Transformer: A Functional-Driven Cycle-Accurate Multicore Simulator

Zhenman Fang[1,2], Qinghao Min[2], Keyong Zhou[2], Yi Lu[2], Yibin Hu[2], Weihua Zhang[2], Haibo Chen[3], Jian Li[4], Binyu Zang[2]
[1]The State Key Lab of ASIC & System, Fudan University. [2]Parallel Processing Institute, Fudan University.
{fangzhenman, minqh, zky, yil, huyibin, zhangweihua, byzang}@fudan.edu.cn
[3]Institute of Parallel and Distributed Systems, Shanghai Jiaotong University. haibochen@sjtu.edu.cn
[4]IBM Austin Research Laboratory. jianli@us.ibm.com

ABSTRACT

Full-system simulators are extremely useful in evaluating design alternatives for multicore. 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 ignored before. Based on the study, 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. Based on Transformer, a graduate student only needs to write about 180 lines of code and takes about two months 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% speedup over GEMS while guaranteeing the cycle-accuracy. A further parallelization between FM and TM leads to 35.3% speedup.

Categories and Subject Descriptors

B.2.2 [**Performance Analysis and Design Aids**]: Simulation

General Terms

Design, Measurement, Performance

Keywords

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 and 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 simulator, researchers have designed and implemented a number of FMs such as Simics [8], QEMU [2], and COREMU [15], and TMs such as GEMS [9], MPTLsim [18] and RAMP GOLD [14]. 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 [4]) or extend QEMU to PTLsim (MARSS [11]). Further, such a tightly-coupled design also makes it hard to efficiently parallelize FM and TM, resulting in inferior performance.

There is actually 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. Though there have been some efforts in trying to explore a loosely-coupled design for multicore simulators [7, 6], their solutions cannot guarantee cycle-accuracy and there is no implemented prototype for multicore simulators.

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%). 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 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 the performance.

We have implemented Transformer based on GEMS [9], a widely-used tightly-coupled simulator, and parallelize FM and TM to achieve better performance. And we plan to release the source code of Transformer to the community in future. Based on Transformer, a graduate student only needs to write about 180 LOCs and takes about two months to extend an X86 functional model (QEMU) in Transformer. Furthermore, experiments with SPLASH-2 [17] and PARSEC [3] show that Transformer achieves about 8.4% speedup compared to GEMS while guaranteeing the cycle-accuracy. And the speedup increases to 35.3% after FM and TM are parallelized.

In summary, this paper makes the following contributions:

- The first comprehensive analysis on the 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 extends 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 demonstrates an example for extending X86 support and evaluates the performance speedup of Transformer. Section 5 describes related work. Finally, section 6 concludes the paper and discusses possible future work.

2. MOTIVATION

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 Figure 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 architecture states and timing. Moreover, TM has to simulate part of the functional model so as to direct the execution of FM. Such a tightly-coupled and complex interaction between FM and TM makes it very difficult to extend a new FM or TM into those simulator frameworks. For example, the developers spends years to combine M5 with GEMS (gem5 [4]) or extend QEMU into PTLsim (MARSS [11]).

In addition, the complex interaction in current tightly-coupled design limits simulation speed. To illustrate this problem, we profile the execution proportion of FM, TM and their interactions (using the experiment setup in section 4.1). First, to support TM, FM has to execute in instruction-by-instruction model instead of fast

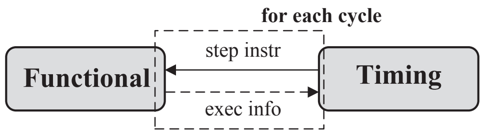

Figure 1: Tightly-coupled Functional and Timing.

binary translation to provide execution information to TM. As a result, FM occupies about 10% 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. Moreover, complex interaction produces about 26% overhead due to complex 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.

- *Branch misprediction:* In modern architectures, branch prediction is usually exploited in the pipeline design to avoid stall 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, not all shared data are protected by lock operations to achieve some harmless operations, such as user-level synchronization. Therefore, FM may execute a different write/read order compared with that of TM, which will diverge the execution path.
- *Interrupt/exception handling:* Interrupt or exception is similar to branch misprediction. TM handles the interrupt or exception (i.e., jumps to the interrupt or exception handling path) in the commit stage after squashing the pipeline. Before that, TM will fetch instructions from the wrong path, i.e., next program counter (PC) instead of the interrupt/exception handler code. However, FM directly simulates the interrupt/exception handling path, which leads to execution path divergence.
- *Shared page access order (i.e., MMU miss order):* In full-system multicore simulation, the system behavior has to be simulated. Although such a design guarantees cycle-accuracy, it involves some additional shared data access 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 a 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, a 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.

Although these factors would lead to execution divergence between FM and TM, they occur rarely. To illustrate this problem,

Table 1: 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

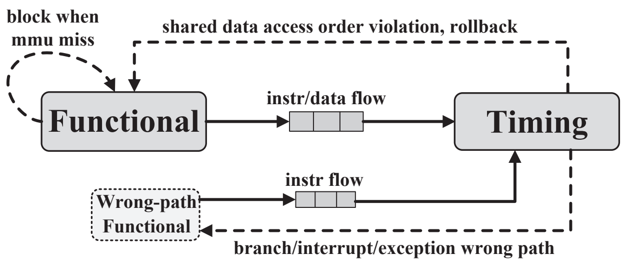
block when mmu miss
shared data access order violation, rollback
Functional
instr/data flow
Timing
instr flow
Wrong-path Functional
branch/interrupt/exception wrong path

Figure 2: The Transformer framework.

we profile the occurrence proportion of each divergence factor in the total execution (using the configuration in section 4.1). As the data shown in Table 1, branch misprediction occurs most and only occupies about 0.53%. The total proportion occurs less than 1%. Therefore, in most cases (more than 99%), there is no execution divergence between FM and TM. This opens the opportunity to use a loosely-coupled design that may result in better extensibility to support other FMs or TMs and superior 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 lightweight scheme to detect and recover from execution divergence to guarantee cycle-accuracy. Then, we illustrate an architecture-independent instruction and data flow interface between FM and TM, to make Transformer more extensible. The overall Transformer framework works as follows, as shown in Figure 2.

In Transformer, 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. 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

As an important factor affecting cycle-accuracy, prior work [6] detects violation in shared data access order through checking whether the loaded values of shared memory between FM and TM are the same. However, based on such an approach, it is difficult to

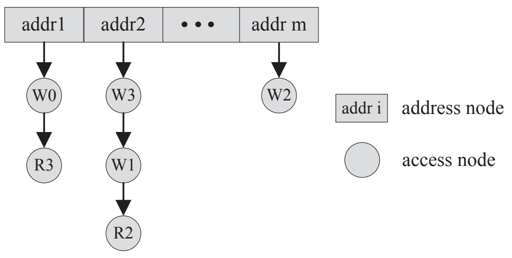

Figure 3: Memory Access Table structure.

know where and when the actual thread interleaving violation occurs. Since the loaded value may be affected by a faraway prior store instruction or two store instructions may have written the same value, the order violation 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, we design a data structure called *Memory Access Table (MAT)* to efficiently record and check the shared data access order. As shown in Figure 3, 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 as to the memory access node. Each memory access node records which core it comes from and its operation type (i.e., read or write). The shared data access recording and checking mechanism works 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 added 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 this node is deleted from MAT. When the memory access list becomes empty, the memory node of this address is also deleted 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 detail of MAT design could be found in the appendix section B.

3.1.2 Shared Page Access Order

For shared page order, i.e., MMU miss order, it is instinct to still use MAT to check the divergence. However, the functionality of MMU is only simulated by FM. In order to check whether the order violates, TM has also to be able to check whether MMU miss or hit. 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 a MMU miss is encountered, we block FM execution until TM directs it to advance, i.e., until the MMU miss instruction commits in TM. 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 a M-MU miss happens, it raises a MMU miss interrupt. As for interrupt handling, it will fetch instructions from the wrong path until the M-MU miss instruction commits. Though 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 the cycle-accuracy. As discussed in [7, 6], we can always deal with the divergence through rolling back FM when a divergence is detected. However, since FM runs ahead, it's difficult to know when to do a checkpoint. Therefore, it will produce large overhead to frequently save the states for checkpoint. Moreover, the rollback strategy can incur double rollback (from right path to wrong path, and again from wrong path to 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.

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. For example, SlackSim [5] 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 large overhead. Therefore, we introduce a lightweight mechanism to roll back FM states.

- For registers, which are lightweight inherently, 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.
- For memory values, we record the old value before 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 a 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 until TM commits all instructions before it. This mechanism avoids I/O rollback.

Optimized strategy: create a wrong-path FM. One problem for the rollback strategy is that it would incur double rollback for branch misprediction and interrupt or exception handling. It first rolls back FM to execute the wrong path when a branch predicts a wrong PC or an interrupt or exception instruction is in its fetch stage. Then, it again rolls back FM to execute the right path when branch misprediction is finished or interrupt or 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 actually are not committed to change architecture states.

When creating a wrong-path FM, we only initialize the register values for it. During wrong-path execution, it uses its own copy of registers. For memory values, it reads from the main FM and MAT, or the values it stores. 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. When branch misprediction is finished or interrupt/exception jumps to the trap handling path, Transformer terminates the wrong-path FM and TM gets instruction and data flow information from main FM again.

3.3 Architecture-Independent Interface

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 detailed simulation. As TM mainly simulates the pipeline dependence and memory behavior, we abstract each instruction as the following architecture-independent information:

- *Pipeline dependence:* Whether an instruction can issue in the pipeline depends on two conditions: 1) whether the functional unit is ready; 2) whether the source operands are ready. The second type of dependence is maintained by registers for computational instructions and by memory 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. Moreover, to detect shared data access order violation, the interface needs to include shared data access order in MAT.
- *Rollback information:* As discussed in section 3.2, we need changed register values for each instruction. Also, we need to save the old memory value for a store instruction.

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 direct rollback.

4. EVALUATION RESULTS

This section evaluates the extensibility and performance of Transformer. As our performance results show that Transformer guarantees the cycle-accuracy compared with GEMS, we omit the results for cycle-accuracy and only present the performance results of Transformer.

4.1 Experimental Setup

Our baseline processor is a 4-core out-of-order SPARC processor with a MOESI cache coherence protocol. Each core has an out-of-order pipeline with yags branch predictor. Detailed configuration is shown in Table 2. We also evaluate an 8-core configuration. Due to the space constraint, the results are shown in the appendix sections.

We use SPLASH-2 [17] and PARSEC [3] 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 [9], a widely-used tightly-coupled multicore simulator. Our Transformer prototype is constructed based on GEM-S and it is with about 5.5K LOCs changes in total: about 1.5K modified LOCs in GEMS to decouple FM and TM, and about 4K

Table 2: Baseline 4-core OoO SPARC configuration.

4-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 cache
Reorder buffer size	128		8-way set associative
Instruction window size	64		64B cache lines
Load-store queue	64		20 cycle latency
Branch predictor	yags predictor	Memory	200 cycle latency
Data speculation	no	Cache coherence	MOESI_CMP_directory

Table 3: Extension efforts comparison.

Simulator	Combining Work	Extension Efforts
gem5 [4]	GEMS + M5	Dozens of person-years
MARSS [11]	PTLsim + QEMU	1.5 years by a 4-people group
Transformer	GEMS + QEMU	About two person-months

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 (3.33GHz, private L1 and L2 cache, 12M shared L3 cache) with 2GB memory.

4.2 Simulation Extensibility

As Transformer exploits the loosely-coupled design and provides an architecture-independent interface (e.g., pipeline dependence, memory information) between FM and TM, the extension becomes much easier. Extending a 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. 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 [2] into our framework to construct an X86 simulator. The reason we choose QEMU as FM simulator is that it well supports X86 and plenty of full-system features and it is open-sourced and widely-used. We first decode X86 instructions into RISC-like micro-instructions using the decoder from PTLsim [11] and then directly map the micro-instructions into the architecture-independent interface.

This extension only consists about 180 lines of code. The whole extension work is done by a graduate student, who is familiar with QEMU but new to GEMS, in about two months. Compared to multiple person-year efforts cost in prior extension work such as gem5 and MARSS, shown in Table 3, much less efforts are needed to extend novel models in Transformer to construct a new full-system multicore simulator.

4.3 Performance Speedup

Speedup 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 Figure 4, the sequential Transformer achieves about 8.4% speedup on average, which is mainly from simpler interaction: 1) less interactions only for rare path divergence cases (less than 1%), where TM revises the execution; 2) TM no longer simulates redundant functional execution.

Speedup 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

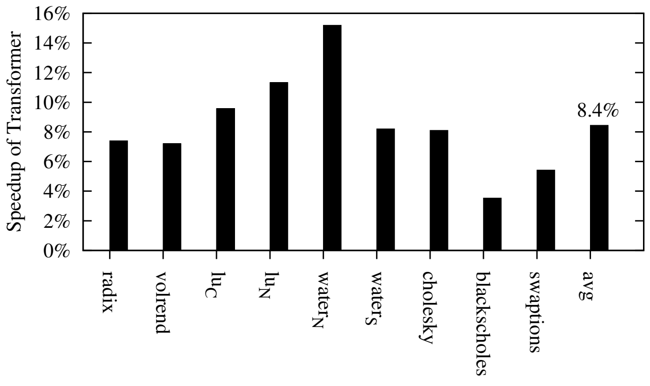

Figure 4: Speedup of sequential Transformer under 4-core configuration.

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.

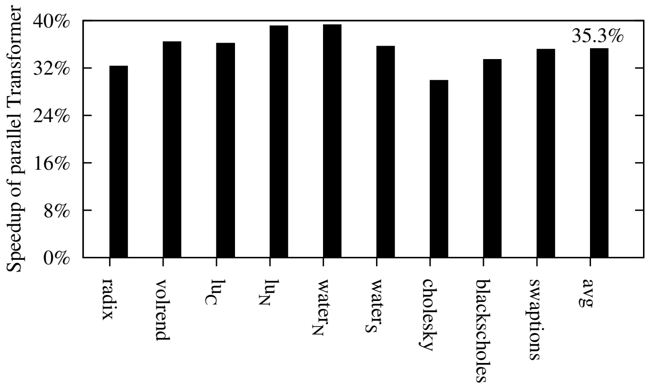

Figure 5: Speedup of parallel Transformer under 4-core configuration.

As the data shown in Figure 5, parallel Transformer achieves about 35.3% speedup against the baseline GEMS simulator. The speedup from parallelizing FM and TM is about 29%. The reason under this speedup is that the parallelization not only distributes the computation into two different cores, but also achieves better instruction and data locality as FM and TM are separated. This speedup is orthogonal to other acceleration techniques such as parallel TM simulation and FPGA-based simulation discussed in section 5. We can combine them to other acceleration techniques to further improve the performance.

5. RELATED WORK

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 [9] simulator, which is used in this paper as the baseline simulator. Other mainstream simulators, such as MARSS [11] and gem5 [4], exploit integrated FM and TM design, even more tightly-coupled than Simics + GEMS.

One of the most closest work to Transformer is UT-FAST [7, 6]. UT-FAST exploits a speculative Functional-First simulation design, in which FM (using QEMU) speculatively executes ahead to provide TM (implemented in FPGA) the instruction stream, and TM rolls back FM if branch misprediction. However, UT-FAST only supports single-core simulation and interrupt/exception handling as well as shared page access are not considered. Though it further discusses its extension for multicore simulation in [6], it only considers memory value violation between FM and TM. On one hand, as we discussed in section 2, the factor of shared page access order is not discussed in their paper, which leads to that their solutions cannot achieve cycle-accuracy. On the other hand, they have not implemented their design in a real simulator. While Transformer has done a real implementation with several novel designs such as wrong-path FM and architecture-independent interface, and guarantees cycle-accuracy. To achieve sampling simulation, Cotson [1] also 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.

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.

Another category of related work is simulation acceleration techniques, including parallel simulation [5, 10], FPGA-based simulation [13, 12], sampling techniques [16, 1], and etc. We use a method orthogonal to the above ones to improve the performance: simple interaction and parallelization between FM and TM.

6. CONCLUSION AND FUTURE WORK

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 and took about two months to extend an X86 functional model (QEMU) based on Transformer. Finally, besides the simple interaction, we further parallelized FM and TM to improve the performance. Experiments showed that it achieved about 8.4% speedup compared to the widely-used tightly-coupled baseline simulator GEMS [9] and 35.3% speedup after parallelizing FM and TM.

There are mainly two directions in our future work. First, we plan to release the source code of Transformer to the public in the near future. Second, we will extend Transformer to support System-on-Chip (SoC) simulation by taking advantage of QEMU and the architecture-independent interface between FM and TM.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 60903015, a joint program between China Ministry of Education and Intel numbered MOE-INTEL-10-04, Key Project of National 863 Program of China under Grant No. 2009AA012201, National 863 Program of China under Grant No. 2012AA010905, Key Project of Major Program of Shanghai Committee of Science and Technology under Grant No. 08dz501600, Opening Project of Architecture Key Laboratory of Institute of Computing Technology in Chinese Academy of Sciences under Grant No. ICT-ARCH2009082009, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

8. REFERENCES

[1] E. Argollo, A. Falcĺőn, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: Infrastructure for full system simulation. *Operating Systems Review*, 43(1):249–261, 2009.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX ATC 2005*.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. *PACT 2008*, pages 72–81.

[4] 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. *Computer Architecture News*, 2011.

[5] J. Chen, L. K. Dabbiru, M. Annavaram, and M. Dubois. Adaptive and speculative slack simulations of cmps on cmps. *Micro 2010*, pages 523–534.

[6] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo. Accurate functinal-first multicore simulators. *Computer Architecture Letters*, 8:64–67, 2009.

[7] 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. *MICRO 2007*, pages 249–261.

[8] 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. *Computer*, 35:50–58, 2002.

[9] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. *SIGMETRICS Perform. Eval. Rev.*, 30:108–116, 2002.

[10] 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. *HPCA 2010*.

[11] A. Patel, F. Afram, S. Chen, and K. Ghose. Marss: a full system simulator for multicore x86 cpus. *DAC 2011*, pages 1050–1055.

[12] M. Pellauer, M. Adlery, M. Kinsy, A. Parashary, and J. Emer. Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. *HPCA 2011*, pages 406–417.

[13] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic. Ramp gold: An fpga-based architecture simulator for multiprocessors. *DAC 2010*, pages 463–468.

[14] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovic, and D. Patterson. A case for fame: Fpga architecture model execution. *ISCA 2010*, pages 290–301.

[15] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang. Coremu: a scalable and portable parallel full-system emulator. *PPoPP 2011*, pages 213–222.

[16] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26:18–31, 2006.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *ISCA 1995*, pages 24–36.

[18] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev. Mptlsim: A simulator for x86 multicore processors. *DAC 2009*, pages 226–231.

APPENDIX

The appendix is organized as follows. Section A gives out the detailed data related to the motivation of the loosely-coupled Transformer framework. The data include the detailed simulation time breakdown of current tightly-coupled simulator design and detailed rate of the factors that would diverge the execution of FM and TM. Section B discusses detailed design of MAT, which is used to efficiently record and check shared data access order violation as discussed in section 3.1. Finally, to further validate the performance of Transformer, section C demonstrates the speedup of sequential and parallel Transformer under the 8-core configuration.

A. MOTIVATION OF TRANSFORMER

This section illustrates detailed data to support the motivation of the loosely-coupled Transformer framework. All these data are evaluated for each benchmark under two different configurations: a 4-core configuration shown in section 4.1 and an 8-core configuration shown in section A.1. We first give out the breakdown of the simulation time in current tightly-coupled simulator design to support that complex interaction produces additional overhead and FM occupies a proportion of the whole execution time that cannot be ignored. Then we demonstrate the detailed rate of four FM and TM path divergence factors, which support that divergences rarely happen and thus loosely-coupled design could achieve simpler interaction.

A.1 8-core Configuration

This section gives the detailed 8-core configuration, which is used to further validate our evaluation results. The detailed configuration is shown in Table 4. Different to the 4-core configuration, the L2 cache is 8M size with 25 cycles latency and the pipeline is simpler: reorder buffer, instruction window and load-store queue are of half size.

Table 4: 8-core OoO SPARC configuration.

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 8MB cache
Reorder buffer size	64		8-way set associative
Instruction window siz	32		64B cache lines
Load-store queue	32		25 cycle latency
Branch predictor	yags predictor	Memory	200 cycle latency
Data speculation	no	Cache coherence	MOESI_CMP_directory

A.2 Simulation Time Breakdown

This section evaluates the detailed simulation time breakdown of current tightly-coupled simulator design for each benchmark under 4-core and 8-core configuration. As the data shown in Figure 6, to support TM, the proportion of FM occupies about 10% of the whole time on average for both 4-core and 8-core configuration, which cannot be ignored any more. For interaction time, it occupies about 26% under 4-core configuration and 17% under 8-core configuration on average. Though the interaction percent decreases for 8-core configuration as TM occupies more, the percent is still significant to slowdown the whole performance.

A.3 Rate of Path Divergence Factors

First we illustrate the rate of the four factors for path divergence to validate that they actually occur rarely, thus motivating our loosely-coupled Transformer design with simple interaction.

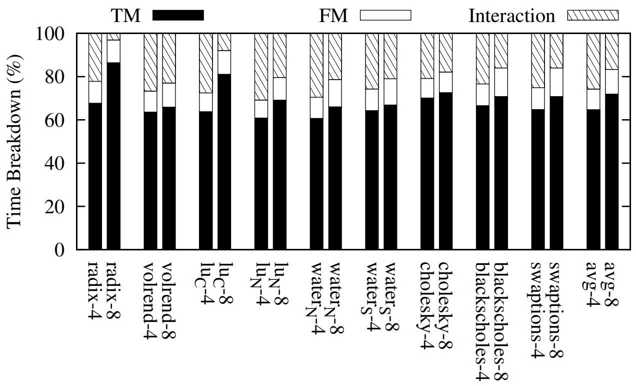

Figure 6: Detailed simulation time breakdown.

Then we further show the I/O rate which involves FM blocking in the "roll back FM" path correcting strategy to validate that it indeed has little influence. All these rate values are shown for each benchmark under both 4-core and 8-core configuration.

Branch Misprediction Rate.

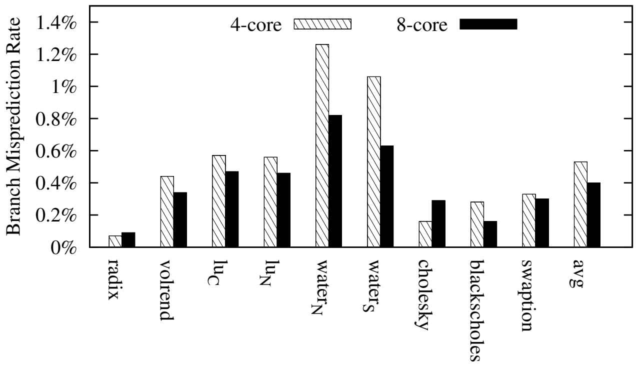

Figure 7: Branch misprediction rate.

Figure 7 shows the branch misprediction rate for each benchmark. As the data shown, the rate of the most occurring branch misprediction factor is only about 0.53% and 0.40% on average under 4-core and 8-core configuration respectively.

Interrupt/Exception Rate.

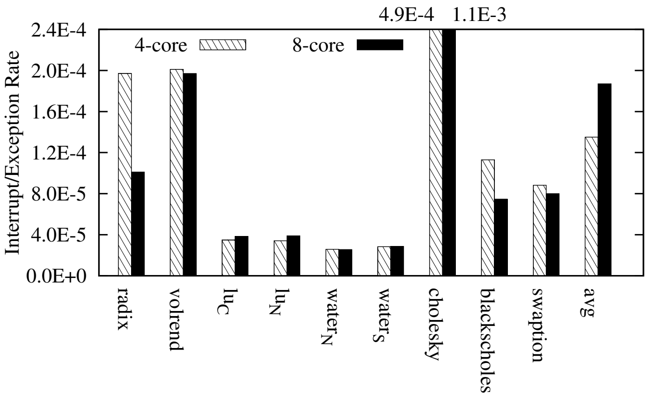

Figure 8: Interrupt/exception rate.

Figure 8 shows the total interrupt and exception rate for each benchmark. As the data shown, the average rate of interrupt 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's less than 0.2%.

Shared Data Access Order Violation Rate.

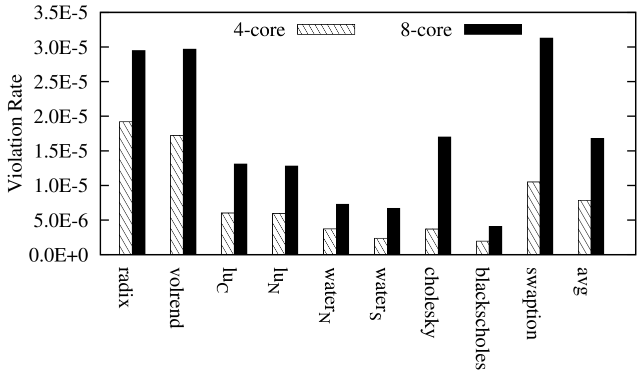

Figure 9: Shared data access order violation rate.

Figure 9 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.

MMU Miss Rate.

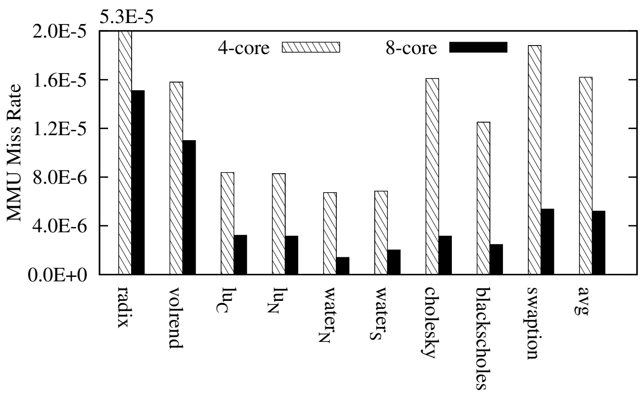

Figure 10: MMU miss rate.

Figure 10 shows MMU miss rate for each benchmark. As the data shown, 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's only about 5.3E-5.

I/O Rate.

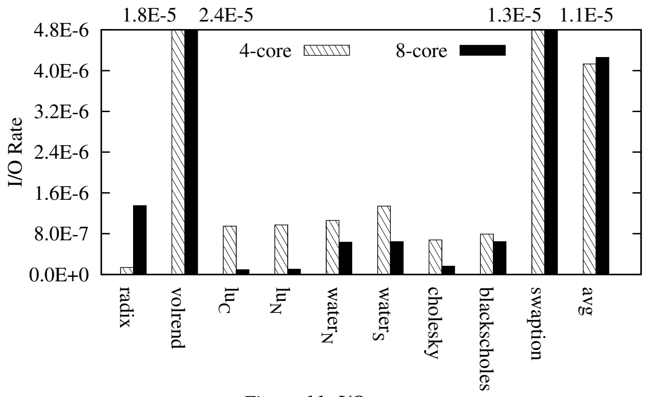

Figure 11: I/O rate.

Figure 11 shows I/O operation rate for each benchmark. As shown, the average rate of I/O operation is only about 4.1E-6 and 4.3E-6 under 4-core and 8-core configuration respectively. Even for the benchmarks (volrend and swaption) with more I/O operations, the percent is less than 2.5E-5.

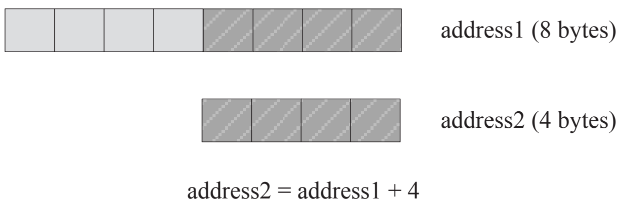

Figure 12: Interleaved cases with different memory address.

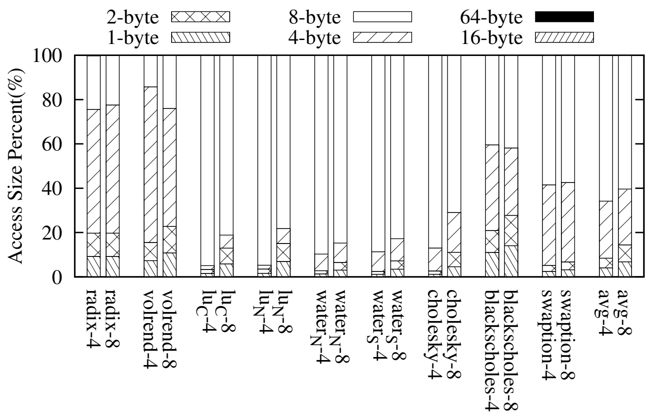

Figure 13: Proportion of each address size.

B. MAT DESIGN

There are two more detailed issues in MAT design, which is used to efficiently record and check shared data access order violation. First, we need to detect access order violation for shared memory access with different addresses. Second, to reduce memory access address list search time in MAT, we hash the memory address.

First, memory accesses with different addresses might access interleaved shared data. As shown in Figure 12, addr1 (access size: 8 bytes) and addr2 (access size: 4 bytes) access 4 bytes shared data. To detect this interleaving order violation, we can divide each memory access into several memory accesses where each accesses only one byte memory. However, this solution is too time-consuming for recording and checking.

Instead, we find most RISC ISAs align memory address (for CISC ISAs, we can divide an unaligned address into two aligned addresses), i.e., n-byte access address is n-byte aligned and it provides us with another opportunity. We have profiled the memory access address and find that more than 99.9% of memory access size is smaller than 8 bytes. The data are shown in Figure 13. Therefore, using 8-byte aligned address can fix in most addresses to avoid dividing the address.

- For memory addresses accessing not larger than 8-byte data, we fixed it into an 8-byte aligned memory address slot and using a 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.

Using this solution, we can efficiently detect all interleaved shared data accesses and detect whether there is order violation.

The second issue is to use hash to reduce memory access address list search time in MAT. The memory address is hashed (e.g., addr mod m) to be an entry of m-entry address array. If two addresses are hashed into the same array entry, they are maintained using a list.

Using these two techniques, to record or check/delete a memory access node, we first map the address into one or more 8-byte aligned addresses. Then for each address, we find the hashed address array entry and search the corresponding address list (much smaller than the design without hash) to find the address in MAT. Finally we record or check the memory access node in the found memory access node list.

C. SPEEDUP UNDER 8-CORE CONFIGURATION

This section illustrates the speedup of sequential and parallel Transformer against the baseline GEMS simulator under the 8-core configuration. The speedup data are shown in Figure 14 and Figure 15 respectively. As the data shown, the average speedup of sequential Transformer under 8-core configuration is about 7.0% while the average speedup of parallel Transformer is about 29.7%. They are smaller that of 4-core configuration because TM under the 8-core configuration occupies more proportion of the execution time, as discussed in section A.2.

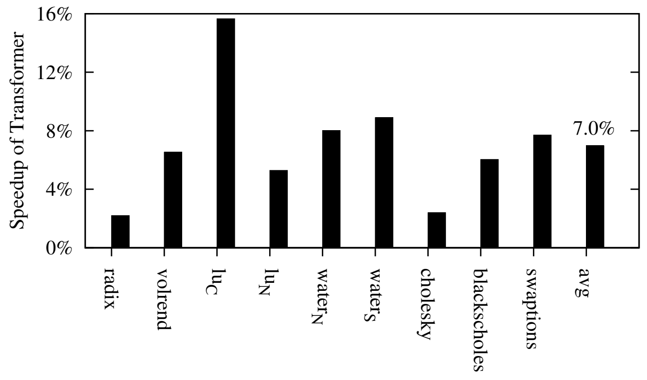

Figure 14: Speedup of sequential Transformer under 8-core configuration.

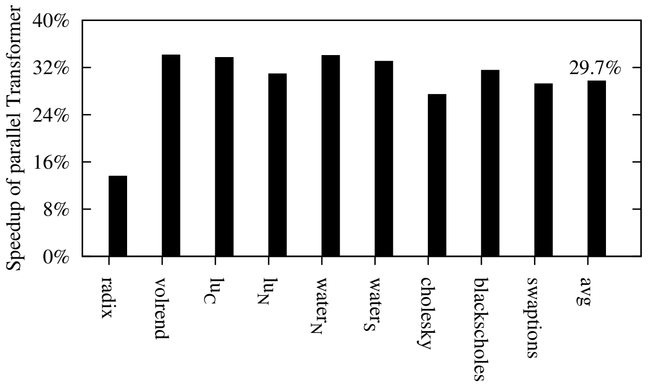

Figure 15: Speedup of parallel Transformer under 8-core configuration.