Parallel Simulation of Multiprocessor Execution: Implementation and Results for SimpleScalar

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

Abstract

In research that relies on simulation in order to predict and compare the performance of proposed computing architectures, multiprocessor simulations have inherent concurrency that can be exploited for parallelization in order to reduce the execution time for a simulation. This paper describes the initial experiences in first introducing multiprocessor simulation support for the detailed out-of-order target simulator from the popular SimpleScalar tool set, and then parallelizing the resulting simulator for execution on a multiprocessor host system. The extended simulator provides the basis for further detailed modeling of target systems with multiple out-of-order processors through parallel simulation on a multiprocessor host. For experiments conducted on a Sun Enterprise 3500 platform, the measured speedup for the initial version of the parallelized simulator reached 4.4 on 6 processors for a selected application from the SPLASH-2 benchmark.

1 Introduction

Simulation is an important technique in computer architecture research, and numerous software tools have been developed for simulating uniprocessor and multiprocessor architectures [1, 2, 5, 8, 9, 11]. Where appropriate, researchers may execute several independent simulations on different host processors at the same time for increased simulation throughput. If a faster turnaround is desired for one particular simulation, however, parallel execution on a host multiprocessor can be considered. In addition to enabling parallel execution, a multiprocessor can provide a larger aggregate memory capacity for large simulations. Few of the existing tools have been produced to execute simulations in parallel on host multiprocessors [4]. It may be possible to partition the timeline for one simulation into several disjoint phases that are simulated in parallel, with appropriate pre- and post-processing to handle the boundaries between phases and aggregation of results. The focus of this paper, however, is to parallelize simulation for execution on a multiprocessor host in a manner that exploits the inherent concurrency in modeling multiple processors in a target system.

SimpleScalar is a popular simulation tool set for computer architecture research that includes functional, cache, and detailed out-of-order simulators [1]. The original release of the software only supported uniprocessor simulation, hence in earlier work by the author of this paper, multiprocessor support was introduced for functional simulation [3]. This paper describes the subsequent extension of the detailed out-of-order target simulator in SimpleScalar with multiprocessor support, and the parallelization of the resulting simulator for execution on a multiprocessor host. Experiments were conducted on a Sun Enterprise 3500 system with six 336-MHz processors and 4-Mbyte caches in order to assess the parallel performance of the basic multiprocessor simulation extensions as a baseline for future comparison as more system details are modeled. For a selected multiprocessor application from the SPLASH-2 parallel benchmark suite, a speedup of 4.4 is achieved on 6 processors.

This paper describes the early experiences in adapting an existing uniprocessor simulation tool to produce a target simulator that models parallel systems with out-of-order processors where the simulator itself is executed in parallel on a host multiprocessor. The contribution is the insight obtained on the approaches to parallelization, and the basis that the modified tool provides for a full simulator that models all aspects of a complete target multiprocessor system with out-of-order processors in detail. At present, the modified simulator models shared memory, and initial support for cache coherence has been developed. Future work will include modeling for other features. This initial work does not address the issue of validation, in part because it is a work in progress, but also because validating simulators is inherently challenging, as discussed in previous work [7].

The remainder of this paper is organized as follows. Section 2 briefly outlines related work. Section 3 discusses

0-7803-7230-1/00/$10.00 © 2001 IEEE

aspects related to extending the out-of-order simulator to model multiple processors. Section 4 discusses the parallelization of the multiprocessor out-of-order simulator. Section 5 describes initial performance results for the parallel simulator. Finally, Section 6 concludes the paper with directions for future work.

2 Related Work

In previous work, the author of this paper enhanced the functional SimpleScalar simulator to model multiprocessor execution [3]. The major changes included supporting multiple register files, adapting the core simulation algorithm for multiple target processors, and introducing new support for thread creation and synchronization operations through system calls and a run-time library. Nonetheless, the resulting simulator was still executed on a uniprocessor host.

Rajwar *et al.* [6] describe a separate project in which they produced a multiprocessor out-of-order simulator that is also based on the SimpleScalar tool set. They indicate, however, that they did not rely on the existing `sim-outorder` simulator, but rather developed their own more detailed simulation method that is slower but somewhat more accurate.

For parallelization of multiprocessor simulation, the most recent notable work is the development of the Wisconsin Wind Tunnel II (WWT II) simulator by Mukherjee *et al.* [4]. WWT II is a direct-execution, discrete-event simulator that can be executed on shared-memory multiprocessors or networks of workstations. WWT II stems from efforts on an earlier parallel simulator that also used direct-execution [7]. Through direct-execution, WWT II models a number of in-order processors in a target parallel architecture in order to calculate the simulated execution time. WWT II synchronizes the processors on the host system that execute the simulation in parallel by dividing simulated time into intervals or quanta. At the end of each quantum of simulated time, the host processors arrive at a global synchronization point for exchanging messages that may affect the simulation state in the next quantum. The quantum duration is based on the minimum amount of simulated time for a meaningful interaction between the simulated elements in the target system.

The authors of WWT II point out, however, that predicting the execution time for target systems with multiple out-of-order processors and blocking caches is a more difficult problem. Since a simulator for this latter system type is much slower and is executed on a single host processor, the purpose of this paper is to describe the initial experiences in parallelizing such a simulator for a multiprocessor host.

3 Modeling Multiprocessors in the Out-of-Order Simulator

Multiprocessor simulation provides a degree of readily-exploitable concurrency for parallelization, hence the first step was extending the out-of-order target simulator in SimpleScalar with multiprocessor support. This section briefly describes the changes that were made for this purpose.

The out-of-order simulator uses the register update unit (RUU) scheme for modeling out-of-order instruction fetching and execution [10]. The core of the uniprocessor out-of-order simulator is found in the file `sim-outorder.c` consisting of more than 4000 lines, and the data structures are allocated implicitly for one simulated processor. The simulation code models register contents and uses numerous other data structures for modeling RUUs, TLBs, caches, a branch history table, the load-store queue, and the speculative state [1]. A counter keeps track of the current simulation time in processor cycles, and on every cycle, new instructions are fetched if resources are available. An event queue also maintains future events marking the completion of various activities and it is checked as simulated time advances forward.

In order to simulate multiple out-of-order target processors, the relevant data structures must be replicated. For the implementation discussed in this paper, the replication was achieved with linear arrays for each data structure. This approach was chosen for its expediency in transforming the code to use the current processor identifier as an index. A number of the important data structures for one processor were already arrays, so for those variables an extra array dimension was introduced. Although the data structures for modeling caches were also replicated for multiple processors, only limited support for enforcing cache coherence was introduced in the cache access function called by the out-of-order simulator, primarily in the form of zero-latency invalidations for writes. As other details for a full simulation are not yet implemented, this capability was disabled for the initial parallel execution experiments undertaken for this paper.

Using the `ocean` program (contiguous data version) from the SPLASH-2 benchmark [12] with a modest problem size of 34×34, the overhead for the array-based replication approach with one simulated processor over the the unmodified uniprocessor target simulation code was 10% on one host processor of a Sun Enterprise 3500 system. As the number of simulated target processors increased to 16, the overhead increased to 30% for interpreting approximately the same total number of instructions as the uniprocessor target simulation. This increase is due primarily to the larger amount of state information for multiple target processors that cannot all be retained in the host processor cache.

An alternative to the array-based approach for extending the data structures is to consolidate all of the data specific for each processor into one large structure, and use a pointer to an instance of the consolidated structure for each processor. This approach can be implemented as a future enhancement, and the extent of the modifications in the code and any resulting overhead can be compared with the array-based approach.

4 Parallelization of the Multiprocessor Simulator

This section describes the parallelization of the multiprocessor out-of-order target simulator. Considerations include the assignment of simulated processors to physical processors, synchronization for advancing simulation time, and practical issues such as the potential for false sharing.

4.1 Host Thread Creation and Assignment of Target Processors

Parallel simulation requires multiple threads to be created on the host multiprocessor. Each host thread must fetch and interpret instructions from the appropriate simulated target processors that are assigned to it, based on the host thread identification. It is convenient to cyclically assign target processors to physical processors in order to automatically provide load balance as the target processors are created in the simulation. If the data in the simulated multiprocessor application is distributed among target processors in a blocked manner, there may be an advantage in considering a blocked assignment of target processors in order to exploit any data sharing between target processors in the host processor caches. As will be discussed later, however, a time-shared scheduling policy for threads executing on the host multiprocessor may negate any caching benefit.

All SimpleScalar simulators include a function `sim_main()` that contains the main simulation loop. For the uniprocessor target simulators, this main loop simulates the cycle-by-cycle activity for a single processor only. The multiprocessor extensions to SimpleScalar introduce a new inner loop that steps through all active target processors in each cycle [3]. For parallelization, the workload in each cycle for different target processors must be executed concurrently on different host processors, hence a new function `par_sim_main()` is introduced as shown in Figure 1. The `sim_main()` function is still called after initialization, but its only purpose now is to create the multiple threads for concurrent execution of the new `par_sim_main()` function. The modified inner loop in the latter function steps through the simulated target processors cyclically in order to provide the load balance.

```
int num_host_procs;

void par_sim_main ()
{
  int my_id = get_my_thread_id ();

  while (TRUE) /* step through time */
  {
    /* step through processors cyclically */
    for (pid = my_id;
         pid < num_simulated_processors;
         pid += num_host_procs)
    {
      /* simulate this processor, if active */
    }
    /* synchronize if quantum has expired */
  }
}

void sim_main ()
{
  int i;

  for (i = 1; i < num_host_procs; i++)
    create_thread (par_sim_main);
  par_sim_main ();
}
```

Figure 1: Parallelization of main simulation loop

The host processors maintain separate time counters that are synchronized at the end of each quantum of simulation time. During the initialization phase before a simulated application enters its parallel execution phase, there may only be one active simulated processor, hence several host processors may initially be idle between synchronizations. Finally, each simulated processor has a flag that indicates whether it is active or not, as implemented in previous work [3]. Synchronization operations that require the simulated processor to wait on a lock or barrier will temporarily mark it as inactive.

4.2 Reduction of False Sharing

For uniprocessor execution of the simulator, false sharing of various data structures that reflect simulated multiprocessor state is not a concern. In extending the out-of-order simulator to model a multiprocessor, data structures are transformed into arrays with one element for each simulated processor. Although convenient from the perspective of source code, this can lead to false sharing when the simulator is executed in parallel on a host multiprocessor. To alleviate this problem, padding is introduced to ensure that data accessed by different host processors

is allocated in different cache lines. Such simple changes reduce parallel execution time by 40% or more and ensure that the speed relative to uniprocessor execution is reasonable. The previously-mentioned alternative of encapsulating processor-specific state information into a complex structure would also alleviate the problem of false sharing and is left for future work.

5 Parallel Simulator Performance

The performance of the base parallel multiprocessor simulator described in the previous section was evaluated on a Sun Enterprise 3500 host system with six 336-MHz UltraSPARC-II processors, each with 4 Mbytes of cache memory. The main memory provided 2 Gbytes of storage. The `pthread` library was used to create and synchronize the actual threads on the hardware. The simulation code was compiled with `gcc` version 2.8.1 using the `-O3` optimization level. The simulated application was compiled with `gcc` version 2.6.3 in the SimpleScalar tool set, with the runtime thread support library described in previous work [3]. The scope of the results reported here is a function of the limited access that was granted to the multiprocessor host system for developmental and experimental purposes. Consequently, the results reported here are from experiments that used a relatively large quantum of 1024 cycles in order to provide an indication of the upper limit on the instruction simulation rate achievable for the parallelized out-of-order simulator.

The `ocean` program with contiguous data allocation was used from the SPLASH-2 benchmark [12]. It is load-balanced for parallel execution in terms of the number of instructions executed per processor. The number of target processors for `ocean` must be a power of two, hence for load balance on the multiprocessor host, it is best to use 1, 2, or 4 host processors. Results on 6 host processors will be reported, but these are not load-balanced simulations. Two problem sizes were considered: 34×34 and 66×66. The L1 cache size on each simulated processor was 8 kbytes and the L2 cache size was 256 kbytes. All caches were direct-mapped with a block size of 16 bytes. As discussed earlier, some initial support for enforcing cache coherence has been developed, but for the sake of the initial experiments to measure performance, this capability was not used. All other parameters related to the processor characteristics used the default values from the original out-of-order target simulator.

Figure 2 provides the simulation rates in instructions per second for different simulations of the `ocean` program using a quantum size of 1024 cycles and no coherence. Modeling the execution of `ocean` with 16 target processors and the larger problem size required more than 13 minutes on one host processor of the Sun Enterprise 3500, but required less than 4 minutes on four host processors, for a speedup of 3.6. Although it is not a load-balanced configuration, simulating 16 processors on 6 host processors required just under 3 minutes, for a speedup of 4.4 over one processor. The simulation rate for the latter case exceeds 254,000 instr/sec. For the smaller problem size on 8 target processors, the lack of load balance on 6 host processors explains the reduction in the simulation rate compared to 4 host processors. Looking towards the future inclusion of more system details in the simulation model and use of a smaller quantum size, the simulation rate can be expected to decline. A coarser task granularity from more detailed modeling can provide the benefit of increasing parallel speedup, but reducing the quantum size can affect granularity in the opposite direction, hence there is a tradeoff that can be investigated in future work.

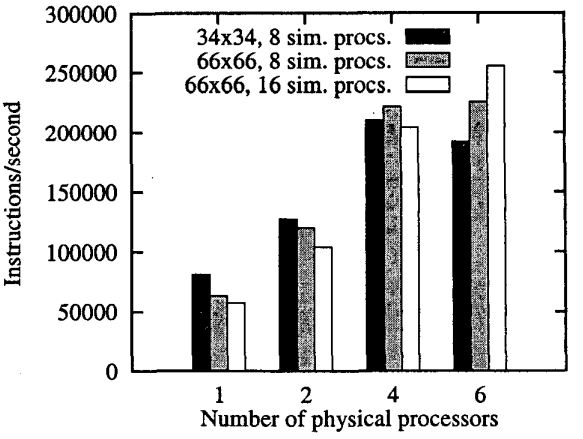

Figure 2: Simulation rates for parallel simulation of multiprocessor `ocean`

Finally, it may be noted that the operating system on the Sun Enterprise 3500 uses a time-shared policy that frequently reassigns threads to different host processors during execution, which reduces cache locality on the real hardware. For comparison, the simulation rate for the largest problem size with 16 simulated processors on one host processor is 57,400 instr/sec, whereas the same simulation executed on a Sun Ultra10/333, a comparable uniprocessor platform with half the cache capacity of a Sun Enterprise 3500 processor yields a higher simulation rate of 71,100 instr/sec. Using a real-time scheduling policy where threads are bound continuously to the same physical processor would potentially yield better parallel execution performance.

6 Conclusion

This paper has described initial experiences in extending the SimpleScalar out-of-order simulator to model multiple processors, and the parallelization of the resulting simulation code for execution on a multiprocessor host. Building on earlier work that introduced multiprocessor support into the functional SimpleScalar simulator, the data structures in the out-of-order simulation code were modified to support multiprocessing. Subsequent parallelization was implemented with code to create and synchronize threads for execution on a host multiprocessor to model the progression of simulated time. The choice was made to extend the various data structures with an array-based approach that required a degree of padding to alleviate false sharing in parallel execution. The performance of the initial version of the parallelized simulator was evaluated on a Sun Enterprise 3500 system, with speedup of 4.4 on 6 processors for a selected program from the SPLASH-2 benchmark.

Future work can consider alternative implementation approaches for replicating the data structures in the simulation code and for synchronizing the host processors in parallel execution. More detailed modeling of a complete target multiprocessor could be undertaken by building on the initial support for enforcing cache coherence and including new support for simulating interconnection and memory behavior. The effect of thread migration on performance can also be studied by considering the use of real-time scheduling to bind the threads performing the parallel simulation to host processors.

Acknowledgements

Funding from Queen's University, Communications and Information Technology Ontario (CITO), and the Natural Sciences and Engineering Research Council of Canada (NSERC) is gratefully acknowledged. Access to the Sun Enterprise 3500 platform was provided by the CASLab facility at Queen's University through the assistance of Greg Macleod.

References

[1] 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.

[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] N. Manjikian. Multiprocessor enhancements of the SimpleScalar tool set. *ACM SIGARCH Computer Architecture News*, 29(1):8–15, March 2001. Software available at `www.simplescalar.org`.

[4] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Fast and portable parallel architecture simulators: Wisconsin Wind Tunnel II. *IEEE Concurrency*, 8(4):12–20, October-December 2000.

[5] 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.

[6] 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.

[7] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[8] 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.

[9] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, San Jose, CA, October 1998.

[10] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transcations on Computers*, 39(3):349–359, March 1990.

[11] 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.

[12] 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, Santa Margherita Ligure, Italy, June 1995.