A High-Parallelism Distributed Scheduling Mechanism for Multi-Core Instruction-Set Simulation

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

ABSTRACT

Ideally, multi-core instruction-set simulation should run in parallel to improve simulation performance. However, the conventional low-parallelism centralized scheduler greatly constrains simulation performance. To resolve this issue, we propose a high-parallelism distributed scheduling mechanism. The experimental results show that our proposed approach accelerates simulation by 6 to 20 times, depending on the number of cores.

Categories and Subject Descriptors

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

General Terms

Performance, Verification

Keywords

Instruction-set simulator, Multi-core simulation, Parallel simulation, Timing synchronization

1. INTRODUCTION

The *Instruction-Set Simulator* (ISS) is already an essential design tool for system development. As multi-core systems are gradually replacing single-core systems, the corresponding *Multi-Core Instruction-Set Simulator* (MCISS) is also becoming more crucial. Intuitively, to attain a MCISS, we can use a single-core ISS to simulate each target core and perform the co-simulation that runs all the ISSs in parallel to gain simulation performance.

Nevertheless, conventional co-simulation approaches such as SystemC [19] usually adopts a centralized scheduler to handle *timing synchronization* between each ISS, as illustrated in Figure 1(a). In order to maintain timing consistency, centralized scheduling always selects the slowest ISS for execution. Even if it allows parallel simulation, only one ISS can actually be executed for most of the time. Therefore, this approach highly limits the degree of parallelism of a MCISS. Considering the fact that the number of cores to simulate continues to increase, it is necessary to leverage parallelism to gain better simulation performance from the computing power of a host multi-core machine.

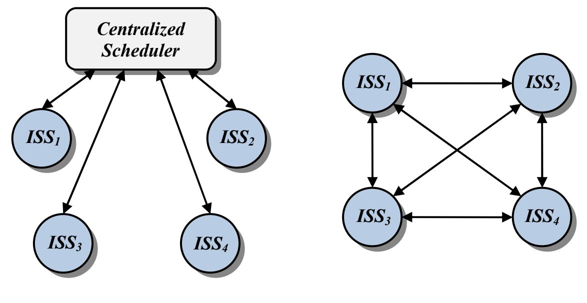
For better MCISS parallelization, this paper proposes a new distributed scheduling mechanism. By allowing each ISS to schedule with others autonomously, as illustrated in Figure 1(b), more ISSs can run at the same time. Furthermore, according to the characteristics of a MCISS, the proposed technique predicts the possible timing of future *sync points* (i.e., the time point for synchronization). Based on this prediction, the time spent on synchronization decisions can be effectively shortened so that our approach enables high simulation performance for a MCISS.

The experimental results show that as the number of cores increases, our distributed mechanism improves the parallel simulation performance by 6 to 20 times over the conventional centralized approach and attains the high simulation speed at 150 to 600 MIPS (million instructions per second). Hence, it can demonstrate our effectiveness on a parallel MCISS.

The remainder of this paper is organized as follows. Section 2 discusses related work. The simulation performance under different scheduling mechanisms is analyzed in section 3. Section 4 describes the proposed distributed scheduling mechanism. The experimental results and a brief conclusion are given in section 5 and section 6, respectively.

2. RELATED WORK

In this section, we first introduce ISS technique background and then discuss how previous co-simulation studies dealt with the timing synchronization issue.



(a) (b)

Figure 1. (a) An illustration of centralized scheduling; (b) An illustration of distributed scheduling.

2.1 Instruction-Set Simulation

In general, there are two major ISS types: *interpretive ISS* and *compiled ISS*.

Like a real processor, an interpretive ISS [1, 2] performs fetching, decoding, and execution for every instruction. Although the implementation of an interpretive ISS is straightforward, the simulation performance is poor, usually only of a few MIPS.

In order to speed up the simulation, a compiled ISS approach is proposed [3, 4]. As opposed to an interpretive ISS, a compiled ISS fetches and decodes all instructions at compile-time. Then, during simulation (i.e., run-time), only the execution part of instructions is performed. The advantage of the compiled approach is clear for cases with loops of repeatedly executed instructions or functions. In other words, the interpretive ISS performs fetching/decoding for each instruction regardless of repeated executions, while a compiled ISS does not. Hence, a compiled ISS performs much faster than an interpretive ISS, usually by dozens to hundreds of MIPS.

Nonetheless, for software applications that are not run-time static, the exact binary codes cannot be specified in advance. To resolve this issue, a *dynamic compiled ISS* [5-9] translates only the piece of codes about to be executed on-the-fly rather than the whole program at the beginning. For better performance, translated codes of possible repeated executions can be cached to reduce translation time. Therefore, a dynamic compiled ISS has both the advantages of the flexibility of an interpretive ISS and the performance of a compiled ISS.

However, when it comes to multi-core simulation, timing synchronization emerges as a critical challenge, as discussed below.

2.2 Timing Synchronization

Timing synchronization is used to keep timing consistency for ensuring accurate concurrent behaviors of multiple simulated components. An intuitive approach is to synchronize all components at every cycle. This approach is usually named the cycle-based or lock-step approach [2, 6]. Though it offers accurate simulation, however, the heavy synchronization overheads would significantly slow down the simulation. Enlarging synchronization intervals could certainly improve performance, but it would also result in inaccurate simulation.

In some cases, properly extending synchronization intervals is helpful. For instance, since the messages communicated through a bus actually determine the interactions between processors, WWT-II [11] argues that instead of one cycle, the message delivery time to target, or bus communication latency, can be set as the basis of the synchronization interval. In this way, the target receiver is guaranteed to be able to process all related arrival messages before responding in action. Then, accurate simulation results can be ensured.

If an architecture has a large intercommunication latency, such as that of the distributed multiple processors system, this enlarged synchronization interval approach will be efficient. Nevertheless, the latency of on-chip communication is considerably smaller for the multi-core processor. Correspondingly, the synchronization interval becomes too short to be acceptable for a high-speed MCISS.

In order to attain a fast and accurate co-simulation, partial order synchronization approaches are proposed [14-16]. The idea is to maintain correct data flow, i.e., data dependency. In reality, programs can only influence each other via their shared memory accesses. As long as the temporal order of all the shared memory accesses is maintained, consistent data dependencies between programs will be obtained. To do so, timing synchronization is only required to perform at each shared memory access. Since the number of shared memory accesses is considerably smaller than the number of total execution cycles, light-weight synchronization efforts allow this shared memory based approach to be more efficient than the lock-step approach. Meanwhile, this approach can guarantee accurate MCISS simulation results.

Following the discussions above, we know that the compiled ISS technique plus the shared memory based synchronization approach can construct a fast and accurate MCISS.

Accordingly, the distributed scheduling mechanism proposed in this paper is designed for this type of MCISS solutions. For a better understanding of the performance impact from different scheduling mechanisms, a comprehensive analysis is provided in the next section.

3. PERFORMANCE ANALYSIS OF CO-SIMULATION

Here we discuss the effect of *centralized scheduling* and *distributed scheduling* on the performance of co-simulation. Typically, co-simulation adopts the centralized scheduling mechanism for timing synchronization [12, 19]. A centralized scheduler is used to control the execution order of simulated tasks. Although it is easier to implement, the parallelism of simulated tasks becomes highly limited. Conversely, a distributed scheduling mechanism allows better parallelism by enabling each task to autonomously synchronize with other tasks without using a centralized scheduler. Details of the two mechanisms are respectively explained and analyzed below.

3.1 Centralized Scheduling

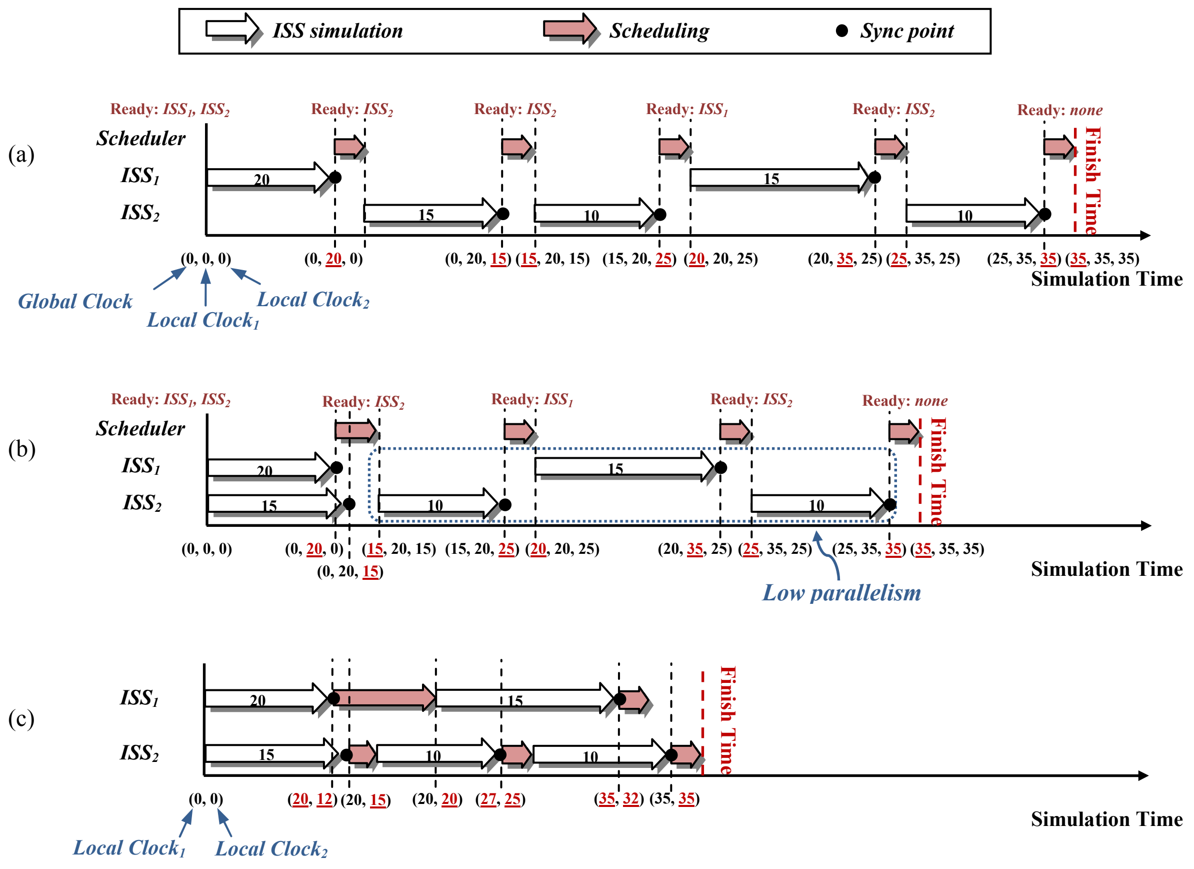
The centralized scheduling mechanism can be either sequential or parallel. The difference is that the sequential version cooperatively executes the tasks, so only one task is executed at one time. On the contrary, in the parallel version, more than one task can execute in parallel.

Figure 2(a) shows the timing diagram of a sequential MCISS example with centralized scheduling. Here, a sync point is annotated in front of each shared memory access, as mentioned previously. When an ISS encounters a sync point, a centralized scheduler will be invoked to determine the next active ISS. To ensure the temporal order of sync points, it always selects the slowest ISS to run.

To do so, the scheduler keeps a global time using a global clock while each ISS has a local clock to record its own local time. At the beginning, both the local clock and the global clock are initialized to zero. An ISS is *ready* only if its local time is of same value as the global time.

For sequential simulation, the scheduler cooperatively selects one of the ready ISSs for execution at a time. For the selected ISS, the corresponding local time advances along with the progress of execution. Then, at the next sync point, the executed ISS is pushed back to a waiting queue and the scheduler selects the next ready ISS to execute. Once there are no more ready ISSs, the scheduler will advance the global clock until an ISS becomes ready.

Note that both the local clock and the global clock represent the *simulated time* from the perspective of the target, i.e., *target time*.



Note: both the local clock and the global clock represent the *simulated time* from the perspective of the target, i.e., *target time*. In contrast, the *simulation time* is the time for the simulation to run on the host, i.e., *host time*.

Figure 2. (a) The sequential simulation timing diagram with centralized scheduling; (b) The parallel simulation timing diagram with centralized scheduling; (c) The parallel simulation timing diagram with distributed scheduling.

In contrast, the *simulation time* stands for the time for the simulation to run on the host, i.e., *host time*. Since the efforts to simulate various types of instructions may be different, a period of task execution with longer simulated time does not necessarily consume longer simulation time.

Ideally, we can shorten the whole simulation time by running the ISSs in parallel. Nevertheless, as illustrated in Figure 2(b), the major problem with the centralized scheduling mechanism is that only the ready ISSs can be actually executed at the same time. This implies that ISSs must have the same local clock time at a sync point, or they cannot be executed simultaneously. Unfortunately, this case rarely occurs in a MCISS with shared memory based synchronization, since programs seldom access shared memory at exactly the same time.

In other words, when the scheduler is invoked, typically only one ISS is ready. For example, in Figure 2(b), both ISS_1 and ISS_2 are ready at the very beginning. Other than that, only one of them is ready at other synchronization point, thereby leading to low parallelism. As a result, the parallel simulation of a MCISS with centralized scheduling barely improves simulation performance in practice.

Next, we will investigate how a distributed scheduling approach can enhance parallelism.

3.2 Distributed Scheduling

The high parallelism of distributed scheduling is achieved by allowing each task to do scheduling autonomously. Each task will track the local clocks of others at its own sync points.

Figure 2(c) depicts the simulation timing diagram of the same example under distributed scheduling. Similar to centralized scheduling, when encountering a sync point, an ISS can safely continue its execution if it has the slowest local clock. The advantage of distributed scheduling is that as long as an ISS has the slowest local clock, it can immediately continue the execution without further waiting for the scheduler.

For instance, when ISS_1 is suspended at the given sync point, i.e., 20, it will resume right after it finds that ISS_2 also advances its local clock to 20.

In contrast, for centralized scheduling, the decision for an ISS to continue execution must be made by the centralized scheduler. Since the centralized scheduler is invoked only when an ISS encounters a sync point, a waiting ISS still has to wait until the next invocation of the scheduler, even if it is indeed the slowest one.

Obviously, distributed scheduling allows the ISSs to attain a higher parallelism degree than centralized scheduling does.

The following section will describe how to further optimize distributed scheduling based on the characteristics of the compiled MCISS using shared memory based synchronization.

4. THE PROPOSED DISTRIBUTED SCHEDULING MECHANISM

Our research shows that we can gain further performance improvement by relaxing the requirement that an active ISS at a sync point can advance its execution only when it is the slowest ISS under the distributed scheduling approach.

Following the same scheduling idea of the shared memory based synchronization approach, the simulation result is still correct if the temporal order of all shared memory accesses (i.e., sync points) is maintained. In other words, an active ISS can safely proceed as long as others' next sync points are timed later than that of the local active sync point, even though it may not be the slowest ISS. This observation provides a performance enhancement opportunity for the distributed scheduling of a MCISS.

To implement this idea, the temporal relationships of sync points must be determined first. Unfortunately, it is impossible to identify a sync point's exact execution time until it is actually executed, since normally programs contain uncertain execution paths. Nevertheless, a prediction to the next earliest possible sync point is feasible and can be used to greatly improve the scheduling performance.

4.1 Static Prediction of Future Sync Points

Here, we propose an approach to make the best-case prediction of future sync points by statically analyzing the control flow graphs (CFG) of a simulated program. The CFG of a simulated program can be obtained at the translation phase of a compiled ISS.

With a CFG, the shortest path from one point to any other point is determinable and the shortest path can be used to estimate the best-case execution time. Based on this idea, the following algorithm is devised to identify the best-case execution time from any given point p to its next possible sync point.

```
PREDICT_NEXT_SYNC_POINT(p)

DEFINITION
    p : a given point
    block_x : the basic block that a particular point x belongs to
    head_x : the head of block_x
    tail_x : the tail of block_x
    TIME_bcet (x) : for a particular point x, the best-case relative
    execution time from head_x to x.

if p's next sync point s is also within block_p
then   return TIME_bcet(s) - TIME_bcet(p)
bcet := infinite
for each succeeding basic block block_i of block_p   do
    bcet' := PREDICT_NEXT_SYNC_POINT(head_i)
    if bcet > bcet'
    then    bcet := bcet'
end for
return  bcet + TIME_bcet(tail_p) - TIME_bcet(p)
```

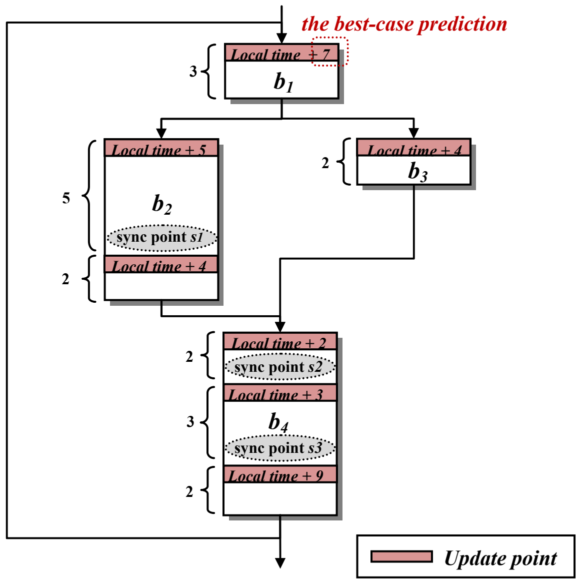


Figure 3. The best-case prediction for the next sync point.

For illustration purposes, assume we are examining the head of basic block b_1 in Figure 3. To predict its next possible sync point, we first check whether the next sync point belongs to the same block. If it does, the relative execution time to the sync point can be calculated directly; otherwise, we traverse its succeeding blocks to make a best-case prediction following the shortest path.

For this case, we have to check b_1's succeeding blocks, b_2 and b_3, in which b_2 has sync point $s1$, but b_3 has no sync point. Hence, we will recursively check succeeding blocks until reaching a sync point. In this case, a sync point $s2$ in the block b_4, a succeeding block of b_3 is identified. Then, the two paths, b_1 (3) → b_2 (5) → $s1$ of total delay 8 and b_1 (3) → b_3 (2) → b_4 (2) → $s2$ of total delay 7, are compared to find the best-case relative execution time, i.e., 7.

4.2 Dynamic Update of Next Sync Points

In order to allow synchronization, it is necessary to obtain information about the execution timing of each ISS's next sync point. Ideally, we can make a prediction for each instruction and update the information after executing one instruction, but this will introduce heavy overheads.

A practical approach is to pre-calculate the best-case delay to next sync point only at two types of points:

- *The tail of each sync point;*
- *The head of each basic block.*

As illustrated in Figure 3, the best-case predictions of the tails of sync point $s1$, $s2$, and $s3$ to their next possible sync point are 4, 3, and 9, respectively. In addition, the best-case predictions of the heads of basic block b_1, b_2, b_3, and b_4 are 7, 5, 4, and 2, respectively.

After encountering these update points during simulation, the predicted execution timing of the ISS's next sync point can be promptly determined summing up its current local time and the best-case prediction. In this way, every ISS can dynamically up-

Table 1. The performance speedup factors of the proposed distributed scheduling against the other three approaches

		Lock-Step		Centralized Scheduling		Distributed Scheduling w/o Prediction	
Benchmarks	**Shared Mem. Ratio**	**2 Cores**	**4 Cores**	**2 Cores**	**4 Cores**	**2 Cores**	**4 Cores**
Radix	0.19%	56.1	79.8	4.6	9.9	1.0	3.6
FMM	0.73%	46.9	80.5	3.7	11.2	1.0	2.5
FFT	1.70%	36.2	47.0	3.2	11.1	1.2	4.1
Ocean	1.82%	31.7	44.4	6.3	8.6	1.3	4.4
LU	3.19%	29.4	40.3	4.6	21.3	1.5	8.2
Barnes	14.21%	19.7	33.9	18.5	57.9	6.3	8.6
Average	3.64%	36.7	54.3	6.8	20.0	2.1	5.2

date the latest information about its predicted next sync point. Based on this information, one can decide whether to wait or continue execution when encountering a sync point.

4.3 Run-Time Scheduling based on Prediction

To illustrate how to do prediction-based scheduling, Figure 4(a) shows an example in which *ISS_2* is encountering a sync point at the time of interest, 25. Following the original distributed scheduling mechanism mentioned in section 3.2, *ISS_2* is supposed to wait for *ISS_1*, since it is ahead of *ISS_1*, which is still at time point 20. Assume that the best-case prediction to the next sync point of *ISS_1* is 15. In other words, the actual execution timing is at least 35, so *ISS_2*'s current sync point (i.e., 25) must be earlier. According to this temporal relationship, *ISS_2* can continue its execution safely without stopping.

Nevertheless, sometimes the best-case prediction may be too early, as shown in Figure 4(b). For this case, *ISS_1*'s predicted next sync point is earlier than the actual one, so *ISS_2* has to wait because of the false-predicted temporal relationship. Fortunately, as *ISS_1* keeps progressing, the prediction will be updated to approach the actual sync point. As long as *ISS_1*'s predicted next sync point becomes later than the current sync point of *ISS_2*, their temporal relationship will be correct, and *ISS_2* can resume its execution immediately. In contrast, the original distributed scheduling mechanism must keep *ISS_2* waiting until the local time of *ISS_1* is later than that of *ISS_2*.

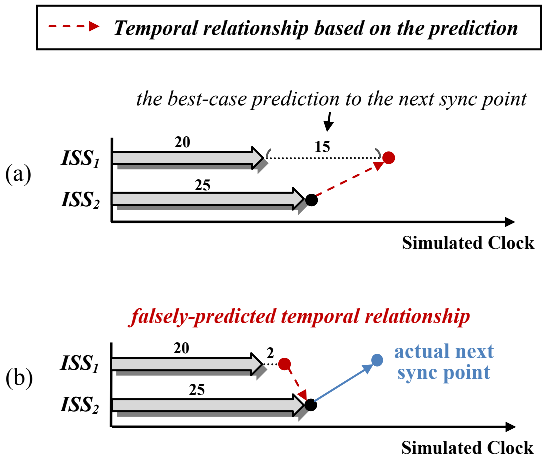

Temporal relationship based on the prediction
the best-case prediction to the next sync point
(a)
ISS1
ISS2
20
15
25
Simulated Clock
falsely-predicted temporal relationship
(b)
ISS1
ISS2
20
2
25
actual next sync point
Simulated Clock

Figure 4. (a) No waiting required if the predicted next sync point of ISS_1 occurs in future time; (b) A falsely-predicted temporal relationship due to prediction.

As a result, the proposed distributed scheduling mechanism would effectively shorten synchronization time in both cases and hence allow higher parallelism for a MCISS.

In practice, each ISS is implemented in the form of a thread or process on a host machine. Our synchronization is done by comparing an ISS's local time with the predicted time of the other ISSs' next sync points, so the prediction will be stored as global data or shared data for inter-communication purposes. Since every ISS can only modify its own prediction and the predicted time is monotonically increased, there is no need to use a semaphore to protect their inter-communication. Hence, the cost of checking the prediction of the other each time is considerably smaller (i.e., a single memory read access), making the scheduling process efficient.

5. EXPERIMENTAL RESULTS

This section summarizes our experimental results. In the experiment, we combine different scheduling mechanisms into an in-house developed compiled MCISS.

The setup is as follows. The target architecture for simulation is AndeStar 16/32-bit mixed length RISC ISA [17]. The parallel programs *Radix*, *FMM*, *FFT*, *Ocean*, *LU*, and *Barnes* from SPLASH-2 [18] are used as benchmark test cases. Our host machine is equipped with an Intel Xeon 2.6 GHz quad-core.

In order to test simulation performance under the maximum parallelism allowed by the host, we evaluate the cases of two and four simulated cores, respectively. Table 1 shows the performance speedup against the other three approaches previously mentioned. Note the *shared memory ratio* is the number of shared memory accesses over all the memory accesses.

The first experiment makes a comparison with the lock-step approach. With the advantage of considerably fewer sync points, the proposed mechanism outperforms it by a factor of 36 to 54 times. In general, the benchmark with a lower shared memory ratio leads to greater performance improvement, since our approach requires fewer synchronization efforts when the number of shared memory accesses is lower.

Compared to the same shared memory based synchronization but under parallel centralized scheduling, our distributed scheduling mechanism still achieves notable improvements of 6 to 20 times. Due to our prediction of future sync points, the speedup is even greater than the maximum parallelism allowed by the number of simulated cores. Hence, it can demonstrate the high parallelism achieved by the proposed mechanism.

Furthermore, in contrast to distributed scheduling without sync point prediction, as mentioned in section 3.2, the proposed mechanism still has 2 to 5 times performance speedup. The enhancement becomes significant as the shared memory ratio grows in general. This is because our prediction method effectively shortens the waiting time on a sync point. Correspondingly, the speedup becomes significant if there are more sync points, i.e., more shared memory accesses.

Overall, the proposed distributed scheduling mechanism attains better improvement as the number of simulated cores increases, indicating that our mechanism enables greater scalability than a conventional centralized scheduling mechanism.

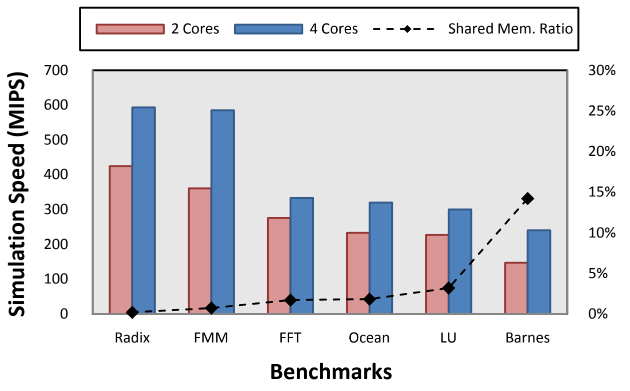


Figure 5. The simulation speed of the proposed approach.

Figure 5 shows our absolute simulation speed at two and four simulated cores respectively, which is also sensitive to the shared memory ratio. The test cases with lower shared memory ratios tend to reach higher speeds. Given different benchmarks, we can perform 150 to 600 MIPS, which means that the proposed approach is feasible for a high-speed MCISS.

In our experiments, the prediction method takes 12.4% extra translation time. Nevertheless, the total translation time is less than 10% of the total simulation time for a compiled ISS in general. Consequently, the overhead due to our mechanism is minor in terms of the whole simulation.

6. CONCLUSION

In this paper, we have proposed a new distributed scheduling mechanism for a parallel compiled MCISS. Our key contribution is to enhance the parallelism of the MCISS so that the computing power of a multi-core host machine can be effectively utilized. Timing synchronization is the bottleneck of parallelism, and the distributed scheduling with our prediction method significantly shortens the waiting time an ISS spends on synchronization. Hence, the proposed mechanism performs 6 to 20 times faster than conventional centralized scheduling approach. The simulation speed of our approach achieves 150 to 600 MIPS given the different ratios of shared memory access.

REFERENCES

[1] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0," in *SIGARCH Comput. Archit. News*, vol. 25, no 3, pp 13-25, 1997.

[2] P. Magnusson et al., "Simics: a full system simulation platform", in *Computer*, vol. 35, no 2, pp. 50-58, 2002.

[3] J. Zhu and D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *Proc. of Conference on Design, Automation and Test in Europe (DATE)*. pp. 62-69, 1999.

[4] M. Burtscher and I. Ganusov, "Automatic synthesis of high-speed processor simulators," in *Proc. of International Symposium on Microarchitecture (MICRO)*, pp. 55-66, 2004.

[5] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *ACM SIGMETRICS*, pp. 128-137, 1994.

[6] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *ACM SIGMETRICS*, pp. 68-79, 1996.

[7] 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 *Proc. of Conference on Design Automation (DAC)*, pp. 22-27, 2002.

[8] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," in *Proc. of Conference on Design Automation (DAC)*, pp. 758-763, 2003.

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

[10] J. Schnerr, O. Bringmann, and W. Rosenstiel, "Cycle accurate binary translation for simulation acceleration in rapid prototyping of SoCs," in *Proc. of Conference on Design, Automation and Test in Europe (DATE)*, 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 *Proc. of Conference on Design, Automation and Test in Europe (DATE)*, pp. 749-753, 2001.

[13] J. Chen, M. Annavaram, and M. Dubois, "Exploiting simulation slack to improve parallel simulation speed," in *Proc. of international Conference on Parallel Processing (ICPP)*, pp. 371-378, 2009.

[14] D. Kim, Y. Yi, and S. Ha, "Trace-driven hw/sw cosimulation using virtual synchronization technique," in *Proc. of Conference on Design Automation (DAC)*. pp. 345-348, 2005.

[15] M. Wu, C. Fu, P. Wang, and R. Tsay, "An effective synchronization approach for fast and accurate multi-core instruction-set simulation," in *Proc. of international Conference on Embedded Software (EMSOFT)*, pp. 197-204, 2009.

[16] M. Wu, W. Lee, C. Chuang, and R. Tsay, "Automatic Generation of Software TLM in Multiple Abstraction Layers for Efficient HW/SW Co-simulation," in *Proc. of Conference on Design, Automation and Test in Europe (DATE)*. pp. 1177-1182, 2010.

[17] AndeStar™ ISA, available at www.andestech.com/p2-2.htm.

[18] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Proc. of international Symposium on Computer Architecture (ISCA)*, pp. 24-36, 1995.

[19] T. Grötker, S. Liao, G. Martin, and S. Swan, System Design with SystemC, Kluwer Academic Publishers, 2002.