3 shows two simple DAGs. In the left DAG, the floating point and the integer add both may issue in the delay slot of the load, and the scheduler thus increases the weight of the load by one for each add. On a single issue machine, this weighting correctly suggests that two cycles load latency can be hidden. On the Alpha 21164, all

three instructions may issue in parallel[4], i.e., placing the adds in the delay slot does not hide the latency of the load. Similarly in the DAG on the right, the Balanced scheduler will give `load1` a weight of 2. Here only one cycle of the latency can be hidden, because only one of the other loads can issue in parallel. The weight therefore does not correctly represent how many cycles of latency can be hidden, but instead how many instructions may issue behind it.

Another implementation problem is that the Balanced scheduler assumes a static load latency of 1 cycle, whereas the machines we use have a 2 or 3 cycle load delay. Since the scheduler covers the static latencies of non-loads first, if there is limited ILP the static latencies of loads may not be covered (as we mentioned in Section 4.1). When we classify some loads as *hits*, we exacerbate this problem because now neither pass assigns these loads any weight. We correct this problem by increasing the weights of all loads classified as *hits* to their static latency, 2 or 3 cycles, which the list scheduler will then hide if possible. This change however breaks the paradigm of Balanced scheduling, as weight is introduced that is not based on available ILP.

5 Experimental Results

We used the SPECfp95 benchmarks, one SPECint95 benchmark, and the Livermore loops in our experiments. The numbers for Livermore are for the whole benchmark with all kernels. We first compiled the programs with Balanced scheduling, overwriting the weights of loads with 1 to achieve a schedule where load latencies are not hidden. We executed this program several times and monitored it with DCPI to collect data for the locality analysis.

We then compiled each benchmark twice, once with Balanced scheduling, and a second time with Balanced scheduling and our locality data. We ran these programs five times on an idle machine, measured the runtimes with DCPI, and averaged the runtimes. The standard deviation of the runs is less than the differences we report. We used the same input in all runs and thus are reporting the upper bound on any expected improvements. The DCPI runtime numbers correspond to numbers generated with the operating system command `time`. We executed the whole experiment twice, once on an Alpha 21064 and once on a 21164. To show the sensitivity of scheduling to the quality of the locality data we use both the strict and the generous heuristic on the Alpha 21164. We expect our heuristic to perform better on the dual-issue 21064 than on the quad-issue 21164 because it needs less ILP to satisfy the issue width. The 21164 is of course more representative of modern processors.

Table 1 shows the number of loads we were able to analyze with the strict heuristic. The total number of loads includes only basic blocks with useful monitoring data, i.e., blocks that are executed several times. The first column gives the percentage of loads inserted during or after scheduling that we cannot use. The second column gives the percentage of loads for which the locality data cannot be evaluated because the instruction that uses it is not in the same block, or because the basic block could not be mapped on

[4] The Alpha 21164 can issue two integer and two floating point operations at once. Loads are integer operations with respect to this rule.

	21064				21164			
program	spill/all	nodata/all	hit/all	hit/anal	spill/all	nodata/all	hit/all	hit/anal
applu	12.9	57.8	13.7	46.7	24.6	46.5	7.9	27.3
apsi	29.2	27.9	16.6	38.6	44.9	17.5	8.6	22.9
fpppp	21.4	70.3	4.3	51.3	18.1	73.6	1.4	17.3
hydro2d	8.2	34.6	22.2	38.9	8.5	33.4	10.2	17.6
mgrid	13.5	42.0	22.4	50.3	15.1	41.5	12.5	28.7
su2cor	17.2	35.2	19.0	40.0	16.9	38.1	6.7	14.8
swim	4.6	25.7	21.1	30.3	5.5	5.5	1.4	1.5
tomcatv	1.8	69.0	6.3	21.7	6.2	33.3	0.7	1.1
turb3d	45.9	25.4	9.5	32.9	47.7	24.8	10.4	37.8
comprs.95	16.2	14.9	31.1	45.1	16.3	14.4	40.5	58.5
livermore	13.9	37.1	25.9	52.9	13.3	40.2	17.1	36.8

Table 1. Percentage of analyzed loads 21064 and 21164

the intermediate representation. Although we use the same binaries, `dcpicalc` produces different results on the different architectures and thus the sets of basic blocks may differ. The third column gives the percentage of loads the strict heuristic classifies as *hits* out of all the loads. It classifies all loads not appearing in columns 1-3 as *misses*. The last column gives the percentage of loads with useful locality data, i.e., those not appearing in columns 1 and 2.

On the 21164, our classification marks very few loads as *hits* for `swim` and `tomcatv`, and thus should have little effect. To address the problem that about half of all loads have no locality data, we need different tools, although additional sampling executions might help.

Table 2 gives relative performance numbers: Balanced scheduling with locality information divided by regular Balanced scheduling. The first two columns are for the strict heuristic which on average slightly improves the runtime of the benchmarks. The two columns give performance numbers for experiments on an Alpha 21064 and an Alpha 21164. The third column gives the performance of a program optimized with locality data produced by the generous heuristic, and executed on an Alpha 21164. On average, scheduling with the generous heuristic degrades performance slightly.

Many blocks have the same schedule in both versions and have only 1 or 2 instructions. 18% of the blocks with more than five instructions have no locality data available for rescheduling because the locality data could not be integrated into the compiler and therefore have identical schedules. 56% of the blocks where locality data is available have either no loads (other than spill loads), or all loads have been classified as *misses*. The remaining 26% of blocks have useful locality data available, and a different schedule. Therefore, the improvements stem from only a quarter of the program.

Although the average results are disappointing, improvements are possible. In two cases, we improve performance by 10% (`su2cor` on the 21064 and `fpppp` on the 21164), and these results are due to better scheduling. The significant degradations of two programs, (`compress95` on the 21064 and `turb3d` on the 21164), are due to flaws in the Balanced scheduler rather than inaccuracies the locality data introduces.

program	strict heuristic		gen. heu.
	21064	21164	
apsi	99.7	100.4	99.9
fpppp	98.1	90.6	101.1
hydro2d	100.4	99.4	101.9
mgrid	101.2	101.7	104.2
su2cor	90.2	99.6	100.4
swim	100.2	99.2	99.3
tomcatv	101.8	99.6	102.9
turb3d	96.8	106.3	105.1
compress95	107.6	98.8	99.5
livermore	100.3	98.7	96.9
average	99.6	99.4	101.1

Table 2. Performance of programs scheduled with locality data.

Further investigation on procedure and block level showed that mostly secondary effects of the tools spoiled the effect of our optimization; the optimization improved the performance of those blocks where these secondary effects played no role. Space constraints precludes explaining these details.

6 Conclusions

In this study, we have shown that it is possible to exploit the run-time information provided by hardware counters to tune applications. We have exploited the locality information provided by these counters to improve instruction scheduling. As it is still difficult to determine statically whether a load hits or misses frequently, hardware counters act as a natural complement to classic static optimizations. Because of the limitations of the scheduler tools we used, we could not exploit all the information provided by DCPI (miss ratio instead of latencies). We believe that our approach is promising, but that it needs new scheduling algorithms that take in to account variable latencies and issue width to be fully realized.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, June 1997.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 72–84, Montreal, Canada, June 1998.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[4] S. Carr. Combining optimization for cache and instruction-level parallelism. In *The 1996 International Conference on Parallel Architectures and Compilation Techniques*, Boston, MA, October 1996.

[5] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction level profiling on out-of-order processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[6] Chen Ding, Steve Carr, and Phil Sweany. Modulo scheduling with cache reuse information. In *Proceedings of EuroPar '97*, pages 1079–1083, August 1997.

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

[8] D. R. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, Albuquerque, NM, June 1993.

[9] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a myrinet-connected shrimp cluster. In *1998 ACM Sigmetrics Symposium on Parallel and Distributed Tools*, August 1998.

[10] J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 151–162, San Diego, CA, June 1995.

[11] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, pages 51–143, 1993.

[12] F. Jesus Sanchez and Antonio Gonzales. Cache sensitive modulo scheduling. In *The 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 261–271, November 1997.

[13] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994.