Received November 21, 2018, accepted December 9, 2018, date of publication December 17, 2018,
date of current version January 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2886630

Epsim: A Scalable and Parallel Marssx86 Simulator With Exploiting Epoch-Based Execution

MINSEONG KIM[1], (Member, IEEE), CHANHYUN PARK[1], (Member, IEEE), MISEON HAN[1], (Member, IEEE), YOUNGSUN HAN[2], (Member, IEEE), AND SEON WOOK KIM[1], (Senior Member, IEEE)
[1]Department of Electrical and Computer Engineering, Korea University, Seoul 02841, South Korea
[2]Department of Electronic Engineering, Kyungil University, Gyeongsangbuk-do 38428, South Korea

Corresponding author: Seon Wook Kim (seon@korea.ac.kr)

ABSTRACT In general, a detailed modeling and evaluation of computer architectures make a cycle-accurate simulator necessary. As the architectures become increasingly complex for parallel, cloud, and neural computing, nowadays, the complexity of the simulator grows rapidly, and thus its execution is too slow or infeasible for practical use. In order to alleviate the problem, many previous studies have focused on reducing the simulation time in a variety of ways such as using sampling methods, adding hardware accelerators, and so on. In this paper, we propose a new parallel simulation framework, called Epoch-based Parallel SIMulator, to obtain scalable speedup with large number of cores. The framework is based on a well-known cycle-accurate full-system simulator, MARSSx86. From the simulator, we build an epoch, that is an execution interval, where the architectural simulation by PTLSim does not involve any interaction with QEMU. Therefore, we can simulate epochs independently, i.e., execute multiple epochs completely in parallel by PTLSim with their live-in data. Our performance evaluation shows that we achieve 12.8× speed on average with 16-core parallel simulation from the SPEC CPU2006 benchmarks and the PARSEC benchmarks, providing the performance scalability.

INDEX TERMS Architectural simulation, epoch-based execution, parallel simulation.

I. INTRODUCTION

Currently, parallel, cloud, and neural computing for big data analysis are ubiquitously emerging technologies around us. For providing fast processing in the analysis, the computing platforms consist of the various state-of-the-art components such as multiple CPUs and GPUs, data and computing accelerators, deep memory hierarchies, large scale memory and I/Os, fast interconnects, and so on. For example, 2∼8 cores and even more than 60 cores per chip [1]–[3] and GPUs with 640 Tensor cores [4] are commercially available, and the data analysis accelerators are widely adopted [5]–[8]. In the memory system, eDRAM and stacked-DRAM based L4 cache [9]–[11] and large-scale 3D-stacked DRAMs [12]–[14] are used. Also, multiple nodes are connected through high-speed links with remote direct memory access (RDMA) [4], [15]–[18].

These platforms become more and more complicated while big data is being generated every day and everyone is connected to each other through internet [19], [20]. The development of the platforms needs to model and evaluate their performance in detail, thus makes a cycle-accurate simulator necessary [21]–[25]. However, the detailed modeling is very costly in development and, what is worse, the cycle-accurate simulation is too time-consuming and infeasible for practical use. Currently, the problem becomes worse since the data size to be simulated grows tremendously everyday [26].

There have been many promising studies for reducing the simulation time by various ways. For example, SimPoints [27] simulates only representative code sections by analyzing frequently executed sequences of basic blocks. FAST [28] is a hardware-assisted scheme that runs a functional model in software while running a timing model using field programmable gate arrays (FPGAs) together. Sniper [29] and IntervalSim [30] use an abstraction model to simulate core performance without the detailed tracking of individual instructions [31], allowing for trading off simulation speed for accuracy. Transformer [32] and P-Mambo [33] use multiple threads and assign them onto multiple cores for the simulation acceleration. Some of the threads perform functional emulation, and the others do cycle-accurate simulation with their interactions and synchronizations.

In this paper, we propose the MARSSx86-based parallel simulation framework, called EPSim (Epoch-based Parallel SIMulator), to exploit large-scale parallelism. We carry out a comprehensive study about MARSSx86, and we find that its two major components, QEMU for emulation and PTLSim for simulation, are tightly integrated like Simics+Gems [32], [34]. We identify their interactions in detail, and separate them for an independent PTLSim simulation by providing necessary data, i.e., live-in data to each PTLSim execution either online or offline. We call the independent simulation interval as *an epoch* where there is no interaction between these two components, thus we can execute each epoch independently, i.e., perform epoch-by-epoch simulation completely in parallel. The live-in data is provided by our Epoch Snapshot Creator (ESC) inside QEMU.

By characterizing the SPEC CPU2006 benchmarks [35], we show that the epoch length is large enough, thus we can obtain the scalable performance with large number of cores by alleviating the parallelization overhead such as the live-in data handling and epoch execution management. Also, for the independent simulation with guaranteeing its accuracy, we apply techniques to warm architectural states in cache, branch predictor and TLB. We verify the correctness against the existing MARSSx86 and measure the performance by executing single and multi-threaded workload onto single and multicore target environments. From the measurement, we show that we achieve up to 13.9× speedup and 12.8× speedup on average with 16 cores, and demonstrate the performance scalability. *To the best of our knowledge, our simulation framework, EPSim, is the first completely parallel simulator to provide the large performance scalability.*

The remainder of this paper is organized as follows: Section II explains MARSSx86 that is the basis for the proposed framework, and Section III describes our research motivation. Section IV describes the implementation of the proposed framework in detail with optimization techniques, and Section V evaluates its performance. Section VI discusses related work, before our conclusions are presented in Section VII.

II. MARSSx86

Fig. 1 shows an overall architecture of MARSSx86 that is a base framework of our EPSim. MARSSx86 is a micro-architectural and system simulator for evaluating and developing ×86 ISA-based platforms, which consists of tightly coupled two components: One is QEMU for full system emulation and the other is PTLSim for cycle-accurate simulation [23], [32], [34], [37].

QEMU can emulate a variety of guest applications on guest operating systems by adopting various types of devices from CPUs to network interface cards (NIC) [38]. QEMU translates the guest's instructions into host's instructions using a code generator and executes the translated code on the host machine at near native speed. The cycle-accurate

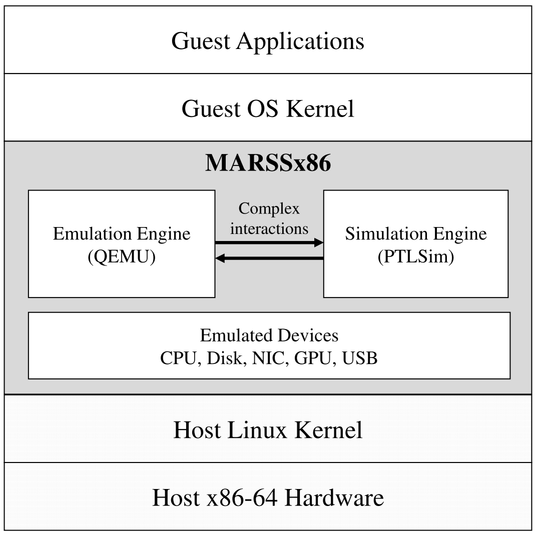

FIGURE 1. MARSSx86 simulator framework [23], [36].

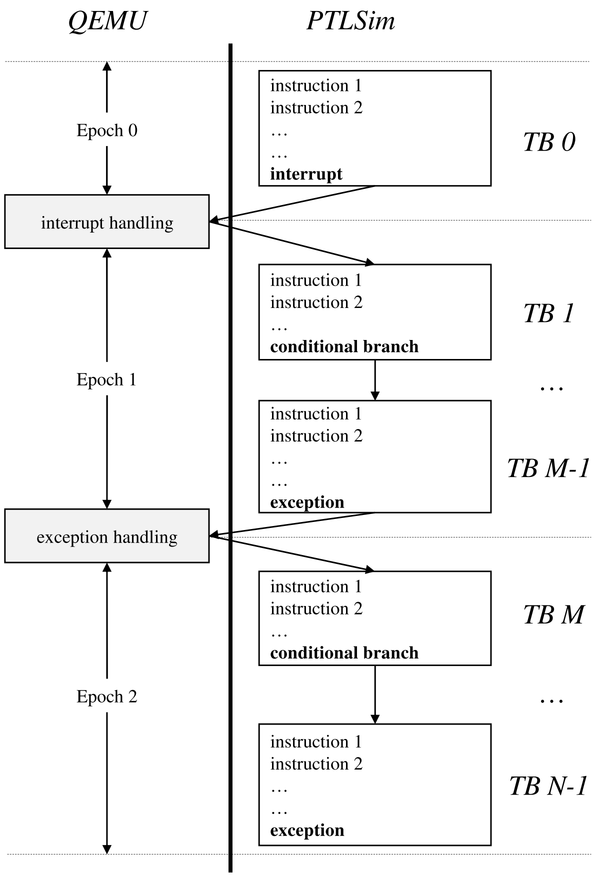

FIGURE 2. Simulation interactions and epochs in EPSim based on interrupt and exception instructions.

simulator PTLSim models out-of-order ×86-based computing platforms in detail [24], and it supports various hardware configurations such as single/multicore, deep memory hierarchy, cache coherence protocols, hardware TLB, branch predictors, peripherals and so on.

Fig. 2 presents simulation interactions between QEMU and PTLSim in MARSSx86. The instructions in a basic block of

TABLE 1. A list of epoch boundary instructions.

Instruction	Operation
sysret	functionally equivalent to sysexit
sysenter/sysexit	fast system call and return from fast system call
cpuid	CPU identification
iret_x/iret	interrupt return
ljmp	jump depending on a value of an access right byte of a descriptor
cli/sti	clear and set an interrupt flag
rdmsr/wrmsr	read and write from/to a model-specific register
rdtsc	read a time-stamp counter
hlt	halt
lldt/ltr	load a local descriptor table or a task register
monitor	set up a monitor address
mwait	monitor wait
vmrun	run a virtual machine
vmcall	call a virtual machine monitor
vmload/vmsave	load and store state from/to a virtual machine control block
stgi/clgi	set and clear a global interrupt flag
skinit	secure init and jump with attestation
invlpga	invalidate TLB entry in a specified address space identifier
lmsw	load a machine status word
invlpg	invalidate TLB entry
fxsave	save x87 FPU, MMX technology, and SSE state
fxrstor	restore x87 FPU, MMX, XMM, and MXCSR state
mov to/from CR/DR/TR	move to/from special registers (CR=control registers, DR=debug registers, TR=test registers)

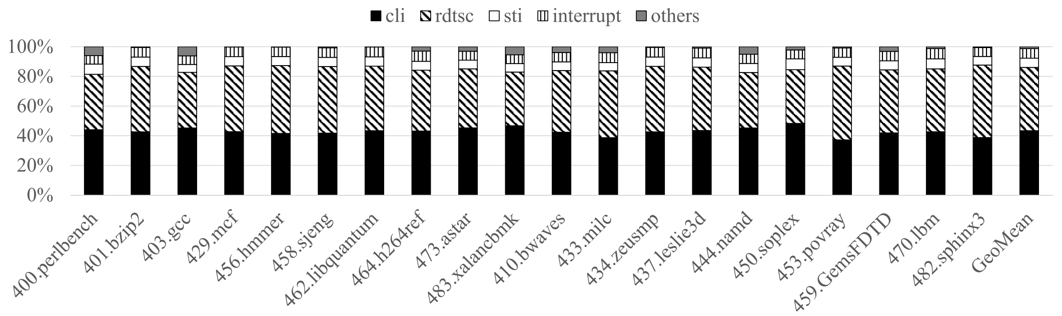

FIGURE 3. Profile of epoch boundary instructions in the SPEC CPU2006 benchmarks.

the guest binary are translated into ×86-like microcodes in another form of a basic block, called a translation block (TB) by QEMU, and the microcodes are simulated by PTLSim. Some instructions, however, cannot be handled only by PTLSim and they are shown in Table 1. We call them as epoch boundary instructions. Once PTLSim encounters such instructions during the simulation, an internal exception is triggered for switching from simulation mode to emulation one, emulated by QEMU's helper functions, and switched back to the simulation mode. For example, in Fig. 2, when PTLSim commits an interrupt instruction of *TB 0*, it calls a related QEMU helper function and provides its CPU state to QEMU so that QEMU can perform the interrupt handling routine. After QEMU handles the interrupt, PTLSim receives the updated CPU state from QEMU and continues the simulation from the next TB, i.e., *TB 1*. QEMU and PTLSim share architectural states for correct execution, and the related overhead accounts for up to 35% of the total simulation time [32]. As a result, such a tightly coupled design for supporting full system simulation slows down the simulation speed [22], [32].

III. MOTIVATION FOR PARALLELIZING MARSSx86

In this section, we present our research motivation for developing EPSim by characterizing an epoch as a parallel simulation unit and calculating the ideal speedup by epoch-by-epoch parallel execution.

We define an epoch as a sequence of instructions that does not involve any interaction between QEMU and PTLSim. Therefore, we can execute each epoch in parallel by PTLSim. For example, in Fig. 2, *epoch 0* is defined as the same as *TB 0*, and *epoch 1* is comprised of multiple translation blocks, i.e., from *TB 1* to *TB M-1*.

In order to characterize the epoch, we profiled the epoch boundary instructions and the epoch length from the SPEC CPU2006 benchmarks [35], and they are shown in Fig. 3 and Fig. 4, respectively. The `cli` and `rdtsc` instructions account for more than three quarters of the epoch

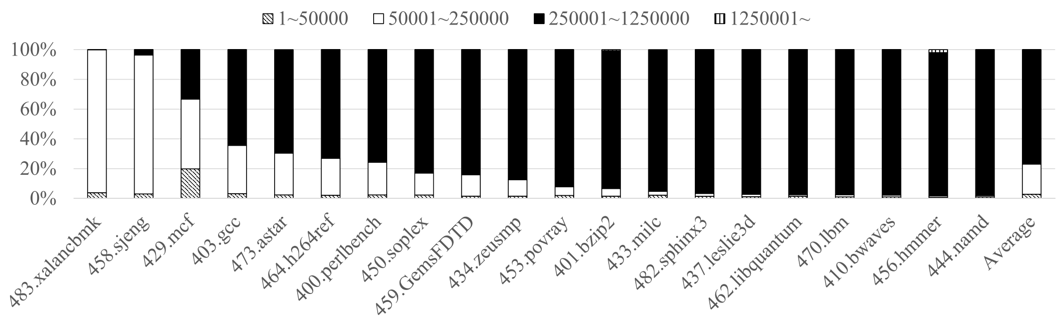

FIGURE 4. Breakdown of the epoch lengths in the SPEC CPU2006 benchmarks.

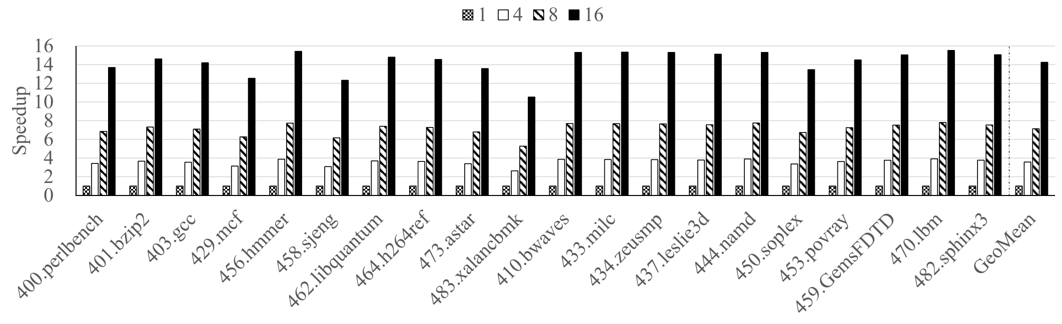

FIGURE 5. Upperbound speedup derived from Fig. 4.

boundary instructions in all benchmarks, i.e., 43% and 42%, respectively. The `interrupt` and `sti` instructions approximately occupies 6% and 6%, repectively, and the others do negligibly.

At the epoch-by-epoch parallel execution, the epoch length should be large enough in order to minimize the parallelization overhead. We broke down the epoch length in ranges of 50,000, 250,000, and 1,250,000 instructions by assuming that we can ignore the overhead when the epoch length is more than 50,000. The epochs whose sizes are more than 50,000 occupy 97.1% of the total instructions on average, and the epochs whose sizes are even more than 250,000 do 76.8%. More specifically, in the case of *444.namd*, 98.2% of instructions are in epochs whose lengths are greater than 250,000 instructions. In *459.GemsFDTD*, 14.3% and 84.2% of instructions belong to epochs with 50,000∼250,000 instructions and greater than 250,000 instructions, respectively. In *483.xalancbmk*, 96.2% of the total instructions are contained within epochs with more than 50,000 instructions. We can conclude that the epoch sizes are large enough for the epoch-based parallel execution.

Fig. 5 shows the ideal speedup that is derived from Fig. 4 by assigning one epoch to one core and ignoring the parallelization overheads with 1, 4, 8 and 16 cores. We obtained the ideal speedup of 1.0× with 1 core, 3.6× with 4 cores, 7.1× with 8 cores, and 14.3× with 16 cores, thus we argue again that the epoch-based parallel execution provides great potential for the simulation acceleration.

IV. EPSim: PARALLEL AND SCALABLE SIMULATION FRAMEWORK

In this section, we describe our proposed simulation framework. We briefly introduce the overall architecture and present major components with performance optimization techniques in detail.

A. OVERALL ARCHITECTURE

Fig. 6 depicts an overall architecture of EPSim where we replace the tightly coupled interaction between QEMU and PTLSim in Fig. 1 with epoch snapshots, generated by *Epoch Snapshot Creator*. The snapshots are necessary live-in data for the epochs' independent execution by PTLSim. The epoch-based parallel execution is scheduled for minimizing load imbalance on multicore and their results are combined at the end of the whole simulation.

B. EPOCH SNAPSHOT AND ITS CREATOR

The Epoch Snapshot Creator (ESC) separates an epoch that consists of one or multiple translation blocks (TBs) from the sequential execution by identifying epoch boundary instructions in Table 1 and generates its snapshot. The structure of the epoch snapshot is shown in Fig. 7, which consists of several live-in components for the epoch's independent execution such as CPU state and data, memory state and data, and additional epoch information.

The CPU state and data, i.e., CPUX86States contain processor features, registers, segments, internal flags, and so on

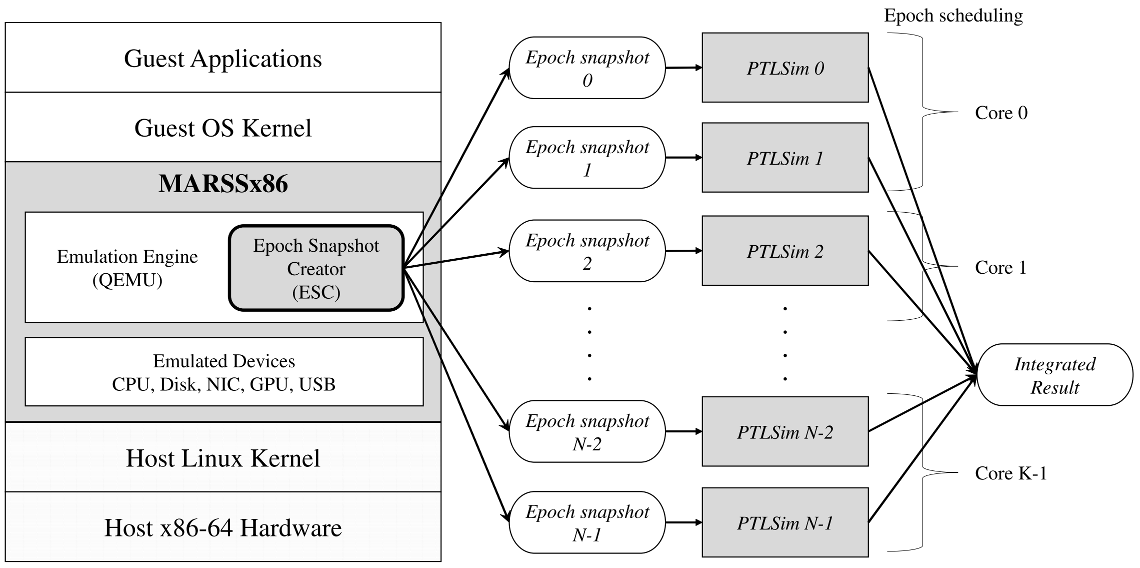

FIGURE 6. Overall architecture of EPSim.

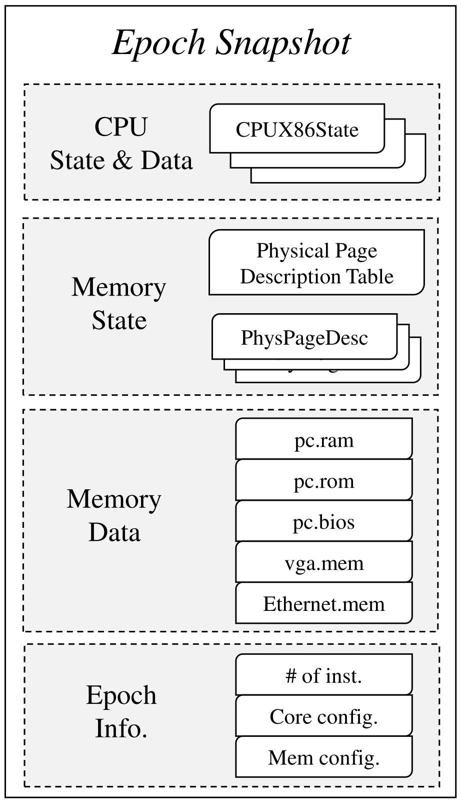

FIGURE 7. A structure of an epoch snapshot generated by ESC.

that are defined in [23]. The memory state are the internal information about memory management in QEMU such as physical page descriptions and their related tables. Also, the memory data consists of codes and data of applications and operating systems such as pc.ram, pc.rom, pc.bios, and so on. These data are stored in the snapshots at the start of an epoch emulation by ESC. Finally, the additional epoch information involves the number of executed instructions in an epoch, the number of cores, memory capacity, and the number of threads per core to deliver simulation environment and to determine the number of instructions to be simulated in PTLSim. These information are stored at the end of the epoch emulation.

C. PARALLEL SIMULATION FRAMEWORK

1) EPOCH'S INDEPENDENT EXECUTION AND CORRECTNESS

We implemented a wrapper that allows PTLSim to read the epoch snapshot instead of communicating with QEMU. The snapshot provides configuration-independent states as shown in Fig. 7 to the PTLSim simulator. However, some hardware states like in cache, branch predictor, and TLB cannot be provided statically. For example, if the hardware configuration is changed or the previous epoch is still being executed, the hardware states can be unknown at the start of the current epoch's independent execution.

In order to reduce the inaccuracy, we adopted hit-on-cold technique [39] in cache that assumes all cold misses are hit. We also applied assume-hit [40]–[42] technique that all misses caused by cold state are hit during the first few thousand cycles in cases of branch predictor and TLB. After all the epoch simulations complete, their results are combined into one.

2) MEMORY OPTIMIZATION FOR MANAGING EPOCH SNAPSHOTS

It is severely time and space consuming to store all the data to be used by PTLSim for every epoch. For example, if the simulated target machine is modeled to have 4 GB main memory and 1,000 epochs are simulated, we require 4,000 GB of storage for all the epoch snapshots. Therefore, we applied

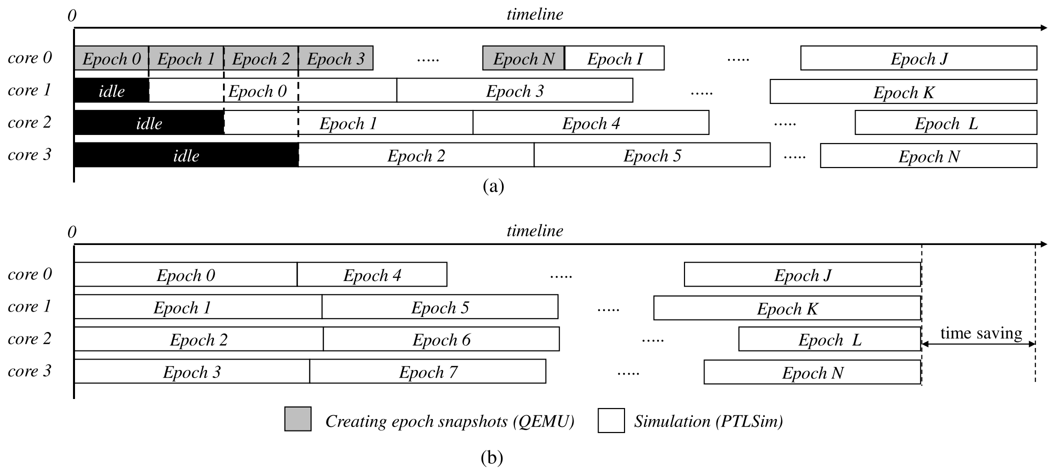

FIGURE 8. **Two parallel execution modes of EPSim. (a) Online. (b) Offline.**

several optimization techniques for reducing epoch snapshot sizes.

First, we only stored the used pages during the emulation to diminish storage usage. We made ESC configure a simulated target memory as a protected mode using *mprotect* command and register a handler for violation of the protected memory. The handler only records the access violated memory regions.

Second, we further reduced the memory usage by storing only dirty data instead of the whole used page. We observed that many recorded data are the same between different epochs, thus the shared data between epochs is stored only once. From the simulation, we found that the target machine with 4 GB memory needs only 17 MB storage on average for the simulation.

Finally, we used shared memory space instead of files for fast sharing epoch snapshots by the PTLSim simulators.

3) SIMULATION MODE: ONLINE VS. OFFLINE

Fig. 8 illustrates two execution modes of our simulation, i.e., online and offline. The online method is designed that epoch snapshot generation and its simulation are performed in parallel. For instance, as shown in Fig. 8 (a), as soon as *core 0* generates epoch snapshot for *epoch 0*, *core 1* starts to simulate *epoch 0*. Similarly *epoch 1* and *epoch 2* are simulated on *core 2* and *core 3* with the generated snapshots by *core 0*, respectively. If *core 0* finishes to generate all epoch snapshots, it simulate epochs like other cores.

The offline method assumes that all snapshots are available before the PTLSim simulation. The epoch snapshots can be reused regardless of alterations to architecture configuration such as target cores, cache hierarchy, and memory timing parameters. Therefore, we can eliminate the overhead of creating the epoch snapshots at every simulation and the idle time of waiting for the epoch to be simulated as shown in Fig. 8 (b).

4) EPOCH-BY-EPOCH SCHEDULING

It is very important to minimize the load imbalance in epoch-based parallel execution since each epoch's simulation by PTLSim takes very long. At the online simulation, we assign and start to simulate an epoch whenever an idle processor becomes available. At the offline simulation, we assign some epochs to be simulated onto one core in one group, and the sum of the epoch lengths in one group is the possibly same as those in other groups by using *Max-Min* algorithm [43], [44]. By using the algorithm, we can minimize the load imbalance.

V. PERFORMANCE RESULT

To evaluate the simulation accuracy and speedup of EPSim against existing MARSSx86, we used the SPEC CPU2006 benchmarks for single simulation and the PARSEC benchmarks [45]–[47] for multicore simulation with 1 billion instructions on 16 2.6 GHz Intel Xeon CPUs without hyper-threading. We also simulated a target machine that contains 4 out-of-order cores with 256 KB L1 D/I private cache, unified 2 MB shared L2 cache, 4 GB of main memory and a combined hybrid bimodal and history based branch predictor.

A. EPOCH CHARACTERISTICS

Fig. 9 depicts the average, maximum, and minimum of the epoch lengths in the SPEC CPU2006 benchmarks. The average epoch length was 0.9 million instructions, and their maximum was 5.5 million instructions in *459.GemsFDTD* benchmark.

As discussed in Section IV-C2, for storing all epoch snapshots we need a storage whose size is 4 GB $\times$ # of epochs without our storage optimization techniques, and the size is impractical for use. Fig. 10 shows the total sizes of epoch snapshots *acceptable* for use by applying our techniques: storing only used pages (FB) and only data differentiation (DIFF). The DIFF technique reduces the sizes significantly from FB, and the reduction ratio was 77.4% on average, especially 83.7% in *483.xalancbmk* and 90.5% in *458.sjeng*.

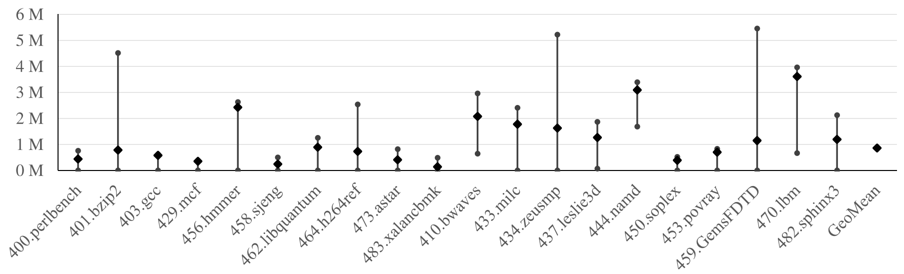

FIGURE 9. The average, maximum, and minimum of the epoch lengths in the SPEC CPU2006 benchmarks.

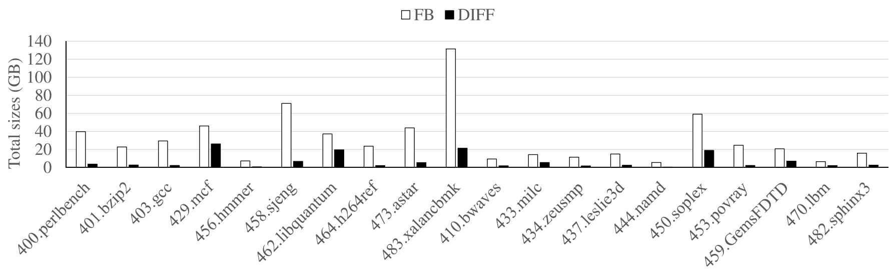

FIGURE 10. The total sizes of epoch snapshots by FB and DIFF in the SPEC CPU2006 benchmarks. FB: Store only used pages. DIFF: Store only data differentiation.

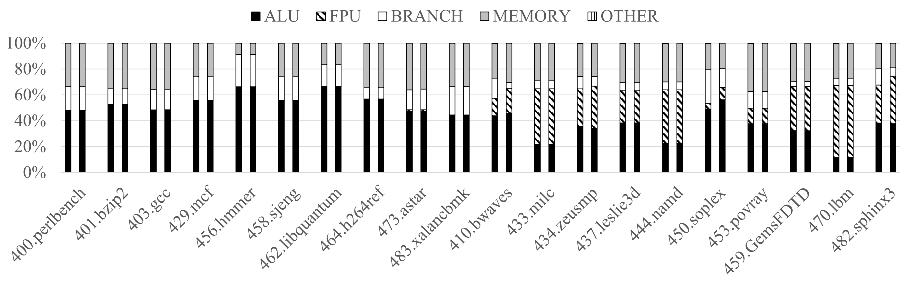

FIGURE 11. Difference of the committed instructions by instruction types between MARSSx86 (left bar) and EPSim (right bar) in the SPEC CPU2006 benchmarks.

The storage reduction is related to the performance improvement and it will be shown in Section V-C.

B. SIMULATION ACCURACY IN SINGLE-CORE SIMULATION

To validate EPSim's accuracy for single-core simulation, we measured committed instructions and architectural statistics. Fig. 11 shows the difference of the committed instructions by instruction types from EPSim and MARSSx86, and we found that the difference is negligible in all the simulated benchmarks.

More in detail, Table 2 compares the committed instructions in *400.perlbench* using the instruction classification of `uops` in MARSSx86, and the difference was less than or equal to 0.02%. The differences of the committed instructions were up to 2.40% and 0.02% on average in the SPEC CPU2006 benchmarks. The difference was not exactly zero because MARSSx86 obtains non-deterministic results by carrying out a full system simulation of OS and applications.

Fig. 12 compares such important architectural statistics from EPSim with MARSSx86 as IPC, L1 cache miss ratio, branch misprediction ratio, and data TLB miss ratio. Again, there was no noticeable difference in the performance. The average IPC difference was 1.93% with maximum of 5.57% in *434.zeusmp* benchmark, the average L1 miss ratio difference was 0.06% with maximum of 1.20% in *450.soplex* benchmark, the average branch misprediction ratio difference was 0.10%, and the average DTLB miss ratio difference was 0.03%.

Table 3 shows more detailed architectural statistics of *400.perlbench*, including the total number of cycles, total number of committed ×86 instructions, IPC, L1 cache miss ratio, branch misprediction ratio of conditional branches, and data TLB (DTLB) miss ratio. The differences of cycles, IPC and L1 cache miss ratio were 2.11%, 2.15% and 0.05%, respectively. The total number of the committed branch instructions showed small difference of 1.49%, and its misprediction ratios were measured as 5.70% and 6.04% for MARSSx86 and EPSim, respectively. Also, The DTLB miss

TABLE 2. Comparison of the committed instructions in 400.perlbench.

instructions		MARSSx86	EPSim	%diff
type	class			
arithmetic (ALU)	addsub	450,865,978	441,942,584	0.02%
	addsubc	0	0	0.00%
	addshift	32,049,674	31,466,790	0.02%
	shiftsimple	21,729,807	21,349,037	0.02%
	shift	579,345	578,919	0.00%
	mul	9,698	9,698	0.00%
	sel	26,714,258	26,151,614	0.02%
	cmp	0	0	0.00%
	bitscan	22,937	22,937	0.00%
	flags	28,200,051	27,673,229	0.02%
	chk	60,823	60,854	0.00%
	vec	219,976	219,976	0.00%
floating-point (FPU)	fpu	451	451	0.00%
	fp-div-sqrt	52	52	0.00%
	fp-cmp	3,591	3,591	0.00%
	fp-perm	0	0	0.00%
	fp-cvt-i2f	859	859	0.00%
	fp-cvt-f2i	1,818	1,818	0.00%
	fp-cvt-f2f	0	0	0.00%
branch	br.cc	168,676,880	165,284,099	0.02%
	jmp	21,007,680	20,619,793	0.02%
	bru	30,998,944	30,369,749	0.02%
	assist	0	0	0.00%
memory	mf	143,582	143,452	0.00%
	ld	267,532,898	262,367,331	0.02%
	st	122,615,935	120,315,479	0.02%
	ld.pre	0	0	0.00%
other	special	0	0	0.00%

TABLE 3. Comparison of architectural performance statistics in 400.perlbench.

Trial	MARSSx86	EPSim	%diff
Cycles	691,327,619	676,751,577	2.11%
# of committed instructions	1,000,000,000	1,000,000,014	0.00%
IPC	1.45	1.48	2.15%
# of L1 cache misses	796,347	796,897	0.07%
# of L1 cache accesses	411,021,657	410,848,793	0.04%
L1 miss ratio	0.20%	0.25%	0.05%
# of committed branch instructions	242,962,893	239,345,799	1.49%
# of mispredicted branches	13,859,490	14,460,033	4.33%
Branch misprediction ratio	5.70%	6.04%	0.34%
# of DTLB misses	9,149,114	9,754,958	6.62%
DTLB miss ratio	1.86%	2.08%	0.22%

ratios were 1.86% and 2.08%, respectively. The differences were definitely acceptable since the simulation results cannot be precisely identical due to the non-deterministic behavior of OS and applications.

C. SIMULATION SPEEDUP IN SINGLE-CORE SIMULATION

Fig. 13 shows the speedup of online and offline method using 1, 4, 8, and 16 cores by averaging three runs of the simulation. On the online simulation, we obtained an average speedup of 0.9× with 1 core, 3.6× with 4 cores, 6.7× with 8 cores, and 12.6× with 16 cores. On the offline simulation, we achieved a little higher speedup than the online, such as 1.0× with 1 core, 3.7× with 4 cores, 6.9× with 8 cores, and 13.0× with 16 cores since the offline method simulates epochs in parallel using the generated epoch snapshots in advance. From the measurement, we assure that our approach shows large performance scalability.

Fig. 14 exhibits the speedup variants from different optimization schemes with the offline method, which were discussed in Section IV-C. FB and DIFF achieved on average 11.6× and 12.3× of speedups and it indicates 0.7× performance improvement by storing only the differences between the epoch snapshots. In particular, DIFF remarkably improved the performance of *483.xalancbmk* and *458.sjeng* benchmarks since the total sizes of their epoch snapshots were significantly reduced from 131 to 21 GB and from 71 to 7 GB, respectively, as shown in Fig. 10. DIFF+SHM obtained 12.9× speedup on average by adopting a shared memory that is 0.6× the speedup improvement. DIFF+SHM+SCHE increased the average speedup to 13.0× by enhancing load balance with *Max-Min* scheduling algorithm. The speedup improvement by SCHE is 0.1× compared with DIFF+SHM in our experiments, but if there will be more cores or simulating longer intervals, SCHE will contribute higher speedup.

D. SIMULATION ACCURACY AND SPEEDUP FOR MULTICORE SIMULATION

Fig. 15 presents the number of committed instructions and the architectural statistics from multicore simulation to prove its accuracy by using some representative benchmarks from PARSEC. The difference of the committed instructions was 0.08% on average. The average IPC difference was 5.36%, and it is greater than the single-core results by higher non-determinism of OS due to multicore scheduling.

Fig. 16 shows the speedup of multicore simulation using the offline method on 1, 4, 8, and 16 cores. We obtained an average speedup of 1.0× with 1 core, 3.6× with 4 cores, 6.5× with 8 cores, and 11.1× with 16 cores. Therefore, EPSim provides performance scalability for the multicore simulation as well as the single-core simulation.

VI. RELATED WORK

In this section, we introduce conventional methods to perform an detailed architectural simulations, i.e., timing model. We also compare and discuss about the research and development to reduce simulation time in the architectural simulations.

A. CONVENTIONAL ARCHITECTURAL SIMULATION

Several architectural simulators with timing model have been popularly used, and their representatives are GEMS [48], GEM5 [22], PTLSim, and MARSS.

GEMS is a detailed multiprocessor simulator that supports many ISAs with various CPU models. The GEMS includes two components: Opal, a detailed model of an out-of-order processor, and Ruby, a detailed model of memory system. In order to support more detailed CPU models and support more ISAs, the GEM5 simulator that combined GEMS and M5 [49] has emerged. GEMS is responsible for the detailed modeling of the memory system, and M5 is for the detailed modeling of the CPU. GEM5 also supports a full system simulator with timing model.

As mentioned in II, PTLSim is a cycle-accurate simulator that models out-of-order ×86-based computing platforms in

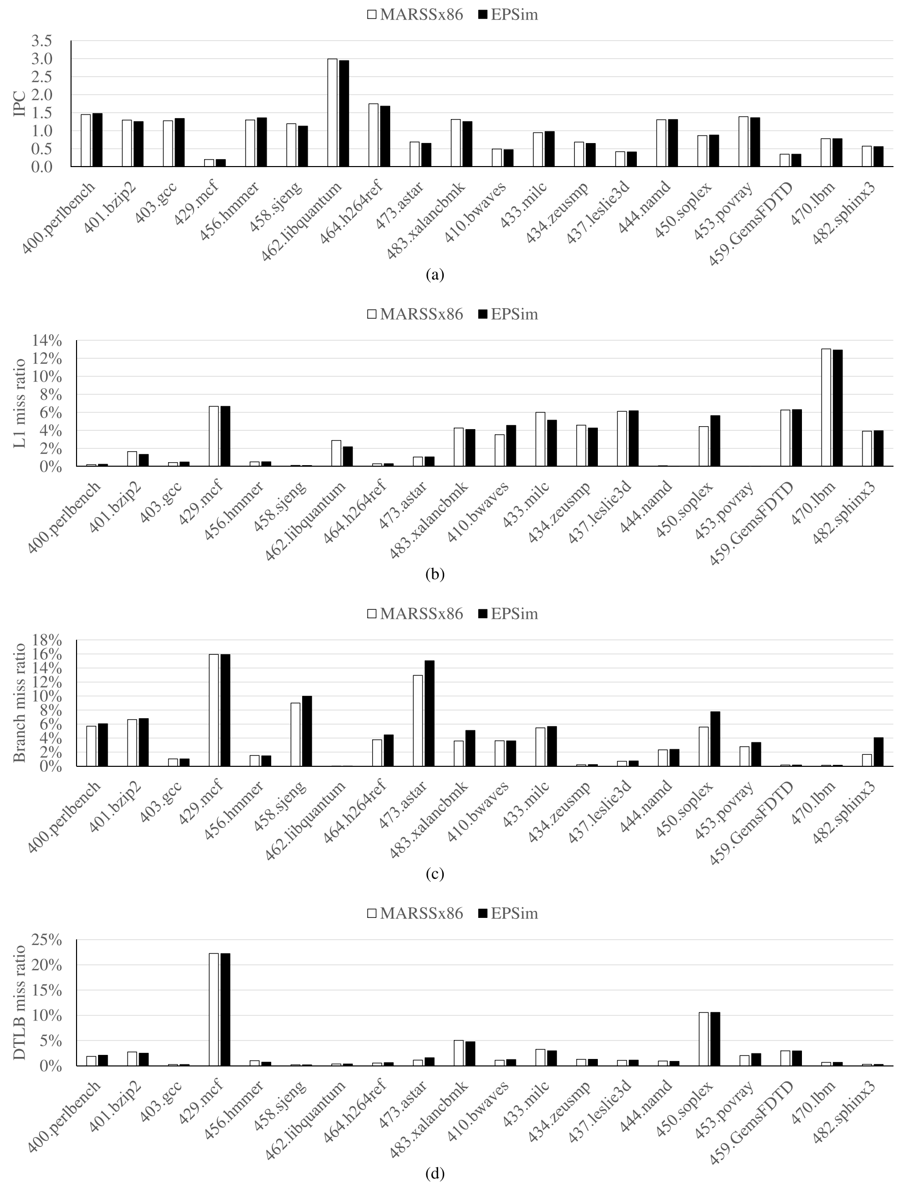

FIGURE 12. **Architectural statistics comparison in the SPEC CPU2006 benchmarks. (a) IPC. (b) L1 cache miss ratio. (c) Branch misprediction ratio. (d) DTLB miss ratio.**

detail. Also MARSS is a full system cycle-accurate simulator that is integrated with QEMU for emulation for full system support and PTLSim for simulation.

However, simulating 464.h264ref benchmarks for SPEC CPU2006 takes more than 5 months for GEM5+Ruby and MARSS [50]. Therefore, researches to reduce the vast simulation time of the architectural simulations have been carried out. In the following sections, we introduce the researches to reduce simulation time.

B. HARDWARE ACCELERATION

Many previous schemes use dedicated hardware resources to accelerate cycle-accurate simulations. FAST partitions the simulation into functional model in software and timing model in hardware FPGA, and executes the models in parallel. ProtoFlex [51], [52] is a hybrid functional simulator where common operations like ALU instructions are simulated on FPGAs whereas complex behaviors like disk I/Os are simulated in software.

Although these schemes achieve remarkable speedup due to the hardware acceleration, they require extra special-purpose hardware, i.e., FPGAs and great effort for the hardware implementation. However, our work does not require hardware resource and implementation.

C. HIGH-ABSTRACTION-LEVEL MODELING

High-abstraction-level modeling simulators attempt to find the midpoint of detailed cycle-accurate simulators and

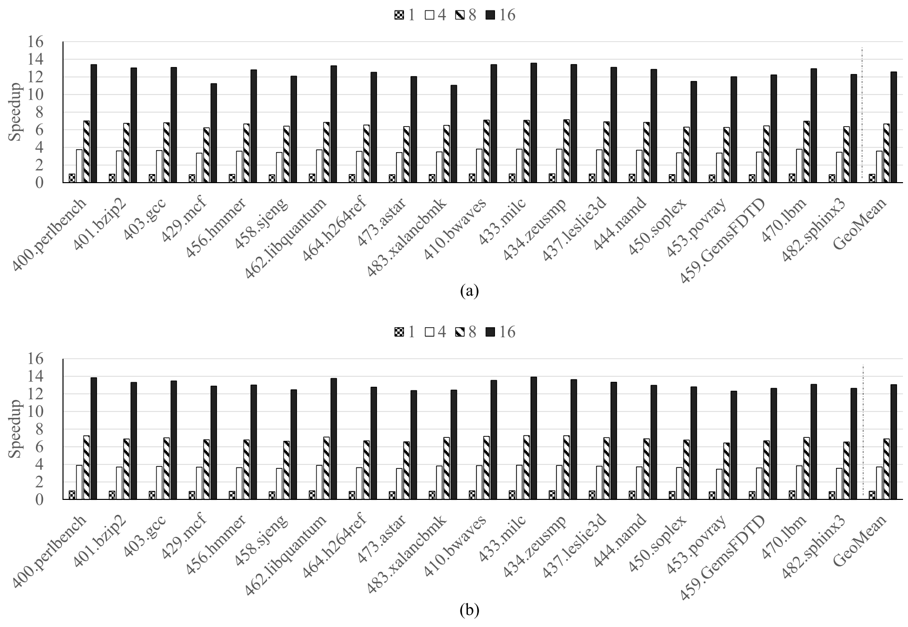

FIGURE 13. Speedup of online and offline methods on different numbers of cores in the SPEC CPU2006 benchmarks. (a) Online. (b) Offline.

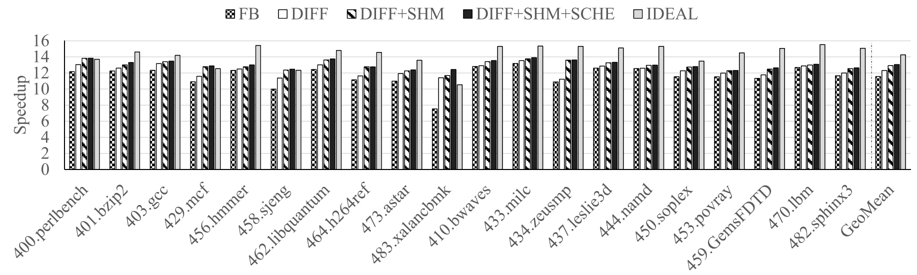

FIGURE 14. Speedup variants from different optimization schemes with the offline method in the SPEC CPU2006 benchmarks. FB: Store only used pages. DIFF: Store only data differentiation. : Use shared memory instead of files. SCHE: Use Max-Min scheduling.

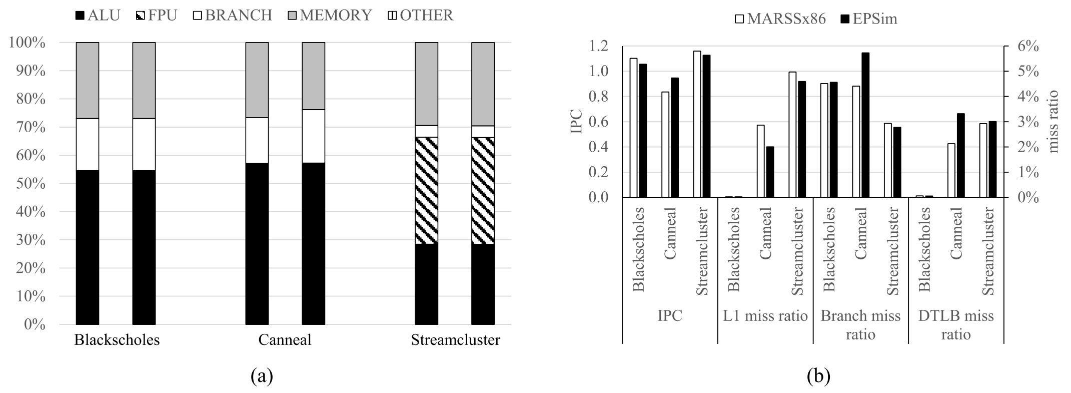

FIGURE 15. Difference of the committed instructions by instruction types and architectural statistics in the multicore simulation between MARSSx86 (left bar) and EPSim (right bar) in the PARSEC benchmarks. (a) Committed instructions. (b) Architectural statistics.

one-IPC simulators that simulate only cores' IPC by trading off accuracy and speed. Sniper [29] and IntervalSim [30] are high-abstraction-level modeling simulators that use interval simulation to balance this trade-off point. These simulators use miss events (branch misprediction, cache and TLB misses) to define the intervals, and adopt an analytical model

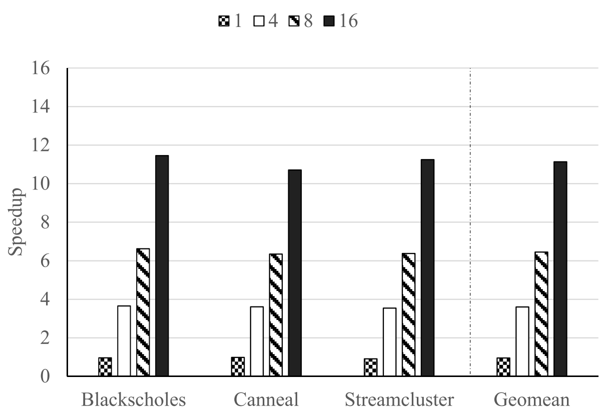

FIGURE 16. Performance of the multicore simulation on 1, 4, 8, and 16 cores in the PARSEC benchmarks.

to predict the core performance without any detailed simulation of the core's pipeline stages. Our scheme parallelizes a detailed cycle-accurate simulator that simulates each core's pipeline stage in detail with epoch-by-epoch simulation, thus we obtain various and accurate performance metrics.

D. SAMPLING-BASED SIMULATION

Sampling-based simulators conduct detailed cycle-accurate simulations about only a few simulation points in order to reduce the simulation time. SMARTS [53] selects the simulation points periodically using a systematic sampling technique that utilizes several sampling variables such as size, mean, coefficient of variation, confidence level, confidence interval, systematic-sampling interval, and so on. SimPoints [27] uses Basic Block Vectors (BBVs) representing the frequencies of executed sequences of basic blocks during a certain interval to identify the same phases of a running application. If two BBVs are similar, then SimPoints assumes that the performance between those two intervals is similar, thus skips the cycle accurate simulation.

DAPs [54] dynamically performs multithreaded simulations at runtime by adjusting the sample length and sampling frequency. If a synchronization event changes the behavior of a multithreaded application, the simulator adjusts the sampling parameters to increase the accuracy of the simulation and reduce time.

Although these simulators offer significant reductions in simulation time, their results are less accurate because of the statistical sampling about a few simulation points. Our simulator simulates all the epochs cycle-accurately, thus it does not lose any accuracy.

E. PARALLELIZING FUNCTIONAL AND TIMING SIMULATIONS

Many simulators have a tightly coupled design between the functional and timing models like MARSSx86 [23]. However, the back and forth execution from their interaction results in significant overhead in the simulation time. Transformer proposes a loosely coupled functional-driven full-system simulator where the interaction is postponed as late as possible for hiding the interaction latency.

However, in our scheme, the functional model and timing model are completely separated by the epochs, i.e. eliminates the interactions, thus the highly performance scalability can be achieved.

F. MULTICORE AND MULTITHREADING SIMULATION

There are active researches to adapt multithreading to architectural simulators on multicore platforms.

P-Mambo [33] is a multi-threaded implementation of Mambo [55] that is the IBM's full system simulator to model PowerPC systems. SlackSim [56] assigns each core of target chip multiprocessors (CMP) onto one thread across several machines and parallelizes the execution by providing some simulation time slack to reduce the synchronization overhead. Graphite [57] is a distributed parallel simulator infrastructure that is designed for many-core processors containing dozens to thousands of cores. It assigns each core tile of the target architecture onto one thread and spreads the threads across multiple machines for parallel execution. BigSim [58] uses a performance prediction model of a target machine and MPI to use large-scale parallel machines.

These simulators parallelize the simulation by assigning target cores onto running platforms, thus the synchronization overhead becomes significant when the number of cores on the platforms increase. The overhead may significantly reduce the performance scalability.

ZSim [59] leverages dynamic binary translation to speed up the sequential simulation and uses two steps of parallelization adaptation for simulating thousands of cores system. First, the simulation is performed in intervals of a few thousand cycles to examine whether there is an interference or not. After that, the simulation continues to be proceeded in parallel with assuming no interference at all. Thus, the interference limits the performance of ZSim.

Prophet [60] is a parallel instruction-oriented simulator that divides a target program into instruction-based intervals for their parallel execution on multicores. They adopt a speculative simulation model to avoid unnecessary interaction between private and shared resources (e.g., private and shared caches). Therefore, they need to update the architectural states after the execution of the intervals. However, the simulator does not have any handler to compensate for the accuracy loss at a miss-speculation.

VII. CONCLUSION

In this paper, we proposed a scalable and parallel cycle-accurate full-system simulator that is derived from MARSSx86. We removed the interaction between QEMU and PTLSim, and defined an epoch to be simulated independently by PTLSim. For the correct simulation, we developed the Epoch Snapshot Creator to provide necessary data to each independent execution of the epochs, and we optimized the data storage size and its management. By minimizing the overhead and executing the epochs completely in parallel, we could achieve the performance scalability. From the comprehensive performance study, we showed that we obtained 12.8× speedup on 16-core execution.

Currently, we perform two researches as future work: One is to reduce the simulation time and the other is to migrate our simulator to the Hadoop infrastructure [61]. The first research is the similar to SimPoint [27]. If some epochs show the similar architectural behavior, we may simulate only one of them, instead of executing all of them. By sampling epochs, we expect that we can significantly reduce the simulation time. Also, we believe that the Hadoop migration will be able to conduct more simulations in shorter time and help architecture researchers and developers to model more aggressive architectures, thus make the cycle accurate simulators practical for use.

ACKNOWLEDGMENT

This work was partly supported by the IT R&D program of MOTIE/KEIT [10052653, Development of processing in memory architecture and parallel processing for data bounding applications] and the SK hynix Inc. research project.

REFERENCES

[1] A. Sodani *et al.*, "Knights landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[2] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2013.

[3] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Berkely, CA, USA: Apress, 2013.

[4] NVIDIA. (2017). *NVIDIA Tesla v100*. [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-v100/

[5] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, pp. 1–19, Mar. 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1603.html#AbadiABBCCCDDDG16

[6] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 2017, pp. 1–12, doi: 10.1145/3079856.3080246.

[7] A. Subramaniyan and R. Das, "Parallel automata processor," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 2017, pp. 600–612, doi: 10.1145/3079856.3080207.

[8] Micron. (2017). *Micron Automata Processing*. [Online]. Available: http://www.micronautomata.com/

[9] G. M. Ung. (2015). *Intel Core i7-5775C Review: The Unwanted Desktop Broadwell Has One Neat Trick*. [Online]. Available: http://www.pcworld.com

[10] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2014, pp. 25–37, doi: 10.1109/MICRO.2014.51.

[11] V. Young, P. J. Nair, and M. K. Qureshi, "DICE: Compressing dram caches for bandwidth and capacity," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 2017, pp. 627–638, doi: 10.1145/3079856.3080243.

[12] D. U. Lee *et al.*, "25.2 A 1.2 V 8 GB 8-channel 128 GB/s high-bandwidth memory (HBM) stacked dram with effective microbump I/O test methods using 29 nm process and TSV," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, Feb. 2014, pp. 432–433.

[13] D. U. Lee *et al.*, "A 1.2 V 8 Gb 8-channel 128 GB/s high-bandwidth memory (HBM) stacked DRAM with effective I/O test circuits," *IEEE J. Solid-State Circuits*, vol. 50, no. 1, pp. 191–203, Jan. 2015.

[14] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2011, pp. 1–24.

[15] G. F. Pfister, "An introduction to the infiniband architecture," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, vol. 42. New York, NY, USA: Wiley, 2001, pp. 617–632.

[16] T. Reuters, "Mellanox infiniband accelerates the Exegy ticker plant at major exchanges," Mellanox Technol., Sunnyvale, CA, USA, Tech. Rep. 1, 2008.

[17] NVIDIA. (2017). *NVLink Hish-Speed Interconnect: Designed for Accelerated Computing*. [Online]. Available: http://www.nvidia.com/object/nvlink.html

[18] R. Nishtala *et al.*, "Scaling memcache at facebook," presented at the 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI), Lombard, IL, USA, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[19] J. Wang, M. Qiu, and B. Guo, "Enabling real-time information service on telehealth system over cloud-based big data platform," *J. Syst. Archit.*, vol. 72, pp. 69–79, Jan. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762116300364

[20] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *J. Syst. Archit.*, vol. 72, pp. 93–107, Jan. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762116300662

[21] 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*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[22] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, doi: 10.1145/2024716.2024718.

[23] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2011, pp. 1050–1055.

[24] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2007, pp. 23–34.

[25] E. Naroska, F. Lai, R.-J. Shang, and U. Schwiegelshohn, "Efficient parallel timing simulation of synchronous models on networks of workstations," *J. Syst. Archit.*, vol. 47, no. 6, pp. 517–528, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762101000236

[26] Z. Bian *et al.*, "Simulating big data clusters for system planning, evaluation, and optimization," in *Proc. 43rd Int. Conf. Parallel Process. (ICPP)*, Sep. 2014, pp. 391–400.

[27] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA, 2002, pp. 45–57, doi: 10.1145/605397.605403.

[28] D. Chiou *et al.*, "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 249–261.

[29] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. SC*, Nov. 2011, pp. 1–12.

[30] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proc. 16th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Jan. 2010, pp. 1–12.

[31] Sniper. (2012). *Interval Simulation*. [Online]. Available: http://snipersim.org/w/Interval_Simulation

[32] Z. Fang *et al.*, "Transformer: A functional-driven cycle-accurate multicore simulator," in *Proc. 49th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2012, pp. 106–114.

[33] K. Wang, Y. Zhang, H. Wang, and X. Shen, "Parallelization of IBM mambo system simulator in functional modes," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 1, pp. 71–76, Jan. 2008, doi: 10.1145/1341312.1341325.

[34] W. Zhang, H. Wang, Y. Lu, H. Chen, and W. Zhao, "A loosely-coupled full-system multicore simulation framework," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1566–1578, Jun. 2016.

[35] M. Kainaga, K. Yamada, and H. Inayoshi, "Analysis of spec benchmark programs," in *Proc. 8th TRON Project Symp.*, Nov. 1991, pp. 208–215.

[36] A. Patel, F. Afram, S. Chen, and K. Ghose. (2017). *Marssx86*. [Online]. Available: http://marss86.org

[37] S.-H. Kang, D. Yoo, and S. Ha, "TQSIM: A fast cycle-approximate processor simulator based on QEMU," *J. Syst. Archit.*, vols. 66–67, pp. 33–47, May 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762116300297

[38] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATEC)*, Berkeley, CA, USA, 2005, p. 41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[39] M. Van Biesbrouck, L. Eeckhout, and B. Calder, *Efficient Sampling Startup for Sampled Processor Simulation*. Berlin, Gemany: Springer, 2005, pp. 47–67, doi: 10.1007/11587514_5.

[40] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Trans. Comput.*, vol. 43, no. 6, pp. 664–675, Jun. 1994.

[41] D. A. Wood, M. D. Hill, and R. E. Kessler, "A model for estimating trace-sample miss ratios," in *Proc. ACM SIGMETRICS Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, New York, NY, USA, 1991, pp. 79–89, doi: 10.1145/107971.107981.

[42] D. A. Wood, M. D. Hill, and R. E. Kessler, "A model for estimating trace-sample miss ratios," *SIGMETRICS Perform. Eval. Rev.*, vol. 19, no. 1, pp. 79–89, Apr. 1991, doi: 10.1145/107972.107981.

[43] K. Etminani and M. Naghibzadeh, "A min-min max-min selective algorihtm for grid task scheduling," in *Proc. 3rd IEEE/IFIP Int. Conf. Central Asia Internet*, Sep. 2007, pp. 1–7.

[44] X. Li, Y. Mao, X. Xiao, and Y. Zhuang, "An improved max-min task-scheduling algorithm for elastic cloud," in *Proc. Int. Symp. Comput., Consum. Control*, Jun. 2014, pp. 340–343.

[45] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *Proc. 5th Annu. Workshop Modeling, Benchmarking Simulation*, Jun. 2009, pp. 1–9.

[46] 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. (PACT)*, New York, NY, USA, 2008, pp. 72–81, doi: 10.1145/1454115.1454128.

[47] C. Bienia and K. Li, "Characteristics of workloads using the pipeline programming model," in *Proc. Int. Conf. Comput. Archit. (ISCA)*. Berlin, Germany: Springer-Verlag, 2012, pp. 161–171, doi: 10.1007/978-3-642-24322-6_14.

[48] D. T. Phillips, M. Handwerker, and G. L. Hogg, "GEMS: A generalized manufacturing simulator," *Comput., Ind. Eng.*, vol. 3, no. 3, pp. 225–233, 1979. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0360835279900160

[49] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006.

[50] D. Kim, C. Celio, D. Biancolin, J. Bachrach, and K. Asanovic, "Evaluation of RISC-V RTL with FPGA-accelerated simulation," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, 2017.

[51] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "PROToFLEX: FPGA-accelerated hybrid functional simulator," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–6.

[52] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "PROToFLEX: Towards scalable, full-system multiprocessor simulations using fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 15:1–15:32, Jun. 2009, doi: 10.1145/1534916.1534925.

[53] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. 30th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 2003, pp. 84–97, doi: 10.1145/859618.859629.

[54] C.-C. Chen *et al.*, "DAPs: Dynamic adjustment and partial sampling for multithreaded/multicore simulation," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2014, pp. 1–6.

[55] P. Bohrer *et al.*, "Mambo: A full system simulator for the PowerPC architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, Mar. 2004, doi: 10.1145/1054907.1054910.

[56] J. Chen, M. Annavaram, and M. Dubois, "SlackSim: A platform for parallel simulations of CMPs on CMPs," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 20–29, Jul. 2009, doi: 10.1145/1577129.1577134.

[57] J. E. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, Jan. 2010, pp. 1–12.

[58] G. Zheng, G. Kakulapati, and L. V. Kale, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, Apr. 2004, p. 78.

[59] D. Sánchez and C. E. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 475, 2013.

[60] W. Zhang, X. Ji, Y. Lu, H. Wang, H. Chen, and P. Yew, "Prophet: A parallel instruction-oriented many-core simulator," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2939–2952, Oct. 2017.

[61] Apache Software Foundation. (2017). *Welcome to Apache Hadoop*. [Online]. Available: http://hadoop.apache.org/

MINSEONG KIM received the B.S. degree in electrical engineering and the Ph.D. degree in electronics, electrical and computer engineering Korea University, in 2011 and 2018, respectively. He is currently with SK Hynix, South Korea. His research interests include compiler construction, microarchitecture, memory systems, and parallel and distributed architectural simulator designs.

CHANHYUN PARK received the B.E. degree in electrical engineering from Korea University, Seoul, South Korea, in 2013, where he is currently pursuing the Ph.D. degree. His research interests include compiler support, microarchitecture, and regular expression.

MISEON HAN received the B.E. degree in electrical engineering and the Ph.D. degree in electronics, electrical and computer engineering from Korea University, Seoul, South Korea, in 2012 and 2018, respectively. She is currently with SK Hynix, South Korea. Her research interests include compiler support, microarchitecture, and memory designs.

YOUNGSUN HAN received the B.E. and Ph.D. degrees from the School of Engineering, Korea University, Seoul, South Korea, in 2003 and 2009, respectively. He was a Senior Engineer with the Samsung LSI, from 2009 to 2011. He is currently an Assistant Professor with the Department of Electronic Engineering, Kyungil University. His research interests include embedded systems, compiler construction, high-performance computing, and SoC design.

SEON WOOK KIM received the B.S. degree in electronics and computer engineering from Korea University, in 1988, the M.S. degree in electrical engineering from The Ohio State University, in 1990, and the Ph.D. degree in electrical and computer engineering from Purdue University, in 2001. He was a Senior Researcher with the Agency for Defense Development, from 1990 to 1995, and a Staff Software Engineer with Inter/KSL, from 2001 to 2002. He is currently a Professor with the School of Electrical and Computer Engineering, Korea University. His research interests include compiler construction, microarchitecture, system optimization, and SoC design. He is a Senior Member of the ACM.