A Shared-Variable-Based Synchronization Approach to Efficient Cache Coherence Simulation for Multi-Core Systems

Cheng-Yang Fu, Meng-Huan Wu, and Ren-Song Tsay
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan
{chengyangfu, mhwu, rstsay}@cs.nthu.edu.tw

Abstract—**This paper proposes a shared-variable-based approach for fast and accurate multi-core cache coherence simulation. While the intuitive, conventional approach—synchronizing at either every cycle or memory access—gives accurate simulation results, it has poor performance due to huge simulation overloads. We observe that timing synchronization is only needed before shared variable accesses in order to maintain accuracy while improving the efficiency in the proposed shared-variable-based approach. The experimental results show that our approach performs 6 to 8 times faster than the memory-access-based approach and 18 to 44 times faster than the cycle-based approach while maintaining accuracy.**

Keywords- cache-coherence; timing synchronization

I. Introduction

In order to maintain the memory consistency of multi-core architecture, it is necessary to employ a proper cache coherence system. For architecture designers, cache design parameters, such as cache line size and replacement policy, need to be taken into account, since the system performance is highly sensitive to these parameters. Additionally, software designers also have to consider the cache coherence effect while estimating the performance of parallel programs. Obviously, cache coherence simulation is crucial for both hardware designers and software designers.

A cache coherence simulation involves multiple simulators of each target core. To keep consistent simulated time of each core, timing synchronization is required. A cycle-based synchronization approach synchronizes at every cycle as shown as in Figure 1(a), and the overhead due to the frequent synchronization heavily degrades the simulation performance. At each synchronization point, the simulation kernel will switch out the executing simulator and put it in a queue according to the simulated time, and then switch in the ready simulator with the earliest simulated time to continue execution. Highly frequent synchronization causes a big portion of the simulation time spent on context switching instead of intended functional simulation.

As far as we know, existing cache coherence simulation approaches are making a tradeoff between simulation speed and accuracy. For instance, event-driven approaches [6-9] select system state changing actions from simulation system as events and keep these events executed in a temporal order according to the simulated time instead of at every cycle. To execute events in a temporal order, timing synchronization is required before each event, as shown as in Figure 1(b). While a correct execution order of events will clearly lead to an accurate simulation result, in practice not every action requires synchronization. If all actions are included as events without discrimination, the synchronization overhead can be massive.

As an example, since the purpose of cache coherence is to maintain the consistency of memory, an intuitive synchronization approach in cache coherence simulation is to do timing synchronization at every memory access point. Each memory operation may incur a corresponding *coherence action*—according to the type of memory access, the states of caches, and the cache coherence protocol specified—to keep local caches coherent.

To illustrate the idea, Figure 2 shows how coherence actions work to keep local caches coherent in a *write-though invalidate policy*. When *Core$_1$* issues a write operation to the address @, the data of @ in memory is set to the new value and a coherence action is performed to invalidate the copy of @ in *Core$_2$*'s cache. Therefore the tag of @ in *Core$_2$*'s cache is set to be invalid. Next, when *Core$_2$* wants to read data from the address @, it will know that the local

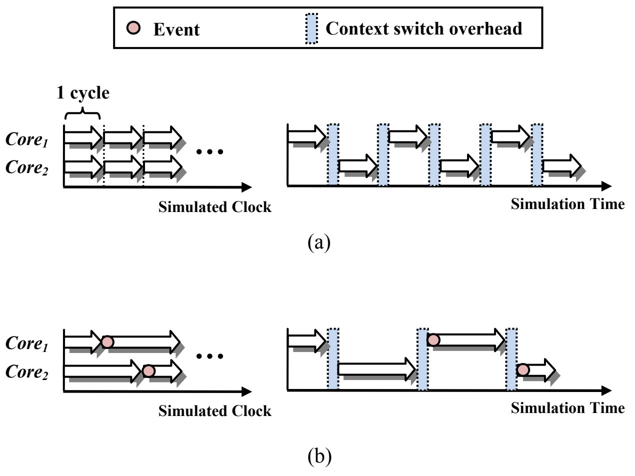

Figure 1: (a) Simulation diagram of the cycle-based approach; (b) Simulation diagram of the event-driven approach.

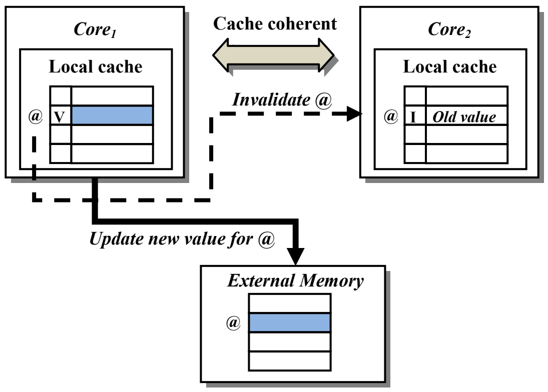

Figure 1: A cache coherent example for a two-core system based on a *write-through invalidate* policy.

cache is invalidated and that it must obtain a new value from the external memory.

Therefore, if timing synchronization is done at every memory access point, the cache-coherent simulation will be accurate. However, Hennessy et. al. [5] reports that, in general, over 30 percent of executed instructions of program are memory access instructions. Hence, this approach still suffers from heavy synchronization overhead.

To further reduce synchronization overhead in cache coherence simulation, we propose a shared-variable-based synchronization approach. As we know, coherence actions are applied to ensure consistency of shared data in local caches. In parallel programming, variables are categorized into shared and local variables. Parallel programs use shared variables to communicate or interact with each other. Therefore, only shared variables may reside on multiple caches while local variables can only be on one local cache. Since memory accesses of local variables cause no consistency issue, the corresponding coherence actions can be safely ignored in simulation. Based on this fact, we can synchronize only at shared variable accesses and achieve better simulation performance while maintaining accurate simulation results. As confirmed and verified by the experimental results, the simulation results are accurate and the speed is 6 to 8 times faster than synchronizing at every memory access and 18 to 44 times faster than the cycle-based approach.

The rest of this paper is organized as follows. Section II reviews related work. Section III explains the idea of using shared variable in cache coherence simulation. Section IV describes implementation details for our simulation approach. Experimental results are presented and discussed in section V. Finally, our conclusion and future work are summarized in section VI.

II. Related Work

In this section, we first review cycle-based and event-driven simulation approaches for multi-core simulation. In addition, we also review the timing-adjustment approach, a variation of trace-driven approach that tries to reduce synchronization overhead against cycle-based and event-driven approaches by adjusting only timing but not event points.

A. Cycle-Based Simulation

Cycle-based approaches [1-4] (or lock-step simulation approaches) synchronize each single-core simulator at every cycle time. All cache data accesses and coherence actions are simulated in a precise temporal order and simulation is guaranteed to be accurate. However, as discussed previously, the issue is that heavy s*ynchronization overhead* slows down the simulation speed. In practice, the performance of cycle-accurate simulation is observed to be only about 1 to 3 MIPS (million instructions per second).

A variation to cycle-based approach is to synchronize at every N cycle instead of every cycle [3-4]. Nevertheless, improved simulation performance comes with a penalty of greater inaccuracy.

B. Event-Driven Simulation

Another approach is event-driven [6-9], which is also known as the discrete event simulation approach. Instead of synchronizing at every cycle, the event-driven approach synchronizes only at where *events* occur and makes sure they are executed in a strictly chronological order.

1) Cache coherence simulation

Among our surveyed event-driven approaches, Ruby and Timepatch [6, 9] focus on handling cache coherence simulation in multiprocessor systems.

Ruby [6] is a well-known event-driven multiprocessor memory system timing simulator. It performs synchronization at the next scheduled message-passing event on the queue instead of at every cycle. In practice, this approach may generate too many events—close to that of lock-step simulation, and perform poorly when applied to cache coherence simulation.

Timepatch [9] exploits the fact that functional and timing correctness of a simulation can be ensured by executing timing synchronization at user-level synchronization operations, such as lock. By keeping the order of user-level synchronization operations, the simulated execution can faithful reproduce the factual execution on the target parallel machine. However, this approach cannot capture data racing situations of unprotected shared variables and can lead to inaccurate results.

2) General multi-core simulation

The remaining event-driven approaches focus on different issues in multi-core simulation [7-8]. These approaches either are inefficient or cannot guarantee accuracy when applied to cache coherence simulation.

CATS is a technique that accurately simulates bus transfer in MPSoC by synchronizing at the boundaries of data transfers, or equivalently granted memory access, on bus [7]. In general, frequent data transfers generate huge number of synchronization points and hence slow down simulation performance for this approach.

The data-dependence-based synchronization approach [8] was the first elaborated shared memory synchronization approach that could maintain multi-core system simulation accuracy. Programs on multi-core communicate through shared memory accesses, so data-dependence exists only between these shared memory accesses. Therefore, synchronizations are needed only at these accesses and accurate simulation result can be ensured. However, this work only considers an ideal cache-less shared memory model. In practical modern multi-core architectures, cache coherence system is required and is critical for system performance. Without considering coherence action handlings, there will be a gap between the estimated and true performance.

C. Timing-Adjustment Approach

In contrast to event-driven approaches, which synchronize at every event, the timing-adjustment approach [10-11] continues advancing simulated time until data transfer or a user-defined time point and performs timing adjustments for any events that may have triggered from other sources.

As an example, virtual synchronization [10] advances simulated time until it reaches the data transfer boundary and then uses resource-access traces to calculate delay from any resource conflicts. Then, the simulated time is adjusted according to the calculated delay.

To reduce synchronization overhead, the Result-Oriented Modeling (ROM) approach [11] enlarges the time interval between synchronizations. Users can arbitrarily decide the length of time interval. When reaching the specified synchronization point after execution, ROM checks if any interrupt occurred before this time point and estimates the impact for revising current simulated time.

Nevertheless, the main problem of this approach is that it does not enforce the execution order and hence may violate dependence relationships between past executions. In this case, the simulation result can be questionable and meaningless.

In order to provide a fast and accurate simulation approach applicable to multi-core simulation with cache coherence, we extend the data-dependence-based synchronization approach [8] and propose a shared-variable-based approach that can handle cache coherence simulation.

Before going into the details of our proposed approach, we first elaborate the cache coherence issues in multi-core simulation in the following section.

III. SHARED-VARIABLE-BASED SYNCHRONIZATION

To effectively reducing synchronization overhead in multi-core simulation, it resides in the fact that only shared variables in local caches can affect the consistency of cache contents. Therefore, timing synchronizations are needed only at shared variable access points in order to achieve accurate simulation results.

A. Execution Order

In cache coherence simulation, it is crucial to know the correct execution order of data access and coherence actions in each cache. As discussed before, parallel programs use shared data to interact with each other, and these shared data may have multiple copies in different local caches on a multi-core system. The correct simulation procedure of cache update coherence actions is essential to maintaining correct cache contents and states of caches without corruption.

We use the case in Figure 3 to illustrate the importance of correct simulation order of data access and coherence actions. Figure 3(a) is a correct simulation of shared data accesses in a cache coherence system. *$Core_2$* executes the *Write* operation between *$Core_1$*'s *$Read_1$* and *$Read_2$* operations. The time to execute the invalidation caused by *$Core_2$*'s *Write* operation is important because it forces *$Core_1$*'s *$Read_2$* operation to re-read data from memory instead of cache. If this invalidation operation cannot be captured between *$Read_1$* and *$Read_2$* operations, The *$Read_2$* operation reads the wrong value and changes the behavior of *$Core_1$*, as Figure 3(b) shows. Clearly, improper execution orders can generate inaccurate simulation results.

B. Synchronization Points

Theoretically, for minimum synchronization overhead, we only need to maintain the execution order of the cohe-

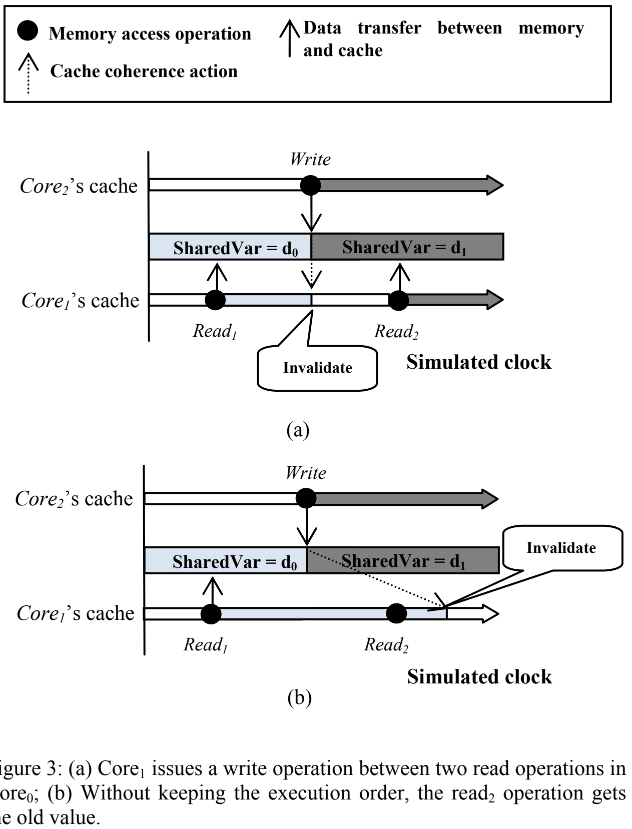

Figure 3: (a) $Core_1$ issues a write operation between two read operations in $Core_0$; (b) Without keeping the execution order, the $read_2$ operation gets the old value.

rence actions and data accesses in cache locations that point to the same shared variable address. However, due to the large memory space required for recording the necessary information, it is infeasible to trace addresses of all coherence actions and data accesses.

A proper balance is to synchronize at every shared variable access point. In practice, coherence actions are used to mark cache status and ensure the consistency of shared data in local caches. Since only shared variables may reside on multiple caches and local variables can only be on one local cache, memory accesses of local variables cause no consistency issues. Hence, the corresponding coherence actions can be safely ignored in simulation. Therefore, it is possible to synchronize only at shared variable access points to achieve accurate simulation results with high simulation performance.

IV. The Proposed Simulation Approach

In this section, we use the multi-core simulation to elaborate our synchronization approach. In this platform, each core is simulated by a single target-core simulator and coherence actions are passed between simulators.

A. Simulation Framework

The simulation framework is shown in Figure 4. As discussed before, for achieving accurate simulation results, we need to make sure that all unprocessed coherence actions have occurred before any shared variable memory access instruction are processed prior to executing the memory access.

One intuitive approach for ensuring the temporal execution order of both coherence actions and shared variable memory access instructions is to perform timing synchronization on all coherence action and shared memory access points.

If this simple idea is implemented using the platform, each single-core simulator submits its broadcasted/received coherence actions and shared memory access events to SystemC kernel [12] and lets the kernel's internal scheduling meachanism perform timing synchronization. In SystemC, timing synchronization is achieved by calling the

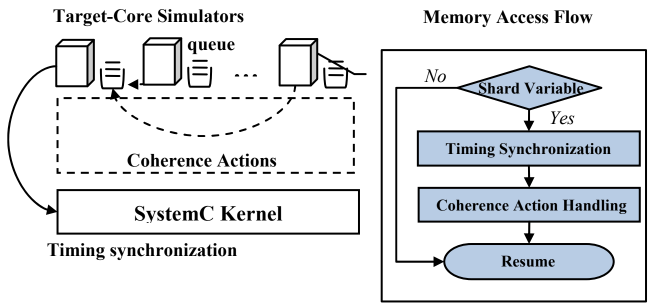

Figure 4: The proposed simulation framework for a multi-core with cache coherence.

wait() function. When executing wait(), the SystemC kernel will switch out the calling simulator and calculate the invocation time according to the wait time parameter of the wait() function. Then, the SystemC kernel selects the queued simulator with the earliest simulated time to continue simulation.

This approach, while providing accurate results, will carry a heavy synchronization overhead and have poor performance. The main reason is that, in reality, each memory access incurs multiple coherence actions and hence induces too many synchronization events. To improve simulation efficiency, we observe that the handling of coherence actions on each single-core simulator can be deferred until encountering a shared memory access point.

For accuracy, all coherence actions occurred prior to a shared memory access must be processed before the memory access point. There are two important considerations associated with this requirement. First, these coherence actions only have to be executed *before* the memory access point, but not necessary at the action occurring time. Therefore, we can simply just queue up the coherence actions and process them when a shared memory access point is reached. By doing so, the overhead is greatly reduced.

Another issue is that we need to ensure that all coherence actions occurring before a shared memory access point are captured in the queue for processing; otherwise, missing-but-needed coherence actions will lead to inaccurate results. This requirement is in fact guaranteed by applying the centralized SystemC kernel scheduler. Note that after timing synchronization, the simulator with the earliest simulated time is selected to continue execution. In this way, the coherence actions broadcasted from other simulated cores must have occurred before the current time point and all related coherence actions should have been captured.

Nevertheless, one question is that whether the queued coherence actions are in correct temporal order. If the communication delay for passing coherence actions is fixed, then the queued coherence actions should be naturally in temporal order since the simulators are invoked following the temporal order of shared memory access points through the centralized SystemC kernel scheduler, as discussed before. In cases where the communication delay to different cores is uncertain, the received coherence actions may not be in the proper temporal order. Therefore, we have to put the coherence actions queue into temporal order before processing them.

With synchronizations only at shared memory access points and all required coherence actions ready in queues, the simulation approach not only performs much more efficiently than previous approaches but also guarantees functional and timing accuracy.

We demonstrate the idea using the example in Figure 5. Figure 5(a) shows the time diagram in terms of the simulated clock domain. In this case, the timing synchronization event is inserted before every shared memory access point. According to the previous discussion, this will ensure the

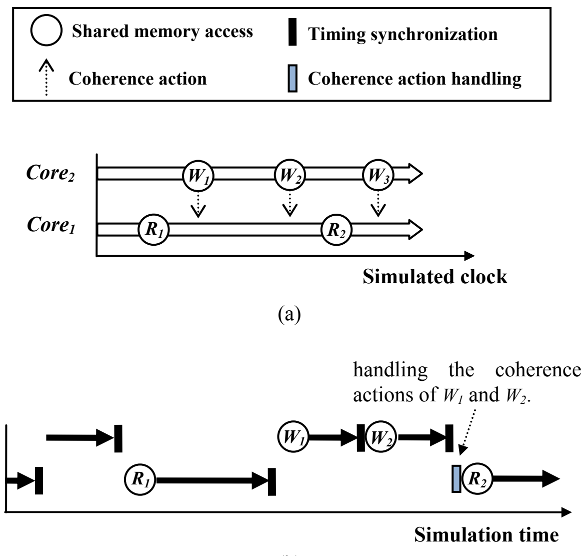

Figure 5: An illustration of coherence event handling in terms of (a) the simulated clock domain and (b) the simulation time domain.

shared memory accesses to be executed in order, i.e., $R_1 \rightarrow W_1 \rightarrow W_2 \rightarrow R_2 \rightarrow W_3$ as illustrated in Figure 5(b).

Assume that the targets of all the given shared memory accesses are the same. Therefore, when $Core_1$ executes R_2, the corresponding data is cached already due to the previous execution of R_1. If there is no other coherence events between R_1 and R_2, R_2 can directly fetch the data from its local cache. However, in this case, $Core_2$ executes W_1 and W_2 which introduce coherence actions to invalidate $Core_1$'s cache content. In our implementation, those actions will be queued until the execution of R_2 and then update the $Core_1$'s local cache. Following this proper processing sequence, the accurate simulation results are guaranteed.

V. Experimental Results

We implement the proposed approach on SystemC2.20simulation framework [12]. The target multi-core system under test is an Andes [13] 16/32-bit mixed length RISC ISA-based system and the corresponding ISS is implemented using static *compiled-simulation* technique [14]. The cache on each core is direct mapped and of 1,024-byte data size and 32-byte line size. The target system adopts the MESI cache coherence protocol [15], which is implemented in most modem multi-core architecture and snoopy coherence mechanism [16] in which every memory access will broadcast coherence actions to all cores.

The host machine is an Intel Xeon 3.4 GHz quad-core machine with 2GB RAM, running Linux OS. We tested our algorithm on three parallel programs from Splash2 [17], which are standard benchmark cases that can be easily configured to machines with different numbers of cores. The three parallel programs are Fast Fourier Transform (FFT), LU, and radix sorting (Radix) with suggested problem size.

Since the number of coherence actions is directly proportional to the number of target cores for the snoopy coherence mechanism, all cases are tested on two and eight cores systems to show the differences of synchronization overheads from core number differences.

In Figure 6, we compare the performance of our proposed synchronization approach with four other approaches with various synchronization frequencies. The first compared is the cycle-based approach, or CB in short, which synchronizes at every cycle. The second is MC (memory accesses with coherence actions), which synchronizes at every memory access and coherence action point and is equivalent to the Ruby approach [6]. Thirdly, MA (memory accesses) synchronizes at every memory access point and is equivalent to CATS approach [7]. Lastly, SMC (shared variable access with coherence actions) synchronizes at every shared variable memory access and the related coherence action point. Our proposed approach is named SMA (Shared variable access) and it processes queued coherence actions at the shared memory access synchronization points.

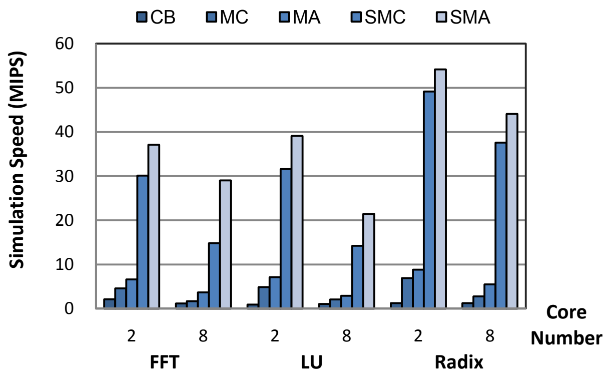

Figure 6: Comparing simulation performance of different synchronization approaches.

As reported by the experiment results, our synchronization approach SMA outperforms CB by 18 to 44 times, MC by 8 to 17 times, and MA by 6 to 8 times.

At a glance, SMA is in the range of 10% to 25% faster than SMC for two-core target systems. As the core number increases, the performance gap between SMA and SMC becomes larger. For an eight-core target system, SMA performs in the range from 1.2 to 2 times faster than SMC.

To study the severity of the synchronization overhead in the approaches mentioned above, we test them using the FFT application on a two-core target system.

As shown in Figure 7(a), CB, MC and MA approaches all have heavy synchronization overheads, ranging from 70% to 80% of total simulation time. Mainly, frequent synchronization requests cause heavy overheads. In contrast, our proposed SMA approach has only a 10% overhead, mainly contributed by much-reduced synchronization points.

We also show the total simulation time comparison in Figure 7(b). It can clearly demonstrate that our proposed SMA approach outperforms all other approaches. An interesting observation we would like to point out is that the dark portion, or the total computation time of instruction executions and cache access actions, is becoming smaller following the sequence of CB, MC, MA, SMC and SMA. Theoretically, the total number of instruction executions and cache access actions should be the same independent of the chosen approach and hence the computation time should be the same. In reality, at each synchronization point the simulation kernel has to call a cache coherence checking routine, which induces some overheads even if there is no cache action. Therefore, depending on the synchronization frequency, the checking routine calling overhead increases accordingly. Our proposed approach, SMA, has the minimum number of synchronization points and hence gives the best results.

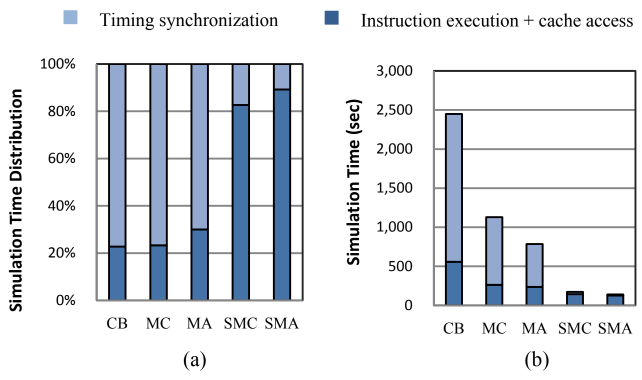

Figure 7: Comparison of synchronization overhead and total simulation time.

VI. Conclusion

We have presented and demonstrated an efficient synchronization approach for multi-core cache coherence simulation. Our major contribution is in successfully devising an effective synchronization approach specific to cache coherence in multi-core simulation. The approach takes advantage of the operational properties of cache coherence and effectively keeps a correct simulation sequence. The experimental results show that our approach outperforms existing approaches and is fast and accurate.

Our future research topic could be extending the proposed synchronization algorithm to simulate multi-tasking target systems. Currently, each core is assumed to take only one application thread. With the constraint relaxed, the approach could be more generally and practically applied.

Acknowledgement

This work was supported by National Science Council (Grant No. NSC99-2221-E007-114-MY3) and the specification of Andes ISA was provided by Andes Technology.

References

[1] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," in*Computer,* vol. 35. pp. 50-58, 2002.

[2] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," in *The Journal of VLSI Signal Processing.*vol. 41, pp. 169-182, 2005.

[3] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: the SimOS approach," in*IEEE Parallel & Distributed Technology: Systems & Applications.*vol. 3, pp. 34-43, 1995.

[4] F. Fummi, M. Loghi, S. Martini, M. Monguzzi, G. Perbellini, and M. Poncino, "Virtual Hardware Prototyping through Timed Hardware-Software Co-Simulation," in *DATE.* pp. 798-803, 2005.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, fourth edition, 2006.

[6] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," in *SIGARCH.*vol. 33, pp. 92-99, 2005.

[7] D. Kim, S. Ha, and R. Gupta, "CATS: Cycle Accurate Transaction-driven Simulation with Multiple Processor Simulators," in *DATE.* pp. 1-6, 2007.

[8] M. Wu, C. Fu, P. Wang, and R. Tsay, "An effective synchronization approach for fast and accurate multi-core instruction-set simulation," in *EMSOFT.* pp. 197-204, 2009.

[9] G. Shah, U. Ramachandran, and R. Fujimoto, "Timepatch: A Novel Technique for the Parallel Simulation of Multiprocessor Caches," in *Proceedings of the ACM SIGMETRICS.* pp. 315-316, 1994.

[10] D. Kim, Y. Yi, and S. Ha, "Trace-driven HW/SW cosimulation using virtual synchronization technique," in*DAC.* pp. 345-348, 2005.

[11] G. Schirner and R. Dömer, "Fast and accurate transaction level models using result oriented modeling," in *ICCAD.* pp. 363-368, 2006.

[12] OSCI,available: http://www.systemc.org/home/

[13] Andes,available: http://www.andestech.com/

[14] M. Burtscher and I. Ganusov, "Automatic Synthesis of High-Speed Processor Simulators," in *MICRO* 55-66, 2004.

[15] M. Papamarcos and J. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ISCA.* pp. 348-354, 1984.

[16] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," in *Computer.*vol. 23, pp. 12-24, 1990.

[17] S. Woof, M. Ohara, E. Torriet, J. Singhi, and A. Guptat, "The SPLASH-2 Programs: Characterization and Methodological Considerations, " in *ISCA.* pp. 24-36, 1995.

[18] M. Wu, W. Lee, C. Yu, and R. Tsay, "Automatic Generation of Software TLM in Multiple Abstraction Layers for Efficient HW/SW Co-simulation," in *DATE.* pp. 1177-1182, 2010.