Synchronization for Hybrid MPSoC Full-System Simulation

Luis Gabriel Murillo, Juan Eusse, Jovana Jovic,
Sergey Yakoushkin, Rainer Leupers and Gerd Ascheid
Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Germany
{murillo,eusse,jovic,yakoushkin,leupers,ascheid}@ice.rwth-aachen.de

ABSTRACT

Full-system simulators are essential to enable early software development and increase the MPSoC programming productivity, however, their speed is limited by the speed of processor models. Although hybrid processor simulators provide native execution speed and target architecture visibility, their use for modern multi-core OSs and parallel software is restricted due to dynamic temporal and state decoupling side effects. This work analyzes the decoupling effects caused by hybridization and presents a novel synchronization technique which enables full-system hybrid simulation for modern MPSoC software. Experimental results show speed-ups from 2x to 45x over instruction-accurate simulation while still attaining functional correctness.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

MPSoC, Virtual Platforms, Hybrid Simulation, HySim, Synchronization, Temporal Decoupling

1. INTRODUCTION

The increasing complexity of modern electronic systems and the spread of multi-processor systems-on-chip (MPSoCs) have demanded a drastic design paradigm shift. Platform architectures and software are designed, developed and evaluated from a system perspective, focusing on components interaction, synchronization and communication. This system perspective is of utmost importance to achieve not only better performance-power ratio, but also to support cutting-edge features required by new products.

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

System simulation plays an important role to support electronic design, as it provides the means to estimate performance, test functionality and guide the development process. Naturally, the complexity of modern systems is reflected in the complexity of their simulators. Current MPSoC simulators comprise several models of different processing elements (PEs), hardware peripherals, accelerators, and communication infrastructures. Specialized simulation frameworks, such as SystemC-TLM2 [4], OVPSim [1], QEMU [2], Synopsys Virtualizer [3] and Simics [6], are widely used to create models at different levels of abstraction and assemble virtual systems. These full software models of hardware systems, also known as Virtual Platforms (VPs), have gained tremendous popularity among designers because of their early availability.

Accuracy and simulation speed can be traded-off in order to create VPs for different use scenarios (e.g. architectural design, software verification). This is typically done by choosing an appropriate level of abstraction when modeling devices and the communication among them. Models of PEs can be found in the form of Instruction Set Simulators (ISSs), either *cycle-accurate* (ISS-CA) or *instruction-accurate* (ISS-IA), and as *host-compiled* simulators. However, VPs are often many orders of magnitude slower than the real systems they represent and further increasing their speed is a difficult challenge.

Although VPs are composed of several models, PEs remain as a major simulator bottleneck due to their complexity. Several techniques, such as dynamic binary translation [17] and just-in-time compiled simulation [13], have been used to increase ISS speed. However, ISSs remain slow if compared to more abstract processor models. Conversely, *host-compiled* or *native* simulators [9, 14, 15, 16], corresponding to one level of abstraction above ISS-IA, provide faster simulation at the cost of accuracy and a limited view of the underlying hardware. The latter limits the applicability of host-compiled simulators for software development, validation and debugging in a practical context.

1.1 Hybrid Full-System Simulation

Hybrid simulation approaches try to bridge the gap between two different abstraction levels and have been used before with different objectives [11, 12]. *HySim* [10] is a hybrid processor simulator that allows bidirectional dynamic switching between a target ISS-IA and a host-compiled simulator, while keeping the *processor-centric* state synchronized between both simulation modes. The target ISS executes processor specific functions, whereas the host-compiled simulator executes target-independent parts of the application.

The host-compiled part of a hybrid ISS runs outside the context of the simulation kernel, hence leading the PE into a future state with regard to the rest of the system. This decoupling, namely *hybridization-introduced decoupling*, is highly dynamic and might decouple PEs long enough to disturb behavior of systems with interrupt-driven functionality, multi-threading, multi-core synchronization, preemptive scheduling, dynamic task migration and other common features of modern OSs and concurrent run-time environments. In consequence, the use of HySim and other hybrid ISS frameworks is limited to MPSoCs where reactive behavior and inter-processor activity do not define the system correctness and usability.

This paper introduces and describes the aforementioned concept of hybridization-introduced decoupling, and proposes a new synchronization mechanism to attain system coherency when using hybrid ISSs, thus allowing proper interaction among PEs and preserving the functional correctness of the system. The mechanism was used to define an improved hybrid processor simulation architecture and extend the HySim framework in order to enable hybridization, software-centric synchronization, traditional temporal decoupling and reactive behavior to coexist in the simulator.

Paper Outline. The remainder of this paper is organized as follows. In Section 2, the necessary background to better understand the concepts herewith discussed is presented. Section 3 introduces the concept of hybridization-introduced decoupling, and proposes a technique to temporally synchronize hybrid ISSs with the simulation kernel and the rest of the system. This is complemented in Section 4 with an approach to keep a behaviorally correct system-wide state in hybrid systems which run parallel applications and utilize software synchronization functions. Section 5 presents results of our synchronization techniques for hybrid ISSs when applied to MPSoC VPs created in two major commercial frameworks, namely Synopsys Virtualizer and Simics. Results cover complex cases of parallel applications with distributed scheduling mechanisms and complete software stacks. Finally, Section 6 presents conclusions of this work.

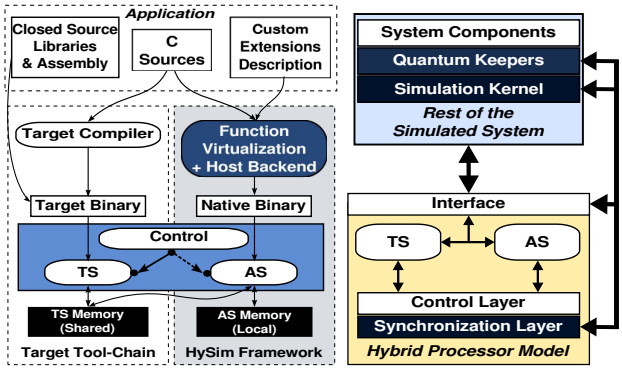
2. BACKGROUND

2.1 ISS vs. Host-compiled

ISS and host-compiled simulators cover different use cases of VPs. An ISS emulates execution of a cross-compiled binary, including a one-to-one match of instruction sets and dynamic effects at the instruction level. ISSs of VLIWs, DSPs and ASIPs model also non traditional resources common to these devices, such as irregular register architectures or extended memory interfaces.

On the other hand, host-compiled approaches model computation at the source code level and execute it directly on the host machine. To take into account dynamic effects, parts of the hardware-specific software layers (e.g. HAL, context switching) are usually abstracted away thus losing the visibility of the target architecture, such as in [14, 15].

In a practical context, modifications for host-compiled simulation (e.g. intrusive instrumentation of software and/or simulation models [9]) might be prohibited due to the presence of third-party IPs, legacy code, or even limitations to alter previously verified systems. For this reason, software development often falls back to slow, traditional ISS simulation. Another reason is the necessity to port or develop OSs, evaluate target-optimized software and run low-level code and middleware. These are critical tasks in almost all signal processing and multi-media software used to power today's telecommunication and consumer electronic devices.



(a) Workflow (b) New core structure

Figure 1: The HySim Framework.

2.2 HySim - A Hybrid Simulation Framework

HySim combines the advantages of a fast native simulator and a target-specific ISS. It was designed mainly to support embedded software development and debugging. The key idea behind it is partitioned execution at function level, which allows bidirectional run-time switching between a target ISS (TS) and a host-compiled abstract simulator (AS).

The major components of the HySim workflow, shown in Figure 1a, are a compiler-like function virtualizer and a control module which switches simulation modes. Function virtualization analyzes application C source code and selects functions with only target-independent features (i.e. *virtualizable*), which can be executed in AS. The selected functions are extended and instrumented using special synchronization APIs which guarantee processor-centric consistency (i.e. registers and memory view) between TS and AS modes [8, 10]. The remaining functions in an application (i.e. *non-virtualizable*) are executed in TS. This set comprises functions with inline (or written in) assembly, *state-altering* functions (e.g. *fopen*), functions with direct stack manipulation (e.g. *setjmp* and *longjmp*), function pointers, and functions without definition (e.g. closed source libraries). The user, using partitioning algorithms for ultra-fast forward breakpoints or manually mapping functions to AS, has the final choice on how to run the simulation. The AS mode relies on dynamic software performance estimation solutions to give a notion of simulated time, which were also introduced in [7]. Although HySim requires the application sources as input, it can be successfully applied to a wide range of practical scenarios because it does not modify the target binary and allows closed source libraries, target-dependent code, and drivers in the application.

2.3 Temporally Decoupled Simulation

Based on the observation that components in a system do not interact with the surrounding environment frequently, some parts might be allowed to run ahead of the rest of the system without consequences. This concept, known as temporal decoupling, has been used in simulation to avoid unnecessary kernel synchronization points and context switches, which cause a significant overhead.

In practice, every PE is assigned a *quantum*, either statically or dynamically, that defines how many simulation steps (i.e. instructions or cycles) it can advance without synchronizing. A small quantum allows to handle external events more accurately but at slow simulation speed, while a big quantum achieves fast speed at the cost of corrupting the timing behavior of the system.

3. HYBRIDIZATION-INTRODUCED DECOUPLING

This section analyzes the decoupling effects in hybrid simulators by (i) introducing the concept of hybridization-introduced decoupling and (ii) proposing a mechanism to ensure proper time-driven behavior in MPSoC system simulators.

In HySim, every function mapped to AS is executed in synchronous mode with the simulated application. Host-compiled execution is incapable of affecting directly the simulated time, neither globally nor locally. Thus, the execution of a virtualized function is performed in *zero time* from the simulator's perspective. Software performance estimation techniques help to obtain timing values for the functions executed natively, which are annotated to the cycle counters of the PEs. However, this causes a hybrid ISS to be temporally decoupled from the rest of the system.

Since switching from TS to AS is performed upon the execution of a function in the application, the introduced temporal decoupling could at best be synchronized at function borders. This is a consequence of having a highly abstracted model for PEs which represents functions as instructions from the ISS perspective. Therefore, interrupts, software-centric synchronization and other events will be delayed to interact with the system state left by the AS execution. Moreover, the loss of accuracy during native mode might disturb timing and change the behavior of OS schedulers. Without any further action, a hybrid ISS might lead to a system crash (e.g. due to unhandled interrupts) or to non-deterministic simulator behavior, thus restricting its use for software development and debugging. This effect, what we call hybridization-introduced decoupling, has some similarities to traditional temporal decoupling scenarios, but poses new constraints that must be handled differently.

3.1 Modified Hybrid Processor Structure

HySim's original hybrid processor model did not consider the necessity to define a synchronization interface for hybridization-introduced decoupling. Therefore, it was necessary to replace HySim's core architecture with a new structure which links the hybrid PE to the simulator kernel's time, as shown by Figure 1b. Our structure features a Synchronization Layer, on top of HySim's Control Layer, that analyzes simulation mode switches and events sent by other system devices. In this way, it is possible to tell the kernel when the hybrid PE should be scheduled again and what simulation mode to be used. Depending on the simulation technology, the link between kernel and PE can be created either directly or through dedicated time manipulation interfaces, like SystemC-TLM2 Quantum Keepers. When used with extensible, API rich simulation frameworks, such as Simics or Synopsys Virtualizer, it is possible to set up the Synchronization Layer in a transparent way and no changes are required in the kernel or other system components.

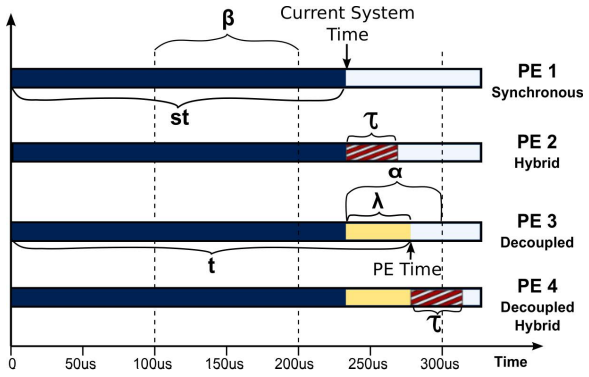


Figure 2: Temporally decoupled simulation.

3.2 Suspension Quantum

The estimated time for a given virtualized function represents at run-time the amount of time a hybrid PE will be ahead of the system. In a PE running synchronously with the system (e.g. triggered by the simulator kernel on every new instruction), a new instruction cannot be executed right after executing a virtualized function without synchronizing with the system. This is because other PEs will stay in the "past" with regard to the PE state that was modified during the virtualized function.

To allow other PEs and system components to reach the new temporal state, it is necessary to introduce a mechanism to suspend a PE for a given amount of time. We define a *suspension quantum* (denoted τ) to be the time a PE will be suspended as system time advances. The suspension quantum is created dynamically upon the execution of a virtualized function, and its length is equal to the estimated time associated to the function. In Figure 2, *PE1* runs always in synchronous mode, whereas *PE2* is synchronized only until it starts executing a virtualized function. This creates a decoupling time equal to τ. To avoid unnecessary kernel synchronizations, a new synchronization point for the suspended PE is set after τ.

In HySim, the larger the code parts executed in native mode, the faster the simulation runs. Functions mapped to AS mode are usually application hotspots that take significant execution time. Therefore, the suspension quantum could possibly take a very large value, causing the PE to lose responsiveness to external events. To avoid this, the suspension quantum must (i) be visible outside the context of a suspended PE and (ii) be breakable by system events.

3.3 Breaking the Suspension Quantum

In an MPSoC, any system component might trigger external events that need interaction with the suspended PE. This is the case for interrupts produced by peripherals, such as timers, accelerators and multi-core mailbox-based communication modules. Losing these interrupts causes systems to change their timing behavior and, in some cases, they even behave incorrectly (e.g. an OS that waits for a timer interrupt to boot). As this situation also happens with dynamic quanta in normal temporal decoupling, some specialized mechanisms allow to break a decoupling quantum and recompute new synchronization intervals. Similarly, the suspension quantum is broken when a PE receives an interrupt or a hardware signal that must be handled "in time".

But, in contrast to normal quanta, the mechanism to break τ needs to take a PE out of its suspension state while

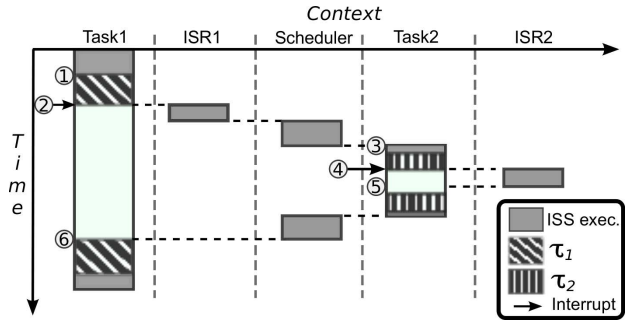


Figure 3: Breaking the suspension quantum.

guaranteeing that the processor jumps into special handling functions (i.e. interrupt service routines (ISR)). The remaining suspension quantum needs to be stored to be consumed later. If the suspension quantum is just canceled and the time is not consumed at all, it will lead to serious inaccuracy errors in the simulation. Moreover, if the remaining suspension quantum is not consumed in the original function execution context, the timing behavior of the application might change considerably.

We introduced a mechanism to ensure that the rest of the suspension quantum is consumed in its proper context, performing the following steps: (i) The hybrid ISS detects an incoming interrupt. (ii) The processor is waken up and the value of the program counter (PC) is taken in the instruction which is aborted by the core interrupt handling mechanisms. (iii) The PC value is associated to a *remaining suspension quantum*. (iv) A breakpoint-like mechanism is activated on the saved PC in order to restore the suspension the next time the processor executes the aborted instruction.

The previous mechanism is specially important in systems with OS and preemptive context switching. Figure 3 shows how a suspension quantum should be broken in an interrupt-triggered scheduling mechanism. In the figure, a task (Task 1) executes one of its functions in native mode (①) and introduces a suspension quantum denoted by τ_1. After some time of suspension, an interrupt (②) arrives, breaking the suspension quantum and triggering the OS scheduler. The scheduler preempts Task 1 and triggers Task 2 (③). The remaining part of τ_1 is only consumed after the context of Task 1 is restored by the scheduler (⑥), because it is tied to the address of the instruction that was interrupted by the incoming signal. In the same way, other nested interrupts that break the quantum in a different context can be supported (as with τ_2 in ④ and ⑤). This approach is limited to applications that do not share functions in their concurrent tasks. However, it can be easily extended by adding OS awareness and detecting a context switch. To avoid errors caused by handling interrupts in a PE with a "future" state (i.e. virtualized function was already executed in zero time and cannot be reverted), this mechanism relies on modifications to the virtualization chain, as discussed later in Section 4.

3.4 Suspension Quantum and Traditional Temporal Decoupling

To achieve maximum simulation speed, it is possible to mix hybridization-introduced decoupling and traditional temporal decoupling, in the same PE. To do so, suspension quanta are used to recompute traditional quanta, and define new synchronization points. Figure 2 shows a processing element (*PE3*) which uses traditional temporal decoupling to run ahead of other system components. In a given point, the PE is assigned a quantum that defines the amount it is allowed to run decoupled. In the meantime, the global system time remains unchanged, and will be modified only after the next synchronization point (i.e. when the quantum is over). In this situation we use the following definitions:

- ***PE Global Quantum*** (β_i)**.** Time unit on which PE_i synchronizes.
- ***Current System Time*** (st)**.** Time elapsed uniformly in all components.
- ***Local Time*** (t_i)**.** Time elapsed in PE_i. Can be greater than st.
- ***Local Quantum*** (α_i)**.** Time remaining from t_i to the end of the next β_i.
- ***Local Time Offset*** (λ_i)**.** Time PE_i is ahead of the system. Difference between t_i and st.

When a hybrid ISS is present, the decoupling parameters need to be dynamically modified depending on the suspension quantum. Therefore, after the execution of a virtualized function the value of t_i is updated, the processor is suspended, and a new β_i is recomputed, according to the following conditions:

1. If the updated local time exceeds the end of the next β_i (i.e. $\tau > \alpha - \lambda$), synchronization is done immediately. The quantum has been overshot, thus a new value for β_i is defined to be used the next time the PE is scheduled. The following operations are performed on the PE timing values:
$$\beta_i' = \beta_i - (\tau - (\alpha - \lambda)) \qquad t_i' = t_i + \tau$$
2. If the updated local time does not exceed the end of the current β_i (i.e. $\tau \leq \alpha - \lambda$), synchronization is performed normally at the next quantum end. In this case, the only operation performed is:
$$t_i' = t_i + \tau$$

These operations need to be performed only when virtualized functions are executed in a given quantum. Otherwise, temporal decoupling is used normally.

4. SYSTEM STATE SYNCHRONIZATION

Allowing PEs to run ahead of others creates momentary inconsistencies in the system. Besides, in shared-memory architectures, the "future" state left by a HySim processor might be wrongly propagated to other processing elements, thus causing a concurrency bug (e.g. atomicity violation, deadlock). This situation is very likely to happen if software synchronization functions (e.g. locks, semaphores, mutexes) or functions unrestrictedly accessing shared memory are virtualized and executed in native mode. Additionally, a memory access in AS mode is not able to trigger behavior in the adjacent peripherals. This is due to the fact that memory accesses do not use the traditional ports or sockets in ISSs, in order to achieve maximum speed. Instead, memory is read by using debug APIs or direct memory interfaces (DMI), thus failing to trigger behavior in other components. Functions that rely on global or static variables which are modified during ISRs cannot be virtualized either.

Because of this, restrictions need to be added to the HySim virtualization chain. In a full system simulation, virtualizable functions are not allowed to:

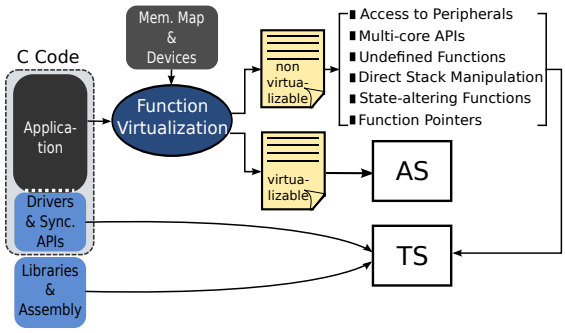


Figure 4: Virtualization chain for state synchronization.

- Perform software synchronization or unrestrictedly access shared memory.
- Interact with peripherals and accelerators.
- Depend on global states modified by ISRs or exception handlers.

All these functions are explicitly excluded from the set of virtualizable functions. Functions without definition are treated in the same way as closed libraries, and are marked as non-virtualizable. In well-formed applications, drivers and the communication and synchronization APIs are clearly separated, and sources can be excluded easily. If the separation is unclear, then the programmer still has the choice to map any virtualizable function to AS or not. Detection of unsafe mappings is done dynamically by address monitoring inside the AS, which uses a memory map description with the location of shared memories and memory-mapped peripherals. Figure 4 shows the modified virtualization flow. Under these constraints, a system with HySim will behave like a temporally decoupled simulator, yet with higher speed and application-defined synchronization.

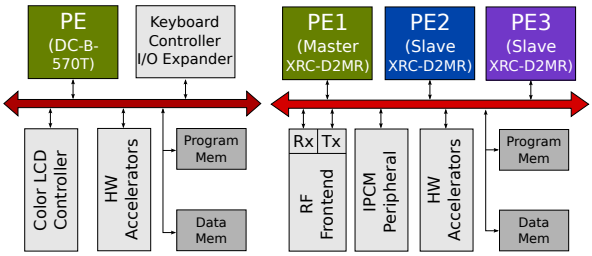
5. TEST CASES AND RESULTS

To test our synchronization mechanism, we used the HySim framework to simulate different scenarios which are prone to behave wrongly in the presence of decoupling. Platforms for multi-media and signal processing were modeled in Simics and Synopsys Virtualizer, whereas Tensilica[5] Diamond and Xtensa ISSs, wrapped with our hybrid architecture, were selected as PEs. The synchronization layer was implemented using extensibility APIs provided by both simulation frameworks (e.g. *Haps*, execution, cycle and step interfaces in Simics; instrumentation points, quantum observers and simulation context handlers in Synopsys).

For the synchronization, function execution times were estimated by sampling the execution of virtualized functions in the ISS, using the statistical sampling theory from [18]. Although software performance estimation is not the focus of this paper, it is worth to note that it introduces certain error in the number of simulated cycles with and without HySim, which will be presented in the results.

All experiments were executed on a host with a 64-bit AMD Phenom Quad-Core Processor running at 2.4GHz, 8GB of memory and Fedora Core 5.

Scenario 1: 3DES on Single-core System. A Simics platform with one Diamond DC_B_570T core was used to execute a simple 3DES encryption/decryption application. Since it does not depend on reactive behavior, the traditional HySim can be used normally. Thus, this scenario allows to obtain the overhead caused by the new hybrid architecture.



(a) Single-core platform (b) Multi-core platform

Figure 5: Test systems.

Table 1 compares the elapsed wall-clock time, the amount of ISS executed cycles, and the speed-up values using the normal ISS, HySim and the synchronization-capable HySim (HySim-Sync). Comparing HySim and HySim-Sync, the results show an overhead, traduced to 5x less speed-up, due to the synchronization layer. The speed-up with HySim-Sync is still a significant ∼27.2x with respect to the normal ISS.

Scenario 2: MJPEG on Single-core System. An enhanced version of the platform from Scenario 1, shown in Figure 5a, was used to execute a Motion-JPEG (MJPEG) player. This platform includes a set of hardware blocks for multi-media acceleration, a detailed model of a color LCD controller and a peripheral for user interaction. In this system, external interrupts are necessary to enable proper behavior of the device drivers. If the drivers lose interrupts, the system could crash with an unhandled interrupt exception. This is the case when simulating it with a big quantum for traditional temporal decoupling or when using HySim without synchronizations. The last situation is particularly difficult to handle since the hybridization-introduced decoupling might induce a crash only when some specific functions are mapped to HySim's AS mode. Comparative results when running the system are illustrated in Table 1. When HySim-Sync is used, all critical functions are marked as non-virtualizable by the framework and the suspension quantum mechanism enables handling the interrupts as expected by the drivers. If compared to the normal HySim, HySim-Sync guarantees the correct operation at the cost of less speed-up (76.9x vs. 45.2x), however, it ensures the usability of the simulator at a considerable high speed.

Scenario 3: Circular-FFT on Multi-core System. A Synopsys platform consisting of three Xtensa XRC_D2MR cores and an AMBA AXI bus was used to execute a Circular-FFT. This application is a token-passing system in which a core owning the token has to perform an FFT over some data and then pass the token to the next core. The tokens are passed using a simple communication protocol over shared memory upon the reception of a timer interrupt. Cores not holding the token wait while polling the shared memory. This system features a special time-triggered, software-based synchronization and its behavior with HySim depends on the FFT processing time and the timer period:

- If the FFT time is much greater than the timer and the polling (i.e. huge input data set), then HySim yields enormous speed-ups. The results table shows a value of 313x when the FFT size is 1024.
- If the FFT time is less than the timer and the polling, then the time saved by native execution is offset by the ISS execution speed during the polling loops. Thus, HySim might achieve marginal or no speed-up.
- If both times are similar, then the system loses determinism and might lock due to unhandled interrupts.

Application	Normal		HySim				HySim-Sync					
	Simulated ISS Cycles(M)	Wall-clock Time(s)	Simulated ISS Cycles(M)	Wall-clock Time(s)	App. in AS(%)	Speedup (times)	Simulated ISS Cycles(M)	Wall-clock Time(s)	App. in AS(%)	Synchro-nizations	Speedup (times)	Estima-tion Error(%)
3DES	2214.3	1625	26.5	50	98.8	32.5	26.5	59.7	98.8	600000	27.2	-6.3
MJPEG	1705.4	1231	8.8	16	99.4	76.9	10.5	27.2	99.3	7444	45.2	-7.4
Circular-FFT	2360	17238	0.75	55	99.9	313.4	642.6	7494	72.8	200	2.3	-33.3
OFDM-Trans	797.6	5816	71.7	574	91.1	10.1	162.3	3061	79.66	1000	1.9	-12.4

Table 1: HySim speed-up in VPs with and without synchronization.

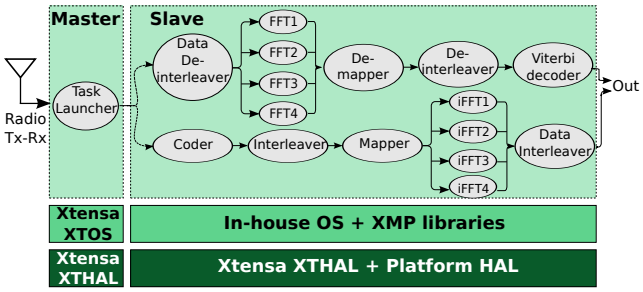


Figure 6: OFDM transceiver application.

HySim-Sync's suspension quantum and interrupt support are necessary to avoid the last situation, however, at the cost of a significant speed-up reduction of 2 orders of magnitude, as shown by the table (313x vs. 2.3x). In this case, the user has to decide whether such trade-off is acceptable based on his knowledge of the application.

Scenario 4: OFDM Transceiver System. The base platform from Scenario 3 was used to set up a full digital wireless transceiver (OFDM-Trans). The hardware was extended with a mailbox-based Interprocessor Communication Peripheral (IPCM) for multi-core support and peripherals to handle input and output data flows, as shown in Figure 5b. The program features a complete software stack consisting of HAL, a priority-based preemption mechanism, a distributed scheduler, and a task management system. On top, the user application implements an OFDM transceiver algorithm which is divided into sub-tasks corresponding to algorithmic kernels, as illustrated in Figure 6. All sub-tasks are mapped to two cores that act as "Slaves", whereas the job management, launching and scheduling are done in the remaining core ("Master"). The Master launches dynamically a reception or transmission job every time a new packet is reported by the radio frontend. Scheduling a sub-task is done by sending a message through the IPCM which interrupts the destination core. The destination core's scheduler receives the order and manages it locally according to task priorities. Since this application performs priority scheduling and preemption based on incoming interrupts, it is mandatory to synchronize frequently the hybrid ISSs and the system. With the normal HySim, the simulation might still be functionally correct if the application code itself is written to be perfectly synchronized. If not, deadlocks and data races will arise in the system due to unsupported interrupt rate in the drivers or due to the randomization of task interleavings in the scheduler. For this system, HySim-Sync achieves 1.9x speed-up and guarantees correct operation and reproducibility. The level of details of other models (e.g. the bus) prevent to obtain more speed-up.

It is worth to mention that speed-up and execution times are not comparable between different simulation tools because the systems contain models at different levels of abstraction (e.g. systems in Synopsys have a detailed AXI bus model, while Simics uses a point-to-point bus).

6. CONCLUSIONS

Hybrid processor simulators are essential to provide both high speed and target-specific functionality. This paper presented an approach to synchronize hybrid processor simulators within full-system simulators in order to attain correctness. The temporal and the state decoupling problems were addressed by (i) defining a specialized temporal decoupling mechanism and (ii) identifying functions that must be avoided in native execution in order to ensure correctness of parallel applications. The proposed mechanisms were used to refine the internal architecture of a representative hybrid simulator (HySim) and analyze it in four application scenarios. Future work should address the application of hybridization to many-core systems as well as its combination with other advanced simulation techniques (e.g. parallelization).

Acknowledgment

This work has been supported by the FP7 Euretile project, the UMIC Research Center and Huawei Technologies. The authors would like to thank Yao Zhiliang and Guo Can from Huawei for their valuable contributions.

7. REFERENCES

[1] Open Virtual Platforms. `http://www.ovpworld.org`.
[2] Qemu. `http://www.qemu.org`.
[3] Synopsys Virtualizer. `http://www.synopsys.com`.
[4] SystemC. `http://www.systemc.org`.
[5] Tensilica processors. `http://www.tensilica.com`.
[6] Windriver Simics. `http://www.windriver.com`.
[7] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *Design Automation Conference (DAC)*, 2008.
[8] L. Gao, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. A fast and generic hybrid simulation approach using C virtual machine. In *CASES*, 2007.
[9] P. Gerin, M. M. Hamayun, and F. Pétrot. Native MPSoC co-simulation environment for software performance estimation. In *CODES+ISSS*, 2009.
[10] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. HySim: a fast simulation framework for embedded software development. In *CODES+ISSS*, 2007.
[11] W. Lee, K. Patel, and M. Pedram. B2sim: a fast micro-architecture simulator based on basic block characterization. In *CODES+ISSS*, 2006.
[12] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha. Hybrid simulation for energy estimation of embedded software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26, 2007.
[13] 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*, 2002.
[14] P. Razaghi and A. Gerstlauer. Host-compiled multicore RTOS simulator for embedded real-time software development. In *Design, Automation and Test in Europe Conference (DATE)*, 2011.
[15] G. Schirner, A. Gerstlauer, and R. Dömer. Fast and accurate processor models for efficient MPSoC design. *ACM Trans. on Design Automation of Electronics Systems*, 15, 2010.
[16] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Design Automation Conference (DAC)*, 2008.
[17] N. Topham, B. Franke, D. Jones, and D. Powell. Adaptive high-speed processor simulation. In *Processor and System-on-Chip Simulation*. Springer-Verlag, 2010.
[18] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical sampling of microarchitecture simulation. *ACM Trans. Model. Comput. Simul.*, 2006.