A Novel Synchronization Technique for Fast and Accurate Multi-core Instruction-set Simulation

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

ABSTRACT
This paper proposes a synchronization technique for fast and accurate Multi-Core Instruction-Set Simulation (MCISS). Traditionally, a lock-step approach, which synchronizes every cycle, is commonly used to achieve accurate simulation results of MCISS. However, this approach results in immense overhead and low simulation speed. Rather than synchronizing every cycle, our approach synchronizes the MCISS based on the data dependency among the simulated programs. Consequently, the synchronization overheads can be highly reduced while the accurate simulation results are ensured. With the proposed technique applied, the simulation speed of MCISS is up to 40 ~ 1,000 million instructions per second (MIPS) in general.

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.

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 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 guaran-

21th VLSI Design/CAD Symposium, Kaohsiung, Taiwan, August 2010
teed. 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 $40 \sim 1,000$ 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 explains the synchronization issues of multi-core simulation and the proposed approach. Our synchronization approach on multi-core ISSs is described in section 3. Sections 4 discuss experimental results. Finally section 5 gives the conclusion and summarizes future work.

2. 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.

2.1 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.

2.2 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 inaccurate simulation results. To avoid dependency violations, the sync points should 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 2, where the MCiSS simulates programs $P_1$ and $P_2$ to the points indicated by the corresponding program counters (Figure 2(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 2(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 2(c). Then if $t_2' > t_1$, $P_1$ can safely proceed before $P_2$ executes $w$, whereas if $t_2' > t_1$, $P_1$ 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.

![Figure 2: 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 $t_2 < t_1$.](image-url)
3. MULTI-CORE SIMULATION

In this section, we will demonstrate how to incorporate 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 3. Each core is 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.

<table>
<thead>
<tr>
<th>Compile Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Executables</td>
<td>ISSs</td>
</tr>
<tr>
<td>binary translate</td>
<td>synchronize</td>
</tr>
</tbody>
</table>

Figure 3: An overview of the multi-core simulation flow.

### 3.1 Compile Time

The conventional binary translation has to be modified for synchronization purposes. Figure 4 illustrates the modified binary translation flow, where the shaded steps are specifically designed for synchronization.

![Figure 4: The flow of binary translation.](image)

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.

### 3.2 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 5. 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.

```c
void sync_function( unsigned int current_point ) {
    for ( int i = 0; i < total_iss_num; i++ ) {
        while( sync_table[ i ].earliest_next_point < current_point ) {
            wait(); /* wait for other ISSs */
        }
        /* end of while */
    }
    /* 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 5: Synchronization during run time.](image)
4. 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].

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>
<thead>
<tr>
<th>Proposed</th>
<th>Lock-step</th>
<th>Cooperative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>&lt;1 MIPS</td>
<td>1 ~ 30 MIPS</td>
</tr>
<tr>
<td>Worst Case</td>
<td>&lt;1 MIPS</td>
<td>1 ~ 30 MIPS</td>
</tr>
</tbody>
</table>

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.

5. 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.

6. REFERENCES


