An Effective Synchronization Approach for Fast and Accurate Multi-core Instruction-set Simulation

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

ABSTRACT
This paper proposes a synchronization approach for fast and accurate Multi-Core Instruction-Set Simulation (MCISS). An ideal MCISS should run accurately in a real-time fashion. In order to achieve accurate simulation results of MCISS, a lock-step approach, which synchronizes every cycle, is commonly used. However, this approach introduces immense overhead and lowers the simulation speed. Instead of synchronizing every cycle, our approach synchronizes the MCISS based on the data dependency among the simulated programs. Therefore, the synchronization overheads can be highly reduced while the accurate simulation results are ensured. With the proposed approach applied, the simulation speed of MCISS is up to 40 ~ 1,000 million instructions per second (MIPS) in general.

Categories and Subject Descriptors
I.6.7 [**Simulation and Modeling**]: Simulation Support Systems

General Terms
Performance, Verification

Keywords
Instruction-set simulator, binary translation, multi-core, synchronization

1. INTRODUCTION
An Instruction-Set Simulator (ISS) has become an essential system-level design tool. With ISSs, software developers can validate their programs without the need of real target machines and thus significantly shorten design turnaround time. Also, the transparency and debuggability of the ISS can help developers quickly converge on design problems. After years of development, the single-core ISS is close to be ideal, i.e. accurate and fast. However, multi-core architecture gradually replaces single-core architecture due to the advance in semiconductor manufacturing process. As a result, in order to maximize the benefit of multi-core architecture, more and more software is now written in parallel programming models. Thus, it is crucial to accurately simulate the interactions among the parallel programs. Unfortunately, due to lack of an effective synchronization approach, the current solution for Multi-core ISS (MCISS) is insufficient for the need of both simulation speed and accuracy.

EMSOFT'09, October 12–16, 2009, Grenoble, France.

In a multi-core system, programs are executed *concurrently*. Intuitively, each core of the target multi-core system can be simulated by an individual ISS. Therefore, the parallel simulation should yield better performance if the host machine is also a multi-core system. However, without proper synchronization, the ISSs cannot guarantee the concurrency. Figure 1(a) shows an example of four ISSs running on two host cores, where the host OS *randomly* schedules ISSs to the underlying host cores. Note that *simulation time* indicates the time to execute ISSs on the host. Assume these simulated programs are launched at the same time. Their target time will become *non-synchronized* as depicted in Figure 1(b). Here, *target time* indicates the time that simulated programs execute on the target. The non-synchronized target time may result in incorrect execution order of simulated programs and hence the simulation results are incorrect thereby.

The traditional *lock-step* approach solves this issue by forcing each ISS to synchronize every cycle as shown in Figure 1(c), so the simulated programs can execute in synchronized target time as shown in Figure 1(d). Each cycle tick is sync point (the point to

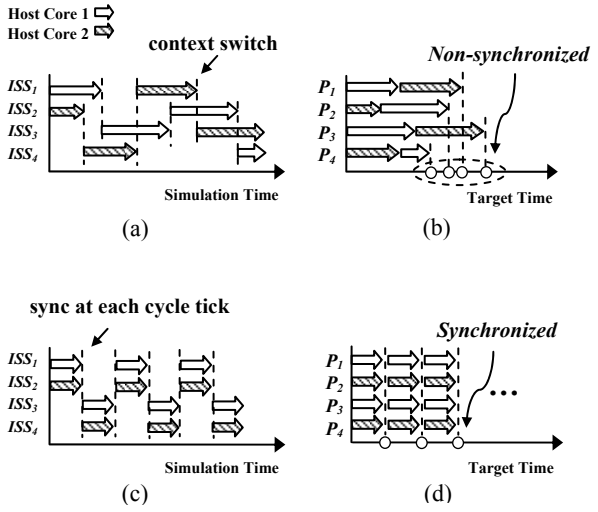

Figure 1: (a) Simulation of four ISSs running on two host cores, (b) Non-synchronized target time of the four simulated programs, (c) Synchronization in a lock-step manner, (d) Synchronized target time affected by the lock-step synchronization.

synchronize). At each sync point, the ISS has to stop and synchronize. The major drawback of this approach is the immense synchronization overhead.

Motivated by the needs of a fast and accurate MCISS, this paper proposes an effective synchronization approach. From our observation, the data dependency among simulated programs is determined by some essential points. As long as the execution order of these essential points is maintained, the accurate simulation results will be guaranteed. Therefore, we devise a mechanism that can ensure the in-order execution of these points. Meanwhile, since only the essential points need to be regarded as sync points, our sync point number is considerably smaller compared with the lock-step approach. As a result, the synchronization overhead can be significantly reduced.

To match with the performance of the new low-overhead synchronization approach, we adopt high speed *binary translation* technique [9] for instruction-set simulation and hence achieve real-time performance for MCISS. The experimental results show that the overall simulation speed is up to 40 ~ 1,000 MIPS (million instructions per second) in general. This proves that our approach is capable of fast and accurate multi-core simulation.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 explains the synchronization issues of multi-core simulation and the proposed approach. The employment of our synchronization approach on multi-core ISSs is described in section 4. Sections 5 and 6 discuss performance analysis and experimental results, respectively. Finally section 7 gives the conclusion and summarizes future work.

2. RELATED WORK

Before discussing synchronization issues, we first introduce the state-of-art ISSs for single processor systems. In general, ISSs can be classified into two types: interpretation-based and compilation-based. The *interpretation-based* ISS is implemented with a main loop which repeatedly processes fetching, decoding, and executing. The implementation is simple, but the simulation speed is slow (few MIPS) [1]. In contrast, the *compilation-based* ISS performs fetching/decoding at *compile time* and executing at *run time* [2-8]. This technique greatly saves fetching/decoding time, so it can speed up to dozens of MIPS. Furthermore, some compilation-based ISSs using *binary translation* directly translate target instructions into host instructions [2][3][4][7], which can achieve simulation speed of hundreds of MIPS. Though the ISSs for single processor simulation above have achieved extreme simulation speed, when it comes to multi-core simulation, the synchronization problem emerges as a major road block.

The synchronization problem for multi-processor simulation has been widely discussed. WWT-II [11] simulates a parallel shared-memory (physically distributed, logically shared) program on a parallel computer. In order to guarantee the correct simulation results, target execution is divided into quanta, *lock-step* intervals, and it is synchronized at the end of each quantum. For a lock-step approach, the quantum length is critical to simulation performance. Since target processors of WWT-II interact through network and therefore have a long quantum, the performance degradation is acceptable. Nevertheless, when the same approach is applied to a multi-core system where interactions are through an on-board memory, the quantum length becomes as short as the latency of memory access. The resulting excessive synchronization overheads will greatly damage simulation performance.

In contrast, Embra [4] simulates a multi-processor system in a round-robin fashion. Each processor is simulated in turns for a configurable time-slice. When the time-slice is chosen to be the execution time of a single instruction, the simulation result is nearly instruction-accurate, but the simulation performance is considerably poor.

Jung et al. [12] apply software analysis to improve performance for multi-processor systems. In a memory-mapped I/O architecture (i.e. same address bus for both memory and I/O devices), interrupts are triggered if particular memory addresses are accessed, so the authors statically analyze simulated programs to make pessimistic predictions of those accesses in advance. During simulation, a central server periodically delivers the information of the next access to each individual ISS. ISSs enter lock-step mode only when interrupts may possibly occur. Hence, its synchronization overhead is lower than that of the lock-step approach. Although the approach can be applied to resolve memory synchronization issue, the huge amount of memory accesses will force the ISSs to enter the lock-step mode frequently. Then the simulation will become as slow as the lock-step approach.

Kim et al. [13] propose a virtual synchronization technique to reduce the synchronization overhead in co-simulation. First, execution traces are generated by simulating each component separately. Then with the traces, the global time information is reconstructed to perform co-simulation. However, since interactions among components are not considered, the virtual synchronization technique cannot guarantee accurate multi-core system simulation.

In summary, the above mentioned approaches apparently are either incapable of managing synchronizations correctly or are unacceptable for its poor performance. The following section will introduce our proposed approach to solve this issue by first analyzing the mechanism of synchronization.

3. SYNCHRONIZATION

To efficiently solve the synchronization issue of multi-core simulation, the key of our approach is to identify the data dependency among simulated programs. Then the ISS of each core is individually synchronized according to this dependency, which can result in accurate simulation. The target architecture we simulate in this paper is a multi-core system with a shared-memory.

3.1 Data Dependency

In order to guarantee correct simulation results, the data dependency must be maintained. In a shared memory model, programs on different cores interact with each other through their data input/output. The data input/output are via memory accesses, so any two memory accesses with data dependency must be executed in order. Such dependency exists when two accesses to the same data (i.e., the same address) have any one of following relationships: (1) *WAW* (write after write), (2) *WAR* (write after read), and (3) *RAW* (read after write) [14]. To ease later discussion, we name *synchronization* the process of maintaining such data dependency relationship and the corresponding memory access point a *sync point.*

In nature, an ISS simulates a program sequentially so that the memory accesses within the same program are always in order. Hence, our synchronization approach only needs to check and

keep proper execution order for memory accesses across different programs. Theoretically, only memory accesses to the same address have to be checked for data dependency issue, but in practice, large memory space makes tracking each different address infeasible. Moreover, the exact accessing address is not always known in advance because indirect addressing mode (i.e. memory address indicated by a register instead of an immediate) is commonly used. A simple minded way is to treat every memory access a sync point, but the excessive number of memory accesses will result in poor simulation performance. Hence, we tend to reduce the number of sync points for better performance.

3.2 Sync Point Reduction

From our observation, data can be further categorized into shared variables and local variables. Only the memory accesses to shared variables actually have data dependency. The sync point number can be greatly reduced by taking these accesses as the only candidates. To identify shared variable accesses for sync points, the following strategy is applied. From the aspect of a program, memory space is partitioned into different segments as shown in Figure 2. Conventionally, a compiler allocates shared variables only at the shared data segment. Although the exact address of a memory access is unavailable, the indicating register may provide the hint of the pointing segment. For instance, a frame pointer register or a stack pointer register always points to the stack segment, where local variables are stored. Therefore, the memory accesses for these local variables can be easily identified and eliminated from sync points. Likewise, though in advance we cannot exclude the private data segment and the text segment from sync points, the address ranges of these segments can be obtained from the target program. Hence, we check and skip the memory accesses to these segments at run time. Consequently, the number of sync points executed is greatly reduced so that the simulation performance is significantly improved.

3.3 Avoiding Data Dependency Violations

Data dependency can be violated if the sync points are executed out-of-order, which will lead to incorrect data access and hence inaccurate simulation results. To avoid dependency violations, the sync points should always execute in order. As a result, we devise a synchronization mechanism for each individual ISS to maintain data dependency with others.

Without sacrificing generality, assume that at a simulation time point of interest, one ISS of the MCISS, say ISS_1 simulating P_1, encounters a sync point s_1. Meanwhile, other ISSs are either executing non-sync point instructions or waiting for wake up. Now we just pick any other ISS, say ISS_2 simulating P_2, and assume its earliest next sync point is s_2. If the estimated earliest target time t_2 of s_2 is later than the target time t_1 of s_1, there will be no data dependency violation. Then ISS_1 may proceed without conflict with ISS_2. If t_1 is later than t_2, and s_1 and s_2 have data dependency, a potential dependency violation may occur. For similar cases, ISS_1 has to wait until ISS_2 reaches the sync point s_2 in order to avoid the possible violation.

The concept of synchronization can be illustrated by the example shown in Figure 3, where the MCISS simulates programs P_1 and P_2 to the points indicated by the corresponding program counters (Figure 3(a)). Now assume P_1 is encountering a read sync point r at target time t_1, and P_2's next sync point, w, is a write access in a succeeding basic block. Though we do not know which branch will be taken, the earliest possible target time t_2 of w (when the branch outcome is for basic block b_{q+1}) can still be estimated. Then if $t_1 < t_2$, w's possible target time t_2' must be later than t_1 as in Figure 3(b). Hence, the ISS of P_1 is safe to proceed. On the contrary, if $t_2 < t_1$, w's possible target time t_2' may be either earlier or later than t_1 as shown in Figure 3(c). Then if $t_2' > t_1$, P_1 can safely proceed before P_2 executes w, whereas if $t_1 > t_2'$, P_1 cannot proceed until after P_2 finishes w. Therefore, to avoid the potential dependency violation, for such a case, the ISS of P_1 should wait until the dependency between t_1 and t_2' is certain. Consequently, by repeating the synchronization process on each of other ISSs, the data dependency can be maintained, and the simulation result is hence guaranteed to be correct.

4. MULTI-CORE SIMULATION

In this section, we will demonstrate how to incorporate of our synchronization approach on the MCISS. For simplicity, assume that the target multi-core executes one program on one assigned core, and cores interact with each other through an external memory. Our simulation flow is as shown in Figure 4. Each core is

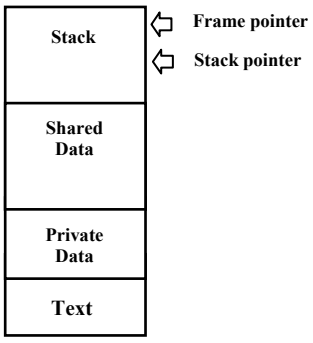

Figure 2: A typical memory space from the aspect of a program.

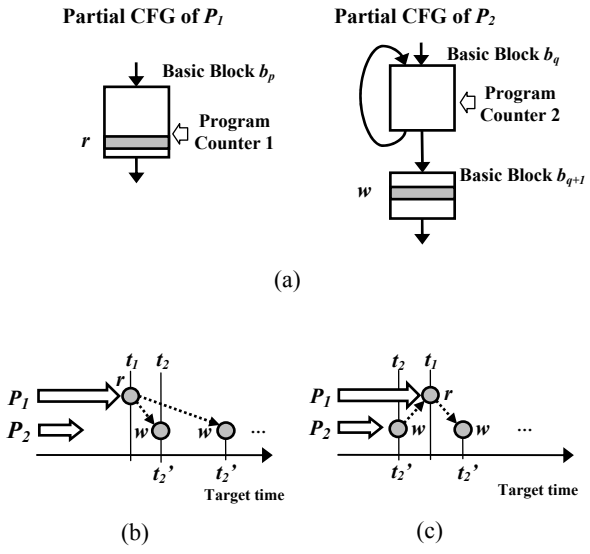

Figure 3: (a) The partial CFGs (Control Flow Graph) of Programs, P_1 and P_2, (b) The possible dependency when $t_1 < t_2$, (c) The possible dependency when $t_2 < t_1$.

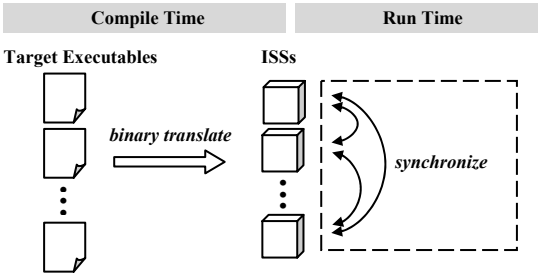

Figure 4: An overview of the multi-core simulation flow.

simulated by a binary translation ISS. At compile time, the target executables are translated into native codes. Then ISSs can simulate the behaviors of the corresponding target executables at run time. Besides, each ISS will perform synchronization to maintain the data dependency. The processes of *compile time* and *run time* are explained in detail.

4.1 Compile Time

The conventional binary translation has to be modified for synchronization purposes. Figure 5 illustrates the modified binary translation flow, where the shaded steps are specifically designed for synchronization.

For each target executable, we can disassemble it into *data* and *text*. The data part will be allocated and initialized on the host memory. We then identify the range of the shared data segment for checking potential sync points during run time. The translation procedure of the text part is further divided into several steps. First, target instructions are translated into intermediate codes for further manipulation. Then, sync points are identified and inserted accordingly. Additionally, in order to determine data dependency, target time information for each sync point is required. This can be obtained by timing annotation techniques, such as those proposed in several prior studies [10][15]. Subsequently, register allocation (i.e., mapping target registers onto host registers) is performed. Finally, all the functionalities are translated into equivalent host instructions for simulation.

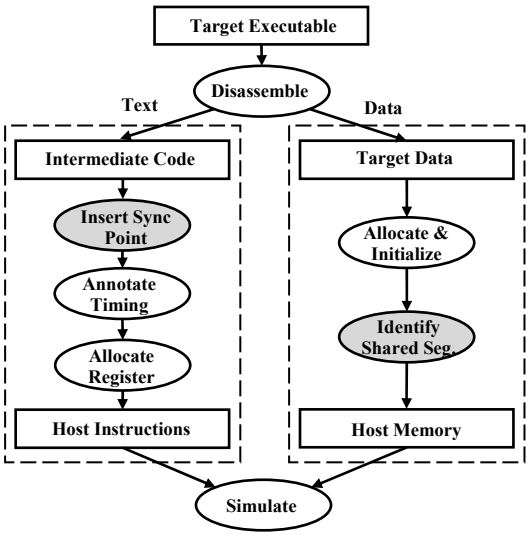

Figure 5: The flow of binary translation.

4.2 Run Time

Here we will focus our discussion on synchronization only, since the rest parts of simulation is similar to a conventional binary translation ISS. The proposed synchronization mechanism is illustrated in Figure 6. When encountering a sync point, an ISS will first check if the memory access is for the shared segment. If it is, a *sync function* will be called for maintaining data dependency; otherwise the sync point will be skipped. The sync function checks the sync table, which contains the information about all the earliest next sync points of ISSs. The function waits until all the others earliest next sync points are later than the current sync point. With this mechanism, the data dependency among ISSs can be ensured. The detailed implementation is as below.

```
1  void sync_function( unsigned int current_point ) {
2      for ( int i = 0; i < total_iss_num; i++ ) {
3          while( sync_table[ i ].earliest_next_point < current_point ) {
4              wait();  /* wait for other ISSs */
5          } /* end of while */
6      } /* end of for */
7  } /* end of sync_function */
```

To keep the sync table up-to-date, each ISS has to update whenever the earliest next sync point changes. This change takes place at the end of a basic block or at a sync point. Since each update consumes only one extra assignment instruction in our implementation, the overhead is considerably small. More importantly, our approach allows each ISS independently synchronizes with each other, without the need of a centralized scheduler. The synchronization approach leverages the parallelism of MCISS and hence greatly minimizes synchronization efforts.

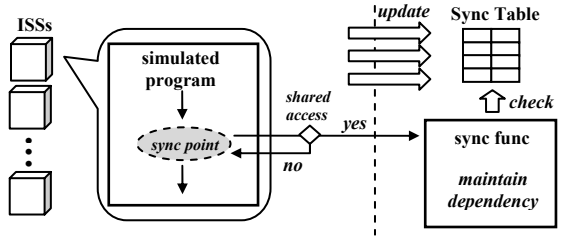

Figure 6: Synchronization during run time.

4.3 Interrupt Supported

Besides memory accesses, interrupts can affect the program execution. The proposed synchronization approach is not limited to synchronizing the interactions through memory. It can be extended to support interrupts.

An interrupt can preempt a program, but it cannot directly affect the execution result of the program. Instead, an interrupt triggers an ISR (interrupt service routine), and the ISR influences the preempted program through memory interactions as well. Therefore, the same synchronization mechanism can be used for maintaining the data dependency between the preempted program and the ISR, and the shared accesses are also treated as sync points. Besides, the execution results of other instructions between sync points will not be affected by interrupts. Consequently, we can handle interrupts when encountering a sync point. In practice, at each sync point an ISS will check whether any interrupt has been triggered. If there was any interrupt, a new ISS will be spawned

for each interrupt to simulate the corresponding ISR. Then based on the triggering time, the ISS of the ISR can be scheduled for synchronization. On the other hand, the ISS of the preempted program re-adjusts its schedule by incorporating the suspending time. Then the ISS waits at the encountered sync point until no dependency violation with the ISS of the ISR can happen. As a result, the effect of interrupts can be correctly simulated.

5. PERFORMANCE ANALYSIS

The effect of a synchronization approach on the simulation performance is analyzed in this section. Assume $t_{sim,i}$ is the time to simulate a target instruction i, then the simulation performance without synchronization is given by

$$P = \frac{n}{\sum_{i=1}^{n} t_{sim,i}} ,$$

where n is the total simulated instruction number. Moreover, suppose there are m executed sync points, and the synchronization time at a sync point j is $t_{sync,j}$. The performance with synchronization is given by

$$P_{Sync} = \frac{n}{\sum_{i=1}^{n} t_{sim,i} + \sum_{j=1}^{m} t_{sync,j}} .$$

Obviously, the number of sync points is the key factor to the synchronization overhead. In our approach, the sync point number varies depending on simulated applications, but it should be much smaller than that of the lock-step approach. The length of synchronization time at sync points also influences the simulation performance. At a sync point, an ISS is required to wait until the slower ISSs catch up, so the length of synchronization time depends on the gap between the given ISS and the slowest ISS. Therefore, the best performance is achieved when all ISSs are executed at nearly the same rate. To help evaluate the impact of synchronization overhead to simulation performance, we further define the synchronization efficiency as

$$E = \frac{P_{Sync}}{P} = \frac{\sum_{i=1}^{n} t_{sim,i}}{\sum_{i=1}^{n} t_{sim,i} + \sum_{j=1}^{m} t_{sync,j}} .$$

With this index, it is obvious that a faster simulator, given shorter simulation time, is more sensitive to synchronization overhead than a slower one. Since the binary translation ISS can perform two orders faster than the conventional ISS, it is crucial to employ an effective synchronization approach. Otherwise the synchronization overheads will dominate simulation performance.

6. EXPERIMENTAL RESULTS

The experimental results of the MCISS with our synchronization approach implemented will be demonstrated in this section. The host machine is equipped with Intel Xeon 3.4 GHz quad-core and 2GB ram, which runs Linux OS. The target machine is Andes 16/32-bit mixed length RISC ISA [16].

Figure 7 shows the simulation performance of different benchmarks. In this experiment, two target cores are simulated. The two benchmarks, Micro-benchmark and Fibonacci, do not have shared memory access. The experimental results show insignificant performance differences before and after the employment of our synchronization approach. The slight performance degradation is mainly due to timing annotation overhead. The synchronization efficiency is around 90%. This in fact demonstrates that our approach has small overhead to those programs without interactions. The other benchmarks, LU, Radix, and FFT, are parallel programs from SPLASH-2 benchmarks [17]. The overhead for synchronizing through the shared memory reduces the efficiency to 70%. Meanwhile, the overall simulation speed can still be up to hundreds of MIPS. Moreover, to test the effect of heavy shared accesses, we modify FFT to FFT-M by increasing the number of shared accesses to once every 12 instructions in average. As a result, the efficiency drops to 10%, but the overall speed remains about 40 MIPS.

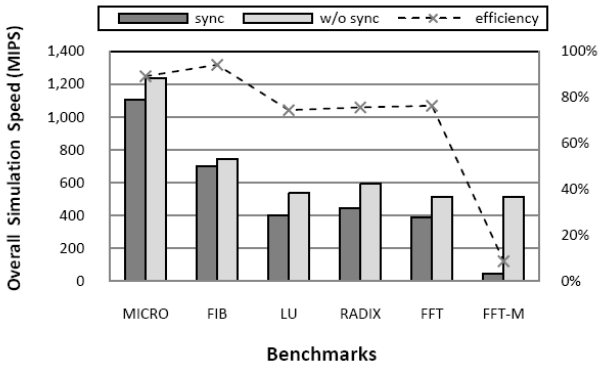

Figure 7: Simulation performance of different benchmarks.

In addition, we evaluate the simulation performance in terms of different number of target cores. Figure 8 illustrates the overall simulation speed from two to eight target cores. Only the test results of FFT are showcased because the results from LU and Radix are similar. For those MCISSs that employ synchronization approach to test cases with less than five target cores, the synchronization efficiency decreases as the number of cores increases. It is because more simulated (target) cores yield more sync points for simulation. Besides, each additional target core implies one more dependency checking at each sync point, and thus more overheads are created. Beyond four cores, the efficiency decrement appears flattened since the incremental overhead caused by additional core becomes minor. After all, the overall simulation speed maintains at 150 ~ 200 MIPS.

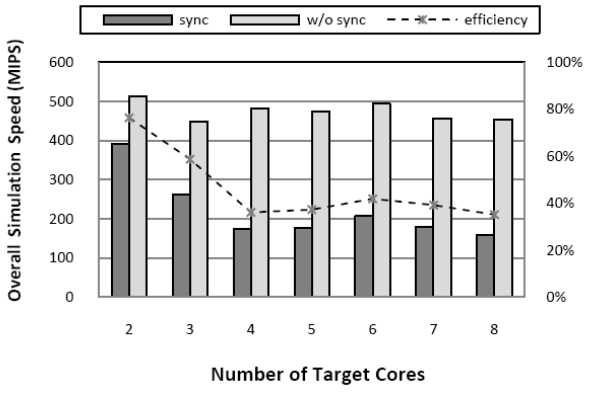

Figure 8: Simulation performance of FFT in different number of target cores.

In contrast, even without synchronization, the overall simulation speed does not improve as the number of target cores increases. This is due to the fact that the simulation itself is a memory-bound application, so the increased ISSs raise the possibility of memory bus contentions and hence limit the simulation speed.

To perform the stress test, we create a special test program and systemically adjust the *share ratio*, the ratio of the number of total executed shared accesses to the number of total executed basic blocks. Here each basic block contains 5.4 instructions in average. The special test program is designed to always make the worst case prediction in our synchronization mechanism, i.e. the earliest next sync point is located in the next one basic block. Figure 9 shows the synchronization efficiency under this worst case impact with different number of simulated cores. The results indicate that the synchronization efficiency improves as the share ratio decreases. It is because lower share ratio implies less synchronization efforts. Similar to the previous experimental result, the synchronization efficiency deteriorates as the number of simulated cores increases. As a result, for the case of 4-core and share ratio 1/1, the efficiency is as low as 4.5%, while the simulation speed still maintains at 34 MIPS.

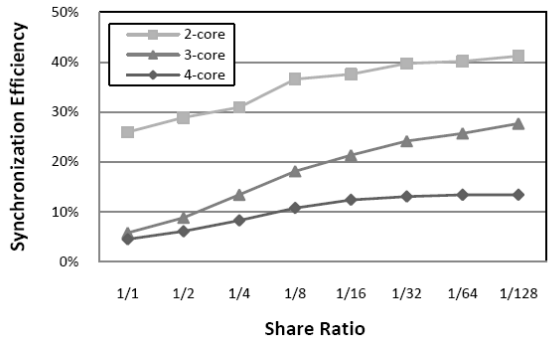

Figure 9: Synchronization efficiency under the worst case prediction in various numbers of simulated cores.

Once beyond four simulated cores, the worst case prediction dramatically degrades the simulation speed to less than one MIPS. In this situation, the host cores are less than the simulated cores, so there will be at least one idle ISS at any time. Moreover, the worst case prediction keeps those executing ISS waiting for the idle one. Thus, most of the execution time is wasted on waiting and context switching.

Table 1 shows the simulation speed comparison with other approaches. For all cases, when the lock-step approach is employed, the simulation speed of a binary translation MCISS is less than one MIPS due to the immense synchronization overhead. In contrast, our approach can achieve 40 ~ 1,000 MIPS for regular applications. Instead of parallel simulation, another approach cooperatively simulates each simulated program in a round-robin fashion. However, this approach is only suitable for conventional compilation-based ISSs, since binary translation ISSs are unable to be serialized. In addition, this cooperative approach cannot benefit from the performance of a host multi-core machine. Consequently, the simulation performance is limited, around 1 ~ 30 MIPS. Only if the application always makes the worst case prediction and the number of simulated cores is over that of the target cores, our approach can be less efficient than the sequential approach. Fortunately, normal applications rarely fall into the worst case type, so generally our approach is more effective than others.

Table 1. Simulation speed comparison with other approaches

Proposed		Lock-step	Cooperative
Normal	40 ~ 1,000 MIPS	< 1 MIPS	1 ~ 30 MIPS
Worst Case	< 1 MIPS		

The accuracy of our approach is verified by comparing the trace of shared memory accesses from our approach with that from the lock-step approach. Identical results and access order prove the accuracy of our approach.

7. CONCLUSION

We have presented and demonstrated an efficient synchronization approach for MCISS. Our major contribution is on clarifying the data dependency issue of a shared-memory multi-core system. The approach can effectively maintain data dependency. The experimental results show that the MCISS using our simulation approach can perform fast and accurate simulation. Although in this paper we only apply the approach to a binary translation MCISS, it can be incorporated by any compilation-based MCISS.

A future research topic could be the target system with multi-tasking. In our current work, we assume each program is fixed on one core. Yet for multi-tasking cases, a program can be dynamically assigned to different cores. The corresponding synchronization mechanism is more sophisticated and needs further investigation.

8. ACKNOWLEDGMENTS

This work was supported by National Science Council (Grant No. NSC96-2628-E-007-144-MY3) and the specification of Andes ISA was provided by Andes Technology.

9. REFERENCES

[1] Simplescalar, available at www.simplescalar.com

[2] J. Zhu and D. D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *DATE '99: Proceedings of the conference on Design, automation and test in Europe*. pp. 62-69, 1999.

[3] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. pp. 128-137, 1994.

[4] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. pp. 68-79, 1996.

[5] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *DAC '02: Proceedings of the 39th conference on Design automation*. pp. 22-27, 2002.

[6] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," in *DAC '03: Proceedings of the 40th conference on Design automation*. pp. 758-763, 2003.

[7] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of the USENIX Annual Technical Conference*, pp. 41-46, 2005.

[8] W. Qin, J. D'Errico, and X. Zhu, "A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. pp. 193-198, 2006.

[9] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Commun. ACM*, vol. 36, no. 2, pp. 69-81, 1993.

[10] J. Schnerr, O. Bringmann, and W. Rosenstiel, "Cycle accurate binary translation for simulation acceleration in rapid prototyping of socs," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. pp. 792-797, 2005.

[11] S. Mukherjee et al., "Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator," in *Concurrency, IEEE,* vol. 8, pp. 12-20, 2000.

[12] J. Jung, S. Yoo, and K. Choi, "Performance improvement of multi-processor systems cosimulation based on sw analysis," in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. pp. 749-753, 2001.

[13] D. Kim, Y. Yi, and S. Ha, "Trace-driven hw/sw cosimulation using virtual synchronization technique," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. pp. 345-348, 2005.

[14] J. Hennessy and D. Patterson, *Computer Architecture: a quantitative approach*, 4th ed., 2007.

[15] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. pp. 3-8, 2008.

[16] Andes, available at www.andestech.com

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. pp. 24-36, 1995.