Load Scheduling with Profile Information*

Götz Lindenmaier
Institute for Program Structures and Data Organization
University of Karlsruhe

Kathryn S. McKinley
Department of Computer Science
University of Massachusetts

Olivier Temam
PRiSM, Versailles University

September 17, 1999

Abstract

Within the past five years, many manufactures have added hardware performance counters to their microprocessors to generate profile data cheaply. Translating aggregate data such as basic block execution frequencies from the executable to the compiler intermediate representation is fairly straightforward. In this paper, we show how to use Compaq's DCPI tool to determine load latencies which are at a fine, instruction granularity and then use them to provide fodder for improving instruction scheduling. We validate our heuristic for using DCPI latency data to classify loads as hits and misses against simulation numbers, demonstrating that we can gather correct latencies cheaply at runtime. 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 has the potential to improve run times by up to 10% on a Compaq Alpha when compared to Balanced scheduling, but that a variety of pitfalls make consistent improvements difficult to attain.

1 Introduction

In this paper, we explore 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)[1]. We use DCPI, the performance counters on the Alpha, and `dcpicalc` a tool for translating the average load stall times computed by DCPI 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 as simulation. For instance, `dcpicalc` often cannot produce the reason for a load stall, or differentiate between multiple reasons for a stall when they exist. 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 our results show this classification matches simulation numbers well.

We present a modified version of Balanced scheduling [15, 17] implemented in the Multiflow Trace Compiler which produce schedules that hide miss latencies with available instruction level parallelism. We communicate our classification to the Balanced scheduler, mapping it from the executable back into the Multiflow's Intermediate Representation (IR), and then produce a new schedule. Although others have used static profile information for the same purpose [12, 9, 7], we are the first to use performance counters at such a fine granularity to improve optimization decisions. Our results include execution time improvements (not simulations!) due to improved schedules on the Alpha 21164 and 21064 of up to 10%, but our average improvements are less than 1%. 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.

Our work also suggests a more interactive runtime system to tune applications. The runtime monitoring tool first collects information for selected compiler optimizations. After each program run, the operating system triggers program recompilation based on this profile information. The com-

*The authors email addresses are goetz@ipd.info.uni-karlsruhe.de, mckinley@cs.umass.edu and temam@prism.uvsq.fr. This work is supported by grants from Compaq, NSF grant EIA-9726401, Darpa grant 5-21425, and an NSF Infrastructure grant CDA-9502639. Kathryn S. McKinley is supported by an NSF CAREER Award CCR-9624209. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

[1] We define ILP for an instruction as the number of other instructions available to hide its latency, and the ILP of a block or a program as the average of the ILP of its instructions.

piler would be designed to exploit run-time information, like the scheduling techniques presented in this study. Naturally, we must still investigate heuristics to manage such an environment across several runs. In this study, we implicitly show that all the components of such an environment exist or can be designed, and demonstrate one such optimization.

The remainder of this paper is organized as follows. 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. We summarize our experiences and conclude in Section 6.

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 an 4-way issue Alpha [4, 10]. Similar hardware now exists on the Intel PentiumPro, Sun Sparc, SGI R10K and in Shrimp, a shared-memory parallel machine [4, 16]. 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 [2] and critical path profiles to sharpen constant propagation [3]. Our work is unique in that it uses information at the instruction level, and integrates it into a scheduler.

Compiler approaches to improve locality [1, 13, 14, 20, 24] transform the loop nest or data layout to eliminate misses, but do nothing to hide the remaining latencies. Hardware and software prefetching identifies missing instructions [5, 8, 19, 21], and issues loads early to hide latencies. Although prefetching is often effective, it may replace data in the cache that is still in use, it may fetch data already in cache (useless prefetches), and it may fetch data that is never used. The approach we use here is complementary to prefetching.

Many researchers have used software profiling and simulation to find critical paths, traces [12], and super blocks [9, 7] to focus instruction scheduling and other optimizations. Here instead we use profile information to influence the order of instructions rather than which instructions should be fodder for the scheduler.

Previous work on using instruction level parallelism (ILP) to hide latencies for non-blocking caches has two major differences from this work [6, 11, 15, 17, 22]. 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, the information it produces, `dcpicalc`, a tool that translates DCPI output to a more useful form, and compares the results to simulation.

DCPI is a runtime monitoring tool that cheaply collects information by sampling hardware counters [4]. On average, it adds one to two percent to a program's execution time. The DCPI hardware also saves the collected data efficiently in a database. It is designed to run 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, e.g., cycles or cache misses. When the counter overflows, it triggers an interrupt. The interrupt handler saves the program counter for the instruction that is at the head of the issue queue, about to execute. `Dcpicalc` then interprets the sampled data off-line to provide detailed information about how often, long, and why instructions stall during the execution of the program. Figure 1 shows the information computed with `dcpicalc` for an example basic block from `compress`, a SPEC'95 benchmark executed on an Alpha 21164.

```
     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
```

Figure 1: Example for the calculation of locality data.

`Dcpicalc` uses the number of times an instruction is sampled to estimate the length of each stall. The basic idea is that 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. Using processor implementation details, `dcpicalc` determines through a static analysis of the binary the reason(s) for some stalls. If DCPI provides information about dynamic events, e.g., cache misses, `dcpicalc` also uses these to determine the cause(s) of a stall. If it has no such information, it gives all possible reasons for a stall.

`Dcpicalc` annotates the assembly instructions with this information. In Figure 1, five instructions stall; each line without an instruction indicates a half cycle stall before the next instruction can issue.[2] 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. `Dcpicalc` determines reasons `a` and `b` from the static known machine implementation, and `i` and `d` from the dynamic information. The two rightmost columns indicate the length of the static stalls determined with the machine model, and the average length of the dynamic stalls, i.e., stalls due to dynamic events, from the DCPI data, respectively.

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 means that the instruction seldom stalls, and when it stalls, the stall is relatively short. Instruction 7 also stalls due to an instruction cache miss, on average 20 cycles, which indicates this stall is frequent and long, and probably accesses a lower level of the cache hierarchy.

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`.
- *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 instruction 1 in Figure 1, *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 cause a stall (*MissMarked = false* and *Stall = 0*) because its latency may be hidden by static or dynamic events. If *StatDist* is larger than the latency of the cache at memory hierarchy level n, then a cache miss in cache $n-1$ which hits in cache n will not cause a

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

stall. If *StatDist* does not hide a cache latency, the *DynDist* may still hide it. This case occurs only if a dynamic stall hides the cache latency on almost every execution of the load. In this case, *DynDist* or *OtherDynStalls* are high.

The information collected with DCPI 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. Our heuristics are based on basic blocks, rather than a trace. We classify a load that is used in a different basic block as a *miss* per default.

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* since our experiments showed that this heuristic works best.

We examined several other heuristics, all of which performed worse in comparison to the simulation and when used for optimization. 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 more loads as *hits* (i.e., the loads should be classified as *misses*).

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 [23] 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.)

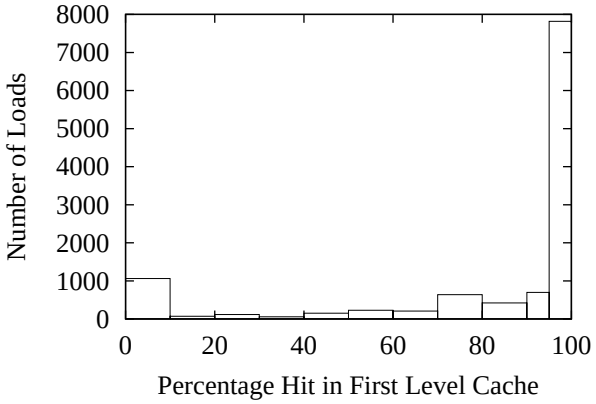

Figure 2: Simulated number of loads

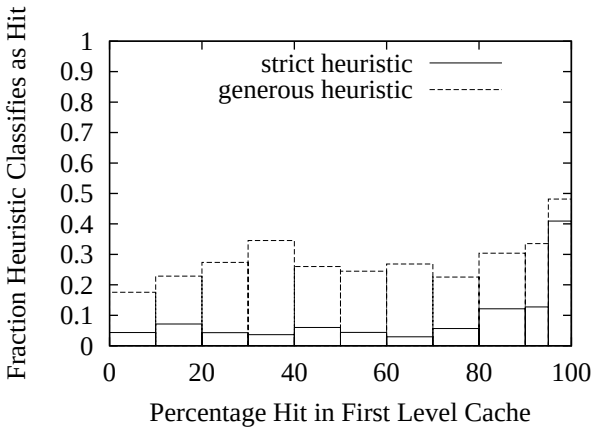

Figure 3: Comparison of heuristics to simulation

The Figures 2 and 3 summarize all analyzed loads in eleven SPEC'95[3] and the Livermore benchmarks. 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.

Figure 3 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

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

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 operations that do not depend on the load in its delay slots (*behind* it) in the schedule. In most programs however, 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 scheduler could differentiate between the expected latency of a miss, placing the most ILP behind the misses with the longest latencies.

Instruction scheduling has an additional advantage as a test optimization for using dynamic locality information: a few inaccuracies in the runtime data can be smoothed out in the final schedule because the plethora of loads in a given trace.

4.1 Balanced Scheduling

We use the Multiflow compiler [12, 18] with the Balanced Scheduling algorithm [15, 17]. The Multiflow compiler includes additional optimizations, e.g., unrolling, to generate ILP and *traces* of instructions that combine basic blocks. We use these transformations. Below we first briefly describe Balanced scheduling and then we describe our modifications to it.

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).[4] 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 [15], 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.[5] 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.

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

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

[5] For the Alpha 21064 and 21164, hits actually need weight also because they have a 3 and 2 cycle latency, respectively, but the implementation of the Balanced scheduler we have assumes a 1 cycle load penalty and unfortunately ignores this latency.

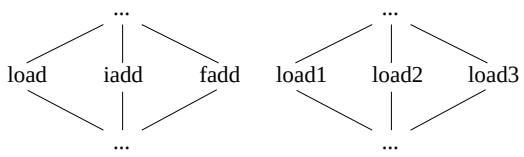

Figure 4: Example: Balanced scheduling for multi issue processors.

to map the locality information to the internal representation. To guarantee that the compiler uses the same representation at the beginning point for the scheduler as on the previous compilation, it must, of course, first perform the same optimizations.

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 first columns of Table 1 and 2. `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 4 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[6], 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. 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

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

program	spill/all	nodata/all	hit/all	hit/anal
applu	12.9	57.8	13.7	46.7
apsi	29.2	27.9	16.6	38.6
fpppp	21.4	70.3	4.3	51.3
hydro2d	8.2	34.6	22.2	38.9
mgrid	13.5	42.0	22.4	50.3
su2cor	17.2	35.2	19.0	40.0
swim	4.6	25.7	21.1	30.3
tomcatv	1.8	69.0	6.3	21.7
turb3d	45.9	25.4	9.5	32.9
comprs.95	16.2	14.9	31.1	45.1
livermore	13.9	37.1	25.9	52.9

Table 1: Percentage of analyzed loads 21064

program	spill/all	nodata/all	hit/all	hit/anal.
applu	24.6	46.5	7.9	27.3
apsi	44.9	17.5	8.6	22.9
fpppp	18.1	73.6	1.4	17.3
hydro2d	8.5	33.4	10.2	17.6
mgrid	15.1	41.5	12.5	28.7
su2cor	16.9	38.1	6.7	14.8
swim	5.5	5.5	1.4	1.5
tomcatv	6.2	33.3	0.7	1.1
turb3d	47.7	24.8	10.4	37.8
comprs.95	16.3	14.4	40.5	58.5
livermore	13.3	40.2	17.1	36.8

Table 2: Percentage of analyzed loads 21164

is of course more representative of modern processors.

Tables 1 and 2 show the number of loads we were able to analyze with the strict heuristic. The total number of loads include 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 in Balanced scheduling. 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 the intermediate representation in the Multiflow compiler. 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 with respect to the 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. Since about half of all loads have no locality data, we hope that more information from additional sampling executions would further improve our results.

Table 3 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.

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 3: Performance of programs scheduled with locality data.

Many blocks have the same schedule in both versions and many blocks 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,

we think that 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.

5.1 Scheduler Sensitivity to Locality Data

In this section, we present more detailed results for a few programs comparing Balanced scheduling with and without locality information. Table 4 gives execution times in cycles of several procedures for the optimal case where there are only static stalls (`best`), i.e., no dynamic events such as data cache misses, the average case (`aver`) for the actual schedules with and without locality information, and the number of cycles spent stalling on data cache misses (`d-stalls`), all measured using DCPI. In the remainder of this section, we first discuss the changes to the schedule due to locality information with respect to no data misses, and then how changes to the schedule effect dynamic events.

When we examine the schedules, we see that locality information does cause more ILP to be placed behind *misses*, and less behind *hits*, i.e., it uses the data effectively. In the `best` column of Table 4, we can see that locality data also improves the schedule when we ignore dynamic latencies. However, the quality of the `best` static schedule is not always correlated to the average execution time. If a poor static schedule hides a larger share of the dynamic latencies, it will out perform a longer static schedule.

The improvements in the static schedules (where dynamic latencies are ignored) are due to the following reason. The weights of instructions with static latencies are the same for both scheduling algorithms because the locality data influences only which dynamic latencies the scheduler hides; the weights for instructions with static latencies are determined before using the locality information. With locality information, the amount of total weight in the graph is actually reduced. For example, consider Figure 5 which shows a DAG and the weight distributed in the second phase due to instruction `inst`. The increment of weight due to this instruction for the four loads is given for Balanced scheduling and scheduling with locality data as pairs bs/bs+ld. The sum of the weight distributed by Balanced scheduling is 1.2, but with locality data it is only 1.0 because the scheduler with locality information does not give any ILP to hits. Thus, all loads get less weight and there is relatively more weight on instructions with static latencies and therefore more emphasis on hiding static

procedure	sched.	best	aver	d-stalls
hydro2d, 21164				
filter, filter fct.	bs	103	126	31.5-32.0
1 hit, 6 miss	bs+ld	100	116	28.0
s1	bs	44	196	36.5-71.0
7 hit, 11 miss	bs+ld	50	199	71.0
livermore kernels, 21164				
1. kernel	bs	29	39	9.5
5 hit, 4 miss	bs+ld	30	40	9.5
2. kernel	bs	64	77	11.0
10 hit, 10 miss	bs+ld	62	75	10.5
3. kernel	bs	41	45	0.0
6 hit, 18 miss	bs+ld	43	52	9.0
7. kernel	bs	59	61	1.5
9 hit, 9 miss	bs+ld	55	62	1.0-4.0
12. kernel	bs	23	25	0.0-0.5
5 hit, 4 miss	bs+ld	26	32	1.5-5.0
13. kernel	bs	78	128	21.5-31.0
5 hit, 11 miss	bs+ld	75	122	23.5-31.0
14. kernel, 1. loop	bs	69	85	0.5-2.5
4 hit, 13 miss	bs+ld	65	112	24.5-34.5
21. kernel	bs	35	43	0.0
5 hit, 11 miss	bs+ld	34	43	0.0
22. kernel	bs	247	294	0.0-2.0
7 hit, 17 miss	bs+ld	249	269	0.0-3.0
su2cor, 21064				
bespol	bs	93	178	0.0
4 hit, 20 miss	bs+ld	89	153	0.5-1.5
sweep	bs	58	452	0.0
6 hit, 18 miss	bs+ld	58	350	0.0
compress95, 21064				
compress, 1. trace	bs	20	49	1.0-3.0
3 hit, 4 miss	bs+ld	20	57	0.5-1.0
compress, 2. trace	bs	8	42	0.0
2 hit, 0 miss	bs+ld	8	54	0.0
getcode	bs	21	34	0.0
6 hit, 3 miss	bs+ld	15	28	0.5

Table 4: Execution time of chosen traces in cycles.

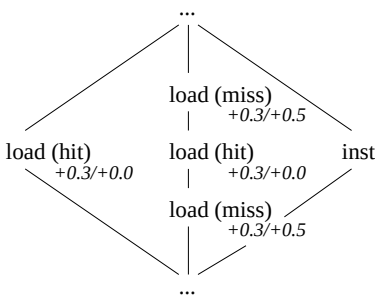

Figure 5: Example: Balanced scheduling with locality data introduces less overall weight.

cycles which on average yields a better schedule.

Another effect we observed on the schedules is that locality data slightly improves the instruction mix. If the loads are scheduled with the static latency of the first cache, loads, integer operations and floating point operations are mixed, so that a schedule utilizing the capabilities of the processor is achieved. Such a schedule sometimes allows three or four instructions to issu at once. Scheduling with balanced weights moves the load instructions to the beginning of the schedule of traces, as the Balanced scheduler gives all loads more weight to hide their dynamic latencies. But therefore many integer operations computing addresses must be moved to the beginning of the schedule, too. Consequently integer and floating point operations are not mixed as well and it is unlikely that more than two instructions will issue simultaneously.

Scheduling with locality data can impact performance in three different ways. (1) The schedule hides the latency of *misses* better. (2) The schedule can either breed or eliminate other dynamic effects, e.g., instruction cache misses or write buffer overflows. (3) Loads that miss (or hit) with the original schedules can hit (or miss) with the new schedule.

The second effect should even out for many traces, as these effects are not controlled by the experiment. We cannot see a direct correlation between the optimization and these effects that we could exploit to avoid them.

We observed that the third effect considerably degrades the performance of the schedules generated with locality data. If a load classified as *hit* now misses in the cache, none of its latency is hidden. There are two possible causes. First, if several loads accessing a cache block are reordered, and a load classified as *hit* gets scheduled as the first of these, this load will now miss in the cache. We observed this effect in several programs, and is not surprising in light of loop unrolling. Second, if loads accessing two different datasets are interleaved differently they might cause more conflict misses.

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). However, in blocks where these drawbacks do not matter we could achieve performance improvements. Another benefit of this study was to investigate how to adapt the Balanced scheduling algorithm to better take into account processor architecture (multiple issue and cache 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.

While we used the DCPI information much like simulation-based profiling information, the low overhead of hardware counters will ultimately allow for background recompilation of an application after an execution, based on profiling information collected by hardware counters. Such a compilation environment which involves the compiler, hardware counters and operating system management for performance purposes will raise issues like the stability of application optimizations, e.g., scheduling, across different datasets.

References

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 28th International Symposium on Microarchitecture*, Austin, TX, December 1993.

[2] 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.

[3] 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.

[4] 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.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[6] 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.

[7] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B.R. Rau, and M.Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th International Symposium on Microarchitecture*, Paris, France, December 1996.

[8] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[9] W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W. W. Hwu. Profile assisted instruction scheduling. *International Journal of Parallel Programming*, 22(2):151–181, April 1994.

[10] 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.

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

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

[13] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, October 1998.

[14] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matix-based approach to the global locality optimization problem. In *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] C. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Boston, MA, October 1996.

[20] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[21] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992.

[22] 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.

[23] 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.

[24] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.