Speeding Up MPSoC Virtual Platform Simulation by Ultra Synchronization Checking Method

Yu-Fu Yeh [+] d94943035@ntu.edu.tw
Chung-Yang (Ric) Huang [+] ric@cc.ee.ntu.edu.tw
Chi-An Wu [+] b91901089@ntu.edu.tw
Hsin-Cheng Lin [†] r99921032@ntu.edu.tw

[+] *Graduate Institute of Electronics Engineering, National Taiwan University*

[†] *Department of Electrical Engineering, National Taiwan University*

***Abstract*—Virtual platform simulation is an essential technique for early-stage system-level design space exploration and embedded software development. In order to explore the hardware behavior and verify the embedded software, simulation speed and accuracy are the two most critical factors. However, given the increasing complexity of the Multi-Processor System-on-Chip (MPSoC) designs, even the state-of-the-art virtual platform simulation algorithms may suffer from the simulation speed issue. In this paper, we proposed an Ultra Synchronization Checking Method (USCM) for fast and robust virtual platform simulation. We devise a data dependency table (DDT) so that the memory access information by the hardware modules and software programs can be predicted and checked. By reducing the unnecessary synchronizations among simulation modules and utilizing the asynchronous discrete event simulation technique, we can significantly improve the virtual platform simulation speed. Our experimental results show that the proposed USCM can simulate a 32-processor SoC design in the speed of multi-million instructions per second. We also demonstrate that our method is less sensitive to the number of cores in the virtual platform simulation.**

Keywords-Virtual Platform Simulation, SoC, Synchronization.

I. INTRODUCTION

To conduct design space exploration in the early stage of the modern System on a Chip (SoC) design flow, designers begin to adopt virtual platform simulation. However, due to the increasing design size and complexity, the validatation of a design becomes time consuming. in this paper, we propose an Ultra Synchronization Checking Method (USCM) for speeding up the virtual platform simulation.

A SoC virtual platform is a software program that is sequentially simulated by a software simulator. To mimic the concurrent hardware behavior in a SoC design, the virtual platform simulator usually employs a synchronization mechanism to ensure proper communications in the right order and accurate computations at the right time. In other words, the simulation kernel needs to iteratively talk to each module on the corresponding triggering events in order to schedule its data computations and communications with proper delays. Consequently, the context-switches, introduced by the synchronization mechanism of the simulation scheduler, usually constitute the majority of the runtime [1].

Several studies have shown that a certain portion of synchronizations can be skipped and the simulation results can still be accurate. Most approaches that skip synchronizations are based on Asynchronous Discrete Event Simulation (async-DES, see more details in Section 2). In contrast to the traditional synchronous DES, in which the simulations of events among different design modules have to follow the chronological order, in async-DES, individual modules can continue its simulation for many cycles (without performing synchronizations) until "synchronization conditions" are met.

Result-Oriented Modeling (ROM) utilizes an optimistic approach to predict the outcome of a simulating process when the process starts [2]. The virtual platform simulator can then continuously simulate this process without any synchronization until the predicted time is up. However, if there are disturbing influences (i.e. causality errors) that might alter simulation results during the simulation, the simulator needs a recovery mechanism to roll back the simulation, thus affecting the simulation speed.

To avoid such errors, Kim [3] adopts a conservative approach, called "virtual synchronization", for its synchronization mechanism in simulating the operating system modelers. It depends on an operating system scheduling technique to determine synchronization conditions. However, because the timing granularity that the operating system scheduling can support is coarse-grained, this mechanism may not be applicable to the virtual platform simulation in which higher simulation accuracy (e.g. cycle-approximate accuracy) is needed.

The methods of virtual synchronization were further improved by Wu [4] and Lin [1], respectively, to support better simulation accuracy. Their key idea is to examine data dependencies among different modules/programs in order to recognize finer-grained synchronization conditions. Specifically, Wu's method adopts the disassembling utility to identify the static/heap memory regions of software programs. Wu utilizes memory information to predict where the data-dependency issues (DD issues) occur in the program and when the simulator needs to perform synchronization for the cycle-based processor model. However, in a SoC virtual platform, apart from the processor model, there are other hardware modules whose memory access information cannot be acquired from the disassembled program. Therefore, this method may be restricted.

On the other hand, Lin's method marks all memory regions that hardware modules potentially access. Some of the memory regions may overlap to each other, and the overlapped regions are the only places where data dependency may occur. During simulation, Lin [1] checks whether the current hardware module is accessing the overlapped regions. If not, there is no

978-3-9810801-7-9/DATE11/©2011 EDAA

need to perform synchronization and thus this module can simulate as long as it stays in its specific memory region. Otherwise, synchronization will be evoked. Although Lin's method is highly applicable to various virtual platform simulations with high accuracy, for Multi-Processor SoC (MPSoC) designs where almost all memory accesses are in the overlapped (shared) memory regions, their algorithm have to take the conservative approach and perform synchronizations in almost all cycles, reducing the performance of xxx.

In this paper, we propose an Ultra Synchronization Checking Method (USCM) that improves previous virtual synchronization mechanisms by monitoring the data dependencies on hardware modules as well as on software programs. We devise data dependency tables (DDTs) so that the memory access information of the hardware modules and software programs can be predicted and checked efficiently. Implemented in a robust data dependency checking flow, our method can greatly reduce the number of synchronization even in the MPSoC virtual platform simulation. In addition, the proposed approach preserves the principles of the conservative approach so that it rarely makes casualty errors and thus avoids simulation recovery/rerun. Our experimental results show that the simulation speed of our simulation engine can achieve multi-million instructions per second even for a 32-processor virtual platform. Moreover, our method is orthogonal to a parallel simulation mechanism that runs on a multi-core/processor host machine [4]. Therefore, our method may further increase the speed of virtual platform simulation in the parallel simulation mechanism.

The rest of this paper is organized as follows. First, we introduce the preliminary knowledge in Section 2 and then describe the details of the proposed USCM in Section 3. Section 4 demonstrates the experimental results and finally, Section 5 concludes the paper and discusses future work.

II. PRELIMINARIES

In this section, we present two techniques adopted in our work, that is, Asynchronous Discrete Event Simulation (async-DES) for synchronization reduction, and Data-Dependency (DD) checking for synchronization determination.

A. *Asynchronous Discrete Event Simulation*

The Asynchronous Discrete Event Simulation (async-DES) is an event-driven simulation algorithm that allows the design under verification to simulate asynchronously. In contrast to the Synchronous Discrete Event Simulation (sync-DES) in which the simulation kernel needs to synchronize each module of the design in every cycle/event to make sure the simulation is in the proper chronological order, the async-DES allows the individual module to simulate as long as possible until the necessary synchronization conditions causing data-dependency issues are met.

As shown in Figure 1, async-DES has a potential advantage in simulation speed because it can reduce the number of context switches introduced by the synchronization process. Nevertheless, it may produce some overheads by ensuring the simulation accuracy. As figure 1-(a) shows, the simulation

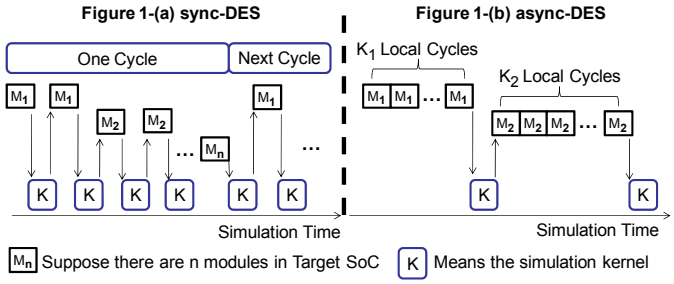

Figure1. sync-DES versus async-DES

kernel needs to spend extra efforts checking whether the synchronization conditions are met (the synchronization condition refers to the situation when the data dependency among different modules occurs). For example, when one module needs some data whose memory location is being updated by another module, its simulation needs to be halted until the memory location is available again. Therefore, a simple-minded algorithm may take a conservative approach to recognize the occurrence of data dependencies and thus perform synchronizations more frequently than necessary, decreasing the efficiency of the sync-DES

Another overhead in async-DES is the need to compensate the communication delays after synchronizations. Please note that when individual hardware modules are simulated continuously in Figure 1-(b), some of the communication delays (e.g. bus contention) might be ignored. For example, if two modules are simultaneously accessing the bus, trying to read/write multiple data from/into different memory locations, bus contentions would take place. In the sync-DES, proper delays will be inserted between data transactions so that the timing accuracy of the simulation can be ensured. However, since there is no data dependency in the async-DES, one module will be chosen to run the simulation as long as possible, and then the others will follow. In such case, it is difficult to observe the bus contentions during simulation. Therefore, we will have to apply techniques such as the trace-driven simulation [3] to restore the accurate simulation time. In short, with such overhead, async-DES can still achieve the same simulation accuracy as the sync-DES.

B. *Data-Dependency Checking*

As mentioned in the previous subsection, efficient data-dependency (DD) checking is the key to identifying the minimal necessary synchronization conditions and thus to promote the simulation speed. While data dependencies can be checked by examining whether modules read from or write into the same memory location, the challenge is that in async-DES, when one module executes in hundreds to thousands of cycles in advance of others, how we predict that it can safely continue the simulation without the worry about the data dependency with other lagging modules.

One solution is to define the memory access region from each module and then to create a map to identify potential data dependencies (cf. [1]). Note that we can construct this data dependency map in the virtual platform initialization phase. Therefore, during simulation, if a non-overlapped memory region is accessed, we can guarantee that there will be no data

dependency and thus no synchronization is needed. In general, for simplified SoCs, different hardware modules may have their own memory maps and thus the abovementioned approach seems to work. However, for modern MPSoCs, it is common to have shared memory among different processors. Moreover, some unexpected events (e.g. interrupt, out-of-order execution) and the embedded software programs will further complicate the memory access regions from different hardware modules. As a result, the above DD checking algorithm may conclude that the data dependencies should occur all the times and thus failing to avoid unnecessary synchronizations in the async-DES.

In the next section, we will present an aggressive DD checking algorithm, called Ultra Synchronization Checking Method (USCM), to reduce the unnecessary synchronizations during MPSoC virtual platform simulation.

III. ULTRA SYNCHRONIZATION CHECKING METHOD

As mentioned previously, data-dependency checking (DD checking) can be very complicated for the async-DES because it needs to accurately predict memory access regions by the software executing on the MPSoC hardware. To precisely judge data dependencies requires not just memory mapping information from the hardware but also the program/data storage of the embedded software. Furthermore, both of them need to be analyzed statically (i.e. at compile time) and dynamically (i.e. during simulation). Hence, we devise a data-dependency table (DDT) mechanism that can facilitate the analysis of various types of memory information. In what follows, we first present two types of DDTs, namely hardware-based and software-based DDTs, and describe how they can be further categorized into hardware static/dynamic DDTs and software static/dynamic DDTs, respectively. Then we consider the features of DDTs and propose the synchronization checking flow to accomplish the USCM.

A. Hardware-Based Data-Dependency Table

In virtual platform simulation, data-dependency issues arise when there are two or more hardware modules accessing the same memory location. In such situations, the simulator must perform synchronizations in order to simulate the hardware modules in a correct chronological order. The data dependency (DD) checking technique described in Section 2.2 determines such synchronization conditions. In other words, if the memory regions for all hardware modules can be recognized, the DD checking mechanism should be able to perform the proper synchronizations. However, the memory regions that the hardware modules access may be specified in compile time and/or changed during simulation. Therefore, we propose two different data dependency tables, that is, Hardware Static-DDT (HW S-DDT) and Hardware Dynamic-DDT (HW D-DDT), for these two scenarios.

1) Hardware Static-DDT

The hardware static DDT is constructed in compile time to record the overlapped memory regions that can be accessed by two or more hardware modules. Without loss of generality, the memory regions that the hardware modules will access can be found as memory maps in the specification sections of the header files. Therefore, our simulation engine will parse these header files to acquire the memory information and record them as hardware static DDT.

```
void pvt_dma::dmaOperation() {
  // Set The memory regions that  dma accesses
  // Embed HW_MAF_ADD func to acquire memory information
  pSync->HW_MAF_ADD(SourceAddr, SourceAddr+M_Size); // add a memory region
  pSync->HW_MAF_ADD(DistAddr, DistAddr+M_Size); // add a memory region
  cout << "DMA begin to move data ....\n";
  for(unsigned int i = 0; i < M_Size; i++){
    ReadMemory(SourceAddr, 4);
    WriteMemory(DistAddr, 4, m_resp_data);
    SourceAddr += 4; DistAddr += 4;
  }
  cout << "DMA Finish move data ....\n";
  // Embed HW_MAF_DEL Func to delete the useless memory information;
  pSync->HW_MAF_DEL(SouceAddr-M_Size, SourceAddr); // Delete a memory region
  pSync->HW_MAF_DEL(DistAddr-M_Size, DistAddr); // Delete a memory region
}

void pSync::HW_MAF_ADD (unsigned int MemBegin, unsigned int MemEnd) {
  HW_DDDT.push_back(new pair(MemBegin, MemEnd));
}
void pSync::HW_MAF_DEL (unsigned int MemBegin, unsigned int MemEnd) {
  itrHWDDDT = HW_DDDT.find(uPair(MemBegin, MemEnd))
  HW_DDDT.erase(itrModDDDT);
}
```

Figure2. An example of MA Function for HW D-DDT

Note that we record only the overlapped memory regions and sort them in an array data structure. During simulation, if a hardware module accesses a memory location, we check whether the corresponding memory address can be found in this hardware static DDT. If yes, the simulator will treat it as a synchronization condition and hand over the control to the simulation kernel. Otherwise, the access is not in the overlapped memory region and thus no data dependency is possible.

2) Hardware Dynamic-DDT

Although the hardware static DDT can identify certain data dependencies to improve the simulation speed (e.g. [1]), it is relatively conservative. When a hardware module accesses the overlapped memory region in the hardware static DDT, it does not necessarily cause data dependencies — the other modules may access this location at some other time.

A brute-force solution to this problem is to monitor the memory accesses of the hardware modules from time to time. However, this causes a big overhead, thus deteriorating the simulation speed. Therefore, we adopt a semi-automatic method in which certain "memory acquiring functions" (MA Functions) are manually embedded in the hardware modules. These MA functions denote the "authorized" memory regions that the modules can access with exclusive privileges. During simulation, these authorized memory regions will be automatically added to and later be removed from the hardware dynamic DDT when the corresponding MA functions are evoked. Consequently, when a memory access region is in the recorded regions of the hardware dynamic DDT, no data dependency can happen.

The example in Figure 2 illustrates our idea. The Direct Memory Access (DMA) controller is a common hardware module in SoCs to move mass data from the source to the

destination memory locations. In the source code, we embed MA Functions, HW_DDDT_ADD() and HW_DDDT_DEL() to acquire the dynamic memory information. That is, when they are evoked during simulation, DMA will be exclusively authorized to access this memory region. Therefore, no data dependency will be possible and thus the async-DES can continue until memory access in other regions takes place.

B. Software-Based Data-Dependency Table

When performing MPSoC virtual platform simulation, there are not just hardware modules to consider; there are also multiple embedded software programs running on multiple processor models. Failing to consider the effects of software programs in data dependency checking may lead to a conservative synchronization mechanism, slowing down the simulation. For example, most of the processor models can access all the shared memory regions. Therefore, when only referring to HW-Based DDTs, we will end up checking synchronizations in almost every cycle.

Next, we will present two software-based DDTs, namely the Software Static-DDT (SW S-DDT) and Software Dynamic-DDT (SW D-DDT). With them, we can further improve the data dependency checking by considering the software effects.

1) Software Static-DDT

As mentioned in [4], only when the software program possesses data-exchanging behavior (e.g. mutex, semaphore) in the shared memory should we consider its data-dependency issues. In other words, if a function of a software program contains only computations within the processor model, the corresponding hardware simulations will not result in data dependencies with other processor models. Therefore, to characterize the impact of the software programs on the synchronization mechanism, we should distinguish the software functions that perform data exchanges with other modules (i.e. the communication functions), with those computation functions. In our proposed USCM, we store the program memory information of the communication functions in the software static-DDT. During simulation, if the current program counter address does not match the memory information recorded in the software static-DDT, we can conclude that the current simulating function is a computation function and thus no data dependencies take place.

In general, it is possible to automatically identify the communication functions in a program (e.g. [6][7]). To prove the concept of the proposed USCM, we utilize interactive debuggers [8] and disassembly utilities [9] to extract the program memory information of the communication functions. We conduct this extraction before simulation and store it in the software static-DDT.

2) Software Dynamic-DDT

In most cases, the communication functions perform not only data-exchanging operations but also non-data-exchanging ones (i.e: buffering). The non-data-exchanging operations only interact with the local variables and thus will not induce data-dependency issues. Therefore, we can further reduce the number of synchronizations by identifying these local variable operations and record their memory information in the software dynamic-DDT. In other words, during simulation, if the current memory access matches the stored information in software dynamic-DDT, we can assure that there is no data dependency issue and the simulation can continue without synchronization.

```
void Comm_Task_1(unsigned int& BufferSize) {
  int* Buffer = new int[BufferSize]; // Buffer is the local variable for example
  // Embed MCF function
  MCF(Buffer, BufferSize, sizeof(int)); // Pass the memory information
  for (unsigned int ii=0; ii<BufferSize; ++ii) {
    buffer[ii] = GetData(); // GetData() refers to mutex_related function
    ..
    PutData(Buffer[ii*3+ii]); // PutData() referes to mutex_related function
  }
  ……
}

void MCF(int StartAddr, int Size, int DataTypeSize) {
  unsigned int Range = StartAddr + Size*(DataTypeSize/sizeof(int))
  // send memory information to DDSC
  *((volatile unsigned int *)DDSC_StartAddr) = StartAddr;
  *((volatile unsigned int *)DDSC_Private_EndAddr) = StartAddr + Range;
  *((volatile unsigned int *)DDSC_Private_Confirm) = 0x1234567;
}
```

Figure3. An example of MCF for SW D-DDT

However, the challenge is how the memory information of the local variables can be recorded during the at-speed simulation. There are two issues: (1) How to identify the local variables, and (2) How to obtain the memory information. For (1), we assume that the identification of the local variables is given. Therefore, we can embed a "memory catching function (MCF)" for each local variable in order to retrieve its memory information (for (2)).

Figure 3 can further illustrate this idea. Please note that we need a medium to collect all local variable information and communicate with the simulation engine. Therefore, we create a specific hardware module, called the data-dependency shadow controller (DDSC), as the medium. Through the DDSC, the simulator can receive the memory information from the communication functions. In the end, MCF will send a confirmation code (i.e. 0x1234567 in this example) to the simulator and then transfer the receiving memory information to the software dynamic-DDT. Similar to the life span of the local variables in a software program, the content of the software dynamic-DDT will also be cleaned up when exiting the current communication function.

C. Ultra Synchronization Checking Flow

As mentioned above, in order to reduce the number of unnecessary synchronizations in virtual platform simulation, we analyze the memory information and categorize the data dependencies into four data dependency tables: hardware static and dynamic, and software static and dynamic DDTs. Intuitively, detections of these data dependencies may correspond to the causality and different runtime complexities. Then the data dependencies lead to different reduction power in the number of synchronizations. Therefore, we should apply

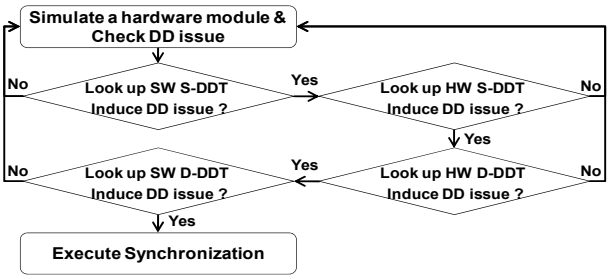

Figure4. The Ultra Synchronization Checking

DD checking wisely in order to achieve the best improvement in simulation.

Figure 4 shows the USCM flow. In short, our criteria is to first check the types of data dependencies with two principles: (1) When proven no data dependency issues in the corresponding category, can assure that no data dependency is possible for other categories either. (2) The chosen type of data dependency checks should be of low computation complexity.

The first analyzed DDT is SW S-DDT. Remember that the contents of SW S-DDT can be constructed and sorted in compile time. Furthermore, no matter what the instructions of the software function are simulated, the program counter of the simulating processor model refers to the same program memory region of the software function. Since the memory regions in SW S-DDT are sorted, we can use binary search to check whether the current program counter belongs to the communication functions in the SW S-DDT. Therefore, the time complexity is O(log(N)), where N is the number of distinct ranges in SW S-DDT. However, when simulating the following instructions in the same function, the time to check whether the simulating function is a communication function or not is only constant time due to the caching effect.

The next step is to analyze the HW S-DDT. Similarly, the memory information gathered in HW S-DDT can be constructed in compile time. Therefore, the searching for data dependency in this category takes O(log(M)) time, where M is the number of hardware memory map regions. This type of checking is also very fast and thus should be applied with the higher priority against SW D-DDT.

Finally, we perform the HW D-DDT and SW D-DDT checks. Because the contents of dynamic DDTs cannot be obtained in compile time and they are not stored in any particular order, we need to conduct linear search in order to determine whether the corresponding data dependencies may occur. Note that we choose to run HW D-DDT before SW D-DDT in our flow because the number of local variables in a function is usually larger than the number of memory regions in the HW D-DDT.

IV. EXPERIMENT RESULTS

To evaluate the effectiveness and robustness of USCM, we conduct several experiments with the original synchronization mechanism (Clock-Step Simulation Method, CSSM) in SystemC and the Data-dependency Virtual Synchronization Method (DAVSM) as in [1]. We compare the number of synchronizations (Num-Sync) and the simulation speed on a MPSoC virtual platform with various numbers of processors.

The experimental settings are as follows. First, we build a virtual MPSoC prototype by SystemC[10] and TLM[5], and modify public programs (i.e: JPEG Encode in [11], and Sparse Matrix Multiplication in [12]) as the parallel programs for the target MPSoC. We then modularize the synchronization mechanism in the target MPSoC, so that the synchronization mechanism becomes replaceable while keeping the original functionalities of hardware modules intact. Furthermore, we build the target MPSoC with two memory systems commonly utilized in MPSoC: the distributed shared memory system and the uniform shared memory system. Then various types of synchronization conditions are examined in the target MPSoC. Finally, our ARM v5Te processor model with ISA and other hardware modules in the target MPSoC are required to be cycle accurate. Such arrangements ensure the virtual platform simulation in high simulation accuracy.

These experiments were conducted on a Linux workstation with Intel Xeon 2.2 GHz and 16 GB RAM.

A. Experimental results on the number of synchronization

Table 1 demonstrates the experimental results on the comparison of the number of synchronizations. We compare three different synchronization mechanisms (CSSM, DAVSM and USCM) on two parallel software programs (JPEG encoding and Sparse Matrix Multiplication) with numbers of CPUs from 1 to 32. The number of the total simulated instructions and the target time (i.e. the time in the target MPSoC) are presented in the second and third columns. Please note that we implement the DAVSM and USCM with cycle accuracy. Therefore, their instruction counts and target time are the same as those of CCSM.

Columns 4, 5, and 6 in Table 1 show the number of synchronizations for CSSM, DAVSM, and USCM,

Table.1 The number of synchronizations versus different numbers of processors for the JPEG-Enc and SMM

JPEG Enc Simulation on Target MPSoC					
Num-CPU	Total Instructions	Target Time (Unit = 10ns)	CSSM Num-Sync	DAVSM Num-Sync	USCM Num-Sync
1	1,025,182,277	1,215,643,107	2,431,286,214	46,134	6,280
2	732,605,477	825,540,481	2,476,621,443	525,371,208	8,380
4	620,999,273	676,731,757	3,383,658,785	525,381,021	12,480
8	569,808,025	608,475,867	5,476,282,803	525,400,626	21,080
16	540,592,595	569,520,124	9,681,842,108	525,439,957	36,680
32	523,610,016	546,873,007	18,046,809,231	525,518,635	71,380

SMM Simulation on Target MPSoC					
Num-CPU	Total Instructions	Target Time (Unit = 10ns)	CSSM Num-Sync	DAVSM Num-Sync	USCM Num-Sync
1	152,601,421	179,173,896	358,347,792	5,267	743
2	111,568,180	124,107,486	372,322,458	87,154,494	998
4	97,557,868	104,716,214	523,581,070	90,443,686	1,508
8	95,378,082	100,388,132	903,493,188	97,022,041	2,528
16	103,032,854	107,751,048	1,831,767,816	110,178,867	4,568
32	124,429,306	130,592,746	4,309,560,618	136,492,548	8,648

Table2. The simulation time (sec) of the experiments

	CSSM		DAVSM		USCM	
Num-CPU	JPEG Enc	SMM	JPEG Enc	SMM	JPEG Enc	SMM
1	3734.15	546.62	173.22	28.87	179.45	29.34
2	2818.85	414.37	441.01	74.83	191.22	31.28
4	2545.14	380.21	453.54	79.58	198.43	35.31
8	2667.44	436.73	494.44	92.05	209.06	38.37
16	3280.79	611.1	539.45	115.83	225.96	46.94
32	4667.14	1161.97	635.17	165.8	263.44	66.57

respectively. The number of synchronizations increases with the increasing number of CPUs, no matter which synchronization mechanism is adopted. However, the number of synchronizations in our USCM algorithm is usually 5 to 6 orders less than that of CSSM and DAVSM. This indicates that our USCM greatly outperforms other mechanisms in reducing the number of synchronizations.

We further compare the growth in the number of synchronizations with respect to the increase in the number of CPUs. The result shows that DAVSM grows much faster than USCM. This means that the USCM algorithm is more robust towards multi-core virtual platform simulations.

B. The improvement of simulation speed

Table 2 compares the simulation time among different synchronization mechanisms. As can be seen, USCM outperforms the other two in all cases. Another measurement compares the simulation speed as the number of instructions executed per second. We use the following formula to compute the simulation speed:

Following the above definition, we plot the impact of the number of CPUs on the simulation speed in Figure 5 the vertical axis is the simulation speed in million instructions per second (MIPs), and the horizontal axis is the number of CPUs. It is clear that both DAVSM and USCM greatly outperform the traditional CSSM approach. This is due to the effects in the synchronization reductions. Nevertheless, UCSM is more robust for MPSoC virtual platform simulation as it is less sensitive to the increase in the number of CPUs. Specifically, for a single-CPU platform, USCM has similar performance with DAVSM. However, for the 32-processor MPSoC virtual platform, the improvement in simulation speed by USCM is as high as 2X and 30X, respectively. In short, USCM can simulate a 32-processor SoC design in the speed of multi-million instructions per second with cycle accuracy.

$$Simulation\ \ Speed = \frac{\sum_{i=1}^{n} The\ number\ of\ \ instructions\ \ performed\ by\ P_i}{Simulation\ \ Time\ \ (\text{sec})}$$

$$n = The\ number\ of\ \ processors,\ P_i = The\ i-th\ \ processor,\ (1 \le i \le n)$$

V. CONCLUSION

In this paper, we propose a novel Ultra Synchronization Checking Method (USCM) for MPSoC virtual platform simulation. By analyzing various types of memory information and categorizing the data dependencies among modules into hardware/software static/dynamic Data-Dependency Tables (DDTs), we can greatly reduce the number of synchronizations in simulation scheduling and still maintain reasonably good cycle accuracy. Our experimental results demonstrate that our method can not only improve the simulation speed by several orders, when compared to the conventional clock-step simulation scheme, but also outperform the other async-DES based approaches, especially for the multi-core designs.

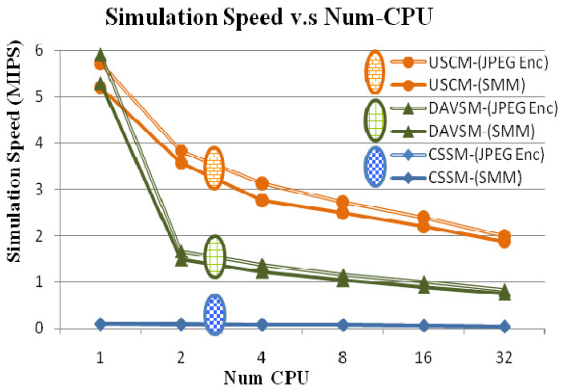

Figure5. The simulation speed versus the different number of processor for JPEG End and SMM

REFERENCES

[1] KH Lin, S.J. Cai and Ric. Huang, "Speeding Up SoC Virtual Platform Simulation by Data-dependency Aware Virtual Synchronization", in Proc. ASP-DAC, Taipei, Taiwan, Jan. 2010.

[2] G. Schirner and R. Dömer, "Fast and accurate transaction level models using result oriented modeling," in Proc. ICCAD, San Jose, CA, Nov. 2006, pp. 363–368.

[3] Y. Yi, "Fast and Accurate Cosimulation of MPSoC Using Trace-Driven Virtual Synchronization", in Proc. IEEE Transaction on Computer-Aided Design of Integrated Circuit and Systems, Dec. 2007, pp. 2186-2200.

[4] MH Wu, "An effective synchronization approach for fast and accurate multi-core instruction-set simulation", in Proc. 5th EmSoft, Nov. 2009, pp. 197-204.

[5] Cai et al, "Transaction Level Modeling: An Overview", in Proc. 1st IEEE Intl. Conf. on Hardware/Software Co-design & System Synthesis, October 2003, pp. 19-24.

[6] M. Girkar, C.D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs", IEEE Trans on Parallel and Distributed System (PDS), Mar, 1992.

[7] S. Gupta et al., "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," in Proc. 16th Int'l Conf. VLSI Design, IEEE Press, 2003, pp. 461-466.

[8] http://sourceware.org/gdb/current/onlinedocs/gdb/, The GDB User Manual, chapter 9, 2010, pp. 96-98.

[9] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited", Proc. of 9th Working Conference on Reverse Engineering (WCRE), 2002, pp. 45–54.

[10] <http://www.systemc.org/downloads/standards/>

[11] <http://www.ijg.org/files/>

[12] <http://www.cise.ufl.edu/research/sparse/SuiteSparse/>.