

Enhancements and Applications of the SimpleScalar Simulator for Undergraduate and Graduate Computer Architecture Education

Naraig Manjikian
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada K7L 3N6
email: nmanjiki@ee.queensu.ca

Abstract—This paper describes enhancements and applications of the SimpleScalar simulator for undergraduate and graduate courses on computer architecture. The enhancements include additional instruction profiling statistics, visualization of data hazards, visualization of memory access behavior, and multiprocessor capabilities with visualization of cache coherence. The applications include prediction of pipelined execution time with stalls for hazards and cache misses, instilling deeper insight into cache locality, parallel programming, and a greater appreciation of cache coherence interactions for multiprocessor programs.

1 Introduction

Many simulation tools for computer architecture research and education have been developed in previous work [2, 3, 8, 9, 11]. Each tool has its own advantages and disadvantages in complexity, efficiency, and value as an educational aid. The SimpleScalar simulator [2], for example, is efficient and includes support for cache and superscalar simulation, and uses a pedagogically-appropriate instruction set derived from the MIPS ISA, but it does not support multiprocessor simulation.

This paper describes enhancements made to SimpleScalar for multiprocessing and several types of visualization, and outlines their application in undergraduate and graduate courses in computer architecture in Electrical and Computer Engineering at Queen's University. Background on courses is first provided, and then simulator enhancements are described for illustrating aspects of pipelining and the memory hierarchy. Multiprocessor enhancements are then described briefly with examples of usage. Conclusions and directions for future work are also provided.

2 Background on Courses

ELEC470 Computer System Architecture This senior *undergraduate* course covers performance, instruction set design, pipelining, caches, memory hierarchy, I/O systems, and multiprocessing. The textbook is by Hennessy and Patterson [5]. The course is lecture-based and does not include a laboratory component (prerequisite courses on digital logic, microprocessor systems, and digital systems engineering make use of programmable logic in laboratory work). A major portion of ELEC470 covers pipelining and memory hierarchy (including cache coherence), hence tools that aid understanding of these concepts have substantial value.

ELEC871 Shared-Memory Multiprocessors This *graduate* course covers shared-memory programming, cache coherence, memory consistency, performance studies, small-scale and large-scale multiprocessor architecture, case studies based on research and commercial implementations, and compiler issues. Selected chapters from the Culler/Singh/Gupta textbook [4] are used with material prepared by the instructor and research papers. Discussion of multiprocessors is made more concrete not only when students write and execute parallel programs, but also when the resulting cache coherence behavior can be studied in detail. Tools that aid in this respect therefore have considerable educational benefit.

Experience with Other Simulation Tools Many uniprocessor and multiprocessor simulation tools have been developed. Examples include SHADE [3], MINT [9], RSIM [8], SimpleScalar [2], SPIM [5], and DLXview [11]. These tools model architecture at varying

degrees of detail and have varying complexity and execution efficiency. Furthermore, the tools differ in whether they support multiprocessing.

The ELEC470 course has previously used DLXview, SPIM, and SHADE. DLXview visualizes pipeline activity, but is not suited for larger programs. SPIM is a functional simulator but does not model caches. SHADE provides instruction profiling and cache simulation for larger programs, but uses the SPARC ISA rather than the MIPS ISA. The ELEC871 course has previously used RSIM for multiprocessor simulation with different memory consistency models, but RSIM requires explicitly-allocated global shared memory and is not a visualization tool.

Both courses now use the SimpleScalar simulator that provides functional simulation, cache simulation, and superscalar simulation. The existing version of SimpleScalar does not support multiprocessing or provide graphical visualization (except for a post-execution trace for superscalar simulation). As part of research activity, however, SimpleScalar was enhanced with multiprocessor support. Other enhancements for visualization were also undertaken to benefit undergraduate and graduate courses on computer architecture, as outlined in the remainder of this paper.

3 Enhancements for Pipelining

Visualization of Data Hazards Courses on computer architecture often illustrate data hazards in pipelining with contrived examples. A more effective approach is to visualize the frequency of hazards in *real* code for a nontrivial program. The SimpleScalar simulator was enhanced with data hazard visualization in ideal pipelined execution, as shown in Figure 1. Register dependences for successive instructions are tracked and displayed with backward-flowing lines. The sequence of code includes subroutine call instructions and the associated stack push operations that demonstrate the frequency of data hazards. Furthermore, the `jal` instructions reveal the presence of a potentially subtle hazard: `jal` modifies the return address register (`$ra` or `$31`) and a subsequent `sw` instruction saves the return address on the stack.

Additional Profiling Data for Pipelined Performance Prediction SimpleScalar includes an instruction profiler that reports various statistics. Sample output for a nontrivial program is shown below:

```
sim_num_insn          75125799
sim_num_refs (load/store) 31609006
```

```
sim_inst_class_prof
#      index      count      pdf
load          26192181  34.86
store         5416825   7.21
uncond branch 1716987    2.29
cond branch   6422889    8.55
int computation 35376655  47.09
fp computation      167    0.00
trap           94     0.00
```

```
sim_branch_prof
#      index      count      pdf
uncond direct      677    0.01
cond direct        6422889  78.91
call direct         92777    1.14
uncond indirect    858191  10.54
cond indirect       0     0.00
call indirect      765342    9.40
```

Enhancements were made to the instruction profiler to collect and report additional statistics on taken branches and load-uses; sample output is shown below:

```
Summary of Taken Branches
=====
uncond direct      677
cond direct        2763305
call direct         92777
uncond indirect    858191
cond indirect       0
call indirect      765342
```

```
loads followed by uses: 11173509
```

The combined statistics can now be used to predict execution time in the absence of cache misses using the equation $T = n_i \cdot CPI \cdot t_c$ as follows:

$$f_{load-use} = \frac{1173509}{26192181}, \quad f_{mispredict} = \frac{2763305}{6422889}$$

$$\begin{aligned} CPI &= 1 + f_{load} \cdot f_{load-use} \cdot (1) \\ &\quad + f_{branch} \cdot f_{mispredict} \cdot (1) + f_{jumps} \cdot (1) \\ &= 1 + 0.3486 \cdot 0.427 + 0.0855 \cdot 0.430 \\ &\quad + 0.0229 \cdot 1 = 1.2085 \end{aligned}$$

Students in ELEC470 perform simulations to obtain the necessary statistics and quantitatively compare performance.

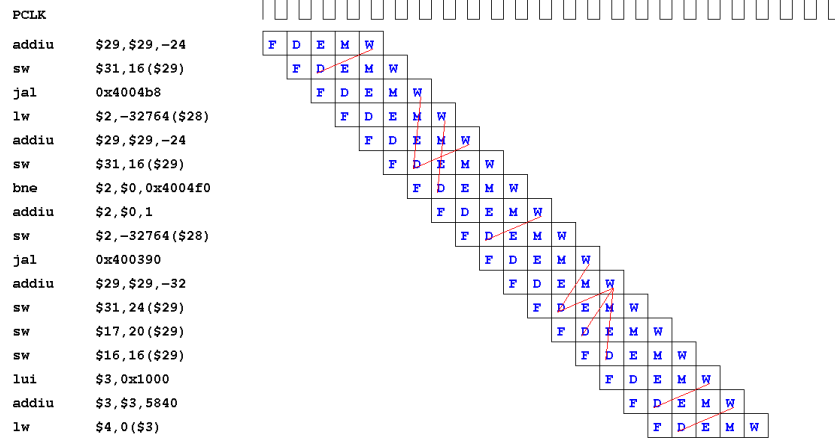
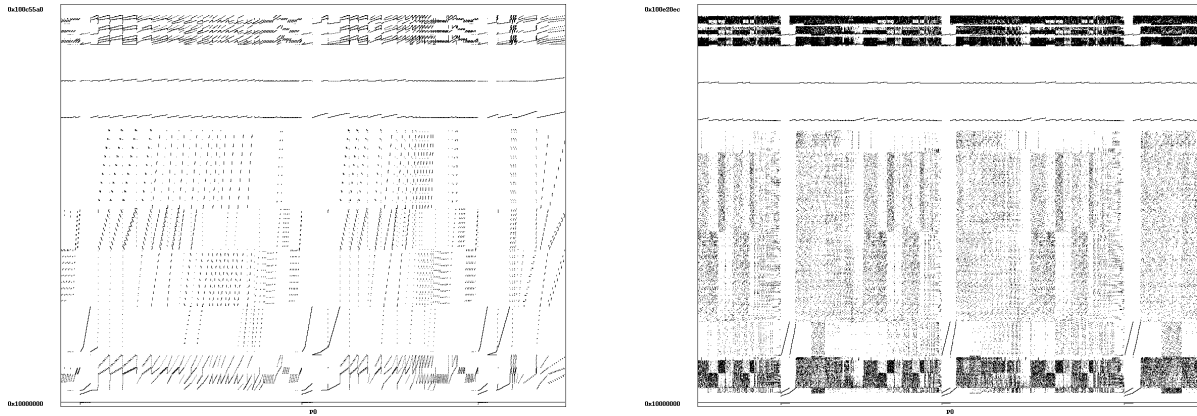


Figure 1: Sample output for visualization of data hazards



(a) s35932 logic circuit

(b) s38584 logic circuit

Figure 2: Visualization of memory access behavior for logic simulation program

4 Enhancements for Cache/Memory

SimpleScalar already includes a uniprocessor cache simulator, hence no significant enhancements were made for that component. The cache simulator produces detailed miss statistics for two-level cache organizations and also TLBs. These statistics reflect the locality characteristics of a program being simulated. What is lacking from a pedagogical perspective, however, is a means to instill a deeper appreciation of those statistics.

To enhance the educational benefit for the study of memory hierarchy, a dynamic visualization of program

memory access patterns was developed. This visualization can aid in understanding differences in cache miss rates obtained with the cache simulator. Sample output is shown in Figure 2. Time progresses along the horizontal axis, and memory addresses increase along the vertical axis. Each pixel represents a contiguous portion of memory whose size is determined by the extent of the data divided by the number of pixels in the vertical dimension. A pixel is set on the screen if the processor touches any location corresponding to that pixel in an interval of time. When the display reaches the right-hand side of the

```

statistic          PID 0    PID 1    PID 2    PID 3
Instructions       36710643 36360475 36399474 36392591
L1_accesses       9013453 8865631 8879753 8880025
L1_miss_ratio     35.79%  35.79%  35.73%  35.78%
L2_writebacks     1177565 1156227 1131910 1164427
L2_accesses      3225877 3172686 3172907 3177088
L2_miss_ratio     35.06%  36.44%  34.93%  34.86%
mem_writebacks    563071  585240  540420  552448
upgrade_requests  25922   23538  25901  24255
external_bus_requests 3445678 3393405 3438897 3441297
snoop_hit_ratio   2.61%   1.93%   2.36%   2.05%
excl_to_shrd_changes 13029   1165   10878   3807
shrd_data_responses 25955   27085  24605  27559
excl_data_responses 13090   1991   13126   2011
external_invalidations 26214   24147  24694  24225

===== Bus activity statistics =====

total_requests    4602671

read_requests     2869730 (62.35% of all requests)
  cache-to-cache xfers 105204 ( 3.67% of total_read_requests)
read_excl_requests 1633325 (35.49% of all requests)
  cache-to-cache xfers 30218  ( 1.85% of total_read_excl_requests)
upgrade_requests  99616  ( 2.16% of all requests)
total_mem_writebacks 2241179 (49.77% of read/read_excl)
sharing_writebacks 105204

```

Figure 3: Abbreviated output from multiprocessor cache simulation for ocean program in SPLASH-2

screen, the display is cleared and drawing recommences from the left-hand side. The display may be paused before being cleared to study the resulting access patterns during simulated execution of complete programs. The horizontal scale may also be controlled for broad views or closer examination of the spatial and temporal locality in data access patterns.

The output in Figure 2 is for a logic simulation program that was developed in earlier research [7]. The results are for two large logic circuits from the ISCAS benchmark [1]. The first circuit, *s35932*, results in a data miss rate of 16% for a 16-kbyte cache with 16-byte blocks. The second circuit, *s38584*, results in a higher miss rate of 25%. The memory access behavior shown in Figure 2 aids in understanding why the second circuit has a higher miss rate.

5 Enhancements for Multiprocessing

In order to expand the applicability of SimpleScalar, enhancements were made to provide support for multiprocessing. The necessary modifications include separate register files for each simulated processor, interleaving of simulated instruction execution from different

processors, support in the simulator for thread management and synchronization, and finally a run-time library for simulated multiprocessor programs. The PARMACS macro package [6] was adapted for use with the the run-time library, thus enabling the simulation of multiprocessor benchmark programs such as those in the SPLASH-2 benchmark [10].

The remainder of this section describes further enhancements to support multiprocessor cache simulation and visualization of cache coherence and memory accesses, provides an example used in courses, and finally shows visualizations for programs from the SPLASH-2 benchmark.

Multiprocessor Cache Simulation An additional enhancement was the integration of an efficient two-level cache simulator based on the MESI protocol with the new multiprocessor version of the simulator. SimpleScalar already includes a detailed uniprocessor multilevel cache simulator. Nonetheless, a custom cache simulator was appropriate for the multiprocessor enhancement in order to support a MESI cache coherence protocol across multiple caches, and to collect multiprocessor-related statistics (only data cache accesses are modelled currently). Sample output is shown in Figure 3 for the *ocean* program from the SPLASH-2 benchmark for a problem size of 130×130 on a 4-processor system with 8-Kbyte L1 caches and 256-kbyte L2 caches. This information is used, for example, in the ELEC871 graduate course to compute bounds and estimates of parallel execution time.

Cache Coherence Visualization Detailed statistics that are reported after execution has completed do reflect multiprocessor program behavior, but may not provide sufficient insight into the dynamics of cache coherence. To aid students in acquiring this insight, a dynamic graphical visualization of cache coherence was incorporated with the multiprocessor enhancements to SimpleScalar described earlier in this section. The multiprocessor cache simulator must already maintain the state of each cache line according to the MESI protocol. Visualization of cache line state changes during the simulated execution of parallel program is therefore a logical extension and a valuable educational tool. The visualization requires a representation of the caches and their states. Colored pixels serve well in this capacity. Each pixel represents a cache line, and its state may be invalid (black),

shared (green), exclusive (yellow), or modified (red). A white pixel is a location in the cache that has not been used. Finally, because of the two-level cache hierarchy, a cache line may be modified in the L1 cache with an out-of-date copy in the L2 cache. For this situation, an additional “modified-above” state (with a color of cyan) is used in the L2 cache.

It is also valuable to dynamically display the L1 and L2 cache miss rates for an interval of time (such as the last 1000 memory references) as a reflection of the variation in locality during program execution. Bar graphs are a convenient means of conveying this information.

Multiprocessor Memory Access Visualization An additional enhancement was to extend the dynamic visualization of uniprocessor program memory access patterns discussed in Section 4 for multiprocessing. A similar display format is appropriate (time horizontally and memory vertically), with multiple narrower displays to reflect the memory accessed by each processor rather than one single display for a uniprocessor. Examples later in this section will illustrate the resulting display.

5.1 Detailed Example Used in Courses

Senior undergraduate students first learn about multiprocessing and cache coherence at the end of ELEC470. Graduate students in ELEC871 have already learned basic concepts for multiprocessing and cache coherence in earlier coursework. To illustrate the most salient aspects of multiprocessing and cache coherence for undergraduate students at the end of ELEC470, and as an initial review for graduate students at the beginning of ELEC871, the standard dot product code example is used. Figure 4 shows the parallel version of this code for the multiprocessor version of the SimpleScalar simulator. The functions for thread creation and synchronization are defined in a run-time library that is linked with the code shown in Figure 4.

The cache visualization support allows more insight to be gained into the parallel program execution and the cache coherence interactions. Figure 5(a) shows the cache contents while the initialization loop in the `main()` function is being executed. Because both arrays are written, the majority of the data is modified in the L2 cache, and the most recent elements that are written are modified in the L1 cache and out-of-date in the L2 cache.

The latter situation is also referred to as the “modified-above” state. Due to mapping conflicts in the L1 cache, one of the arrays has more data in the “modified-above” state than the other array. The small area beneath the array data corresponds to accesses made to the stack region.

Following initialization, multiple threads are created, and each processor executes its loop to perform a partial sum for the dot product. Figure 5(b) shows the cache contents after all threads complete their execution. Because all array elements were written initially by processor P0, the other processors must retrieve modified cache lines for the arrays from the L2 cache of processor P0. Furthermore, the other processors only read the array data. Hence, the state of the cache lines from both arrays will become shared, with a copy retained by processor P0. There is one portion of each array that is accessed only by processor P0. This portion of each array remains in the modified state in the L2 cache of processor P0. Finally, processor P0 executes other code after all threads reach the barrier, resulting in additional data being loaded into the L2 cache.

5.2 Visualization for SPLASH-2 Programs

For the ELEC871 graduate course that explores shared-memory multiprocessors in detail, the study of application behavior benefits from the use of representative programs such as those found in the SPLASH-2 benchmark suite [10]. The Culler/Singh/Gupta textbook [4] that is used in the ELEC871 course also relies on these benchmarks as examples, and provides a great deal of statistical information in various chapters. Visualization can, however, bring these programs to life in a manner that raw statistics alone cannot match. To this end, the multiprocessor-enhanced version of SimpleScalar with visualization of cache coherence and memory access behavior is used as a pedagogical aid.

Figure 6 shows the cache coherence visualization output for the `LU`, `barnes`, and `water` programs from the SPLASH-2 benchmark. Figure 6 also shows the memory access patterns for each processor. Variation in L1 and L2 miss rates is reflected in the bars at the top of the cache displays in Figure 6; this provides additional insight into program behavior beyond a static overall miss rate reported at the end of simulated execution.

```

#define NUM_THREADS 8
#define N          10000

double dot_product, a[N], b[N];
Lock    the_lock;
Barrier the_barrier;

void    ParallelFunction ()
{
    int    i, start, end, thread_id = get_my_thread_id ();
    double local_sum;

    start = thread_id * N / NUM_THREADS;
    if (thread_id == NUM_THREADS - 1)
        end = N - 1; /* last thread goes to end of arrays */
    else
        end = (thread_id + 1) * N / NUM_THREADS - 1;
    local_sum = 0.0;
    for (i = start; i <= end; i++) /* compute partial dot product */
        local_sum += a[i] * b[i];
    lock (the_lock); /* atomically update final result */
    dot_product += local_sum;
    unlock (the_lock);
    barrier (the_barrier, NUM_THREADS); /* wait for all threads */
}

main ()
{
    int    i;

    for (i = 0; i < N; i++) /* initialize arrays with some values */
        a[i] = b[i] = (double) 1.0/(i+1);
    init_barrier (&the_barrier);
    init_lock (&the_lock);
    for (i = 1; i < NUM_THREADS; i++) /* create additional threads */
        create_thread (ParallelFunction);
    ParallelFunction (); /* main thread calls this directly */
    printf ("The dot product is: %g\n", dot_product);
    return 0;
}

```

Figure 4: Parallel version of dot product program

6 Conclusion

This paper has described enhancements to the SimpleScalar simulator for undergraduate and graduate courses on computer architecture. Visualization of data hazards, additional instruction profiling statistics, and vi-

ualization of memory access patterns are enhancements made to the uniprocessor simulator. New multiprocessor support was also added to SimpleScalar, followed by enhancements to perform multiprocessor cache simulation and coherence visualization. Memory access visualization was also extended to the multiprocessor version.

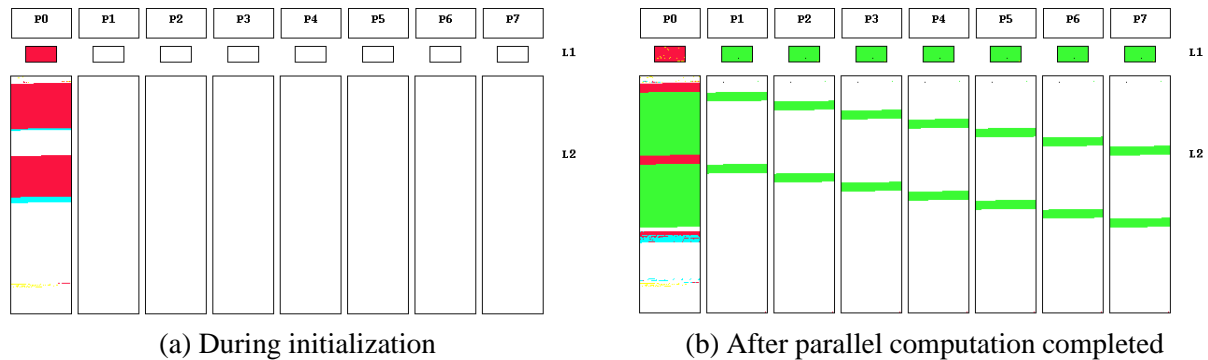


Figure 5: Visualization of multiprocessor execution and cache coherence for dot product

Future work includes extending the multiprocessor support to the superscalar version of SimpleScalar. Data hazard visualization may be extended to illustrate the effect of forwarding and load-use stalls as the resolution to the hazards. Visualization of branch prediction is also a possible enhancement, for example, by displaying both the contents of branch prediction tables and bar graphs on prediction accuracy.

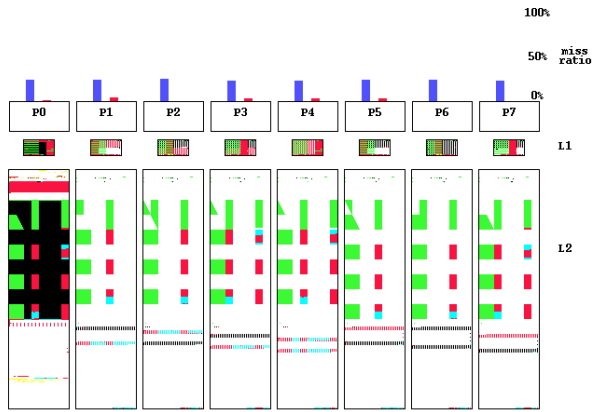
The SimpleScalar software, including the multiprocessor enhancements discussed in this paper, are available at the Website www.simplescalar.org.

Acknowledgements

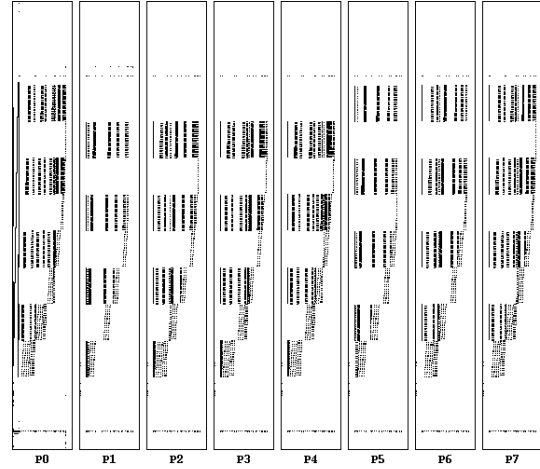
The multiprocessor enhancements described in this paper were initiated as part of research projects funded by Queen's University, Communications and Information Technology Ontario (CITO), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

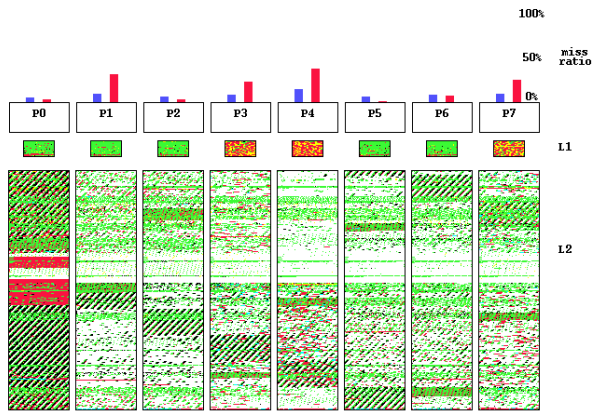
- [1] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1929–1934, May 1989.
- [2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Tech. Report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.
- [3] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. of the 1994 ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.
- [4] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA., 1999.
- [5] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, CA., 2nd edition, 1998.
- [6] E. Lusk et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.
- [7] N. Manjikian and W. M. Loucks. High performance parallel logic simulation on a network of workstations. In *Proc. 7th Workshop on Parallel and Distributed Simulation*, pages 76–84, San Diego, CA., May 1993.
- [8] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual—version 1.0. Tech. Report 9705, Dept. of Electrical and Computer Eng., Rice University, August 1997.
- [9] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proc. 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, Durham, NC, January 1994.
- [10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.
- [11] Y. Zhang and G. B. Adams. An interactive, visual simulator for the DLX pipeline. *Newsletter of the IEEE Computer Society Technical Committee on Computer Architecture*, September 1997.



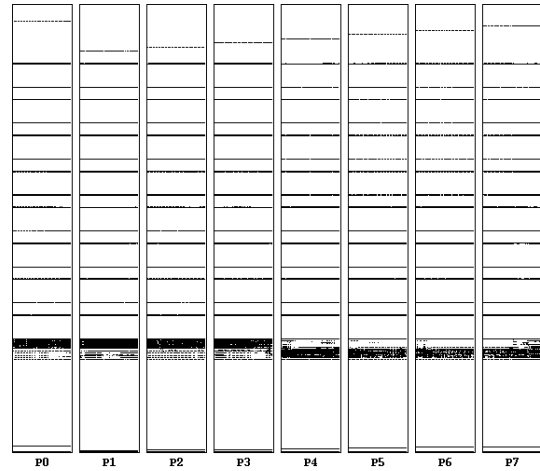
(a) LU 128×128 : cache coherence



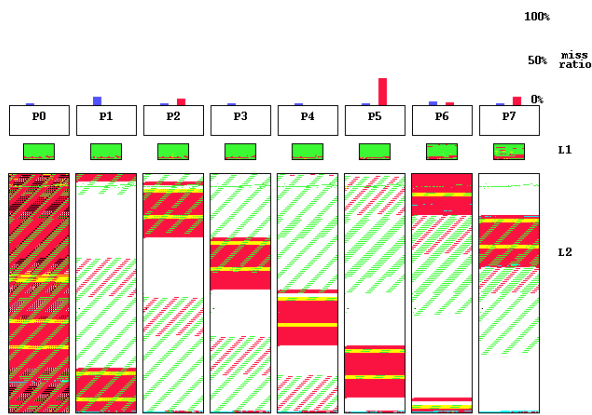
(b) LU 128×128 : memory accesses



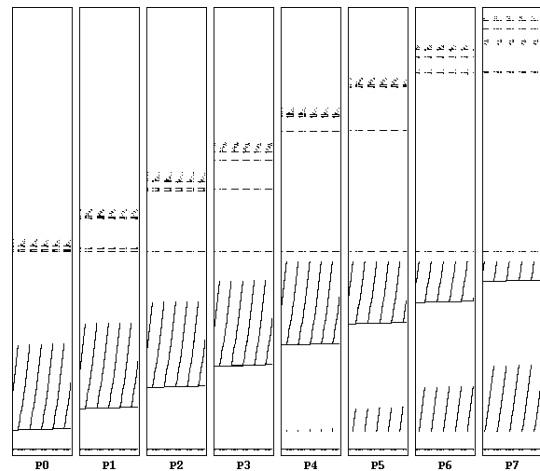
(c) barnes 2048 particles: cache coherence



(d) barnes 2048 particles: memory accesses



(e) water-nsquared 512 particles: cache coherence



(f) water-nsquared 512 particles: memory accesses

Figure 6: Visualization of multiprocessor execution and cache coherence for SPLASH-2 programs