Using SimICS to Evaluate the Penny System

Johan Montelius, Peter Magnusson
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, SWEDEN
{jm,psm}@sics.se
http://www.sics.se/

Abstract

We demonstrate the benefits of instruction-set simulation in the evaluation of a parallel programming system, Penny. The simulator is a reliable tool in exploring design alternatives for improving performance and can greatly help in understanding program behavior. The results obtained improved the performance of Penny and highlighted the importance of the caches.

1 Introduction

Instruction-set simulation is a useful tool both for performance debugging and for explaining the behavior of a system. In this paper we show how the instruction set simulator SimICS is used to improve and explain the performance of Penny, an implicit parallel programming system.

A tool such as SimICS can provide a wide range of statistics on hardware events triggered by software activity, including data cache hits and misses, instructions executed, and virtual memory performance. It can relate these events to particular lines of code, greatly simplifying the task of understanding the performance of software systems and, hopefully, improving it.

The simulator shows that the Penny system would scale almost perfectly if only the number of executed instructions are taken into account, i.e. for an idealized multiprocessor. The total number of executed instructions when sixteen processors are used increase only by 10% compared to the one processor execution. This would indicate a speedup of 15, in contrast to the actual speedup of 7.7. The study shows how important good cache performance is to a system such as Penny, and demonstrates how an instruction set simulator can relate cache events to program code.

1.1 The Penny system

The Penny system [12] is a implementation of AKL [7] on a shared memory architecture. It will automatically extract parallelism in an AKL program. No user annotations are required, thus relieving the programmer from adding explicit information to control parallelism. The system can utilize both *and*- and *or*-parallelism in the program. There is no compiler support to

extract parallelism. All detection and scheduling of parallel tasks is done automatically at runtime. Penny is complete with a parallel garbage collector and, for an experimental system, is quite stable.

The heart of the system is a threaded code emulator implemented in C using the GNU C compiler (version 2.x) where labels can be handled as data. The machine only defines sixty-three abstract machine instructions so the emulator itself is rather small. The instruction set is very similar to the instruction set used in the WAM [14].

When a program is executed, a fixed set of *workers* are created. The number of workers will determine the level of parallelism, so there is no advantage to create more workers than available number of processors. Each worker will dynamically be assigned work during an execution. If a worker runs out of work it will steal tasks from another worker.

During an execution the workers build and modify a shared *execution state*. The execution state consists of a tree structure of *goals* and *continuations*, and a set of AKL *terms*. Each goal in the execution state is either ready to be executed or suspended on some AKL variable. When a variable is assigned a value the goals suspended on the variable will be scheduled for execution. It is the worker that assigned the value that is responsible for executing the goals.

Continuations represents sequences of un-executed goals and are in the same way owned by a particular worker. The right to execute a goal or to select a goal from a continuation can be stolen by another worker. The whole execution state is therefore accessible to all workers.

The data structures that are used to represent goals and continuations can be explicitly reclaimed by the workers. This improves cache performance since the same cache-lines can be reused immediately. AKL terms are not explicitly reclaimed although work on compile time analysis shows that this is quite possible [4]. The terms are therefore allocated on a *heap* that is subjected to garbage collection. A parallel stop-and-copy garbage collector is used. It is important that the garbage collector is parallel since the garbage collection time would otherwise increase in proportion to the execution time.

The Penny compiler compiles AKL programs to abstract machine instructions. The Penny compiler is itself written in AKL and consists of about three thousand lines of AKL code, that can in turn be executed on Penny itself with good parallel performance.

1.2 The SimICS Simulator

There are a large number of techniques and tools available to analyzing the behavior of a software system. Regardless of how it is done, we need to solve three problems. First, we need to execute the actual instructions specified by the executable binary. Second, we need to perform the system services required by the program, if any. The first two problems thus involve recreating the execution environment. Third, we need to generate information

about the execution over and above the actual program result.

There are essentially three starting-point strategies:

Instruction set simulation, also called instruction-level or program-driven simulation, is the naive brute-force approach, whereby each instruction in the program is simulated one at a time. This provides an accessible and in some sense correct target machine model for instrumentation, and places minimum restrictions on the architectural relationship between the host and target. Program-driven simulation is probably the oldest strategy [5].

Execution-driven simulation, also called program augmentation, involves running a modified program binary. The modifications can be induced at any stage during generation of the binary, either by modifying intermediate program formats (source code, assembly code, object file, or executable binary) or any of the compiler tools (preprocessor, compiler, assembler, or linker). Traditional profilers generally fall into this category.

Host-supported simulation, historically called emulation, requires hardware monitors or other special host hardware features which provide tools to gather statistics or otherwise control the execution of a program.

Naturally, as with any artificial taxonomy, the above classification is by no means strict—mixing methods is common.

It is beyond the scope of this paper to go into any detailed discussion of trade-offs between the various strategies. Simply put, instruction set simulation is the slowest and most flexible, host-supported simulation is the converse (fast but inflexible), and execution-driven simulation holds the middle ground.

SimICS [9, 10, 11] is an instruction set simulator that has borrowed many design principles from g88 [3]. SimICS takes the brute force approach, modelling the target architecture on an instruction-by-instruction level. This results in a lower performance compared to execution-driven or host-supported systems—namely a slow-down of approximately $25 - 100$ per simulated processor compared to the real execution.[1] We also need to deal with a significantly more complex software engineering problem in building the simulator. This effect is, of course, difficult to quantify, but it is significant.

On the plus side, SimICS provides full profiles—both of execution (instructions), data cache events, and virtual memory events—and does so within a traditional debugger environment. This allows a detailed, interactive analysis of parallel programs.

1.3 Our Target Machine

Our workhorses for Penny timings (and SimICS runs) have been two SPARCcenter 2000 (SC2000) multiprocessors with 8 and 20 processors, respectively.

[1] Thus if an execution takes two seconds using one processor and one second using two processors the simulation will in both cases take between 50 and 200 seconds, i.e. a slowdown of $30 - 100$ per processor.

The SC2000 is a bus-based shared-memory multiprocessor from Sun Microsystems. Figure 1 is a sketch of a generic shared-memory multiprocessor—though the SC2000 is considerably more complex, the figure serves to highlight some principal features.

Each processor has two on-chip caches, one for instructions and one for data. These are generally small, because on-chip area is a scarce resource. On the SC2000's processors, 50MHz SuperSPARCs, the data cache is 16Kbytes, four-way associative with 32 byte long cache lines. The instruction cache is 20Kbytes, 5-way associative with 64 byte cache lines. Despite their small size, the first level caches consume half of the SuperSPARC's 3 million transistors.

The processors connect to an off-chip cache, the second level cache, which on the SC2000 is 2Mbytes, direct-mapped with 64-byte cache lines. These caches are connected to a bus, whereby they can communicate with the main memory and/or other caches. This communication is controlled by SuperCache controllers and "Bus Watcher" chips. There are actually 2 buses on the SC2000, dual 40MHz XDBuses, with an effective read/write throughput of 500Mbytes per second.

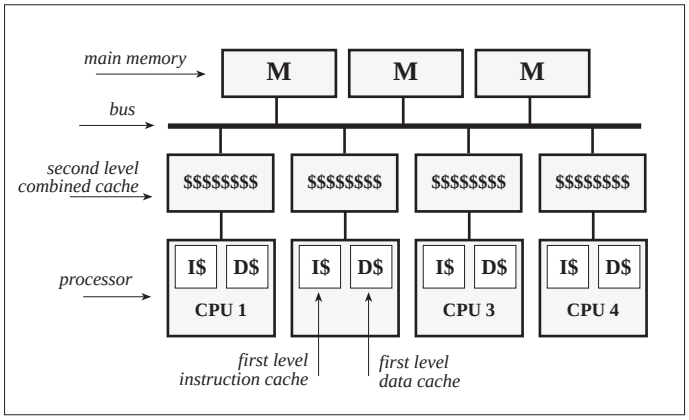

Figure 1: Generic shared-memory multiprocessor

A throughput of 500Mbytes per second may sound high, but each SuperSPARC processor is capable of executing three instructions per cycle, including supplying 64 bits per cycle from memory, so a 20-processor SC2000 could conceivably request 8 *billion* bytes per second from the memory system. Hence two layers of caches. The first level, being on-chip, can react with new data in one cycle. The second level takes 5-10 cycles, whereas accessing the main memory takes 20-60 cycles. A high cache hit rate is therefore crucial to good performance.

SimICS emulates the cache hierarchy in figure 1, which is close enough to real life to give a good prediction of performance of an application. In fact, during our profiling we initially had significant discrepancies between predicted performance and measured values, until we discovered that both the

SC2000 machines we used for timing measurements had faulty SuperSPARC processors with only 4Kbyte caches, not 16Kbyte. The faulty processors had gone unnoticed for several years, despite the machine being used extensively for benchmarking of parallel programs. To remain comparable to available hardware at the time of this study (autumn 1995), all simulated values in this paper assume a 4Kbyte first-level data cache with 32-byte cache lines, and a 2Mbyte second-level integrated cache with 64-byte cache lines, both direct-mapped. Instruction cache behavior is irrelevant in this study as the core interpreter in Penny is small.[2]

2 Using SimICS

In this section, we show how we used SimICS to study Penny from various perspectives. As input to Penny we used an implementation of the Smith-Waterman algorithm. The algorithm is used to compare DNA sequences and the benchmark has good parallel performance.

```
(gdb-simics) prof-weight 32 20
Weighted profiling results:
  Physical    Virtual    ( source )
  0x000ac860  0x00029860 (pid 1002)    285085.00
  0x000aca60  0x00029a60 (pid 1002)    282070.00
  0x000a61a0  0x000231a0 (pid 1002)    232325.00
  0x000a8540  0x00025540 (pid 1002)     95540.00
  0x000a8480  0x00025480 (pid 1002)     90155.00
  ...

Total profiled:   1644459.00 (56%)
Not shown:        1269168.00 (44%)
```

Figure 2: `prof-weight` listing

2.1 Performance debugging

The main concern when improving the performance of a system is to locate the code that consumes the most resources. SimICS allows different costs to be assigned to events and can then generate statistics and display the most expensive parts of the program. Figure 2 shows the output from one such command.

[2] Interpreters in general have the effect of converting instruction cache pressure to data cache pressure.

For our benchmark set and on our target machine the second-level data cache and TLB misses turn out to be the most important events.[3] These misses easily stall the CPU if the result is needed soon, which is often the case. The output in figure 2 is from running the benchmark with 6 workers. The cost of a TLB miss and a read miss was set to 5 and a write miss to 1. The figure lists the 5 most expensive blocks of instructions where each block is 32 bytes long (8 SPARC instructions).

```
0x29868 :   0 54628 2388    63500  1 ld  [ %i2 + 4 ], %o0
0x2986c :   0     0    0    60500  1 cmp  %o0, 1
0x29870 :   0     0    0    65000  1 be,a   0x2992c
0x29874 :   0     0    1        0  1 ld  [ %i2 ], %i2
0x29878 :   0     0    0    68500  1 cmp  %o0, 1
0x2987c :   0     0    0    64000  1 bcs,a   0x2992c
0x29880 :   0  3481   33    57500  1 ld  [ %i2 ], %i2
0x29884 :   0     0    0     1000  1 cmp  %o0, 2
0x29888 :   0     0    0      500  1 bne,a   0x2992c
0x2988c :   0     0    1        0  1 ld  [ %i2 ], %i2
0x29890 :   0    36    0      500  1 ld  [ %i2 + 8 ], %l0
```

Figure 3: Assembler listing of the most expensive block

```
477  0   245   92   1000  4 for(tc = ta->trd; tc != NULL; tc = tc->nxt) {
478
479  0 58109 2423 380500 10  switch(tc->type) {
480                           case DEAD_CELL:
481                             break;
482
483                           case FREE_CELL:
484                             break;
485
486                           case CHB_CELL: {
487  0    36    0    500  1     templc = tc->box.chb;
```

Figure 4: Source code listing of the most expensive block

The listing shows that almost 10% of the cost is found in the block starting at virtual address `0x29868`. The assembler listing of the block in figure 3 gives us: the number of write misses, the number of read misses, the

[3]The Translation Lookaside Buffer (TLB), also known as the Address Translation Cache (ATC), caches virtual-to-physical translations to improve the performance of virtual memory.

number of TLB misses, and the approximate number of times the instruction was executed and whether (1/0) the instruction was decoded.[4]

As the listing makes evident, the `ld` instruction on address `0x29868` causes a read miss almost every time it is executed. The instruction corresponds to the source code shown on line 479 in figure 4 where the profiling data has been accumulated for each source line.

The source code in figure 4 is from the garbage collector, and implements part of the algorithm for distributing tasks among workers. The read misses severely decreased overall performance, since the code is a part of a sequential phase of the garbage-collector.

Many profilers would have identified the procedure as a potential performance problem, but would not have explained why—namely that one line of assembler traversing a list misses the second-level cache over 80% of the time, and misses the TLB almost 4% of the time. This in turn was caused by the creation of the list having been spread across multiple processors, and would not have been a performance problem in a sequential version of Penny.

An implementation technique that avoids building the list had been sketched out a year earlier, but the previous benchmarking techniques had not seen the traversal as a potential problem. Making this correction to Penny improved performance significantly, as seen in figure 5.

Figure 5: Improvements of the Smith-Waterman benchmark, following removal of sequential bottleneck in garbage collection (note: log-log scale)

2.2 Deciding on prefetching

By generating annotated source code the emulator could be studied in detail. We noticed an exceptionally high read miss rate in the decoding of the *unify*

[4] This corresponds closesly with having been prefetched.

instructions.

The *unify* instructions are used to access the components of an AKL term. Terms are represented by tagged pointers and are accessed by first executing a *get* instruction. The *get* instruction will first look at the tag of the term and only after having verified the tag will it follow the pointer to verify the functor. A unique tag is used for list terms, so in this case only the tag needs to be verified. The following *unify* instructions will then access the arguments of the term.

The reason for the high miss rate in the *unify* instructions was that when the instructions were used they were often reading AKL terms that had been constructed by another worker. The terms had thus been constructed in one cache but were often read by another processor, forcing communication over the bus.

The remedy to this problem was to add a prefetch instruction (coded in C) in all *get* and *unify* instructions. The prefetch would read the next argument position so that the following *unify* instruction would find the value in the cache. The time to read the argument would hopefully overlap with useful work.

The fix did not work as expected. The read misses in the *unify* instructions did decrease but we had added a large number of read instructions, most of which contributed nothing since the data was already in the first-level cache. The net effect was slower execution.

The reason for this was obvious. When a functor or argument of a term was inspected the following argument would in seven cases out of eight be in the same cache line of the first level cache. The same would hold for any prefetch instruction in the *unify* instructions. The only place where a prefetch instruction would make any sense was in the *get* instruction that was responsible for verifying a list cell. The *get* instruction itself would only read the tagged pointer to the cell but not the *car* nor *cdr* of the cell.

Removing all but the one effective prefetch left us with an overall performance improvement of 3-4% for several of the workloads.

In this example, we were able to use SIMICS both to follow where the cache misses moved to, and to quickly quantify the overhead induced by the fix. Note that an optimizing compiler could not have identified this prefetch, since it wouldn't know about the restrictions placed on the sequences of abstract instructions—this required the intervention of the Penny designer, using SIMICS to explore trade-offs.

2.3 The danger of locking

The Penny system uses locks in two different situations. The first is in the internals when workers move between different parts of the execution state or steal tasks from each other. The second situation is when AKL variables are locked in order to add a new binding or suspension.

Both of these situations could cause a hot-spot in the implementation.

Since all locks are spin locks, a worker stalls if a lock is held by another worker, so it is interesting to know how often the locks are actually missed and how long it takes for a worker to acquire a missed lock.

To get an idea how often locks are missed, SimICS *counters* were placed around the lock primitives. A SimICS counter is a special SPARC instruction inserted in the source code that the hardware (the "real" processor) will treat as a no-op, but which SimICS uses to start or stop event accumulators. The technique is thus similar to execution-driven simulation, but with much lower perturbation.

```
Counter 5:

  Number of times counter activated: 79949
  Total tick count spent in counter: 1729885

  Detailed counts:

    read operations                  254074
    write operations                 192874
    xmem (swap) operations           41178
    read cache misses                83
    write cache misses               6296
    xmem (swap) cache misses         4444
    cache replacements               184
    cache net invalidates            2424
    number of TLB misses             467

Counter 6:

  Number of times counter activated: 1
  Total tick count spent in counter: 6

  Detailed counts:

    read operations                  2
    cache net invalidates            1
```

Figure 6: Example of a statistics vector (reported by counters)

We ran the Smith-Waterman benchmark with sixteen workers. The AKL variable lock counter statistics for one CPU are listed in figure 6. Counter 5 was entered when a worker examined a potential variable and counter 6 was entered if there was a collision. As we can see almost eighty-thousand variables were examined resulting in more than forty-thousand lock (swap) operations. In only one (!) case does a collision occur. The results were similar for all kinds of locks in the system.

These figures indicate that the locks in the Penny machinery are not a

performance bottleneck. In fact, it might be worthwhile redesigning some of these locks to be more aggressive in assuming low contention.

The use of SIMICS counters in this analysis greatly simplified instrumentation of the locks. We added half a dozen different types of counters, to a few dozen different procedures and macros. Though it required modification in the source code and re-compilation, the binary can run on the real machine unchanged with an insignificant effect on performance. In fact, in the real execution this instrumentation adds less than 4 no-op instructions for every 10000 "real" instructions. Thus there is no real need to maintain separate versions.

3 The limiting factor

Though we managed to improve both the initial performance and speedup (parallelism) of the Penny system, there remains a significant sequential element that could not be easily explained. As seen earlier in figure 5 the speedup is not linear, but decreases. The limiting factor could of course be in the Smith-Waterman algorithm itself but there are no sequential components in the benchmark that could explain the limited speedup. The answer lies in the cache performance of the Penny system.

3.1 Perfect parallelism

Table 1 shows statistics gathered from executing the benchmark on the improved Penny system. The reported numbers are the sum of events generated by all the processors. The cache statistics are for a 2Mbyte cache, i.e. our target machines' second level cache.

	Number of workers				
	1	2	4	8	16
runtime (ms)	13822	7238	3996	2487	1789
read operations ($x10^6$)	124	124	125	127	129
read cache misses ($x10^3$)	8.63	257	287	384	467
write operations ($x10^6$)	47.5	47.5	47.6	48.1	48.1
write cache misses ($x10^3$)	137	271	294	412	418
executed instr. ($x10^6$)	630	631	634	656	669

Table 1: Profiling and timing for all processors

As the table shows, the increased number of processors does not induce a large overhead in the number of executed instructions. The instruction count only increases by 10% when sixteen workers are used and the number of read and write instructions increase even less.

In Table 2 we show the actual reduction in execution time with each doubling of workers, contrasted with the reduction in various operation types per processor. For example, in going from 4 to 8 workers, execution time is

	number of workers			
	1-2	2-4	4-8	8-16
runtime	0.52	0.55	0.62	0.72
read operations	0.50	0.50	0.51	0.51
write operation	0.50	0.50	0.51	0.50
executed instr.	0.50	0.50	0.52	0.51

Table 2: Actual execution time, contrasted with instruction counts.

cut by 38%, whereas the number of read and write instructions are reduced by 49% each and the instruction count is reduced by 48%, all on a per-CPU basis. In other words, the complexity and nature of the computation on a coarse level does not change much, but becomes evenly spread over an increasing number of workers. To account for the non-linear scaling of performance, we need to look for other effects.

3.2 Cache misses

The explanation is partly found in the number of read and write misses. Table 3 shows estimated reduced execution time based on the cache misses only. When going from one to two workers the number of read misses increases dramatically, which represents a small number of capacity misses changing to a large number of coherency misses caused by inter-processor communication. The write miss count also increases but not as dramatically. The increased number of read and write misses explains why the initial speedup is only 1.9 and not closer to 2.0 as indicated by the operation count.

	number of workers			
	1-2	2-4	4-8	8-16
runtime	0.52	0.55	0.62	0.72
read misses	15	0.56	0.67	0.61
write misses	0.99	0.54	0.70	0.51

Table 3: improvements of cache performance

Once the initial penalty of running in parallel has been taken the miss rate increases less dramatically. When going from two to four or from four to eight workers the increased miss rate appears sufficient to explain the actual limited speedup. But, when going from eight to sixteen workers the increased miss rate alone cannot explain the poor performance.

3.3 Explaining overall performance

We ran various combinations of Penny—using eight versions of Penny itself (with different improvements and modifications), several benchmark inputs in addition to Smith-Waterman, and three levels of parallelism (4, 8, and 16

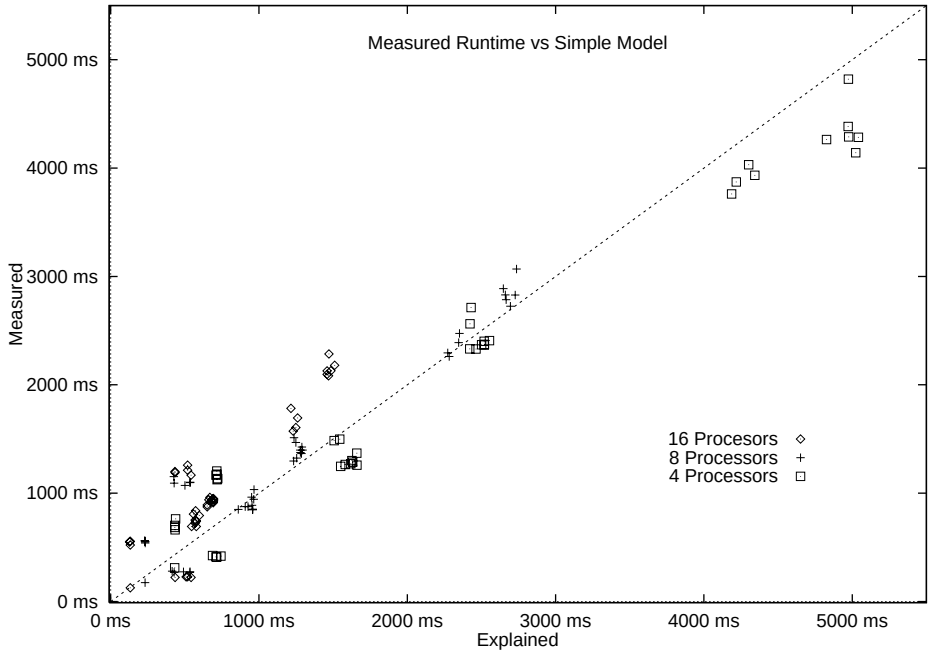

Figure 7: Results of explaining performance using a simple best-fit

workers). We measured the median time out of 31 runs, giving us 96 timing points. For each point, we used SIMICS to generate 12 aggregate values, covering the different cache and TLB miss types in the target system, for one of the processors.

We next selected three variables that we presumed to be important for explaining performance. A multiple regression of these variables against the database, and fudging, results in the following relation between the explained time T in milliseconds, read operations R, first-level read misses L_1, second-level read misses L_2:

$$T = 0.1 \times R + 0.33 \times L_1 + 10 \times L_2$$

In figure 7 we have plotted the explained time T against the measured time. The correlation is 0.96, not exceptional but clearly a good indicator. Observing the figure, we note that the performance of large configurations (16 processors) is overestimated and, conversely, the small configuration (4 processors) is underestimated. We suspect the reason for this is that we are lacking a fourth coefficient to measure bus contention, an issue that becomes significant as the number of communicating processors increase.

Misses to the second level cache cause the bus load to increase. The bus is a globally shared resource, so when it approaches saturation it stalls new accesses. As we increase parallelism, the miss rate of reads and writes both increase, which is natural since we are spreading work and communicating more. At the same time, execution time is decreasing, compounding pressure

on the bus. This could explain the abnormally high coefficient for read misses to the second level cache (10), which of course is the one of the three closest correlated with bus contention and thus has to carry the bus contention load in the regression.

The conclusion is that the bus capacity starts affecting overall performance at 16 workers (something we could confirm over a year later by running the same benchmark on the next generation SPARC multiprocessor with a faster bus).

In this analysis, SIMICS reports sufficient detail to help us reconstruct what is causing speedup to begin trickling off. We now know that for larger configurations, we should focus on second-level cache misses. Many of the data structures in Penny have been optimized to be cache-line aligned, etc. The SIMICS' source-line profiling of second-level cache misses could be used to evaluate different design changes aimed at reducing the amount of data communicated.

4 Concluding remarks

The brief examples in this paper have been anecdotal, and were intended to underline SIMICS' ability to zoom in on and study performance problems, or to explore design alternatives. The criteria we used for deciding what was "good" were a small number of characteristics, essentially the type of data listed in the *counters* example in figure 6. That these relatively simple statistics are good guidelines can be shown by correlating them against a large number of performance measurements from the real target machine. The cache performance is so important for the overall performance of the system that it is very hard to tune a system without having access to cache miss statistics.

A parallel system can have a perfectly scalable behavior with respect to the number of instructions executed and still not show linear speedup. Only when the cache misses are taken into account can the performance of the system be understood. We have shown a strong correlation between the actual execution time and read operations. It indicates that a second level read miss is two order of magnitudes more expensive than a first level hit. This in turn suggests that a read miss rate of the second level cache of over one percent will dominate the execution time.

Analysis of cache performance of logic programming systems is not new [6] but our approach is quite different. We have analyzed the performance of a parallel system on an existing parallel architecture. The analysis is done not through a generated trace file of selected read and write operations but from all operations actually performed by the system.[5] The number of read and write operations performed by the larger benchmarks is over 100 mil-

[5]We have restricted ourselves to a user-level study, i.e. excluding the effects of an operating system.

lion. The total number of instructions executed is for some benchmarks over 600 million. SIMICS also allows us to do interactive performance debugging since hot-spots are easily located. We know not only the overall cache performance but can pinpoint the instructions that generate misses.

In general, we note the importance of profiling tools keeping abreast of what hardware events are actually triggered by the software, why, and roughly what cost they correspond to. For the architecture modelled in this study, the performance of the memory hierarchy (data caches), the virtual memory system (TLB), and instruction count was adequate. Subsequent generations of parallel architectures will undoubtably introduce new events. For example, we would expect future large parallel machines to exhibit more complex memory hierarchy behavior, in which case we would need to extend SIMICS to maintain separate profiles for a larger family of memory hierarchy events.

Acknowledgments

The parallel implementation of AKL has been developed using the AGENTS 1.0 [8] system as a starting point. Haruyasu Ueda did much of the implementation and analysis of the scheduler [13]. Gallal Atlam and Kahyri Ali, designed and implemented the garbage collector [1, 2].

Bengt Werner co-designed much of the SIMICS front-end semantics. Anders Landin has been an enthusiastic supporter of SIMICS and has contributed much to the discussion of what a user needs to know about program/architecture interaction. David Samuelsson wrote much of the SPARC V8 interpreter. Henrik Forsberg wrote much of the Unix emulation.

Thanks to Peter Fritzson at Linköping University for access to a 20-processor SC2000 for the Penny timings.

Various parts of this work have been sponsored by Ellemtel in the Enterprise and Hubble projects, Sun Microsystems in the SOS project, the European Commission in the GPMIMD project, ACCLAIM Esprit project, EP 7195 and SICS.

References

[1] K. A. M. Ali. A parallel copying garbage collection scheme for shared-memory multiprocessors. *New Generation Computing*, 13(4), December 1995.

[2] G. A. M. A. Atlam. *Parallel garbage collection in a multiprocessor implementation of a concurrent constraint programming system.* Phd thesis, Menoufia University, Egypt, January 1997.

[3] R. C. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of Winter '90 USENIX Conference*, pages 53–63, January 1990.

[4] P. Brand, D. Sahlin, and T. Sjöland. Assessment of a storage optimization tool for AKL. Esprit, PARFORCE deliverable, D.WP2.3.5.M3.

[5] S. Gill. The diagnosis of mistakes in programmes on the EDSAC. In *Proceedings of the Royal Society Series A, Mathematical and Physical Sciences*, volume 206/1087, pages 538–554. Cambridge University Press, May 1951.

[6] M. Hermenegildo and E. Tick. Memory referencing characteristics and caching performance of and-parallel prolog on a shared-memory multiprocessor. *New Generation Computing*, 7(11):37–58, 1989.

[7] S. Janson. AKL a multiparadigm programming language. Uppsala Thesis in Computing Science 19, SICS Dissertation Series 14, Uppsala University, SICS, 1994.

[8] S. Janson and J. Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), SICS, 1992.

[9] P. S. Magnusson. A design for efficient simulation of a multiprocessor. In *Proceedings of MASCOTS*, pages 69–78, January 1993.

[10] P. S. Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the '97 Winter Simulation Conference*, 1997. (to appear).

[11] P. S. Magnusson and B. Werner. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, pages 62–73, 1995.

[12] J. Montelius. Exploiting fine-grain parallelism in concurrent constraint languages. Uppsala Thesis in Computing Science 28, SICS Dissertation Series 25, Uppsala University, SICS, 1997.

[13] H. Ueda and J. Montelius. Dynamic scheduling in an implicit parallel system. In *Ninth International Conference on Parallel and Distributed Computing Systems*, September 1996.

[14] D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, SRI International, 1983.