A Critical-Section-Level Timing Synchronization Approach for Deterministic Multi-Core Instruction-Set Simulations

Fan-Wei Yu[1], Bo-Han Zeng[2], Yu-Hung Huang[3], Hsin-I Wu[4], Che-Rung Lee[5] and Ren-Song Tsay[6]
National Tsing Hua University
Department of Computer Science
Hsinchu, Taiwan
{fwyu[1], bhzeng[2], yhhuang[3], cherung[5], rstsay[6]}@cs.nthu.edu.tw, hiwu@gmail.com[4]

***Abstract*—This paper proposes a Critical-Section-Level timing synchronization approach for deterministic Multi-Core Instruction-Set Simulation (MCISS). By synchronizing at each lock access instead of every shared-variable access and using a simple lock usage status managing scheme, our approach significantly improves simulation performance while executing all critical sections in a deterministic order. Experiments show that our approach performs 295% faster than the shared-variable synchronization approach on average and can effectively facilitate system-level software/hardware co-simulation.**

Keywords—Deterministic, Multi-core instruction-set simulations, Timing Synchronization

I. INTRODUCTION

Multi-core architectures have become mainstream and have gradually replaced single-core architectures due to the power wall issue. As a result, parallel programming models are required for developing software applications that leverage the computing power of multi-core systems. Nevertheless, parallel applications are very difficult to debug. In general, the key to debugging parallel applications is to capture concurrent behaviors precisely. The Multi-core Instruction-Set Simulations (MCISS) approach is considered the most promising solution for this issue. Therefore, in recent years there have been many research projects focusing on developing fast and accurate MCISSs for accurate concurrent behavior simulations.

The most important goal of any simulator is to reproduce the behaviors of an actual physical target system faithfully, but this goal has long been a great challenge for MCISS. For a multi-core system, each core can execute a task concurrently, and different task-interleaving orders may result in different execution behaviors and lead to different results. Mainly, different interleaving orders imply potentially different shared-data access sequences.

We use the example in Fig. 1 to illustrate the importance of having a deterministic interleaving order. We assume that there are two processes and each runs on a separate ISS. If both processes access the same shared data *x*, then due to the possible execution speed difference between different ISSs, the access orders of *x* can be different, such as in the cases shown in Fig. 1(b) and 1(c). For this example, we have a non-deterministic final value of *x*.

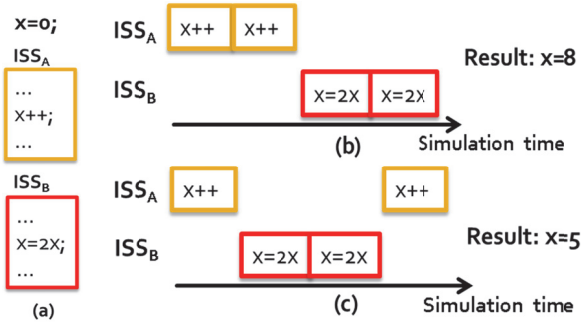

Fig. 1: (a) Two processes, each running on an ISS, access a same shared-data *x*; (b), (c) Different interleaving orders result in different outputs.

Therefore, to reproduce the shared-data access orders faithfully, an intuitive solution is to simulate the system operation cycle-by-cycle [1], [2], [3]. This type of approach can guarantee accurate simulation results, but its excessive simulation time makes it unfeasible for realistic system design use.

Instead of synchronizing at each cycle, Wu et al. [4], [5] develop an efficient shared-variable synchronization approach based on timing annotation techniques and have a deterministic MCISS for accurate concurrent system behavior simulations. Although Wu's approach synchronizes only at shared-variable locations and is much more efficient than the cycle-based approaches, it still incurs significant synchronization overhead as it has to consider all *possible* shared-variable locations to be safe.

With proper synchronization, a deterministic interleaving order can be maintained. However, synchronization also introduces certain overhead, which reduces the simulation speed. Different synchronization granularity induces different simulation speed versus accuracy. Wu et al. [6] introduce three TLM-abstraction layers which include *Instruction-Level*, *Data-Level* and *Shared-Variable-Level* to cover possible timing synchronization points between concurrent hardware and software components.

In practice, however, we find that it is sufficient to observe only the interleaving order of critical sections instead of all shared variables as most parallel application developers do use critical sections to protect shared data and avoid race conditions. We

978 -3-9815370-0-0/DATE13/©2013 EDAA.

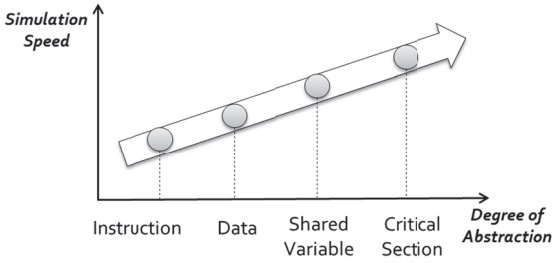

Fig. 2: The correlation between simulation speed and the degree of abstraction.

therefore extend the abstraction layer to a higher modeling abstraction named *Critical-Section-Level,* as shown in Fig. 2.

We hence develop a much more efficient deterministic *Critical-Section-Level* MCISS approach by leveraging the timing annotation techniques and focusing on synchronizing the execution of critical sections as well as the access order of protected shared data inside critical sections. This approach is good for most practical uses since the number of critical sections is much less than the number of shared variables considered in Wu's approach. In case the shared variables outside of critical sections are suspected of inducing bugs, designers can switch to the shared-variable approach for detailed debugging.

Nevertheless, it is a challenge to handle the simulation of atomic instructions properly to reproduce the same atomicity in an actual physical target system faithfully. An idealistic approach is to translate each target atomic instruction into an equivalent host atomic instruction [7]. The atomicity is then naturally reproduced and protected by the host machine. However, a direct translation may not always be possible as different architectures support different types of atomic instructions.

To resolve this issue, we developed a generic and effective solution by recording and checking the usage status of each lock memory address. Additionally, we apply timing synchronization [4] and annotation [8] techniques to guarantee that all mutually exclusive critical sections are executed in a deterministic order.

We have implemented our proposed approach and tested it on a few standard benchmark cases. In general, our proposed simulator performs 295% faster on average than the shared-variable approach. In sum, our approach can greatly help designers with system verification because it is able to reproduce the same deterministic order and the atomicity of the executed programs.

The remainder of this paper is organized as follows. Section 2 discusses related work. The proposed deterministic timed MCISS technique is elaborated in Section 3. We show the correctness of synchronization on Critical-Section-Level in Section 4. We then report the experimental results in Section 5 and give a brief conclusion in Section 6.

II. RELATED WORK

For multi-core simulations, each core can concurrently execute a task, and different task-interleaving orders may result in different execution results. The issue of determinism is widely discussed as it deeply complicates the debugging of parallel programs and limits the ability to test multithreaded code. Therefore, correctly yet efficiently reproducing the concurrent behaviors of multiple simulated cores has emerged as one of the most critical challenges.

QEMU [9], a widely adopted emulation approach, utilizes a dynamic binary translation technique but synchronizes only at basic block boundaries and performs up to hundred MIPS (Million Instructions per Second). QEMU simulates a target multi-core design in a round-robin sequence. Each target core is simulated for a time-slice of a few basic blocks. QEMU has become mature after years of active development by the open source community. A wealth of support for different architectures and device models makes it a popular simulation platform. However, current QEMU cannot guarantee determinism and cannot take advantage of the parallelism available in the underlying multi-core host architectures.

To further improve the performance of QEMU, COREMU [10] attempts parallelization by leveraging available multiple host cores. COREMU emulates multiple cores by creating multiple instances of the existing sequential emulator, QEMU, and uses a thin library layer to handle the inter-core and device communication. COREMU shows negligible uniprocessor performance overhead and scales much better compared to QEMU. Both QEMU and COREMU honor atomicity and are good for many applications, but they consider no timing information and cannot faithfully simulate concurrent system behaviors or guarantee deterministic results. In other words, even with the same inputs, the results can be different if executed at different times and in different environments.

To reproduce consistent task-interleaving orders faithfully in a deterministic simulation, an intuitive solution is adopting CA (cycle-accurate) based simulators [1], [2] that simulate system behaviors cycle-by-cycle. However, the performance of CA-based simulators is inadequate due to the excessive synchronization overhead.

Instead of synchronizing at each cycle, Gligor et al. in [11] performs synchronizations both at the basic block boundaries and predefined intervals of time. While it can achieve about 17x speedup compared to the cycle-accurate approach, the simulation performance is only about 3 MIPS and the results contain 10% errors. Similarly, Graphite [12], a distributed parallel multicore functional simulator, also applies a periodical synchronization approach. Each ISS randomly chooses another ISS for clock time comparison and the front-running ISS is put to sleep and woken up later at a dynamically determined period of time. Although efficient, the periodical synchronization approaches cannot guarantee accurate executions.

To guarantee deterministic results for parallel applications, Kendo [13] is a software-based multithreading approach that enforces a deterministic interleaving of lock acquisitions based on a deterministic scheduling algorithm. Nonetheless, the specific scheduling order introduces considerable performance overhead. In contrast, the DMP approach [14] leverages the Transactional Memory technology and implements fully deterministic inter-thread communications. It encapsulates each data access in a transaction and atomically executes the transactions with a deterministic commit order. Although this approach has negligible performance overhead, it has to rely on a specific transaction-memory-equipped hardware that is not generally available.

As discussed in the introduction section, the shared-variable synchronization approach developed by Wu et al. [4], [5] is efficient and guarantees deterministic and accurate concurrent system behavior simulations. However, the synchronization overhead of the shared-variable approach is still significant.

In practice, software developers most likely use critical sections to protect critical shared accesses. Since the number of critical sections is much less than the number of shared variables, the critical-section approaches are more efficient. However, all existing approaches [10], [15] honor only the atomicity constraints but not the execution order of critical sections and hence cannot guarantee deterministic simulation results.

Some attempt to ensure equivalent atomicity between the target and host machines. For instance, PTLsim [15] is a cycle accurate full system x86-64 micro-architectural simulator that allows each atomic operation to acquire a specific lock and manages physical lock addresses using a shared-controller to all simulation threads. Although this approach avoids using a global lock to all memory regions so that parallel accesses to different memory regions are allowed, the extra simulation overhead for lock-status monitoring is simply impractical.

Instead, COREMU [10] adopts compare-and-swap instruction from specific host machine architectures such as x86 to implement equivalent atomic instructions. An issue is that the equivalent atomic instruction such as compare-and-swap instruction may not be available on ARM or other architectures. Therefore, the approach is limited and is not portable.

To resolve the issues discussed above, we propose an efficient critical-section level deterministic timed MCISS approach which is not limited to the underlying host architectures. The details are elaborated in the following section.

III. CRITICAL-SECTION-LEVEL TIMING SYNCHRONIZATION

With all shared variables protected in critical sections, a consistent lock-unlock order can guarantee deterministic multi-core instruction-set simulations. We extend Wu's timing synchronization framework [4] and develop a fast and accurate deterministic critical-section-level MCISS.

As discussed in the introduction section, most parallel program developers do use critical sections, which are composed of lock-unlock operations, to protect accesses of shared variables. With this fact, synchronizations of the shared variables inside critical sections can be omitted as they follow the order of lock operations.

We use the case in Fig. 3 to illustrate how the synchronization of critical sections works. Here we have a two-core target system execution with two ISSs entering a critical section each in the indicated chronological order. Using Wu's *Shared-Variable-Level* approach, it synchronizes, as shown in Fig. 3(a), at every shared-variable access point, which can be either a lock access point marked as a red dot or a regular shared-variable access point marked as a green dot. In contrast, for our proposed *Critical-Section-Level* synchronization approach it is sufficient to consider synchronization only at lock accesses, as shown in Fig. 3(b), while achieving the same simulation results.

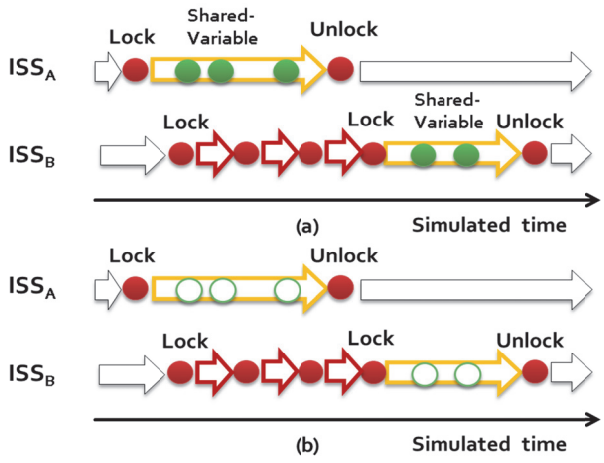

Fig. 3: A two-core target system simulation example shows that ISS$_A$ locks in a critical section first, and then ISS$_B$ tries to acquire the lock to enter in a critical section. (a) In the *Shared-Variable-Level* approach, all lock access points (red dots) and regular shared-variable access points (green dots) are considered synchronization points; (b) In the preliminary *Critical-Section-Level* approach, only lock-variable access points (red dots) are considered.

In the following, we elaborate details of issues and solutions associated with the proposed *Critical-Section-Level* synchronization approach. To simplify the discussion, the sync point before each lock or unlock operation is referred to as a lock sync point. We begin by discussing how to identify lock sync points.

A. Identifying Lock Sync Points

To identify a lock operation for a sync point, we usually do so by recognizing certain architecture-specific lock instructions. To identify an unlock operation, we usually check if the *move* (or memory store) instruction occurred after the identified lock instruction. Note that the unlocking *move* instruction always writes a certain value (1 in x86) to the lock address to release the lock so in practice we only need to check if the data address is a target lock address. If the addresses match, then this will be an unlock operation. Sync points are inserted before each lock and unlock operation.

To manage the lock status precisely, we simply use a lock address table to record each unique lock address and its associated status. The lock status is set when the lock is acquired and is cleared when the lock is released.

Depending on the host architectures provided, we detect lock operations by identifying the atomic read-and-update instructions such as test-and-set [16], compare-and-swap [16] or a pair like load-link/store-conditional [17]. These lock instructions are used to implement lock operations for parallel programming purposes and enforce atomicity across operation threads. We use x86 assembly codes of spin lock and spin unlock in Fig. 4 to demonstrate how to identify lock sync points through architecture specific lock and move instructions.

Spin lock is a technique used in the Linux multiprocessor kernel [18] environment for lock acquisition. If a processor finds that the target lock is *open*, it can acquire the lock for subsequent executions. Conversely, if the processor finds that the target lock is *closed* (or *locked*) by another processor, it *spins* around by repeatedly executing an instruction loop until the lock is released (or *opened*). The spinning process represents a *busy wait* and consumes actual CPU time

As illustrated in Fig. 4(a), the x86 lock instruction *decb* decreases the spin lock value. A test is then performed on the sign flag. If it was cleared, or the spin lock was set to 1 (unlocked), then execution continues to the line labeled L_3. Otherwise, a busy-waiting loop at the line L_2 is executed until the spin lock assumes a positive value. Then it renews execution from the line L_1 and checks if it is

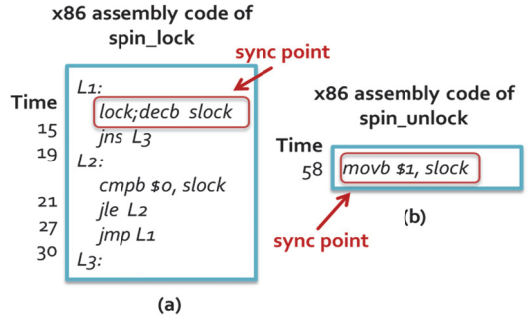

Fig. 4: (a) The x86 assembly codes of spin lock and (b) spin unlock.

indeed safe to proceed without having the lock acquired by another processor.

Conversely, the corresponding unlock operation cannot be easily identified as it is simply just a *move* instruction, which sets the value 1 into the lock memory address to release the acquired lock, as illustrated in Fig. 4(b). Nevertheless, since an unlock operation has to follow and match with a lock operation to form a critical section, we only need to examine the *move* instructions executed after a lock instruction is identified. As the number of move instructions inside a critical section is usually small, the extra checking overhead induced by our lock sync point identification approach is negligible in practice.

When an ISS encounters a move instruction inside the critical section, it looks up the lock address table and checks if the data address of the move instruction is in the table. If so, then it is an unlock sync point. Then the ISS performs the unlock operation and leaves the critical section to continue its non-critical section execution until next lock sync point.

In this way, our synchronization approach can systematically enforce the deterministic execution order of lock-unlock operations by maintaining the lock acquisition status. Also, the integrity of atomicity is guaranteed by maintaining the lock acquisition status and the annotated time of each instruction. We therefore have a generic solution that is applicable to different host architectures and can guarantee true atomicity exactly as in the physical target system.

B. Spin-waiting Optimization

To further improve the performance, we find that we can eliminate unnecessary synchronization overhead by optimizing the spin-waiting lock acquisition process in simulation. The two-core target system simulation example in Fig. 5(a) demonstrates a preliminary Critical-Section-Level simulation result without spin-waiting optimization. Suppose that ISS_A acquires a lock first and then ISS_B tries to acquire the lock later and is in spin-waiting mode before the lock is released by ISS_A. With each lock instruction considered as a sync point, the process of spin-waiting introduces excessive sync points, as shown in Fig. 5(a).

In fact, when ISS_B is attempting lock acquisition, we can easily identify whether the lock has been acquired by others by checking the status of the corresponding lock entry on the lock address table. If the corresponding lock status indicates that the lock

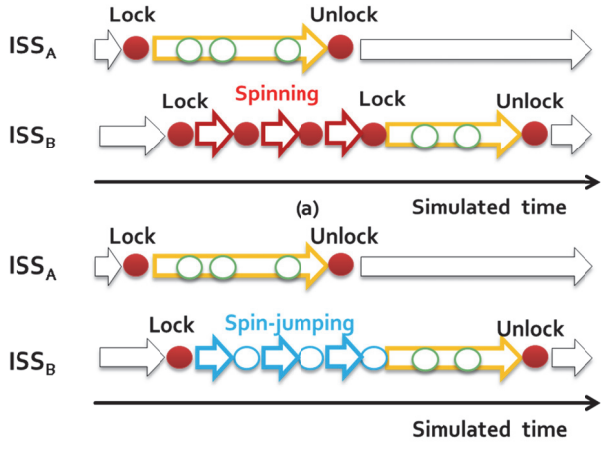

Fig. 5: A Critical-Section-Level two-core target system simulation example illustrating spin-waiting optimization. (a) A preliminary approach synchronizing at every lock operation, including spin-waiting lock instruction. (b) Spin-jumping for reduced overhead.

is held by another ISS, then we simply put ISS_B into a *spin-jumping* mode without doing actual synchronization until the lock is released, as shown in Fig. 5(b). The name *spin-jumping* indicates that it does not require synchronization at each lock acquisition attempt but only the first lock attempt. Therefore, the overhead is reduced. Most importantly, the execution order of critical sections is still correctly maintained with spin-jumping.

The accurate lock activating time for ISS_B can be easily calculated based on its initial lock acquisition time and the time span of each spinning period. In the next section, we elaborate the details of timing calculation for spin-jumping and lock activation.

C. Lock Activation

An interesting and non-trivial issue is that in case of multiple lock-competing ISSs, the lock winner is determined by its lock activating time, but not the initial lock acquistion time. To ensure correct timing behavior, we have to compute the exact activating time of the lock-waiting ISS through the following spin-jumping time calculation mechanism. The spin-jumping time can be calculated as

$$T_{jump} = \left\lceil \frac{T_{lock\_release} - T_{lock\_attempt}}{T_{spinning\_period}} \right\rceil * T_{spinning\_period},$$

where $T_{spinning\_period}$ is the period of time for repeating lock acquisition attempt, or (t_1-t_0) as in Fig. 6. $T_{lock\_release}$ is the time when the lock-holding ISS releases the lock, or t_2 in Fig. 6. $T_{lock\_attempt}$ is the time when the lock-competing ISS first attempts its acquisition of the lock, or t_0 in Fig. 6. Since the lock-releasing time may not match the spinning period, we take the ceiling of $\left(\frac{T_{lock\_release} - T_{lock\_attempt}}{T_{spinning\_period}}\right)$ to compute the precise spin-jumping time and add it to the initial lock acquisition time t_0 to derive the actual lock activating time t_3.

If there are more than one lock competitors, we calculate the possible lock activating time according to the above method and consider the lock winner to be the one whose lock activating time is closest to the lock release time. In Fig. 6, ISS_C is the lock winner since the lock activating time of ISS_C, t_3, is the closest to the lock release time, t_2, compared to another lock activating time of ISS_B, t_4. In case of a tie, then we simply follow the priority rule of the target system to determine the winner. Afterward, the new activating time is re-computed based on the timing of the new winner.

D. The Critical-Section-Level Simulations

We now use the flow chart in Fig. 7 to summarize our proposed Critical-Section-Level approach. For executions on each ISS, once a lock operation is identified, a lock sync function is invoked.

As shown in Fig. 7(b), the lock sync function first confirms if the local time of the invoking ISS is the minimum one among all

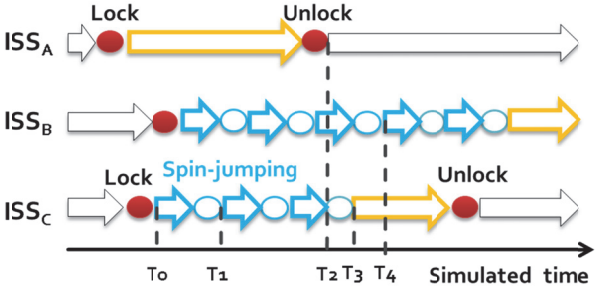

Fig. 6: An example illustrating timing calculation of spin-jumping.

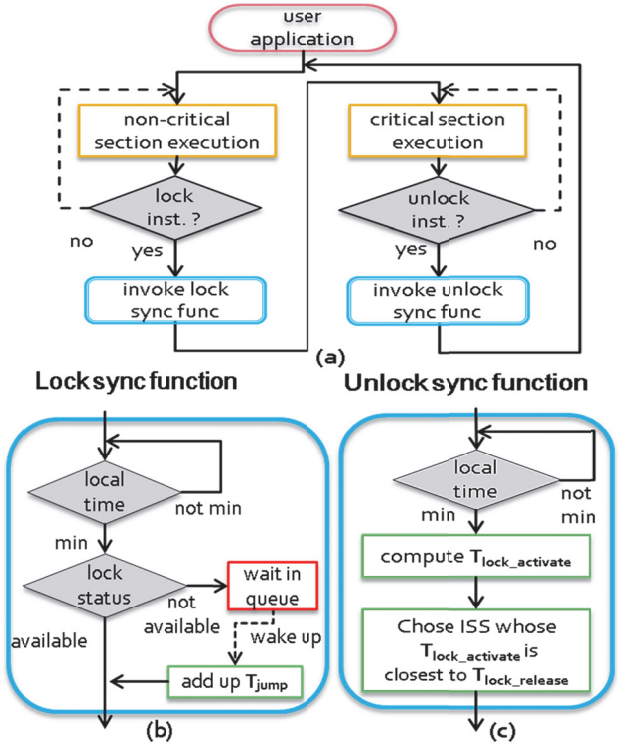

Fig. 7: The flow charts of (a) the proposed Critical-Section-Level Synchronization approach, (b) the lock sync function and (c) the unlock sync function.

sync points (including lock sync and unlock sync points). After being confirmed to be of the minimum local time, the lock sync function then checks whether the target lock is free. If it is free, then the ISS acquires the lock and enters critical-section execution. If the lock is not available, then the ISS waits in a lock-specific queue until the lock is released. Note that in this case it is important to add the spin-jumping time to the waiting ISS for accurate timing.

As depicted in Fig. 7(c), if an unlock operation is identified, the unlock sync function compares the local time of the invoking ISS with other sync points to confirm that it has the minimum local time before releasing the lock. Then it computes the lock activating times of all lock-waiting ISSs and selects the one with the closest lock activating time to the lock release time as the winning lock; other waiting ISSs simply stay in the queue for next lock release point.

Unlock operations are usually done by *move* instruction and each unlock operation has to match and occur after a lock instruction. Based on this fact, we simply check the data address of every *move* instruction after an identified lock instruction or in a critical section until we recognize that the data address is a lock address. Only then the unlock operations release the lock.

For cases with nested locks or recursive locks, i.e., having lock sync points inside a critical section, we have to follow the same lock acquisition flow to determine when these nested locks can be acquired or released. When encountering an unlock operation, we have to make sure whether we are still in a nested critical section before switching to a non-critical section mode for execution. If so, we free only the current critical section and continue monitoring higher level critical sections until all critical sections are freed. We repeat the above process until the user application terminates.

IV. Correctness Proof

In this section we show the correctness of our proposed Critical-Section-Level synchronization approach. The *Machine Correctness Theorem* in [19] has proved the correctness of the shared-memory synchronization approach, which guarantees the same results as that from the original program. In our case, when all shared variables are protected within critical sections, we can prove that our approach of synchronizing at lock-unlock operations does guarantee the same simulation results as those of the shared-variable approach, and hence as those of the original program.

A shared-variable synchronization execution, E, and its critical-section-level synchronization execution, E', are equivalent if and only if the execution orders of all the critical sections in every program of E' is the same as program of E, and moreover, all the shared variables are protected inside the critical sections.

The following notations are used for clarity. Let s stand for a shared variable, l stand for a lock operation and u stand for an unlock operation. T and T' are the access time on the shared-variable synchronization execution, E, and the critical-section-level synchronization execution, E', respectively.

Assume shared variables $s_1, s_2,\ldots s_n$ are protected inside the critical section. Below, we show the denotation of the access time order of the critical section on the shared-variable synchronization execution E:

$$T(l) < T(s_1) < T(s_2) < \cdots < T(s_n) < T(u).$$

Below, we show the denotation of the access time order of the critical section by synchronizing at the lock-unlock operation on the critical-section-level synchronization execution E':

$$T'(l) < T'(u).$$

Since the critical section is always executed atomically, the access time of shared variables inside critical section falling before the access time of the lock operation and after the access time of the unlock operation is shown below as

$$T'(l) < T'(s_1) < T'(s_2) < \cdots < T'(s_n) < T'(u).$$

Therefore, the access time order of the shared variables inside critical section on the critical-section-level synchronization execution is equivalent to the shared-variable synchronization execution.

According to the *Machine Correctness Theorem*, the critical-section-level synchronization execution is equivalent to the shared-variable synchronization and the shared-variable synchronization execution is proved to be as same as the original program. Consequently, by maintaining the execution order of critical sections among simulated programs, the correctness of multi-core simulation is guaranteed.

V. Experimental Results

We have implemented and tested our proposed deterministic Critical-Section-Level MCISS simulation approach with excellent results. Our experimental setup uses a host machine equipped with an Intel Xeon 2.67 GHz quad-core and 4GB RAM. The target architecture for simulation is Andes 16/32-bit mixed length RISC ISA [20]. We tested our implementation with 4 target cores using the multi-core benchmark test cases FFM, FFT, Ocean, LU, and Barnes from SPLASH-2 [21] and a deterministic execution test case from RACEY [22].

The simulation performance comparisons are summarized in Fig. 8. In general, we can achieve a 295% speed up on average against the shared-variable approach. The observed top simulation

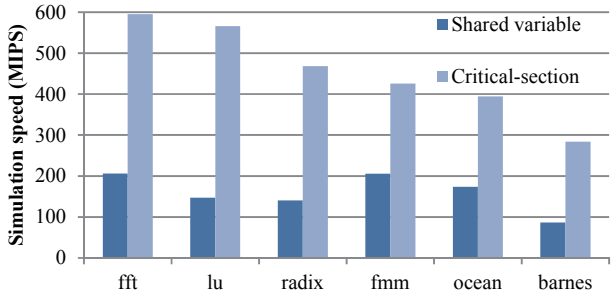

Fig. 8: A comparison of the shared-variable approach and the critical-section-level approach simulation speeds.

speed is improved to 596 MIPS. The average sync number ratio of lock sync points over shared-variable sync points of SPLASH-2 is 14.5%, which shows that our proposed approach has much less sync overhead compared to the shared-variable approach.

The accuracy of our approach is verified by comparing the computational results and the trace of critical section execution order of our critical-section level approach to that of the shared-variable approach, and we confirm that our approach is 100% accurate. Additionally, the test results, particularly those of RACEY, show that our proposed Critical-Section-Level approach is deterministic. In contrast, when running both SPLASH-2 and RACEY benchmarks with no synchronization, the results are non-deterministic and the variance of total simulated instruction count is about 50%.

We also confirm that the total number of sync points determines the sync overhead. From the test results of RACEY, we observe that the total sync overhead is proportional to the number of sync points. As shown in Table 1, the sync number ratio is 43%, which is almost equal to the sync overhead time ratio of 42%. The sync number ratio of RACEY is higher because it is a lock intensive program for determinism checking. Note that the sync number ratio is the number of lock sync points over all shared-variable sync points and that the sync overhead time ratio is the sync overhead time of our approach over that of the shared-variable approach.

VI. CONCLUSIONS

In this paper, we have proposed a new Critical-Section-Level timing synchronization approach for deterministic MCISS. Our proposed approach synchronizes only at the lock-variable access points instead of every shared-variable access point. The experimental results show that our Critical-Section-Level synchronization approach is 100% accurate compared with the shared-variable approach while performing 295% faster. Our future work is to include other synchronization points from other possible source-level operations such as barrier, signal, and wait. Additionally, it is also important to consider the scalability of the simulation approach.

TABLE 1: Comparison of the sync number and sync overhead of Critical-Section-Level and Shared Variable Level.

	Sync Number	**Sync Overhead**
Lock Sync	126523	17.4(ms)
Shared-Variable Sync	293125	41.6(ms)
Ratio	≈43%	≈42%

VII. REFERENCES

[1] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, 1997.

[2] 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," *Computer*,. 2002.

[3] J. Schnerr, O. Bringmann, and W. Rosenstiel, "Cycle Accurate Binary Translation for Simulation Acceleration in Rapid Prototyping of SoCs," in *DATE '05*

[4] M.-H. Wu, C.-Y. Fu, P.-C. Wang, and R.-S. Tsay, "An effective synchronization approach for fast and accurate multi-core instruction-set simulation," in *EMSOFT '09*

[5] M.-H. Wu, P.-C. Wang, C.-Y. Fu, and R.-S. Tsay, "A high-parallelism distributed scheduling mechanism for multi-core instruction-set simulation," in *DAC '*11

[6] M.-H. Wu, W.-C. Lee, C.-Y. Chuang, and R.-S. Tsay, "Automatic generation of software TLM in multiple abstraction layers for efficient HW/SW co-simulation," in DATE '10.

[7] R. Lantz, "Parallel SimOS - Performance and Scalability for Large System Simulation," PhD Thesis, Standford University, 2007.

[8] K.-L. Lin, C.-K. Lo, and R.-S. Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," in *ASPDAC '*10.

[9] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the UNENIX Annual Technical Conference*, 2005.

[10] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "COREMU: a scalable and portable parallel full-system emulator," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011,

[11] M. Gligor, N. Fournel, and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible MPSoC simulation," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2009.

[12] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010,.

[13] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: deterministic shared memory multiprocessing," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[15] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *ISPASS 2007.*

[16] Intel Corp., "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M.".

[17] E. Jensen, G. Hagense, and J. Broughton, "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors," Lawrence Livermore National Laboratory, Technical Report,1987.

[18] D. Bovet and M. Cesati, *Understanding the Linux Kernel, Second Edition*, 2nd ed. O'Reilly Associates, Inc., 2002.

[19] M.-H. Wu, C.-Y. Fu, P.-C. Wang, and R.-S. Tsay, "A Distributed Timing Synchronization Technique for Parallel Multi-Core Instruction-Set Simulation," *accepted by ACM Trans. Embed. Comput. Syst.*, 2012.

[20] Andes Technology Corp., "AndeStar instruction set architecture manual/Andes programming guide." 2008.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," *SIGARCH Comput. Archit. News*, 1995.

[22] M. Xu, R. Bodik, and M. D. Hill, "A 'flight data recorder' for enabling full-system multiprocessor deterministic replay," in *Computer Architecture, 2003.*