A Loosely-Coupled Full-System Multicore Simulation Framework

Weihua Zhang, Haojun Wang, Yunping Lu, Haibo Chen, *Senior Member, IEEE*, and Wenyun Zhao

Abstract—Full-system simulation is critical in evaluating design alternatives for multicore processors. However, state-of-the-art multicore simulators either lack good extensibility due to their tightly-coupled design between functional model (FM) and timing model (TM), or cannot guarantee cycle-accuracy. This paper conducts a comprehensive study on factors affecting cycle-accuracy and uncovers several contributing factors less studied before. Based on these insights, we propose a loosely-coupled functional-driven full-system simulator for multicore, namely Transformer. To ensure extensibility and cycle-accuracy, Transformer leverages an architecture-independent interface between FM and TM and uses a lightweight scheme to detect and recover from execution divergence between FM and TM. Built upon Transformer and its foundational simulator components, a graduate student only needed to write about 180 lines of code to extend an X86 functional model (QEMU) in Transformer. Moreover, the loosely-coupled design also removes the complex interaction between FM and TM and opens the opportunity to parallelize FM and TM to improve performance. Experimental results show that Transformer achieves an average of 8.4 and 7.0 percent performance improvement over GEMS in 4-core and 8-core configuration while guaranteeing cycle-accuracy. A further parallelization between FM and TM leads to 35.3 and 29.7 percent performance improvement respectively.

Index Terms—Functional-driven, multicore simulation, full-system, extension

1 Introduction

Full-system simulation is a key tool to evaluate new ideas in architectural design. Generally, there are two basic models in a full-system simulator: *functional model* (FM), which provides a full-system execution environment to execute operating systems and applications then collects the resulted instruction flow and data access information; and *timing model* (TM), which simulates micro-architectural behavior of the instruction flow generated by FM. Due to the importance of full-system simulators, researchers have designed and implemented a number of FMs, such as Simics [10], QEMU [5] and COREMU [17], and TMs, such as GEMS [11], MPTLsim [19] and RAMP GOLD [16]. However, FMs and TMs are usually tightly coupled together in a full-system simulator and it is usually hard to extend new FMs or TMs in the simulator. For example, developers have spent years to combine M5 with GEMS (i.e., gem5 [7]) or extend QEMU to PTLsim (MARSS [13]). Further, such a tightly-coupled design also makes it hard to efficiently parallelize FM and TM, resulting in inferior performance.

- *W. Zhang and H. Wang are with the Software School, Fudan University, Shanghai, China and the Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China and Parallel Processing Institute, Fudan University. E-mail: {zhangweihua, wanghaojun}@fudan.edu.cn.*
- *Y. Lu and W. Zhao are with the Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China and the School of Computer Science, Fudan University, Shanghai, China and Parallel Processing Institute, Fudan University. E-mail: luyping@sina.com, wyzhao@fudan.edu.cn.*
- *H. Chen is with the Institute of Parallel and Distributed Systems, Shanghai Jiaotong University, Shanghai, China. E-mail: haibochen@sjtu.edu.cn.*

Manuscript received 30 Mar. 2015; revised 10 June 2015; accepted 3 July 2015. Date of publication 12 July 2015; date of current version 18 May 2016. Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2455499

There is a good reason to take the tightly-coupled design in current mainstream full-system multicore simulators. To guarantee cycle-accuracy, such as faithful instruction execution behavior and timing, they usually use TM to drive the execution of FM: in each cycle, TM advises FM on which instruction FM should execute; FM will also report to TM with information regarding the executed instruction, to let TM maintain correct architecture states and timing information. Such a tightly-coupled and complex interaction between TM and FM limits both extensibility and performance of full-system simulators.

There have been some efforts in trying to explore a loosely-coupled design for multicore simulators, which exploit a speculative Functional-First simulation design. However, not all accuracy factors are included in their design. Thus, it cannot guarantee cycle-accuracy. Moreover, they have no implemented prototype for multicore simulators. COTSon [2] uses Timing feedback mechanism in its framework, but it is mainly used for sampling direction, rather than accuracy assurance.

In this paper, we first present a comprehensive study on the limiting factors that lead to execution divergence between FM and TM. We show that besides traditional well-known factors such as branch misprediction and shared data access order, interrupt/exception handling and shared page access order also lead to execution divergence and thus cycle-inaccuracy in a loosely-coupled design. To understand the probability of occurrence of these factors, we profile the proportions of these events in a set of benchmarks and find that these events happen very infrequently (less than 1 percent). This indicates that for most cases, there is no execution divergence between FM and TM.

Based on the above analysis, we propose Transformer, a loosely-coupled, functional-driven simulation scheme for full-system multicore simulation. In Transformer, FM runs

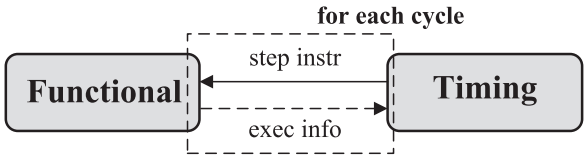

Fig. 1. Tightly-coupled FM and TM.

ahead and provides instructions and data access information to TM. TM then uses such information to simulate the detailed timing of micro-architecture. Transformer also provides a lightweight scheme to detect and recover from execution divergence, thus ensures cycle-accuracy. Basically, Transformer rolls back FM to the path indicated by TM. For branch misprediction and interrupt/exception handling, Transformer uses an additional simple FM to generate the instruction flow information in wrong path to feed TM, so as to further reduce the interaction between FM and TM caused by the rollback scheme. Further, to make Transformer extensible, we provide an architecture-independent instruction and data flow interface between FM and TM.

In Transformer, the interaction between FM and TM is much simpler and thus provides great flexibility to extend with new FMs or TMs. Further, as FM and TM are now loosely-coupled, it also opens the opportunity to parallelize FM and TM to improve performance.

We have implemented Transformer based on GEMS [11], a widely-used tightly-coupled simulator, and parallelize FM and TM to achieve better performance. Based on Transformer, a graduate student only needed to write about 180 lines of codes (LOCs) and took about 350 working hours (about two months) to extend an x86 functional model (QEMU). Furthermore, experiments with SPLASH-2 [18] and PARSEC [6] show that Transformer achieves about 8.4 and 7.0 percent performance improvement compared to GEMS while guaranteeing cycle-accuracy in 4-core and 8-core configuration. Moreover, the performance increases to 35.3 and 29.7 percent after FM and TM are parallelized respectively.

In summary, this paper makes the following contributions:

- The first comprehensive analysis on factors leading to execution divergence between FM and TM, which uncovers that interrupt/exception handling and shared page access are also limiting factors to cycle-accuracy.
- A loosely-coupled full-system multicore simulation framework that is extensible, fast, and cycle-accurate, as well as a set of techniques to detect and recover from execution divergence.
- An experimental evaluation that confirms the effectiveness and efficiency of Transformer and a case study that extended QEMU in Transformer to demonstrate the extensibility.

The rest of the paper is organized as follows. Section 2 discusses the motivation of the loosely-coupled design and comprehensively analyzes which factors affect cycle-accuracy. Section 3 proposes the Transformer framework, describes the lightweight cycle-accurate solutions and discusses the architecture-independent interface. Section 4 presents a case study for extension and evaluates the performance improvement of Transformer in Section 5. The related work is discussed in Section 6. Finally, Section 7 concludes the paper and discusses possible future work.

TABLE 1
Simulation Execution Time Breakdown

Config	FM (support TM)	TM	Interaction
4-core	9.5%	64.7%	25.8%
8-core	11.4%	71.9%	16.7%

2 Motivation

In this section, we first analyze the limitations of a tightly-coupled design for full-system multi-core simulators. Then, we analyze factors that may lead to divergence between FM and TM.

2.1 Limitations with a Tightly-Coupled Design

To achieve cycle-accuracy (e.g., guarantee correct interleaving in parallel applications), existing full-system multicore simulators usually exploit a tightly-coupled timing-driven design. As shown in Fig. 1, in each cycle, TM directs FM with which instructions should be executed and FM feeds back the executed results to TM to maintain correct architectural states and timing information. Then, TM has to simulate part of the functional model so as to direct the execution of FM. Moreover, TM needs to update its own states according to FM's execution results. Such a tightly-coupled manner leads to a complex interaction between FM and TM, which makes it very difficult to extend a new FM or TM into those simulator frameworks. For example, the developers spend years to combine M5 with GEMS (gem5 [7]) or extend QEMU into PTLsim (MARSS [13]).

In addition, the complex interaction in current tightly-coupled design also limits simulation speed. To illustrate this problem, we profile the execution proportion of FM, TM and their interactions (using the experiment setup in Section 5.1). From the data shown in Table 1, we can obtain the following observations.

- To support TM, FM has to execute in an instruction-by-instruction model instead of fast binary translation to provide execution information to TM. As a result, FM occupies about 10 percent of the whole execution time, which cannot be neglected any more. However, it is impossible for a tightly-coupled design to gain performance improvement through parallelizing FM and TM.
- The complex interaction produces about 16-26 percent overhead due to complicated control logic and frequent state transformation with poor locality.

2.2 Factors to Cycle-Accuracy

To gain insight into possible solutions to loosely-coupled cycle-accurate design, we study the factors leading to execution divergence between FM and TM. Besides traditional well-known factors such as branch misprediction and shared data access order, we find that interrupt/exception handling and shared page access order also lead to execution divergence and thus cycle-inaccuracy in a loosely-coupled design.

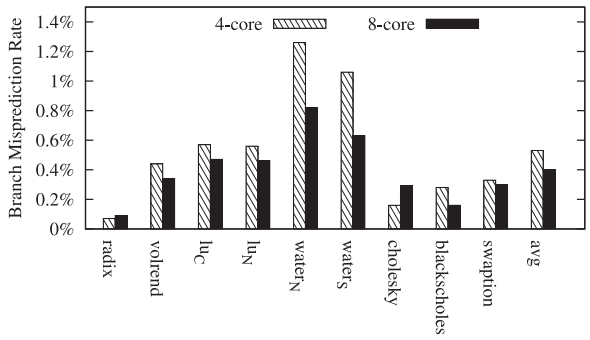

Fig. 2. Branch misprediction rate.

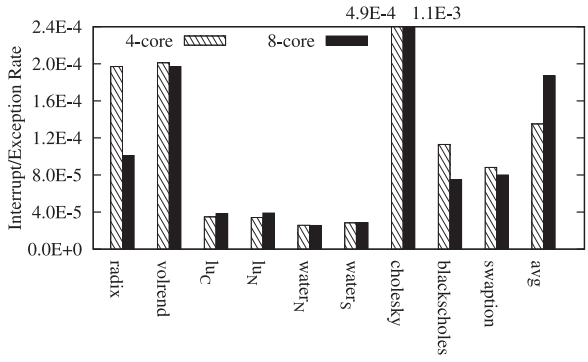

Fig. 3. Interrupt/exception rate.

- *Branch misprediction:* In modern architectures, branch prediction is usually exploited in the pipeline design to avoid stalls caused by branch instructions. The branch could be mispredicted to execute a wrong path in TM. However, FM always executes the instructions on the correct path, leading to execution divergence with actual architectural execution (i.e., TM).
- *Shared data access order:* In parallel applications, shared data accesses are distributed among all the threads and not all of them are carefully designed to guarantee a deterministic access order. The Out-of-Order execution model in pipeline and the unpredictable scheduling of threads will result in the incorrect access order of shared data in FM, compared to the correct access order decided by TM's detailed timing simulation. That is, FM may execute a different write/read order compared with that of TM, which may diverge to another execution path.
- *Interrupt/exception handling:* Interrupt or exception is similar to branch misprediction. To process the interrupt or exception, FM will directly jumps to the interrupt/exception handler and simulates along the handling path. However, TM cannot find an interrupt or exception operation until it commits an interrupt or exception-related instruction. Before that, TM will fetch instructions from wrong path, i.e., next PC instead of the interrupt/exception code, thus leading to execution path divergence.
- *Shared page access order (i.e., Memory Management Unit (MMU) miss order):* The system behavior has to be simulated in a full-system multicore simulation. Although such a design guarantees cycle-accuracy, it involves some additional shared data accesses among different threads, which would further lead to path divergence between FM and TM. The divergence will take place under two conditions. First, two memory operations in different threads may access data within the same page. When this page is not in memory, the first access will result in an MMU miss and its corresponding thread has to include the operations to process the MMU miss. Second, two pages (suppose A and B) accessed by two data accesses might be mapped to the same entry in the page table. Suppose page A is in memory while page B is not present. If the access to page B is executed first, page A will be split out. When page A is accessed again, an MMU miss occurs. However, if page A is executed first, no MMU miss will occur. Since both of these two conditions are related to MMU miss, we will also refer to this factor as MMU miss order.

Some advanced processor features like load speculation [20] may also cause execution divergence. Though such features are usually not supported in current mainstream processors, it is easy for Transformer to handle such a divergence. Correct memory access sequence from FM is stored in an architectural-independent interface (Memory Access Table (MAT) in Section 3.1). After a memory operation is committed, the corresponding item in MAT is removed. Therefore, if a mis-speculation is detected for load speculation, TM only needs to mark the load as un-speculation, squash the pipeline and re-execute the simulation from the load instruction.

2.3 Rate of Path Divergence Factors

Although these factors would lead to execution divergence between FM and TM, they rarely occur in actual. To illustrate this issue, we profile the occurrence proportion of each divergence factor in the total execution (using the 4-core and 8-core configurations in Section 5.1). The occurrence rate of each factor means its average percentage in the execution of an application. For example, a 1.25 percent occurrence rate of misprediction means 1.25 mispredictions per 100 instructions on average.

2.3.1 Branch Misprediction Rate

Branch misprediction only affects the performance of the timing model. Actually, the functional model always know the right path of branches. The timing model also knows the right path instructions and it simulates branch prediction to get the prediction results. If the prediction result indicates a wrong prediction, the timing model will get wrong-path instructions to simulate the behavior. Fig. 2 shows the branch misprediction occurrence rate for each benchmark. The data means the average number of mispredictions for each instruction. High misprediction rate will only affect execution performance of timing model. As the data shows, the occurrence rate of branch misprediction is only about 0.53 and 0.40 percent on average under 4-core and 8-core configuration respectively. Even for the benchmarks whose branch predictor fails more than 50 percent (e.g., *lu* and *water*), the occurrence rate of branch misprediction is less than 1.3 percent.

2.3.2 Interrupt/Exception Rate

Fig. 3 shows the total interrupt and exception rate for each benchmark. As the data shows, the average rate of interrupt

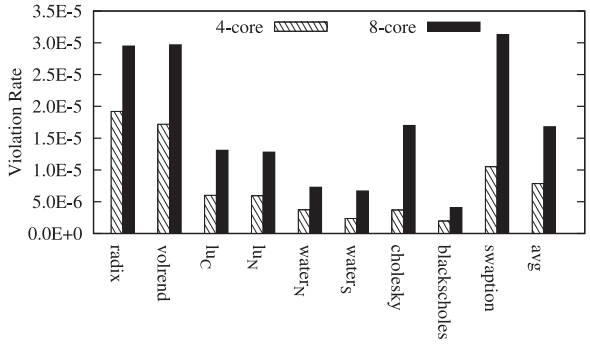

Fig. 4. Shared data access order violation rate.

and exception is only about 1.3E-4 and 1.9E-4 under 4-core and 8-core configuration respectively. Even for the benchmark (cholesky) with most occurring proportion, it is less than 0.2 percent.

2.3.3 Shared Data Access Order Violation Rate

Fig. 4 shows the shared data access order rate for each benchmark. As the data shown, the average rate of shared data access order violation is only about 7.8E-5 and 1.7E-5 under 4-core and 8-core configuration respectively.

2.3.4 MMU Miss Rate

Fig. 5 shows the MMU miss rate for each benchmark. The average rate of MMU miss is only about 1.6E-5 and 5.2E-6 under 4-core and 8-core configuration respectively. Even for the benchmark (radix) with most occurring proportion, it is only about 5.3E-5.

As shown in the above data and Table 2, branch misprediction occurs the most frequently but still only occupies about 0.53 percent. The total frequency of path diversities is less than 1 percent. Therefore, in most cases (more than 99 percent), there is no execution divergence between FM and TM. This provides the opportunity to use a loosely-coupled design that may bring better extensibility to support other FMs or TMs and higher performance due to possible parallelization.

3 The Transformer Framework

This section presents the design of our loosely-coupled framework called Transformer. We first describe a light-weight scheme to detect and recover from execution

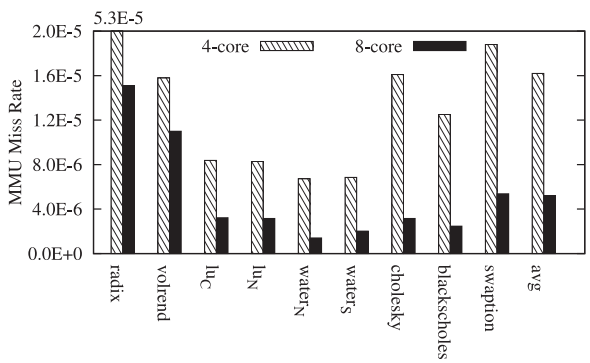

Fig. 5. MMU miss rate.

TABLE 2
Proportion of Path Diversities

Path divergence Source	Proportion
Branch Misprediction	5.3E-3
Interrupt/Exception Handling	1.4E-4
Shared Data Access Order Violation	7.9E-6
MMU Miss	1.6E-5

divergence to guarantee cycle-accuracy. Then, to make Transformer more extensible, we illustrate an architecture-independent instruction and data flow interface between FM and TM. The overall framework of Transformer works as shown in Fig. 6.

As analyzed in Section 2.3, all the factors leading to execution divergence between FM and TM occur very infrequently. Therefore, a loosely-coupled function driven model is applied in the design of Transformer. In other words, FM in most cases generates the architecture-independent instruction and data flow information (e.g., pipeline dependence, memory access address) to TM. TM simulates the detailed micro-architecture using instruction and data information provided by FM and detect whether there is execution divergence. When a divergence factor is detected, different strategies (roll back FM and create a wrong-path FM) are applied to revise the divergence execution.

3.1 Divergence Detection

To guarantee cycle accuracy in a loosely-coupled design, the first thing is to detect when and where an execution divergence occurs. Among the four factors, it is easier to detect branch misprediction and interrupt or exception handling. For branch misprediction, we can detect the divergence through checking whether the target address of a branch instruction in TM is the same as that in FM. If they are different, a divergence occurs. For interrupt or exception handling, whenever it occurs, a divergence happens. Therefore, we will mainly focus on how to detect the divergences caused by shared data access order and shared page access order.

3.1.1 Shared Data Access Order

Data access order violation is an essential factor affecting cycle-accuracy. Thus it is a simple choice to detect the violation in shared data access order by checking whether the loaded values of shared memory between FM and TM are identical. However, based on such an approach, it is difficult to learn where and when the actual thread interleaving violation occurs. Since the loaded value may be affected by a distant prior store instruction or two store instructions may write to the same address, the order violation

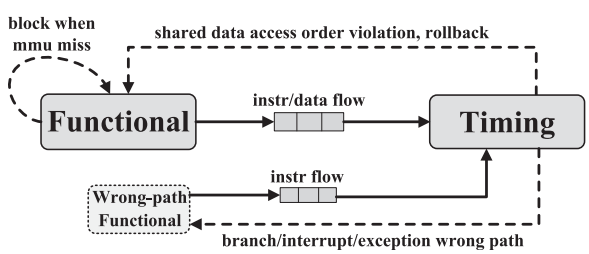

Fig. 6. The transformer framework.

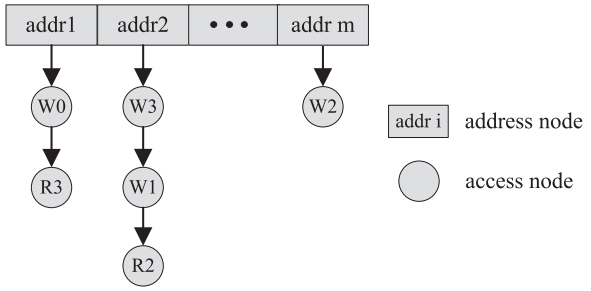

Fig. 7. Memory access table structure.

information may have already been lost when the loaded value is detected to be violated. To overcome this problem, we use a more accurate method: when FM executes instructions, it records its access order for each shared datum. When TM commits the memory instruction, it checks whether its access order is the same as that of FM. If it is different, a divergence occurs.

To achieve this goal, we design a data structure called *Memory Access Table* to efficiently record and check the shared data access order. As shown in Fig. 7, MAT is a two-dimensional table. The first level is a hashed list of memory addresses and we will call the node as the memory address node; for each address, it maintains a list of memory accesses from different cores and the node in it will be referred to as the memory access node. Each memory access node records which core it comes from and its operation type (i.e., read or write). The mechanism of recording and checking the accesses to shared data is demonstrated as follows:

- *Shared data access order recording:* When FM executes a memory instruction, it first checks whether there is a memory address node in MAT for the accessing address. If not, a new address node is created and inserted into the end of memory address list. Otherwise, an access node for this operation is inserted to the end of the memory access list for its address.
- *Shared data access order checking:* Order violation is checked by TM. Since the memory operations in a memory access list are inserted based on their execution sequence in FM, it is easier for TM to check the violation. When TM commits a memory instruction, it only needs to check whether there is no store node before it in the memory access list. If so, there is no violation. And then, this memory operation is committed and its node is deleted from MAT. When the memory access list becomes empty, the memory node of this address is also removed from MAT. Otherwise, the violation is reported.

After the order checking, the node of a memory operation will be deleted from MAT. Therefore, the size of MAT should not be larger than the number of memory instructions that FM executes exceeding TM, which makes MAT relatively small and low-overhead. More details of MAT design can be found in the Section 3.3.

3.1.2 Shared Page Access Order

For shared page access order, i.e., MMU miss order, it is instinct to apply MAT again to check the divergence. However, the functionality of MMU is only simulated by FM. In order to check whether the order violates, TM also has to be able to check whether MMU misses or hits. As a result, the information of entire page table has to be transferred from FM to TM as well, which will lead to more interactions between FM and TM.

To simplify the design, our solution is to avoid this type of divergence. Whenever an MMU miss is encountered, we block FM execution until TM directs it to advance, i.e., until the MMU miss instruction commits in TM. After that, FM will process the MMU miss and update its status. Thus all the MMU misses are processed in the correct order with the guidance of TM, which guarantees the accuracy of MMU simulation.

However, this may bring the danger of draining pipeline in TM, i.e., no instructions are provided by FM. Actually, the pipeline draining will never happen due to the *wrong-path FM* mechanism discussed in Section 3.2. In TM, when an MMU miss happens, it raises an MMU miss interrupt. As for interrupt handling, it will fetch instructions from the wrong path until the MMU miss instruction commits. Although we block the execution of FM, we will create a wrong-path FM and provide instruction flow to TM, which can avoid pipeline draining.

3.2 Divergence Revision

When a path divergence is detected, we need to revise the simulation to keep cycle-accuracy. The simple waiting mechanism makes FM to wait for TM, and then continue simulation in TM's direction. However, such a manner is unduly timing-consuming, which needs frequent and complex interaction between FM and TM. We can also deal with the divergence through a rolling back mechanism, by which FM rolls back its execution to the last checking point when a divergence is detected. However, since FM runs ahead, it is difficult to know when to do a checkpoint. Therefore, it will produce tremendous overhead to save the states for checkpoint frequently. Moreover, the rollback strategy can also incur double rollback (from the right path to the wrong path, and again from the wrong path to the right path) for branch misprediction and interrupt or exception handling. Therefore, besides the rollback strategy, we will also exploit some other optimized strategy: to create a wrong-path FM to execute the wrong path to provide the instruction information to TM for branch misprediction and interrupt or exception handling.

3.2.1 Basic Strategy: Roll Back FM

To roll back FM, we need the correct architecture states at a rollback point, including registers, memory values, MMU states and I/O states. The direct solution is to checkpoint architecture states in a fixed time period. For example, SlackSim [8] uses the *fork* system call to do checkpoint. However, it is difficult to know when a checkpoint is required. Moreover, saving all states will produce significant time and space overhead. Therefore, we introduce a lightweight mechanism to roll back FM states.

- For registers, which are inherently lightweight, TM maintains a copy of these states for rollback. At

initialization, TM reads these values from FM. Then, when each instruction is executed, FM transfers the changed registers to TM. Finally, TM updates the copy when it commits an instruction. Therefore, the register states in TM are updated by the instructions without divergence, which can guarantee correct register states for roll back.

- For memory values, we record the old value of each store instruction in MAT for rollback. When a divergence is detected, we only need to restore these old values from MAT, which greatly reduces memory checkpoint and rollback overhead.
- As discussed in Section 3.1, to avoid shared page access order divergence, we block FM when an MMU miss (note that only MMU miss changes MMU states) occurs until TM directs it to advance its execution. Thus, MMU states are always correct in FM and there is no need for rollback.
- As some I/O operations cannot be rolled back, we simply block the execution of FM when I/O operations occur until TM commits all instructions before it. This mechanism avoids I/O rollback.

3.2.2 Optimized Strategy: Create a Wrong-Path FM

Although a rollback strategy can be used to handle the divergence problem, it is not efficient enough. For branch misprediction and interrupt or exception handling, it would incur double rollback. When a branch predicts a wrong PC or an interrupt or exception instruction is in its fetch stage, TM first rolls back FM to execute the wrong path. Then, it again rolls back FM to execute the right path when branch misprediction is finished or interrupt/exception instruction jumps to the trap handling path in the commit stage.

To further optimize the rollback strategy, we create a *wrong-path FM* to execute the wrong path to provide TM with the instruction information. However, no data information is transferred because the wrong path instructions are not committed to change architecture states actually.

In fact, wrong-path FM is a simplified FM, which has the similar functionalities with the main FM. The simplification is from two aspects: 1) When creating a wrong-path FM, we only initialize the register values for it. In detail, the initial values of these registers are the execution results of the instruction which is just before the mispredicted instruction. That means, in wrong-path FM initialization, we first execute all the instructions in the pipeline and apply the states updating caused by these instructions. For memory values, it reads from the main FM and MAT, or the values it stores in its load-store queue. Since MAT records all the memory traces, wrong-path FM will get the correct memory value according to the timestamp of the mispredicted instruction. While for MMU states and I/O states, it reads directly from the main FM since wrong-path instructions are not committed and cannot change these states. 2) During wrong-path execution, it uses its own copy of registers. Since TM simulation does not need the actual value of each register, wrong-path FM needs only to provide TM with the instruction information as described in Section 3.3. Thus the execution results of wrong-path instructions are only effective in wrong-path FM and will not affect the simulation in TM. When a branch misprediction is finished or an interrupt/exception jumps to the trap handling path (correct path), Transformer terminates the wrong-path FM and TM receives instruction and data flow information from main FM again. Specifically, after TM calculates the target address of the branch instruction, it will raise a misprediction exception if the target address is different from the predicted target address. Moreover, this exception will cause the termination of wrong-path FM and a squash of pipeline.

Branch instruction may also occur in wrong-path instructions. For each of these instructions, wrong-path FM will record the current states in a checkpoint and execute along the predicted path. If the misprediction instruction in the correct path, which initializes the wrong-path FM, first raises an exception, wrong-path FM will discard all the checkpoints and terminate. If misprediction exception is raised by wrong-path branch instructions, wrong-path FM will roll back to the related checkpoint and continue execution along another path. In fact, in our evaluation, wrong-path execution will meet 3-4 branch instructions at most, but almost no roll back really happens (less than five times in one benchmark).

3.3 Architecture-Independent Interface

3.3.1 Interface Overview

For the sake of extensibility, we design an architecture-independent interface in Transformer between FM and TM. FM only needs to map the instructions to the interface and TM only needs to read the interface to do the detailed simulation. We abstract and generalize the information needed for the detailed simulation in TM as two main parts: pipeline dependence information and memory information. Pipeline dependence information includes the function unit usage, register usage, etc. Memory information is used for the simulation of memory hierarchy, and includes PC address and memory address for memory instructions.

In order to guarantee cycle-accuracy, we also need to consider the divergence factors, as discussed in Section 2.2. For interrupts and exceptions, we set a flag for the instruction and record its detailed type. For branch instructions, we provide target PC addresses for both the two branch directions. When TM simulates the branch instructions, TM will predict which direction it will take and directly read the corresponding address. In order to process shared page access order divergence, we set a flag to illustrate whether this instruction will cause an MMU miss. As for shared data access order divergence, we insert an access node into MAT for each memory instruction. In addition, in order to support the rollback mechanism as discussed in Section 3.2, we need to record the changed register values for each instruction and the old memory value for each store instruction.

In general, we design our architecture-independent interface as containing the following information:

- *Pipeline dependence:* Whether an instruction can issue in the pipeline depends on two conditions: 1) whether the functional unit is available; 2) whether the source operands are ready. The second type of dependence is maintained by registers for computational instructions and by memory

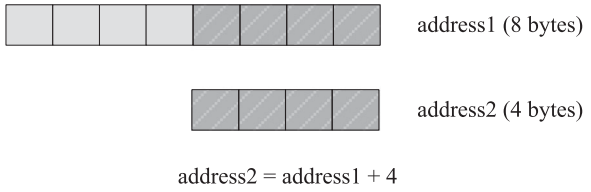

Fig. 8. Interleaved cases with different memory address.

address for memory instructions. Thus, we abstract the pipeline dependence of an instruction as three factors: functional unit for this instruction, source/destination register ID, and memory address.

- *Memory information:* For instruction cache simulation, we need the PC address for each instruction. For data cache simulation, we need memory address for memory instructions. Both the addresses include virtual address and physical address.
- *Accuracy information:* In order to guarantee cycle-accuracy, we need to record interrupt/exception information, branch target PC addresses for both directions, and MMU misses. We also need to record changed register values for each instruction and old memory values for store instructions, as the support of rollback mechanism. Moreover, to detect shared data access order violation, the interface needs to include shared data access order in MAT, which will be introduced in detail later.

Such an architecture-independent interface provides Transformer with more flexibility to extend the state-of-the-art FMs or TMs. Since a loosely-coupled design and clear interface, a new FM only needs to map its instruction information to the interface and support the rollback or the block strategy. It does not need to know other details in TM. Moreover, for a new TM, it only needs to read the interface to do detailed simulation and generate necessary checking information to detect and deal with divergence conditions.

3.3.2 MAT Design

In the architecture-independent interface, MAT is the most complex one because it includes both regular memory information and the information for memory order violation. Besides reading memory information for detailed timing simulation, TM also needs to retrieval it for order violation detection. Its design will influence not only the extensibility, but also the performance. Therefore, two issues should be considered carefully for MAT design. The first one is how to organize these memory items in MAT since different memory operations will access the data with different data length. The second is how to retrieval an item in MAT efficiently.

There are memory accesses with different data length in an application, such as one-byte datum, two-byte datum or four-byte datum. Memory operations with different data sizes might have overlap when they access their data. The memory accesses in Fig. 8 is such an example. In this example, Addr1 is an 8-byte *write* operation and addr2 is a four-byte *read* access. They have 4-byte overlap for shared data accesses. To detect this interleaving order violation, we can divide each memory access into several memory accesses where each only accesses one-byte length. However, such a solution will lead to more items are inserted into MAT with more space overhead. Moreover, because one memory operation may correspond to multiple items in MAT, it will involve more time overhead to maintain MAT, such as deleting items when a memory operation is committed.

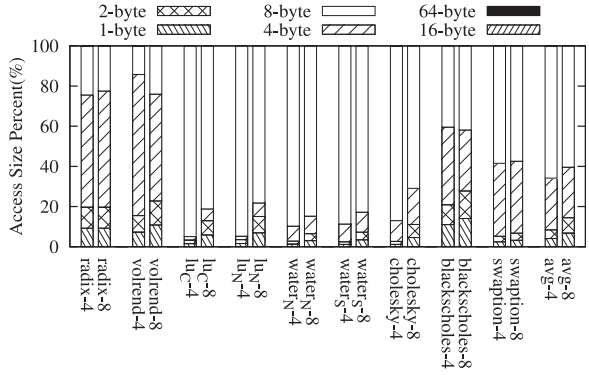

Fig. 9. Proportion of each address size.

In modern process design, most Reduced Instruction Set Computer (RISC) Instruction Set Architectures (ISA) align memory address, i.e., n-byte access address is n-byte aligned. Moreover, for Complex Instruction Set Computer (CISC) ISAs with various address length, an unaligned address can be divided into two aligned addresses. Therefore, to avoid the constraints of a division strategy, we apply an up-bound alignment method for MAT organization. When a memory address is inserted into MAT, it is converted to an up-bound value to represent its address. To determine the up-bound value, we profiled the memory access address in different applications and find that more than 99.9 percent of memory access size is smaller than 8 bytes. The data are shown in Fig. 9. Therefore, using 8-byte aligned address can fit most addresses to avoid dividing the address. Therefore, we set it as the default up-bound value. If a memory assess length is larger than 8-byte, it will be split into multiple addresses aligned to 8 bytes. The detailed process to maintain MAT is shown as follows.

- For memory addresses accessing not larger than 8-byte data, we put it into an 8-byte aligned memory address slot and using an 8-bit bitmap to record which bytes it accesses in the corresponding 8 bytes. By doing so, we define shared memory access, i.e., interleaved accesses, as two memory accesses 1) from different cores, 2) fixed into the same 8-byte address slot, and 3) the *and* operation result for two bitmaps does not equal zero, i.e., two addresses are interleaved.
- For memory addresses accessing larger than 8-byte data (e.g., 16-byte, or 64-byte), we divide the address to several 8-byte aligned addresses.

The second issue for MAT design is how to retrieve an item in MAT when checking the order violation. Here, we use a hash table to reduce the retrieval overhead of a memory address in MAT. To achieve this goal, MAT is organized as a hash table, which use *mod m* operation as the hash function. Each item in the hash table points to a link list. All the memory addresses with the same hash value are saved in the same link list. When a memory address is ready, it will be inserted at the head of the corresponding link list. When

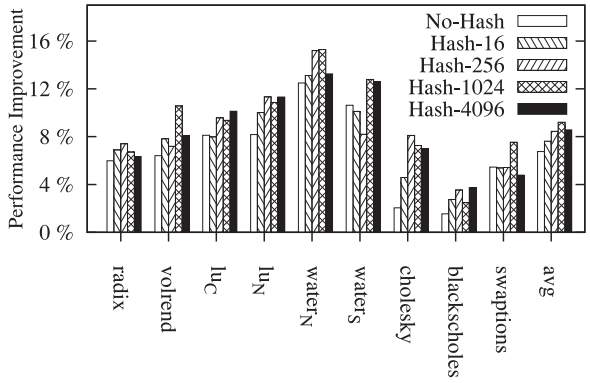

Fig. 10. Performance improvement of different MAT hash sizes.

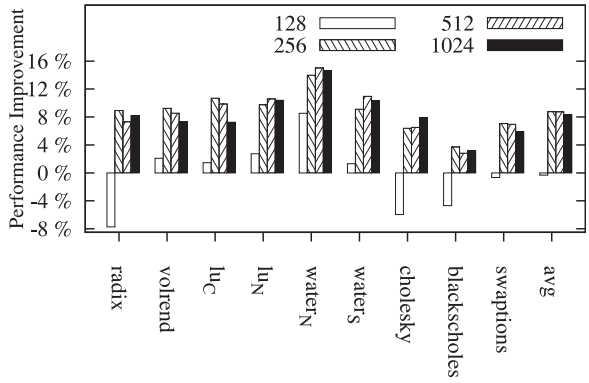

Fig. 11. Performance improvement of different buffer sizes.

a memory operation is committed, the corresponding list is retrieved to check whether there exists a violation. Since the value m decides the length of each link list, it will influence the performance when retrieving a memory address in MAT. To decide the parameter of the hash function, we conduct an evaluation with different hash sizes. As the data shown in Fig. 10, when the hash value (m) is 256, we can achieve the best performance, which brings 8.44 percent performance improvement on average compared to the original GEMS. Therefore, we apply this configuration as the default parameter for the design of hash table.

Since the buffer size of architecture-independent interface also influences the performance, we also collect the performance data in different buffer size. As the data shown in Fig. 11, when the buffer size reaches 256, the performance will change little. Therefore, we set 256 as the default buffer size for the architecture-independent interface.

4 Case Study: Extending QEMU

In our current implementation, we mainly based on GEMS, which use Simics as its FM. QEMU, as an open-sourced emulator, has been widely used in architecture designs and system research. To illustrate the extensibility of Transformer, we extended QEMU into Transformer as a new FM.

4.1 Challenges and Requirements

QEMU is a fast machine emulator based on dynamic translation. It emulates several CPUs, such as x86, PowerPC, ARM, and SPARC, on different hardware platforms. While integrating a QEMU version with RISC ISA, such as MIPS and ARM, the work for extension is straight-forward because the implementation of Transformer is based on GEMS, which is SPARC based and belongs to RISC ISA. We only needed to extend QEMU to make it be able to collect the execution information from applications and translate it into the formats of architecture-independent interface. TM gets such information and directs the execution of QEMU when divergence is detected. However, x86 emulation is an important component in QEMU. As the most widely used CISC ISA, x86 has significantly different features from those of RISC ISAs, such as complex instruction semantics and various instruction length. Thus a mechanism is required to remove the gap between CISC ISAs and RISC ISAs.

There are two major challenges here. The first challenge is how to translate these CISC instructions into the formats of architecture-independent interface because these instructions have more complex structure. The second one is how to simulate the timing behavior of these CISC instructions. RISC instructions are with fixed lengths. In contrast, CISC instructions are with varied lengths. Such a difference requires a different I-Cache simulation mechanism. Moreover, to guarantee cycle-accuracy, Transformer needs to roll back the execution of FM when a divergence is detected. However, there is an obstacle for CISC instruction simulation. When a CISC instruction is executed, it is usually mapped to several RISC-like micro-instructions. When an interrupt occurs during the simulation, we need to record the correct architectural and micro-architectural states. This means if some micro-instructions of one CISC instruction have already been committed and the others have not, we are unable to guarantee the correctness of the recorded architectural states because the CISC instruction has partially committed.

4.2 Implementation

In the current design of QEMU, it uses Tiny Code Generator (TCG) as a median layer between the target ISA and the host ISA. TCG is a RISC-like intermediate representation, which is similar to the micro-instructions translated from CISC instructions. A target instruction is first translated into a TCG representation (one or multiple TCG instructions). Then these TCG instructions are mapped onto the host machine. To bridge the gap between CISC ISA and RISC ISA, we extract the execution information for TM from TCG directly. Pipeline dependence and memory information are extracted from TCG and written into the communication buffer between FM and TM. TM reads the buffer and reorganizes the information with its internal instruction form, which we call *microinstruction*, for detailed simulation. To guarantee precise exception and rollback, the microinstructions belonging to the same CISC instruction (x86 instruction here) will not be committed until all of the microinstructions are ready to be committed. If a CISC instruction causes an interrupt or a branch misprediction, the interrupt or branch information will be processed at a CISC instruction level. Thus we guarantee the semantic correctness for a CISC ISA in our framework.

To simulate the behavior of instruction cache, the microinstructions decomposed from one CISC instruction share the same PC address, which is the actual address of the CISC instruction. Only the address of the first

TABLE 3
Baseline 4-core/8-Core OoO SPARC Configuration

4-core/8-core SPARC configuration			
Per-core parameters		Memory Hierarchy Parameters	
Pipeline width	4	L1 Cache	split I/D cache
Functional units	4 Int add/mul, 2 Int div, 2 Load		each 64KB
	2 Store, 2 Branch, 4 FP add		2-way set associative
	2 FP mul, 2 FP div/sqrt		64B cache lines
Integer FU latencies	1 add, 4 mul, 20 div		2 cycle latency
FPFU latencies	2 default, 4 mul, 12 div, 24 sqrt	L2 Cache	unified 4MB/8MB cache
Reorder buffer size	128/64		8-way set associative
Instr. window size	64/32		64B cache lines
Load-store queue	64/32		20/25 cycle latency
Branch predictor	yags predictor	Memory	200 cycle latency
Data speculation	no	Cache coherence	MOESI_CMP_directory

microinstruction will be used to access I-cache normally. After this address finishes the cache access, all the related microinstructions will be removed from the interface buffer without any other cache access.

5 Evaluation Results

In this section, we first present our experience to illustrate the extensibility and high productivity of Transformer. Then, we evaluate the performance improvement of Transformer brought by simple interaction and parallelization between FM and TM. Finally, we discuss the accuracy influence of different divergence factors.

5.1 Experimental Setup

Our baseline processor is a 4-core/8-core out-of-order SPARC processor with an MOESI cache coherence protocol. Each core has an out-of-order pipeline with yags branch predictor. Detailed configuration is shown in Table 3.

We use SPLASH-2 [18] and PARSEC [6] benchmark suites for evaluation. The benchmarks run on a Solaris 10 operating system with reference input. The baseline simulator is Simics 3.0.31 + GEMS 2.1.1 [11], a widely-used tightly-coupled multicore simulator. Our Transformer prototype is constructed based on GEMS and it is with about 4.2 K LOCs changes in total: about 1 K modified LOCs in GEMS to decouple FM and TM, and about 3.2 K added LOCs to guarantee cycle-accuracy and provide architecture-independent interface between FM and TM. All the experiments are executed on a 6-core Intel I7 980 CPU (3.33 GHz, private L1 and L2 cache, 12 M shared L3 cache) with 2 GB memory.

5.2 Simulation Extensibility

As Transformer exploits the loosely-coupled design and provides an architecture-independent interface (e.g., pipeline dependence, memory information and accuracy maintaining information) between FM and TM, the extension becomes much easier. Extending an FM only needs to map the executed instructions into the interface information, which is generally direct available through instrumentation. While extending a TM only needs to make TM read directly from interface, instead of doing detailed decode itself and interacting frequently with FM. As a result, the extension efforts of constructing multicore simulators, measured in man-months, can be significantly reduced.

To demonstrate the extensibility of Transformer, we have extended a functional model QEMU [5] into our framework to construct an x86 simulator. This extension only consists about 180 lines of code. The whole extension work was done by a graduate student, who is familiar with QEMU but new to GEMS, in about 350 working hours (about two months).

TABLE 4
Extension Efforts Comparison

Simulator	Combining Work	Extension Efforts
gem5 [7]	GEMS + M5	Dozens of person-years
MARSS [13]	PTLsim + QEMU	1.5 years by a 4-people group
Transformer	GEMS + QEMU	About two person-months

Compared to multiple person-year efforts cost in prior extension work such as gem5 and MARSS, shown in Table 4, Transformer provides an alternative solution with its loosely-coupled framework, to ease the efforts and difficulties in simulator researches.

5.3 Performance Improvement

5.3.1 Improvement of Sequential Transformer

We first evaluate the performance of the sequential Transformer framework, i.e., loosely-coupled Simics and GEMS, against the tightly-coupled baseline simulator Simics and GEMS. As shown in Fig. 12, the sequential Transformer achieves about 8.4 and 7.0 percent performance improvement on average for 4-core configuration and 8-core configuration, which is mainly from simpler interaction: 1) fewer interactions only for rare path divergence cases (less than 1 percent), where TM revises the execution; 2) TM no longer simulates redundant functional execution. The 8-core configuration performance improvement is smaller than that of 4-core configuration because TM under the 8-core configuration occupies more proportion of the execution time.

5.3.2 Improvement of Parallel Transformer

Due to the loosely-coupled design between FM and TM, we can parallelize FM (i.e., Simics) and TM (i.e., GEMS) with pipeline parallelism. The parallelized FM and TM works as two threads: FM thread produces instruction and data flow information to a buffer; TM thread reads the buffer, simulates micro-architecture, and revises FM or creates a wrong-path FM (in the same thread) if necessary.

As the data shown in Fig. 13, parallel Transformer achieves about 35.3 and 29.7 percent performance improvement against the baseline GEMS simulator for 4-core configuration and 8-core configuration. The performance improvement from parallelizing FM and TM is about 29.4 and 24.4 percent. The reason for this improvement is that the parallelization not only distributes the computation into two different cores, but also achieves better instruction and data locality as FM

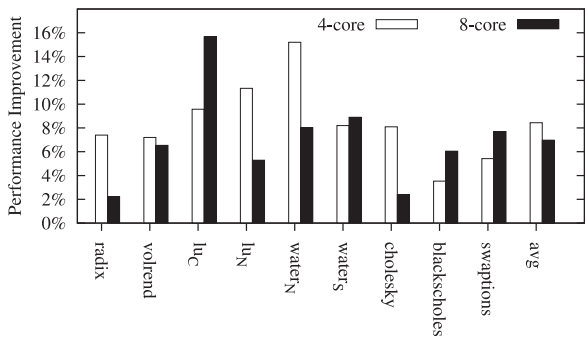

Fig. 12. Performance improvement of sequential transformer.

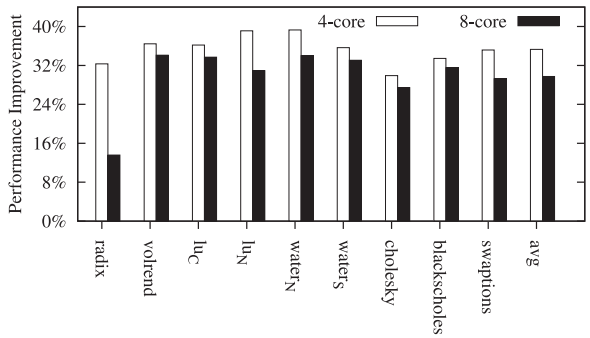

Fig. 13. Performance improvement of parallel transformer.

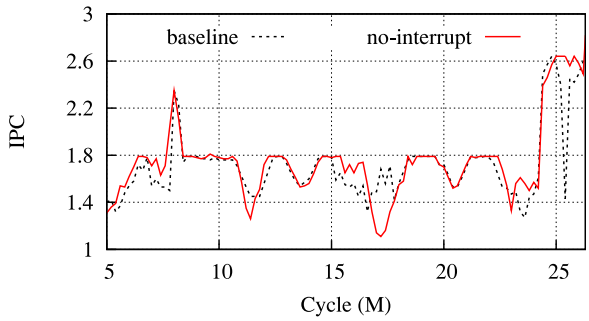

Fig. 15. Interrupt/exception influence.

and TM are separated. Moreover, in a timing-first simulation model, TM has to include an FM part to check whether its states are consistent to those got from FM. In a contrast, the TM in Transformer only needs to simulate the timing behavior without a redundant functional execution. Therefore, while parallelizing FM and TM in Transformer, the execution of TM will dominate the total execution time. This is the reason why the performance gain of parallel Transformer exceeds the potential improvement (the performance gain from only removing FM from GEMS) shown in Table 1.

5.4 Analysis of Divergence Factors

In order to validate our analysis of cycle-accuracy factors resulting in execution divergence between FM and TM in Section 2.2, we take a detailed evaluation on the influence of accuracy, which can provide some insights to full-system multicore simulation designs.

5.4.1 Influence of IPC

First, we use a full-feature version simulator as the baseline and evaluate different versions, which ignores one divergence factor respectively. Then for each version, we compare the average Instructions Per Cycle (IPC) values of the 10,000 cycles intervals throughout the application between the baseline and the factor-removed versions. The comparison results are shown as the curves in the following figures. The *y*-axis of each point in the curve represents the average IPC of the 10,000 cycles interval and the *x*-axis represents the cycle count from the beginning. Considering the variation of different benchmark applications, we just analyze one application as an example for each divergence factor. All the data are evaluated under the 4-core configuration and collected after the initialization part (as the 5 M-25 M cycles curve shown in the figures).

Branch misprediction influence. Fig. 14 shows the branch misprediction influence on water-N benchmark from SPLASH-2 of the intervals from 5 to 20 M cycles as an example. From the figure we can find obvious variances between the two curves. The variance is not that distinct but continuous, resulting from the relatively high occurrence rate of branch misprediction.

Interrupt/exception influence. Fig. 15 shows the interrupt and exception influence on Lu-C benchmark from SPLASH-2. The two curves lap over each over mostly, because of the interrupts happen rarely. The obvious variance occurs at 18 and 25 M cycles respectively. In general, the interrupt-removed version shows little error rate with the baseline version.

Shared data access order violation influence. Fig. 16 shows the shared data access order influence on swaptions benchmark from PARSEC. The IPC fluctuates because of the application behavior. We can also find some variances in the figure. Although the shared data access order does affect the scheduling of the threads interleaving, the total violation influence is not that obvious.

MMU miss influence. Fig. 17 shows the influence of shared data access order on lu-N benchmark from SPLASH-2. We can find a variance at 6 and 26 M cycles. In fact, MMU miss is not often and shared page access order violation occurs more rarely. That is the reason why the two curves in Fig. 16 are almost the same.

As above data shown, the divergence factors will mainly cause variances in a local area. To illustrate their global influence on accuracy, we collect the global IPC error rate when removing them. The data are shown in Fig. 18. As the data shown, except branch misprediction with 6.6 percent average error rate, the other factors have little accuracy influence on simulation.

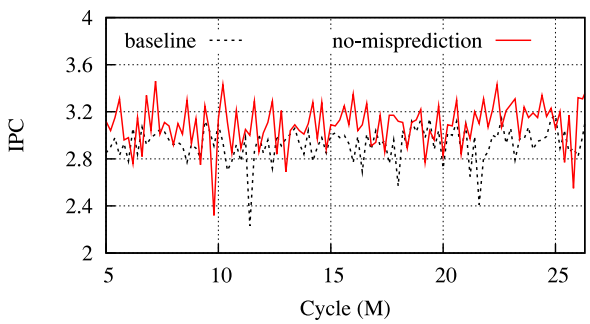

Fig. 14. Branch misprediction influence.

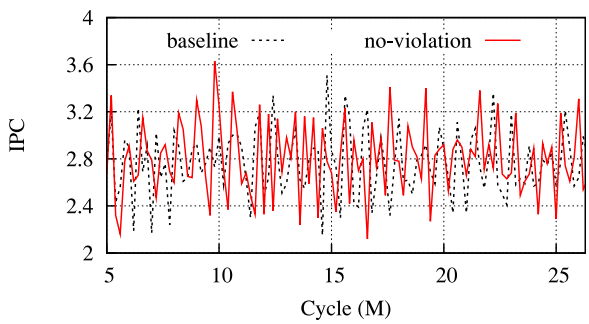

Fig. 16. Shared data access order violation influence.

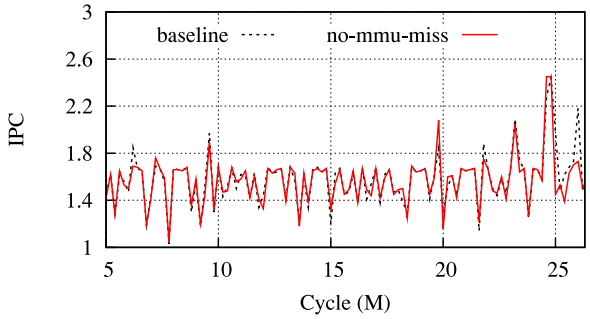

Fig. 17. MMU miss influence.

5.4.2 Influence of Divergence Instructions

Besides the occurrence rate of accuracy factors shown in Section 2.3, we also collect the number of divergence instructions to illustrate the influence of different accuracy factors. The percentages of these divergence instructions are shown in Table 5.

For branch misprediction factor, each misprediction will cause 7.73 wrong-path instructions, on average, fetched into the pipeline and squashed out later. These instructions are 1.77 percent of the total instructions of the applications.

The processing of interrupts or exceptions are similar to the branch misprediction, which uses the wrong-path FM to provide wrong-path instructions and relative information. On average, each interrupt or exception will introduce 15.7 divergence instructions across the workloads. The number of divergence instructions is larger than that of branch misprediction because an interrupt or exception divergence can only be found when it is committed in the commit stage of the pipeline. Since interrupts and exceptions occur rarely, the percentage of divergence instructions is 0.2 percent on average and the influence is tiny.

When a shared data access order violation happens at the commit stage of the pipeline, FM will roll back to the correct states. Since TM simulates the timing of the instruction and thus is always correct, FM will roll back all its wrongly-executed instructions. Moreover, TM gets the instructions from FM through the communication buffer, TM also needs to squash pipeline and re-fetch the correct instructions. In fact, the pipeline is almost full during the normal execution and TM will usually squash all the instructions in the instruction window. Thanks to the low occurrence rate of shared data

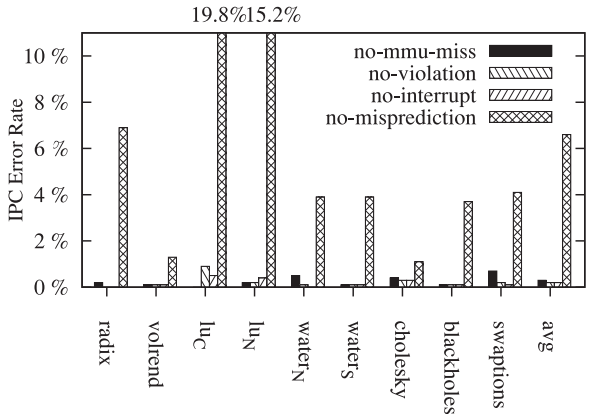

Fig. 18. Total influence.

TABLE 5
Influence of Divergence Instructions

Factor	Proportion
Branch misprediction	1.77%
Interrupt/exception handling	0.2%
Shared data access order violation	0.48%
Shared page access order violation	0.1%

access order violations, the percentage of divergence instructions in TM is no more than 0.5 percent. However, the case of FM is a little different. In sequential version, the step of FM and TM are carefully controlled that FM advances and fills in the instruction buffer when TM needs to fetch instructions into the pipeline. Thus in the worst occasion, FM will roll back all the instructions in the pipeline instruction window, which is similar as TM. However, in the parallel version, FM is much faster than TM and the instruction buffer is almost full all the time through. Thus FM will roll back all the instructions in the pipeline instruction window and communication buffer. Because of the relative larger size of communication buffer, FM will roll back more instructions, at most 6.8 percent of the total instructions. However, since most of the instructions in communication buffer is useless and are not executed at all in TM's perspective, FM's influence is very tiny.

In our framework, we avoid the occurrence of MMU miss order violation, so it will not cause divergence instructions in Transformer. But in the previous evaluation, as shown in Section 2.3, in the worst case (radix benchmark with the most occurrence proportion), the violation occurrence rate is 5.3E-5 at most. If each violation causes TM squash all the instructions in the instruction window, divergence instructions are 0.33 percent of total instructions at most, and 0.1 percent on average.

6 Related Work

In this section, we will discuss related works from two perspectives: simulator productivity and acceleration techniques.

6.1 Simulator Productivity

Existing full-system multicore simulators usually exploit a tightly-coupled FM and TM design to achieve cycle-accuracy. A good example is the widely-used Simics + GEMS [11] simulator, which is used in this paper as the baseline simulator. Other main-stream simulators, such as MARSS [13] and gem5 [7], exploit integrated FM and TM design, even more tightly-coupled than Simics + GEMS.

To achieve sampling simulation, Cotson [2] exploits a functional-directed loosely-coupled design, where FM executes ahead for most cases and TM gives feedback to the FM periodically. However, Cotson does not provide cycle-accurate solutions to revise those path divergences and only periodically provides feedbacks to FM. Within LSE framework [21], simulators can be constructed from a machine description that closely resembles the hardware, ensuring fidelity in the model. LSE abstracts the component as a black box and defines the communication between the components, thus supporting the extension of other components

or models. But its definition also limits the convenience in extending new components into its framework.

In contrast, Transformer gives a comprehensive analysis to which factors will affect cycle-accuracy in loosely-coupled design, i.e., branch misprediction, interrupt/exception handling, shared data access order, shared page access order. We further provide several lightweight solutions to detect and revise the simulation instead of simply rolling back FM using heavy-overhead checkpoint mechanism. Moreover, we design an architecture-independent interface between FM and TM to make it more extensible.

6.2 Acceleration Techniques

Another category of related work is simulation acceleration techniques, including parallel simulation, Field-Programmable Gate Array based (FPGA-based) simulation, sampling techniques and analytical modeling.

For parallel simulation, SlackSim [8] uses multi-threads to simulate different cores and a master thread to control steps of different cores to guarantee the correct access order of shared resources. Because the maximum distance of threads' steps is usually set as last level cache hit latency, SlackSim has lots of synchronization among the threads. Graphite [12] also parallelize the timing simulation and uses an algorithm to randomly synchronize the simulation threads, which reduces the number of synchronization and achieves a high performance. However, it cannot guarantee cycle accuracy because lots of threads are out of cycle limit threshold. Sniper [22], which combines Interval Simulation [23] and Graphite together, also proves that the relaxed synchronization models can lead to significant errors.

FPGA-based simulators, such as RAMP GOLD [15], HASim [14] and UT-FAST [9], use FPGA to help achieving high performance and accuracy. The FPGA acceleration usually brings considerable improvement on performance, from hundreds of KIPS to tens of MIPS. However, FPGA simulators cost too much on developing time and efforts, and the complexity, which is a major restriction on further optimization. Moreover, it is quite difficult to extend or modify components and models in FPGA platforms.

Sampling techniques like ESESC [3] and DAPs [4] exploit cyclic behavior in applications to accelerating the simulation speed. It aims at the sampling of the multi-threaded benchmarks basing on the randomicity of transactions in the server workloads.

There are also some works using analytical models to accelerate the timing simulation of micro-architecture. Interval Simulation [23] is an application level simulator, and mainly focuses on the simulation of miss events which will affect the normal execution of instruction traces. ZSim [24] introduces its instruction-driven approach, which uses the instruction-centric timing model for fast simulation.

Thanks to the extensibility of Transformer, this framework is orthogonal to the above techniques focusing on accelerating TM part. In fact, the above acceleration techniques can be applied to Transformer framework with little modification, which is one of our future work.

7 Conclusion

In this paper, we proposed Transformer, an extensible, fast, and cycle-accurate loosely-coupled full-system multicore simulator. We first presented a comprehensive analysis to four factors affecting cycle-accuracy in loosely-coupled design and provided lightweight solutions to detect and revise these divergence factors to ensure cycle-accuracy. Then we further designed an architecture-independent interface between FM and TM, which makes Transformer more flexible to extend state-of-the-art FMs and TMs. As demonstrated, a graduate student only wrote about 180 lines of code to extend an X86 functional model based on Transformer. Finally, besides the simple interaction, we further parallelized FM and TM to improve the performance. Experiments showed that in 4-core and 8-core configuration, it achieved about 8.4 and 7.0 percent performance improvement compared to the widely-used tightly-coupled baseline simulator GEMS [11] and 35.3 and 29.7 percent performance improvement after parallelizing FM and TM.

Acknowledgments

We are grateful to supports from the National High Technology Research and Development Program of China (No. 2012AA010905), the National Natural Science Foundation of China (No. 61370081), the Key Project of Major Program of Shanghai Committee of Science and Technology under Grant (No. 13DZ1108800). We would like to thank all our anonymous reviewers for valuable feedback on the paper. Haibo Chen is the corresponding author.

References

[1] Z. Fang, Q. Min, K. Zhou, Y. Lu, Y. Hu, W. Zhang, H. Chen, J. Li, and B. Zang, "Transformer: A functional-driven cycle-accurate multicore simulator," in *Proc. 49th Annu. Des. Autom. Conf.*, 2012, pp. 106–114.

[2] E. Argollo, A. Falcn, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: Infrastructure for full system simulation," *Operating Syst. Rev.*, vol. 43, no. 1, pp. 249–261, 2009.

[3] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, Washington, DC, USA, 2013, pp. 448–459.

[4] C.-C. Chen, Y.-C. Peng, C.-F. Chen, W.-S. Wu, Q. Min, P.-C. Yew, W. Zhang, and T.-F. Chen, "Daps: Dynamic adjustment and partial sampling for multithreaded/multicore simulation," in *Proc. 51st Annu. Des. Autom. Conf.* New York, NY, USA, 2014, pp. 129:1–129:6.

[5] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, pp. 41–41.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," in *Proc. Comput. Archit. News*, 2011, pp. 1–7.

[8] J. Chen, L. K. Dabbiru, M. Annavaram, and M. Dubois, "Adaptive and speculative slack simulations of CMPs on CMPs," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 523–534.

[9] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 249–261.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Comput.*, vol. 35, pp. 50–58, Feb. 2002.

[11] C. J. Mauer, M. D. Hill, and D. A. Wood, "Full-system timing-first simulation," in *Proc. Int. Conf. Meas. Modeling Comput. Syst.*, 2002, pp. 108–116.

[12] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proc. IEEE 16th Int. Symp. High Perform. Comput. Archit.*, 2010, pp. 1–12.
[13] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *Proc. 48th Des. Autom. Conf.*, 2011, pp. 1050–1055.
[14] M. Pellauer, M. Adlery, M. Kinsy, A. Parashary, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 406–417.
[15] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic, "RAMP gold: An FPGA-based architecture simulator for multiprocessors," in *Proc. 47th Des. Autom. Conf.*, 2010, pp. 463–468.
[16] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovic, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 290–301.
[17] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "Coremu: A scalable and portable parallel full-system emulator," in *Proc. 16th ACM Symp. Principles Practice Parallel Program.*, 2011, pp. 213–222.
[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.
[19] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev, "MPTLsim: A simulator for X86 multicore processors," in *Proc. 46th ACM/IEEE Des. Autom. Conf.*, 2009, pp. 226–231.
[20] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," *ACM SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 181–193, 1997.
[21] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 29–40.
[22] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–12.
[23] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, Feb. 2010, pp. 307–318.
[24] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 475–486.

Weihua Zhang received the PhD degree in computer science from Fudan University in 2007. He is currently an associate professor of the Parallel Processing Institute, Fudan University. His research interests are in compilers, computer architecture, parallelization, and systems software.

Haojun Wang is now a graduate student in the Software School of Fudan University and working in the Architecture group of Parallel Processing Institute. His work is related to computer architecture, simulation, parallel optimization and so on.

Yunping Lu is currently working toward the PhD degree in the School of Computer Science at Fudan University. Her research interests are in compilers, computer architecture, parallelization, and systems software.

Haibo Chen received the BSc and PhD degrees in computer science from Fudan University in 2004 and 2009, respectively. He is currently a professor in the School of Software, Shanghai Jiao Tong University, doing research that improves the performance and dependability of computer systems. He is a senior member of the IEEE and the IEEE computer society.

Wenyun Zhao received the masters degree from Fudan University in 1989. He is a full profes- sor of the School of Computer Science at Fudan University. His current research interests include software reuse, software product line, software component, and architecture.