Multiprocessor Enhancements of the SimpleScalar Tool Set

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 multiprocessor enhancements of the SimpleScalar tool set. The core simulation code has been modified to support multiprocessing, and a run-time library has been introduced for thread creation and synchronization. Measurements using the SPLASH-2 parallel benchmark suite [13] indicate that the multiprocessor enhancements introduce simulation overhead of 30%-50% relative to the original uniprocessor simulator. An idealized multiprocessor cache simulator has also been developed, and a dynamic visualization tool for cache coherence has also been developed. These multiprocessor enhancements are available at the WWW site for the SimpleScalar tool set.

1 Introduction

A number of *execution-driven* simulation tools have been developed to trace program execution and simulate computer systems with varying degrees of accuracy. Although some tools include support for simulating the execution of multiprocessor programs, others do not. *SimpleScalar* [1] is one of the many simulation tool sets that have been developed for computer architecture research. It has become a popular tool among the research community. For example, 9 of the 29 papers presented at the 27th International Symposium on Computer Architecture in June 2000 made use of SimpleScalar in some manner. SimpleScalar is efficient and flexible, and includes support for fast functional simulation, cache simulation, and detailed superscalar simulation. Since its original release, however, it has been a uniprocessor simulator and has not included support for multiprocessor simulation.

This paper describes enhancements for including multiprocessor support in the SimpleScalar tool set. The enhancements include fast functional multiprocessor simulation, as well as multiprocessor cache simulation to collect detailed statistics on cache coherence activity as well as misses and writebacks. The multiprocessor cache simulation also includes support for graphical visualization of multiprocessor cache contents and coherence activity during the course of parallel execution. Experimental results indicate that the overhead introduced by the modifications to support multiprocessor simulation can range from 30% to 50% for the fast functional simulator. The relative overhead is diminished in the more detailed cache simulator. Even with the graphical capability enabled, the simulation rate is approximately 300,000 instructions/second on a 333-MHz Sun Ultra/10 workstation.

The premise of this paper is that multiprocessor programs require thread creation and synchronization support, but such support need not require new instructions in simulation. Conse- quently, the existing system call mechanism in SimpleScalar i adapted to support multiprocessing. This enhancement is couple with a run-time library that insulates application programs (an the programmer) from the underlying details. The run-time librar provides straightforward functions to create threads and perfor synchronization with locks, barriers, and semaphores. These functions set register arguments and execute a system call instructio that is intercepted by the simulator in order to perform the low-level thread management or synchronization tasks. The run-tim library functions may be used directly in application programs Alternatively, programming interfaces such as the portable PARMACS macro package [5] may be supported using this library.

The enhancements are for SimpleScalar, version 2.0, using the portable instruction set architecture (PISA) that i derived from the MIPS ISA. The multiprocessor simula tor package is available at the SimpleScalar WWW site `http://www.simplescalar.org`. The package include the modified simulator code, the run-time library code that i linked with application programs, and sample programs to illus trate how applications are compiled and linked with the library fo multiprocessor simulation.

The remainder of this paper is organized as follows. Section describes related work. Section 3 outlines the basic multiproces sor support introduced in SimpleScalar. Section 4 provides per formance results to assess the overhead in the modified simulato Section 5 describes a multiprocessor cache simulator with visual ization of cache coherence behavior. Finally, Section 6 conclude the paper and suggests directions for future work.

2 Related Work

In the past decade, numerous simulation and instruction tracin tools have been developed for studying both uniprocessors an multiprocessors. Examples include SHADE [2], MINT [12], Aug mint [8], RSIM [9], SimICS/sun4m [6], SimOS [11], and Sim pleScalar [1]. These tools tools have varying degrees of efficienc related to the level of detail being modelled. Some tools such a SHADE [2] are designed to be highly efficient for rapid uniproce sor instruction profiling and cache simulation. Some tools such a MINT [12] serve as efficient front ends for providing uniprocess and multiprocessor memory reference streams. Some tools are in tended to model superscalar processors in detail at the expense long simulation times. RSIM [9] is one such tool that also mode a distributed shared-memory multiprocessor with different mem ory consistency models.

The SimpleScalar tool set [1] provides an efficient yet flexible simulation platforms for applications ranging from fast functional simulation to detailed (hence considerably slower) superscalar simulation. The only major feature that it lacks is multiprocessor support. The subject of this paper is the introduction of the necessary multiprocessor simulation support and its dissemination to the research community. It is worth noting that there is at least one other effort that has adapted SimpleScalar for multiprocessing, but this adaptation has not yet been released [10].

Characterization of multiprocessor programs is another important area of related work, as exemplified in the work of Woo *et al.* on the SPLASH-2 benchmark suite [13]. Their results characterize programs in terms of concurrency, communication-to-computation ratio, and locality. The results are obtained with an idealized multiprocessor cache simulator. It is argued by Woo *et al.* that nondeterministic behavior in multiprocessor execution makes comparisons difficult for different configurations, hence idealized behavior is valid for the sake of providing a reference.

Another area of relevance concerns performance prediction, tuning, and debugging tools, particularly those focusing on cache performance. Examples include Cprof [4] and MemSpy [7], among others. Such tools rely on simulation to gather detailed statistics on cache misses and correlate them to source code and program data. Systems with hardware monitoring now allow such statistics to be gathered during full-speed execution for off-line analysis with software tools [14].

A final area of relevance is the use of visualization and animation to study cache behavior. Previous work reported by Dongarra [3] has, for example, used these ideas for matrix-oriented algorithms by instrumenting programs to produce traces that are then fed to a post-execution visualization tool that specifically examines array data accesses. In contrast, the visualization of cache coherence behavior to be described in this paper applies to all data accesses and is performed as part of the simulation without requiring any modifications to the application program under study.

3 Adding Multiprocessor Support

Adding multiprocessor support to SimpleScalar requires a number of modifications and additions, including multiple register files, adapting the core simulation algorithm for multiple processors, and new support for thread creation and synchronization operations through system calls and a run-time library. The following paragraphs examine each of these areas in more detail.

Multiple Register Files SimpleScalar models the integer register file and floating point register file using one-dimensional arrays in the simulator code. Other registers such as the program counter are represented as scalar variables. To support multiple register files corresponding to multiple processors, it is necessary to introduce an additional array dimension. For example, the integer register file "`regs_R[SS_NUM_REGS]`" must be augmented with a new dimension, "`regs_R[MAX_PROCS][SS_NUM_REGS]`," in order to provide enough storage for the state of multiple processors. A similar modification is made to other register declarations. The consequences of these changes are significant: all locations throughout the code where these register data structures are accessed must be modified to use the current processor identifier as an additional array index. Such modifications naturally introduce execution overhead in the simulator to perform the necessary array indexing. The extent of this overhead will be characterized in the experimental results to be reported in Section 4.

Execution Strategy for Core Simulation Algorithm The existing core simulation algorithm in SimpleScalar consists of a loop that retrieves instructions for a single thread of execution and simulates their effects on registers and memory. Simulation of multiple processors on one physical processor requires interleaving operations on different register sets. Because the total number of parallel threads of execution may not be known in advance, this interleaving of instructions must dynamically accommodate new threads as they are created. Finally, simulation of parallel programs with synchronization constructs will also occasionally require temporarily suspending the simulated execution of a given thread until the necessary synchronization has been performed.

The modifications necessary for adapting the core algorithm for multiprocessing include a counter for the number of created threads of execution (which corresponds to the number of simulated processors), an array of flags to reflect the state of each simulated processor (active or inactive), and a new inner loop that steps through all active processors in order to fetch and interpret one new instruction for each processor. The resulting code to support multiprocessor simulation is outlined in Figure 1. Each element in the `active[]` array is a flag that indicates whether a given processor can proceed with executing the next instruction, i.e., the processor is not blocked on a synchronization event. Note that the inner loop steps through all created processors, a number that increases as new threads are forked during parallel execution.

New System Calls for Multiprocessing The `syscall.c` module in SimpleScalar defines a collection of services related to memory management, file system input/output, and other miscellaneous functions. Each service is identified by a predefined system call request code that is passed in simulated register $2 at the time that the service is requested through the simulated execution of a `syscall` instruction. The `switch` statement in the main loop of the simulation algorithm shown in Figure 1 interprets this `syscall` instruction and causes the `ss_syscall()` function in the `syscall.c` module to be invoked. This function examines the value in simulated register $2 in order to select the appropriate operation. Additional information needed to perform the desired operation is passed in other simulated registers.

To incorporate services for supporting parallel execution, new system call codes have been defined for thread creation and termination, initialization of synchronization variable, and synchronization operations. For each of these new codes, corresponding cases have been added in `syscall.c` to perform the required tasks related to thread management and synchronization.

Thread Creation and Multiple Stacks Parallel execution requires thread creation, hence simulated parallel execution must

```
while (TRUE)
{
  for (pid = 0;
       pid < num_created_processors; pid++)
  {
    /* skip if inactive */
    if (! active[pid])
      continue;

    inst = <fetch instr. with regs_PC[pid]>;

    /* assume no branch */
    next_PC = regs_PC[pid] + SS_INST_SIZE;

    switch (SS_OPCODE(inst))
    {
       /* jumps/branches change 'next_PC' */
       /* 'pid' identifies register set */
       case <opcode1>:
         <perform operations for opcode1>
         break;
       case <opcode2>:
         <perform operations for opcode2>
         break;
       .
       .
       .
    }

    /* save updated PC for this processor */
    regs_PC[pid] = next_PC;
  }
}
```

Figure 1: Modified simulation algorithm

also support this feature. The information necessary for creating a new thread consists of the initial instruction address and the initial stack pointer value. The system call to create a new thread supplies the initial instruction address; this address will be the initial value of the program counter register for the new thread. The initial stack pointer address may be provided explicitly with the system call, or it may be set automatically. For simplicity, the latter option is chosen for SimpleScalar.

An important consideration for managing multiple threads is allocating sufficient stack space for local data in subroutines executed by different threads. In the SimpleScalar portable instruction set architecture for *uniprocessor* execution, the stack convention specifies that the initial top of stack is at virtual address 0x7FFFC000, whereas the starting address for static data is 0x10000000. Heap storage begins immediately after the static data. The stack may grow downward until it meets the end of the heap data area; there is approximately 1.8 Gbytes of space between addresses 0x10000000 and 0x7FFFC000.

For *multiprocessor* execution, however, there is no specific convention, except that all data and stack storage must still reside within the range 0x10000000 to 0x7FFFC000. A new convention is therefore adopted for the multiprocessor enhancements described in this paper whereby new threads are assigned stack space below the arbitrarily-chosen address 0x70000000. In this manner, the main thread may have up to 256 Mbytes of stack space. If a different value is desired for the maximum stack space in the main thread, a different address other than 0x70000000 may be chosen.

The maximum stack size for other dynamically-created threads may be specified with a parameter s. Thus, the first new thread will be assigned the region from (0x70000000$-s$) up to 0x6FFFFFFF for its stack, with its initial stack pointer value set to 0x6FFFFFF0. The next thread will be assigned the region from (0x70000000-2 s) up to (0x70000000$-s-1$), and so on. The current implementa- tion uses a value of $s = 1$ Mbyte. The heap data may grow upwar from 0x10000000 to the beginning of the thread stack space (i.e. the stack space for the most recently created thread). Note that *al* memory contents—code, static and heap data, stack data—may be shared among the threads in this multiprocessor model. Thus, any thread may access data on the main thread stack to provide the se- mantics of a cactus stack. Less logical, but nonetheless feasible in this model, is the sharing of stack data between threads other than the main thread. The validity of any stack data being share depends, of course, on proper programming and synchronizatio to ensure that the values on the stack are in scope.

Maintaining the State of Synchronization Variables Synchro- nization operations involving locks, barriers, and semaphores are invoked through system calls. Modelling the effects of synchro- nization requires maintaining the state of each synchronizatio construct in the simulator (i.e., whether a lock is free, whether any threads are blocked on a semaphore, or whether all threads have arrived at a barrier). New code has been introduced to check an update the state of synchronization variables, and alter the statu of a thread between active and inactive, as appropriate.

When the application program invokes a run-time library func- tion to initialize a lock, semaphore, or barrier variable, it is as- signed a unique internal identifier that is passed back to the appli- cation program. This identifier is an index into an array for each type of synchronization variable. Subsequent synchronization re- quests from the application must be made with this identifier.

Each element in an array of synchronization variables main- tained by the simulator is a pointer to a queue of threads. For sim- plicity, each thread is statically assigned a node that may be presen in exactly one of these queues. This approach places restriction on threads in that only one lock may be held at any given time, an a thread may not become blocked on a semaphore or barrier whil a lock is being held. These restrictions should not present difficul- ties for typical multiprocessor programs where locks are used i the conventional manner for short critical sections.

Run-time Library Support for Thread Creation and Termina- tion The modifications to `syscall.c` in SimpleScalar permi the creation of a new thread through a system call, given the ini- tial instruction address (stack space is allocated automatically, a described above). Application programs normally invoke a librar routine to create a new thread, passing a function pointer as a pa- rameter. The expectation is that when the thread completes th execution of this function, it terminates automatically. Compiler generated code for such functions, however, normally includes a return-from-subroutine instruction at the end. Hence, some mean

```
typedef int Lock;
typedef int Barrier;
typedef int Sema;

void init_lock (Lock *lock_ptr);
void init_barrier (Barrier *barrier_ptr);
void init_sema (Sema *sema_ptr,
                int initial_sema_count);

void lock (Lock lock_id);
void unlock (Lock lock_id);

void barrier (Barrier barrier_id,
              int num_threads);

void sema_wait (Sema sema_id);
void sema_signal (Sema sema_id);

void create_thread
        (void (*function_ptr)(void));

int  get_my_thread_id (void);
```

Figure 2: Run-time library function interfaces

of properly terminating the execution of the thread is necessary to preserve the desired semantics of parallel execution.

The solution implemented for SimpleScalar is to provide a run-time library routine that accepts a function pointer from the user program and invokes a system call to create a new thread. The system call passes not only the user-supplied function pointer to `ss_syscall()` in the simulator code, but also a pointer to a special "wrapper" function that is part of the run-time library, but not visible to the programmer. The code in `syscall.c` arranges for the new thread to begin executing the wrapper function, rather than the user-specified function. The initial register contents for the new thread include the user-supplied function pointer as a normal argument to the wrapper function, which then performs a regular function call using this pointer. After a normal return from the user-specified function, the wrapper function invokes a system call to terminate the thread. In this manner, no specialized compiler support is required for threads; the necessary functionality is encapsulated entirely in `syscall.c` and a simple run-time library.

Run-time Library Support for Synchronization Additional run-time library functions are included to invoke system calls for initializing synchronization variables and performing synchronization operations. These system calls cause the new cases in the `syscall.c` module of SimpleScalar to be executed. If the calling thread must be suspended to model synchronization, the code in `syscall.c` marks the thread as inactive. Consequently, the remaining instructions after the `syscall` instruction will not be executed, and the run-time library function will not yet return. A later synchronization operation performed by another thread will ultimately unblock the suspended thread and mark it as active again. The next time that the inner loop shown in Figure 1 visits the reactivated processor, the execution of the run-time library function will resume, and there will ultimately be a return to the simulated application code that called the library function.

Interfaces for Run-time Library Functions The current collection of run-time library functions is shown in Figure 2. Each type of synchronization variable has an initialization function whose purpose is to set the variable in the application program with a unique identifier value for later use. The synchronization functions are the conventional ones and do not need further explanation. To create a new thread, only the pointer to a user-supplied function is required; proper thread termination is managed internally by the wrapper function in the run-time library. Finally, any thread may obtain its unique thread identifier whose value is in the range 0 to (*num_threads*−1).

4 Simulator Performance

To measure the overhead introduced by the modifications for supporting multiprocessor simulation in SimpleScalar, experiments were conducted with the `ocean` program from the SPLASH-2 benchmark suite [13]. The `ocean` program uses a multigrid algorithm to solve difference equations for modeling ocean water circulation. This program has two versions, and for the experiments reported here, the version with non-contiguous partitions was used (the difference between versions is not important for functional simulator performance).

All of the experiments were conducted on an unloaded Sun Ultra10 workstation with a 333-MHz UltraSPARC-IIi processor, 2 Mbytes of secondary cache memory, and 128 MBytes of main memory. The `gcc` compiler, version 2.8.0, was used with optimization level -O3 to generate native SPARC code for the simulators. The `gcc` compiler, version 2.6.3, was used with optimization level -O2 to generate SimpleScalar code for the application programs to be simulated.

A *uniprocessor* version of the program (i.e., an executable with no calls to thread creation or synchronization functions) was generated in order to measure the performance of the existing `sim-fast` simulator in SimpleScalar. Instruction counting was disabled for somewhat higher efficiency (uniprocessor instruction counts were obtained with a separate simulator that had counting enabled). The simulation rate for the uniprocessor application program using `sim-fast` is then given by the ratio of the number of simulated instructions and the execution time.

The *multiprocessor* version of the application program was then simulated on 1, 2, 4, 8, and 16 processors using the new multiprocessor `sim-mpfast` simulator. Instruction counting was *enabled* for all of these multiprocessor simulations. The `sim-mpfast` simulator reports not only the total number of instructions simulated by all processors, but also reports the number of instructions executed by each processor.

The simulator performance results for `ocean` are provided in Table 1. All of these results were obtained with a problem size of 130×130 and all other parameters at the default settings. When comparing the simulation rates, the base `sim-fast` simulator with instruction counting disabled is only 36% to 56% faster than the `sim-mpfast` simulator with instruction counting enabled. The differences in rates are illustrated graphically in Figure 3. Clearly, overhead must be incurred from introducing support for multiprocessor simulation (e.g., an additional array dimension for

Table 1: Instruction counts and exec. times for `ocean`

	Total instrs.	Time (sec)	Sim. rate (instr/sec)
sim-fast	144,195,791	25	5,767,832
sim-mpfast -p1	144,203,990	39	3,697,538
sim-mpfast -p2	144,455,617	37	3,801,463
sim-mpfast -p4	145,881,735	35	4,052,270
sim-mpfast -p8	146,586,998	34	4,188,199
sim-mpfast -p16	152,537,382	35	4,237,149

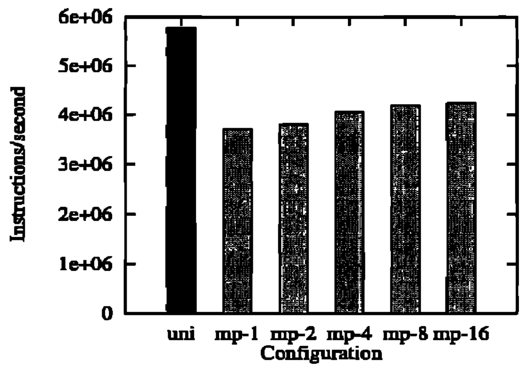


Figure 3: Simulation rates for `ocean`

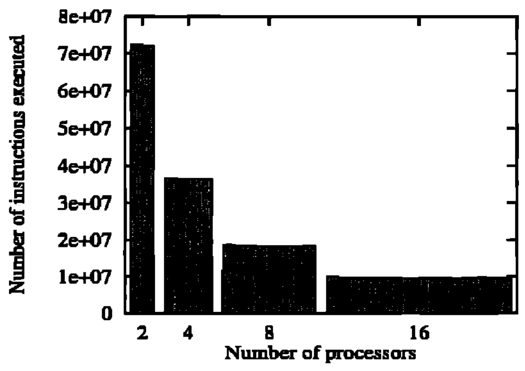


Figure 4: Load balance for `ocean`

register data structures and extra instructions for array indexing). Nonetheless, the overhead is not excessive, and when either of these base simulators is used as the front end for a detailed back-end simulator that models caches, memory latencies, and interconnect behavior, the difference becomes less significant.

To compare simulated execution performance using SimpleScalar with native execution performance, the `ocean` program was compiled for native execution on the Sun Ultra10/333 using `gcc` version 2.8.0 and optimization level -O2. The measured execution time with a problem size of 130×130 was 1.13 seconds. The number of SPARC instructions executed natively was determined by tracing the program with the SHADE [2] tracing tool, which reported a total of 164,151,503. Hence the native execution rate is approximately 145 million instructions/second, or 25 times faster than `sim-fast`.

It is also instructive to examine the load balance in simulated parallel execution. Using the output from the new `sim-mpfast` simulator that provides the number of instructions executed by each processor, the results are shown graphically for each simulation run in Figure 4. Clearly, the load balance remains good a more processors are used, and the parallel efficiency is high.

4.1 Detailed Investigation of Overhead

An interesting feature that is clearly evident in Figure 3 is the ir crease in the simulation rate as more processors are used. Thi trend would, on the surface, appear to be counterintuitive, as would be reasonable to expect that with a larger amount of state t maintain for more processors, the simulation would incur a highe data miss rate on the physical processor executing the simulation

There is, however, an explanation of the observed trend in pe formance. Analysis of the assembly-language output from th `gcc` version 2.8.0 compiler on the Sun Ultra10 shows that the na tive SPARC code for the nested loop structure in Figure 1 become more efficient as more processors are simulated. The reason is th there is a short sequence of 21 SPARC instructions that are onl executed as initialization prior to entering the inner loop in Fi ure 1. With more processors, the time needed for this initializatic code is amortized across more loop iterations (i.e., more proce sors), resulting in greater simulation efficiency. The same initia ization code is generated for all optimization levels of the `gc` compiler. A similar sequence of initialization code is generated b the Sun native compiler, version 4.2, at optimization level -xO4.

To confirm the explanation above, the SHADE [2] tracing to was used to *trace the SPARC instructions that would be execute in order to simulate the SimpleScalar instructions* for the `ocea` application with a problem size of 130×130. This "simulatic of a simulation" revealed that a total of 15,422,818,313 SPAR instructions would be executed for a 1-processor simulation the multiprocessor `ocean` application that interprets 144,207,33 SimpleScalar instructions. Because only one processor is use the number of times that the inner loop in Figure 1 is entered given by the total number of instructions. As a result, the inn loop overhead is given by $(144,207,336 \cdot 21)$=3.03 billion instru tions. In contrast, SHADE revealed that a *16*-processor simulatic required only 12,566,448,420 SPARC instructions to interpret a t tal of 152,540,035 SimpleScalar instructions for all 16 processor With parallel execution, the inner loop is entered only 9,934,6(times, as reported by `sim-mpfast`. Consequently, the overhea is given by $(9,934,602 \cdot 21)$=0.209 billion instructions. The diffe ence in the overhead is $3.03 - 0.209 = 2.821$ billion instruction The difference between the *total* number of SPARC instructions $15.422 - 12.566 = 2.856$ billion instructions, which agrees we with the difference in overhead, and therefore provides confirm tion of the explanation given for this trend.

5 Applications

5.1 Multiprocessor Cache Simulation

Cache simulation for uniprocessors requires modelling a tag arr and maintaining statistics on hits and misses. Generalizing cacl simulation for multiprocessors based on a snooping bus requir not only maintaining multiple tag arrays, but also modelling i teractions such as invalidations and cache-to-cache transfers. Fu thermore, when modelling multilevel caches, additional comple

```
statistic                   PID 0     PID 1     PID 2     PID 3

L1_accesses                 9013419 8865631 8879753 8880025
L1_stores                   1718436 1632550 1632209 1632081
L1_hits                     5766532 5670567 5686535 5679854
L1_misses                   3225863 3172686 3172907 3177088
L1_miss_ratio               35.79%  35.79%  35.73%  35.78%
L1_upgrades                 21024   22378   20311   23083
L1_upgrade_miss_ratio       0.23%   0.25%   0.23%   0.26%
L2_writebacks               1177565 1156227 1131910 1164427
L2_accesses                 3225863 3172686 3172907 3177088
L2_hits                     2089908 2015371 2059021 2068383
L2_misses                   1131057 1156155 1108296 1107533
L2_miss_ratio               35.06%  36.44%  34.93%  34.86%
L2_upgrades                 4898    1160    5590    1172
L2_upgrade_miss_ratio       0.65%   0.71%   0.64%   0.73%
mem_writebacks              563071  585240  540420  552448
excl_to_mod_changes         583562  582997  580879  583003
read_requests               707565  743560  711151  707440
excl_data_returns           671273  696659  667425  666264
read_excl_requests          423492  412595  397145  400093
upgrade_requests            25922   23538   25901   24255
external_bus_requests       3445678 3393405 3438897 3441297
snoop_hits                  89931   65386   81208   70614
snoop_hit_ratio             2.61%   1.93%   2.36%   2.05%
excl_to_shrd_changes        13029   1165    10878   3807
shrd_data_responses         25959   27085   24605   27559
excl_data_responses         13090   1991    13126   2011
external_invalidations      26214   24147   24694   24225

===== Invalidation set size statistics =====

#procs   frequency
0        1605073
1        126591
2        924
3        353

avg. invalidation set size = 0.07

===== Total bus activity statistics =====

total_requests              4602657

read_requests               2869716 (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)

read & read_excl requests           4503041 (97.84% of all requests)
total_mem_writebacks                2241179 (49.77% of read/read_excl)
sharing_writebacks                  105204
writebacks per read/read_excl       0.52
writebacks per bus request          0.51
```

Figure 5: Output from multiprocessor cache simulation

ity stems from having to model multi-level inclusion with different cache line sizes for each level.

The multiprocessor enhancements to SimpleScalar described in this paper were augmented further with customized multiprocessor cache simulation code to model the aspects described above. A MESI protocol is modelled. All instructions and memory access are still assumed to take only one cycle, hence the details of bus contention and arbitration are not simulated. Previous work [13] has taken a similar approach for multiprocessor cache simulation. The approach is valid because the nondeterminism that is inherent in multiprocessor execution already affects the statistics. Idealized multiprocessor cache simulation still provides a useful characterization of multiprocessor execution and coherence interactions.

Numerous statistics are maintained in the multiprocessor cache simulation code for processor-side and bus-side actions. Sample output is given in Figure 5 for the `ocean` program with a problem size of 130 × 130, L1 cache size of 8 kbytes, L2 cache size of 256 kbytes, and a common cache line size of 16 bytes. Similar experiments using a line size of 64 bytes for the L2 cache—while still enforcing inclusion for 16-byte lines in the L1 cache—result in the expected reduction in L2 miss rates and bus transactions.

5.2 Cache Coherence Visualization

A further enhancement to the multiprocessor cache simulator described in Section 5.1 is the use of graphics to visualize cache coherence. An abstract representation of processors and caches is depicted in a window on the screen. Caches are shown as rectangular boxes, and each cache line is represented by a pixel. Different pixel colors represent cache line states (i.e., red=modified, green=shared, yellow=clean-exclusive, black=invalid, white=unused). In simulated execution, changes in cache line states in all affected caches are reflected with color changes to animate the dynamics of cache coherence.

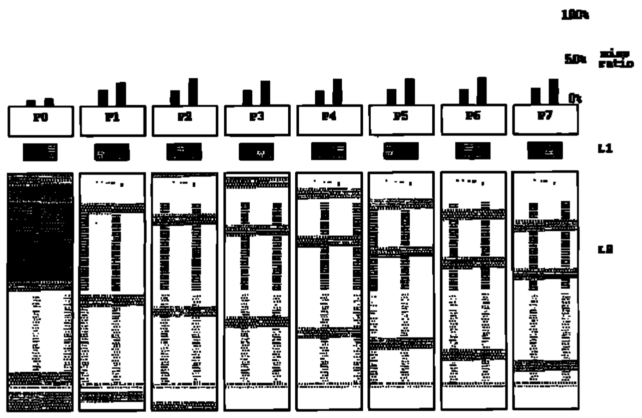
Figure 6 illustrates snapshots of this dynamic visualization for two kernels from the SPLASH-2 suite. Figure 7 provides similar snapshots for three SPLASH-2 applications. All of these views are for 8-processor simulations with L1 caches of 8 kbytes, L2 caches of 256 kbytes, and a line size of 16 bytes. The images provide striking contrasts in cache storage utilization, data access patterns, data contiguity, and data sharing. The `fft` kernel, for example, exhibits the characteristics of the blocked transpose algorithm. Blocking is also clearly evident for the `lu` kernel. For the particle-based `barnes` and `fmm` applications, there is some regularity underlying the otherwise random appearance in the data access patterns. The differences between the contiguous and non-contiguous versions of `ocean` are clearly evident. The insights gained from these visualizations can then be combined with detailed characterization of the type reported by Woo *et al.* for an enhanced understanding of application behavior.

Two bars are also shown for each processor to reflect the L1 and L2 miss ratios for the preceding n memory references by that processor (i.e., it is *not* the cumulative miss ratios since the beginning of execution). Over the course of execution, it is possible to observe different phases with frequent L1 and L2 misses, frequent L1 misses but few L2 misses, or few L1 misses but frequent misses for the few times that the L2 cache is accessed.

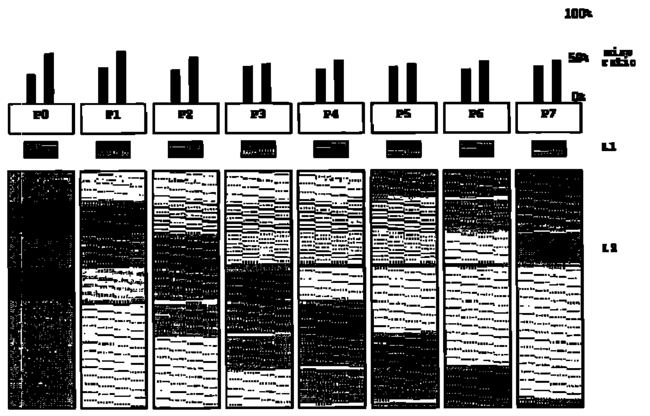
Finally, as an aid in tracking cache accesses within one processor, an additional *modified-above* cache line state is used in the L2 cache. If a cache line has been modified in the L1 cache, and the updated data has not yet been written back to the L2 cache, the line is marked as *modified-above* in the L2 cache to indicate that it is "stale" data and a different color (cyan) is used to reflect this state. Thus, as the processor sweeps through a large array to modify its contents, for example, a clearly-distinguishable block of a different color sweeps through the L2 cache. As modified L1 lines are replaced and written back to the L2 cache, the color for these lines in the L2 cache reverts to that for the normal modified state (red).

5.3 Performance with Graphics

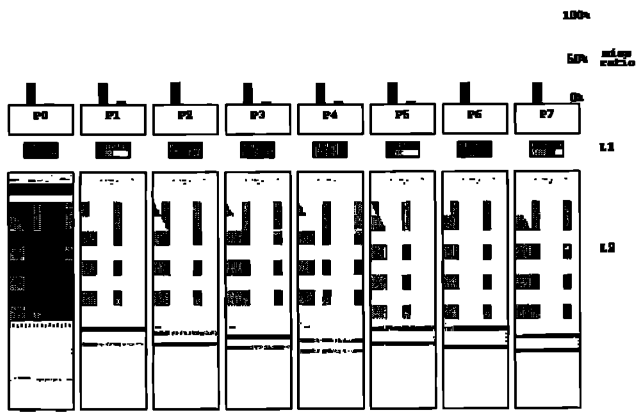
The use of graphics increases the overhead relative to the fast functional multiprocessor simulation provided by the enhancements described in Section 3. For comparison, Table 2 presents performance results for the applications of multiprocessor simulation described in this section. The results are for the `ocean` program on 4 processors with a problem size of 130 × 130, and the simulations were executed on a Sun Ultra10/333 workstation. Results are also provided for the uniprocessor version of `ocean` with the same problem size running under the functional, cache, and superscalar simulators that are provided in the original uniprocessor version of SimpleScalar. All results reported in Table 2 are for simulations that model only the data caches with sizes of 8 kbytes



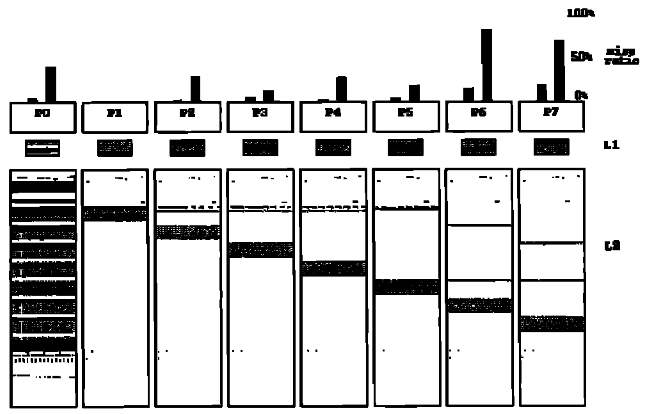
(a) fft 2^{12}



(b) fft 2^{14}



(c) lu, non-contiguous, 128 × 128



(d) lu, contiguous, 128 × 128

Figure 6: Cache coherence visualization for SPLASH2 kernels

Table 2: Comparison of simulation performance

	Instr.	Time (sec)	Instr./sec
sim-mpfast -p4	145,881,735	35	4,052,270
sim-mpcache -p4	145,862,950	60	2,431,049
sim-mpcache -p4 with graphics	145,884,856	499	292,354
sim-fast	144,195,791	25	5,767,832
sim-cache	144,177,153	109	1,322,726
sim-outorder	144,179,328	1,362	105,859

for L1 and 256 kbytes for L2. In this manner, there is no additional overhead for modelling TLBs or instruction caches. The intent is to provide a broad performance comparison across varying levels of detail, both with and without the use of graphics. The results indicate that the execution time for the idealized multiprocessor cache simulation with graphical animation is quite reasonable.

6 Conclusion

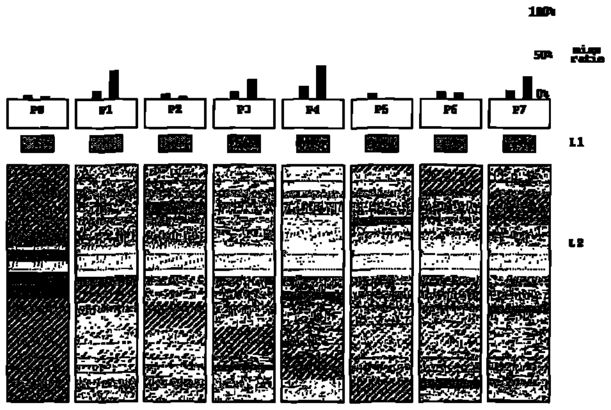
This paper has discussed the addition of multiprocessor support in SimpleScalar for functional simulation. The enhancements for multiprocessing include multiple register files, a new inner loop in the core simulation algorithm, new system calls for multiproces ing, and run-time library support for simulated application pr grams. Experimental results indicate that the simulation overhe for functional multiprocessor simulation is 30%-50%, and that tl overhead is reduced as more processors are used. The applic tions developed using this enhanced version of SimpleScalar i clude multiprocessor cache simulation and cache coherence vis alization. The software for the multiprocessor enhancements d scribed in this paper is available from the SimpleScalar WWW si at `http://www.simplescalar.org`.

Acknowledgements

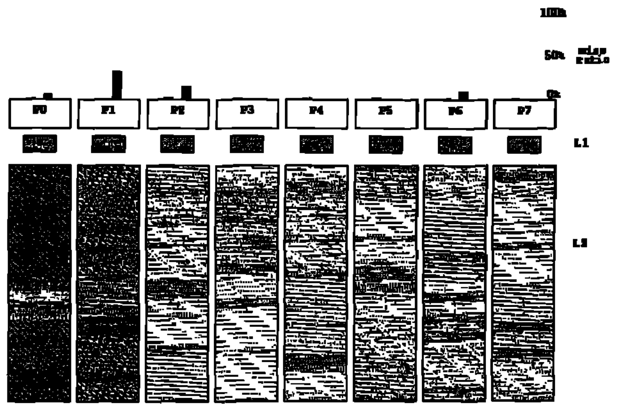
The production of the software described in this paper was su ported in part by funding from Queen's University, Communic tions and Information Technology Ontario (CITO), and the Natu Sciences and Engineering Research Council of Canada (NSERC

References

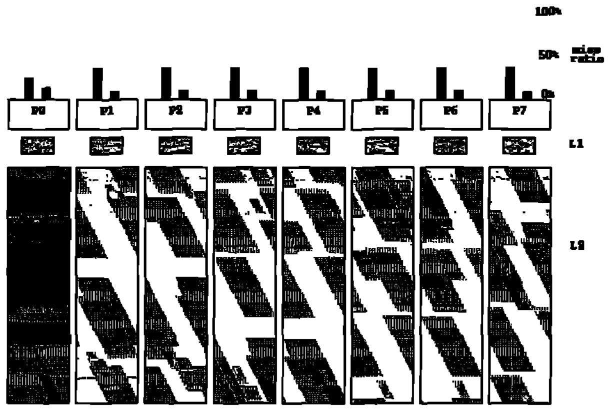
[1] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2 Tech. Report 1342, Computer Sciences Department, University Wisconsin-Madison, June 1997.



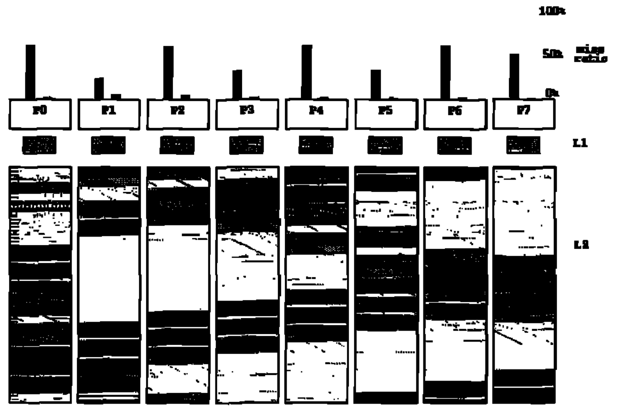
(a) barnes, 2048 particles



(b) fmm, 2048 particles



(c) ocean, non-contiguous, 130 × 130



(d) ocean, contiguous, 130 × 130

Figure 7: Cache coherence visualization for SPLASH2 applications

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[3] J. Dongarra. A tool to aid in the design, implementation, and understanding of matric algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9(2):185–202, June 1990.

[4] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.

[5] E. Lusk et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[6] P. S. Magnusson et al. SimICS/sun4m: A virtual workstation. In *Proceedings of the 1998 Usenix Annual Technical Conference*, New Orleans, LA, June 1998.

[7] M. Martonosi, A. Gupta, and T. E. Anderson. Tuning memory performance of sequential and parallel programs. *Computer*, 28(4):32–40, April 1995.

[8] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of the 1996 International Conference on Computer Design*, October 1996.

[9] 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.

[10] R. Rajwar, A. Kägi, and J. R. Goodman. Improving the throughput of synchronization by insertion of delays. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, Toulouse, France, January 2000.

[11] M. S. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[12] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, Durham, NC, January 1994.

[13] 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.

[14] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996. Available at http://www.supercomp.org/sc96/proceedings/.