Using SimICS to evaluate the Penny system

Johan Montelius and Peter Magnusson
{jm,psm}@sics.se
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

We demonstrate the benefits of instruction-set simulation for the evaluation of the Penny system. The usage of a simulator can greatly help understand Penny's behavior for a varied set of benchmarks, in addition to serving as a reliable tool in exploring design alternatives for improving performance.

1. The Penny System

The Penny system [9] is a implementation of AKL[5] on a shared memory architecture. It will automatically extract parallelism in an AKL program and, during runtime, schedule tasks on a number of *workers* to achieve speedup. Each worker is running as a Solaris thread and all threads run in the same Solaris process. The threads share the same address space and have full access to all data structures in the system. This makes the distribution of tasks very easy.

No user annotations are required nor is there any compiler support to extract parallelism. All detection and scheduling of parallel tasks is done at runtime. This is a big advantage compared to systems where the user must provide explicit information so that the compiler can make a static distribution of tasks. The system can utilize both and- and or-parallelism in the program. It is complete with a parallel garbage collector and, for an experimental system, quite stable. The system currently runs under SunOS 5.4 but should not be hard to port to other operating systems.

1.1. An abstract machine

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.

The emulator 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 [10].

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 creating 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.

1.2. The execution state

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. The right to execute a goal or use 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 worker that executes the goal. This improves cache performance since the same cache-lines can be reused immediately. AKL terms can of course not be explicitly reclaimed and are therefore allocated on a *heap* that is subject 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. Garbage collection time normally stays well below 10% even for parallel executions where the garbage collector has a hard time keeping up with the overall increased performance.

2. The SimICS Simulator

When analyzing the behavior of a software system, 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 strategies to solving the first problem and the choice of which to use is the chief characteristics of the analysis method:.

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 [3].

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).

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.

Each of these three strategies place limits on how well we can solve the remaining two problems, handling system services and extracting statistics. Performance wise, host-supported simulation is faster than execution-driven, which in turn is faster than instruction-set simulation. The main performance penalty occurs between execution-driven and instruction-set strategies and can easily be several orders of magnitude. Conversely, instruction-set simulation is more flexible than execution-driven, which in turn is more flexible than host-supported simulation.

Restrictions inherent in host-supported simulation, other than sheer availability, is selection of what statistics to gather; the hardware support will dictate what studies can be done. Restrictions in common for all execution-driven approaches, that the authors have found, includes lack of support for one or more of: run-time generated code, multiple processes (i.e. workloads), multiple address spaces, system-level (operating system) code, determinism and interactive control of the execution. Approaches that relax these constraints tend to impose new restrictions on the type of programs that can be studied, including programming paradigms and/or source code modifications.

SimICS [7] [8] is an instruction-set simulator that has borrowed many design principles from g88 [2]. SimICS takes the brute force approach to all three problems mentioned at the beginning of this section. For this, SimICS take two penalties—first, we accept a lower performance than specialized approaches. This impact is today on the order of 5-10, or a slow-down of approximately 50 per simulated processor. Second, we 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.

SimICS provides full profiles, both of execution (instructions) and data cache events, and does so with a traditional debugger environment. This allows a detailed, interactive analysis of parallel programs.

The contribution of SimICS has been to achieve a competitive performance point while at the same time avoiding all the restrictions listed above for host-supported or execution-driven simulation.

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. 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 a peak sustainable read/write throughput of 500Mbytes per second.

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. Therefore, all simulated values in this paper assume a 4Kbyte first-level cache with 32-byte cache lines, and a 2Mbyte second-level cache with 64-byte cache lines, both direct-mapped.

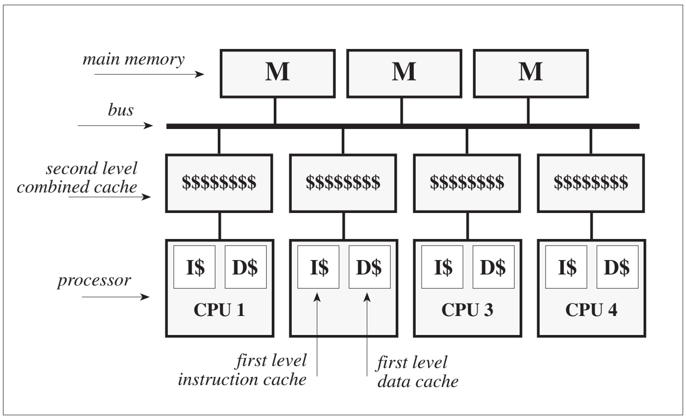

Figure 1: Generic shared-memory multiprocessor

4. Using SimICS

In this section, we will use SimICS to study Penny from various perspectives. As input to Penny we've selected a small number of AKL programs, listed in table 1. These are used to exercise Penny with a variety of problems. Though running on the same emulator, these programs trigger remarkably different behavior.

We begin by generating global statistics to characterize the input programs. In table 2, we've run SimICS on a base version of Penny, with the four example programs and two sizes of machines, 4 or 16 processors. The first line in the table is execution time as measured on the actual target machine, defined as total runtime minus initialization. These timings have a standard deviation of approximately 1.5%. The remaining numbers are reported by SimICS running the same workloads. The numbers are the sum of of all CPUs.

4.1. Detecting sequential components

We begin with running a very simple AKL program that solves the towers of Hanoi puzzle. Hanoi is an effective test of how well recursive definitions are executed. Very good speed-up is generally obtained for this benchmark if the sequential component is left out.

When Penny ran Hanoi with four workers under SimICS, simulating the first-level cache, we first looked for read cache and TLB misses. (A TLB miss generally causes a cache miss on our target machine, so these events are almost equivalent.) These misses easily stall the CPU if the result is needed soon, which is often the case.

It turned out that a single assembler instruction caused almost 14% of all read misses. The instruction profiling furthermore revealed that roughly half of all instructions executed were spent in the one enclosing line of C (specifically, 4 lines of assembler).

The implicated code was not part of the main Penny machinery but was a clean-up procedure that runs after an execution has completed. During execution Penny builds up a linked list of structures, one for each spawned goal. In the Hanoi program the list is only traversed at the end of the execution to determine that no suspended goals exist. Since the result already has been delivered, the time to traverse the list has never shown up during measurements, which invariably focused on the parallel portions. When the time to do the clean-up was reported as part of the execution time the existence of a performance bottleneck was obvious.

Application	Description
Hanoi	Solves the towers of Hanoi (with 16 disks). This program divides a size N problem into two size $N-1$ problems, ideal for parallelization. The generated solution is then counted, introducing a sequential component.
Life	Plays the game of life. Life involves very little computation, since the rules for determining each generation are very simple. It is heavy on communication, however, since each object needs to communicate with all the neighbors. Simulates 40 generations in a 20x20 world.
Trace	Ray-tracing. Renders a small image of a mouse made out of spheres. Very little communication since the rays can be calculated independently, but heavy on calculation.
Smith	DNA sequence matching using the Smith–Waterman algorithm. This is a relatively simple algorithm, but it is hard to parallelize because the synchronization is very fine-grained. The problem solved is matching two 400 elements long sequences against each other.

Table 1: Sample benchmarks used to exercise Penny

The problem existed for other programs as well, though slightly less dramatic. The `prof-weight` output in figure 2 is from running Smith with 6 workers monitoring the second level cache. The figure lists the the 5 most expensive (the cost of a TLB miss and a read miss was set to 5 and a a write miss to 1) blocks of instructions where each block is 32 bytes long.

The listing shows that almost 10% of the cost is found in the block starting at address $0x29868$. The assembler listing of the block in figure 3 gives us the:

1. number of write misses,
2. number of read misses,
3. number of TLB misses,
4. approximate number of times the instruction was executed and
5. whether (1/0) the instructions was executed at all.

As seen the `ld [ %i2 + 4 ], %o0` instruction on address $0x29868$ causes a read miss almost every time it is executed. The instruction correspond to the source code shown on line 479 in figure 4 where the profiling data has been added up for each source line.

The source code is from a procedure that is part 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 being spread across multiple processors, and would not have been a performance problem in a sequential version of Penny.

	penny-hanoi		penny-life		penny-trace		penny-smith	
	4 cpus	16 cpus	4 cpus	16 cpus	4 cpus	16 cpus	4 cpus	16 cpus
Exec time (ms)	1134	1166	1301	739	2373	923	4142	2083
Instr. count	93619500	263373500	159754500	140831500	416811500	365338500	772478500	728961500
Instr. footprint	624	704	1296	1361	2123	2405	2377	2653
TLB misses	6893	61267	482783	710159	108140	231670	94289	314461
Read count	16933754	58589306	29944621	31322773	75663119	77990473	123426208	135877087
L1 miss rate	5.35%	2.56%	9.75%	9.54%	8.43%	8.92%	5.83%	5.74%
L2 miss rate	0.56%	0.41%	0.88%	1.66%	0.06%	0.13%	0.45%	0.55%
Write count	6879330	9387005	13245667	13430186	33281214	33555036	45725860	45882835
L1 miss rate	5.64%	8.09%	10.31%	11.97%	7.18%	8.00%	6.68%	6.92%
L2 miss rate	1.65%	3.41%	2.55%	4.93%	0.94%	1.17%	0.92%	1.26%

Table 2: Sample benchmarks, real and simulated characteristics

```
(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

The procedure is executed during garbage collection and is responsible for distributing work among workers. The read misses in the procedure severely decreased the performance of the garbage-collector since the procedure is a part of a sequential phase of the garbage-collector.

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

4.2. Deciding on prefetching

A collection of abstract machine instructions are used to read and decompose AKL terms. A sequence of instructions are used for decomposing a term, the first instruction of which, called a *get* instruction will verify the type of the term. Subsequent instructions, called *unify* instructions will, in the case of a compound term, access the arguments of the term. In the case of a simple list term this results in three instructions—one for verifying that the term is a list tag and two for accessing the *car* and *cdr* of the list cell.

```
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

```
0   245   92   1000  4 for(tc = templa->tried; tc != NULL; tc = tc->next) {
0 58109 2423 380500 10  switch(tc->type) {
                         case DEAD_CELL:
                           break;
                         case FREE_CELL:
                           break;
                         case CHB_CELL: {
0    36    0    500  1     templc = tc->box.chb;
```

Figure 4: Source code listing of the most expensive block

Terms are represented by tagged pointers. A *get* instruction will first look at the tag of the term and only after having verified the tag follow the pointer to verify the functor. A unique tag is used for list terms, so in this case only the tag need to be verified.

We noticed an exceptionally high read miss rate in the *unify* instructions that accessed the components in a term. The reason 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.

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.

Figure 5: Improvements of the Smith benchmark

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.

4.3. How dangerous is a lock?

The Penny system uses locks in two different situations. The first is in the internals of the Penny system 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, *counters* were placed around the lock primitives. Counters are added by modifying the source code i.e. by adding SimICS macros that will turn the counters on and off.

We ran the Smith 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 (xmem) operations. In only one (!) case does a collision occur. The results were similar for all kinds of locks in the system.

```
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)

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.

4.4. Read and write operations

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 easily be explained.

Table 3 shows statistics gathered from executing the Smith benchmark on the improved Penny system. The reported numbers are from one of the CPUs. The cache statistics are for a 2Mbyte cache, i.e. our target machines second level cache.

The number of read and write operations is a good measure of the amount of "work" performed, and the figures in the table indicate an uncanny linearity in the load per processor, i.e. no significant overhead is added. Yet the speedup is not linear.

Table 4 shows the improvements for every doubling of number of workers. The execution time is reduced to 52% as (speedup 1.9) when going from one to two workers. The efficiency then drops and moving from eight to sixteen workers only reduces the execution time to 72% (speedup 1.39).

The explanation for the decreasing speedup cannot be found in the number of read or write operations or the total number of instructions executed. As seen the number of read and write operations per processor (this is statistics from one of the processors) is more or less cut in two for every doubling of workers.

	Number of workers				
	1	2	4	8	16
runtime (ms)	13822	7238	3996	2487	1789
read operations (x10^6)	124	59.5	29.8	15.2	7.74
read misses (x10^3)	8.66	121	71.7	43.6	29.0
miss rate	0.007%	0.204%	0.240%	0.287%	0.375%
write operations (x10^6)	47.5	22.3	11.1	5.60	2.81
write misses (x10^3)	138	143	80.9	52.8	34.0
miss rate	0.292%	0.643%	0.725%	0.942%	1.211%
Instr. count (x10^6)	610	308	156	81	42

Table 3: Profiling (of one CPU) and timing of Penny running Smith

	number of workers			
	1-2	2-4	4-8	8-16
runtime	0.52	0.55	0.62	0.72
read operations	0.47	0.50	0.51	0.51
write operation	0.47	0.50	0.50	0.50
instruction count	0.50	0.51	0.52	0.52

Table 4: improvements of operation count

The explanation is found in the number of read and write misses. Table 5 shows the same relative changes but for cache misses. When we go from one to two workers the number of read misses increases with a factor fourteen (!) and the number of write misses stays the same. At the same time the total number of read and write operations has been cut in half. This is the most likely explanation why the initial speedup is only 1.9 and not closer to 2.0 as indicated by the operation count.

The read and write misses does behave better once we use more than two processors but does not come close to the figures reported for the instruction counts. It should however be noted that the read and write misses are reduced to 66% or less while the execution time is reduced to 72%. The poor performance when using sixteen workers cannot be explained only by the number of read and write misses alone. This is investigated further in the next section.

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 even further.

	number of workers			
	1-2	2-4	4-8	8-16
read misses	14.0	0.59	0.61	0.66
write misses	1.04	0.57	0.65	0.64

Table 5: improvements of cache performance

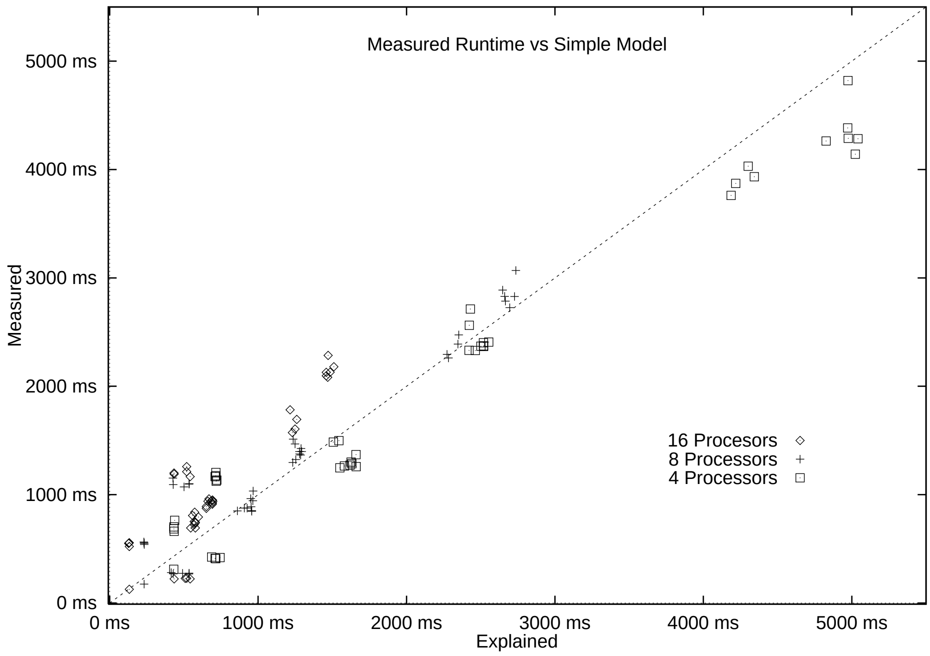

Figure 7: Results of predicting performance using a simple best-fit

4.5. Explaining overall Penny performance

The brief examples so far in this section 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.

We ran various combinations of Penny—using eight versions of Penny itself (with different improvements and modifications), the four benchmark inputs, and three levels of parallelism (4, 8, and 16 workers). We measured the median time out of 31 runs, giving us 128 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.

We next selected three variables that we presumed to be important for explaining performance, namely the number of memory reads, number of read misses to the first level cache, and the number of misses to the second level cache. A multiple regression of these variables against the database, and fudging, results in:

$0.1 * Reads + 0.33 * ReadMisses_{L1} + 10 * ReadMisses_{L2}$

In figure 7 we have plotted the "explained" time against the real 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, 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.

5. Concluding remarks

Instruction-level simulation is a very powerful tool when a parallel system is constructed. 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 perfect behavior with respect to the number of instructions executed and still not show linear speedup. Only when the cache misses are taken in to account can the performance of the system be predicted. The importance of good cache performance can not be overestimated. We have shown a strong correlation between the actual execution time and read operations. It idicates that a second level read miss is two order of magnitudes more expensive than a first level hit. This suggest 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 [4] but our approach is quite different. We have analyzed a the performance of a parallel system on an existing parallel architecture. The analysis is done not trough a generated trace file of selected read and write operations but from all operation actually performed by the system. The number of read and write operations performed by the larger benchmarks is over 100 million. The total number of instructions executed could be as high as 700 million. SimICS also allows 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.

6. Acknowledgments

The parallel implementation of AKL has been developed using the AGENTS 1.0 [6] system as a starting point. Haruasu Ueda did much of the implementation and analysis of the scheduler. Gallal Atlam and Kahyri Ali, designed and implemented the garbage collector [1].

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 Entreprise 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] Galal Atlam and Johan Montelius. Parallelization of Garbage Collection in a CCP System, Penny. Acclaim deliverable 4.1/3, SICS, June 1995.

[2] Robert C. Bedichek. Some Efficient Architecture Simulation Techniques. In *Proceedings of Winter '90 USENIX Conference*, pages 53–63, January 1990.

[3] 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.

[4] M. Hermenegildo and Tick E. Memory Referencing Characteristics and Caching Performance of AND-parallel Prolog on Shared-Memory Multiprocessor. *New Generation Computing*, 7(11):37–58, 1989.

[5] Sverker Janson. AKL A Multiparadigm Programming Language. Uppsala Thesis in Computing Science 19, SICS Dissertaion Series 14, Uppsala University, SICS, 1994.

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

[7] Peter Magnusson. A Design for Efficient Simulation of a Multiprocessor. In *Proceedings of MASCOTS*, pages 69–78, January 1993.

[8] Peter Magnusson and Bengt Werner. Some efficient techniques for simulating memory. Technical Report T94:16, Swedish Institute of Computer Science, August 1994.

[9] Johan Montelius and Khayri A. M. Ali. An and/or-parallel implementation of AKL. *New Generation Computing*, 13(4), December 1995.

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