MRP: Mix Real Cores and Pseudo Cores for FPGA-based Chip-multiprocessor Simulation

Xinke Chen [1,2], Guangfei Zhang[3], Huandong Wang[4], Ruiyang Wu[1,2], Peng Wu[1,2], Longbing Zhang[1]
[1]State Key Laboratory of Computer Architecture, ICT, CAS, Beijing 100190, China
[2]University of Chinese Acedemic of Science, Beijing 100094, China
[3]Shannon Laboratory, Huawei Technologies Co., Ltd
[4]Loongson Corporation, Beijing 100190, China
{chenxinke, wuruiyang, wupeng, lbzhang}@ict.ac.cn, zhangguangfei@huawei.com, wanghuandong@loongson.cn

***Abstract*—Facing the speed bottleneck of software-based simulators, FPGA-based simulation has been explored more and more. This paper proposes a novel methodology to simulate a chip-multiprocessor (CMP) on the limited FPGA resource. By mixing real cores and pseudo cores together (MRP), we can simulate a multicore system with fewer FPGA resource requirements and achieve a much higher simulation speed. We propose several methods to construct the pseudo cores. We implement our idea on a dual Virtex-6 FPGA board to simulate a general-purpose 4-core high performance CMP processor. Comparison experiments against the corresponding tape-out chip prove the effectiveness of MRP. We also evaluate MRP prototype's performance by running SPEC CPU2006 benchmarks on an unmodified Linux operating system, achieving tens to hundreds speedup compared to two other commonly-used simulators.**

Keywords—Simulation, Emulation, CMP, Multicore, Pseudo core, FPGA

1. Introduction

In the exploration of computer architecture, it's hard to weigh the pros and cons of new design ideas and architecture parameters by doing models. The complicated interaction of processor microarchitecture makes it impossible to predict chip performance by theoretical derivation. Simulation is the only choice in design phase. An ideal simulation is to run the exactly same programs of target application to evaluate the performance of an undertaken design. Good simulators are: (i) fast, fast enough to run some sorts of real benchmarks, (ii) accurate, accurately predicting the metrics being measured, (iii) low cost, including not only the hardware investment, but also the engineer efforts needed to develop and use the simulators, (iv) transparent, providing enough visibility into the simulated system for debugging and analysis.

Industrial and academic architects traditionally use software-based simulators, like Simple-scalar, SimOS, gem5, etc. [1, 2, 3] While software simulators have the advantages of transparency, relatively low cost and cycle-accuracy, they have a fatal disadvantage that is its low speed. Typically, software simulators run at several hundred Hz to 100 KHz at which speed it is impractical to run real programs. After the processor architecture evolved into multicore era, the speed gap is further enlarged for two reasons: one is the increased simulation complexity, and the other is that software simulators do not benefit much from the parallelized architecture provided by modern CMP.

Facing the speed bottleneck of software-based simulators, more and more researchers turned to explore the use of FPGA (Field Programmable Gate Array) to accelerate the multicore simulation [4]. Modern FPGA is already capable of running at tens to hundreds of MHz clock frequency. And following Moore's Law, the capacity of FPGA also increases geometrically. Currently, one high volume FPGA is capable of mapping a full moderate-sized processor core. However, the FPGA capacity is still the main constraint to prototype a large scale design, like a CMP.

Some EDA companies provide commercial hardware-accelerated simulation platform that provides automated RTL synthesis and implementation tool flow and also toolset for graphic display and probing, like EVE Zebu [5], Cadence Palladium, etc., which are usually named as emulators. Emulators are accurate, transparent, and relatively fast, usually running at 1~2 MHz, but they are very expensive, costing hundreds of thousands to millions of US dollars.

Trace-based simulation [6] has been used to study the performance of specific components like branch predictor, cache and memory controller. Trace-based simulator is easy-to-use and usually fast. Unfortunately, trace-based simulation does not capture the feedback among the various components of the whole system. So it is more suitable to be used for components simulation other than full system simulation.

In this paper, we propose MRP multicore simulation methodology: mixing real cores and pseudo cores together for FPGA-based multicore simulation. MPR can overcome the FPGA capacity constraints effectively and at the same time alleviate the weaknesses of pure trace-based simulation.

We propose and implement several types of pseudo cores for MRP and validate their effectiveness by comparison experiments against the corresponding tap-out ASIC chip. The experimental results show that MRP has a relative error rate of about 8% which is the fidelity to the simulated design not the uncertainty of the method, but achieves a speedup of about 800 over a popular software simulator. Though the accuracy is sacrificed a little, the huge speed advantage enables us to run the real world programs and do a thorough study by exploring various hardware configurations which is also important for an architecture exploration.

This work is supported by the National Sci&Tech Major Project (No. 2009ZX01028-002-003, 2009ZX01029-001-003, 2012ZX01029-001-002-002), National Natural Science Foundation of China (No. 61432016, 61221062, 61100163, 61133004, 61173001, 61232009, 61222204), the National High Technology Development 863 Program of China (2012AA010901, 2012AA011002, 2012AA012202, 2013AA014301)

This paper is organized as follows. Section 1 summarizes the pros and cons of several different types of simulations and introduces our method. Section 2 further explains the MRP method, and Section 3 gives the detailed MRP implementation and validation. Section 4 evaluates the performance of MRP. Then section 5 introduces the most related work. Finally, we discuss and conclude this paper in section 6.

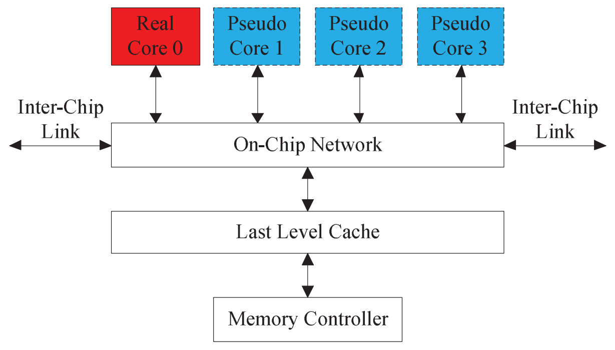


Figure 1. An instance of MRP simulation method

2. MRP Multicore Simulation Method

Figure 1 illustrates an instance of MRP where a four-core CMP system is simulated. The red block with real border represents the real core and the blue blocks with dashed border represent the pseudo cores we add in.

Why fake a core?

FPGA-based prototype has long been used to validate the correctness of a new designed processor core before final tape-out. To deal with the capacity constraints of FPGA, it is easy to come up with the idea that trimming the original multicore system into a single core system. Engineers usually remove the identical components, like the multiple processor cores, multi-banked caches, etc., and preserve only one copy of them. While this method is useful for the verification of core architecture, it is insufficient for the validation of the correctness and performance of the whole multicore system. Because the shared resources in a multicore architecture, like on-chip network, last level cache and memory controller, etc., usually expose different behaviors under single-core and multicore workloads. Thus, it is not accurate to predict their performance by doing a uniprocessor simulation.

In a CMP system, the equipped multiple cores are usually identical to each other, thus it is a huge waste to map all of them into the limited FPGA resource. And the interface of processor core is mainly used for data transfer (the coherence messages can also be treated as data). Base on this observation, we come up with the idea: why don't we keep one (or a few) real core (s) and at the same time use some pseudo cores to replace the rest identical cores to fake a multicore environment? The pseudo cores do not necessarily fulfill the real core's functions, and all they need to do is generating memory accesses that seem like coming from real cores.

Because the pseudo core does not have the complex superscalar out of order pipeline and ALU/FPU components (even the private cache can be removed or at least be simplified), its area cost will be trivial. So, many pseudo cores can be added into the *target* system but at the same time does not increase the FPGA resource requirements significantly. The decreased resource requirements not only reduce the engineer efforts needed for prototyping but also make it possible to map the whole CMP system into one or a few FPGAs, which in turn increases the simulation speed that the *host* system can achieve (by convention, we use the term *target* to mean the system being simulated and the term *host* to mean the system that runs the simulator).

Why does faked multicore environment work?

In a CMP system, the interference among multiple cores mainly comes from the shared resources: shared on-chip network, shared last level cache, and shared memory controller, etc., and is usually irrelevant directly to the inner status of each individual core. From the perspective of the shared components, the requests they received are generated from a real core or a pseudo core makes no difference. So our faked multicore environment is capable of studying the interactions between these shared components and the processor cores. However, as some real cores are preserved, we can run real programs and study how they interact in CMP environment which is different from pure trace-based simulation.

For example, if our goal is to verify the correctness of a directory-based cache coherence protocol, we can fake a private L1 cache which behaves exactly the same as a real L1 cache when dealing with cache coherence messages. But the real data are not needed, and only the tag and directory bits need to be stored. For a typical cache organization with 64-byte cache line size, the area cost of a faked cache is about only one eighth of the real one. Stimulus can be added to the faked cache to generate requests to the shared address space in order to study how the coherence protocol works.

For the study of on-chip network and memory controller, as their tasks are mainly on data routing and data access, it makes no difference that which master sends the requests. So MRP is quite suitable to study the behaviors of them under multicore pressure.

How to fake a core?

If a faked multicore environment is capable of simulating the CMP behavior, then comes with the key question: **how to fake a core?** The interface of a processor core is mainly used for data transfer. To fake a program run is to fake the requests the core would generate if the program was actually executed.

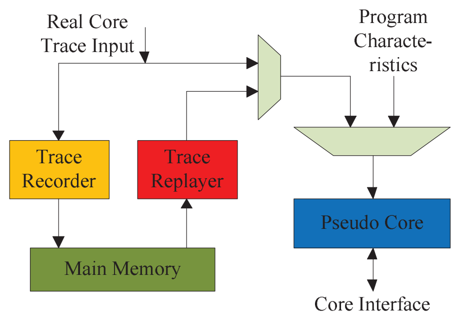


Figure 2. Structure of pseudo cores

Figure 2 shows the generalized structure of our pseudo core. In this figure, three types of pseudo core are shown. Two types of pseudo core receive real core's trace and generate requests similar to the reference trace. The reference trace can either come from the real core's real-time trace or come from the stored offline trace of the real core. The third type of pseudo core does not rely on trace input. It generates memory accesses that obey predefined patterns directly according to configurable parameters.

For homogeneous workload, it is natural to come up with the idea that one can feed the real core's interface signals into the pseudo cores, and with some simple transformations, let the pseudo cores generate requests with the same characteristic of the ones generated by the real core. This strategy is used to simulate the scenario in which the multiple cores are running the same application simultaneously.

For heterogeneous workloads, the problem becomes more complicated as we cannot use the real core's real-time trace directly. To deal with this problem, we present two methods in this paper. The first one is that one can gather the most important microarchitecture feathers, like L1 cache miss intervals, read/write ratio, address pattern, etc., and then configure the pseudo cores to generate accesses that are statistically equal with these characteristics. This strategy is applicable to simulate some programs that have monotonic behavior, like a *stream*. However, for programs that have multiple execution stages and irregular memory access characteristics, this scheme may not work.

To deal with the challenge of simulating heterogeneous workloads, a "trace recording and replaying" mechanism is proposed. Our idea is like this: when simulating a mixture of different programs, we can run the to be imitated program on the real core and record its trace first, then we replay this trace through the pseudo core and at the same time run the target program on the real core to get its performance metrics, like execution time, IPC, etc. By replaying the trace, we "rerun" the program approximately. While the previously recorded trace is replaying, the real core's trace can also be recorded for use in next iteration. This method can be used to simulate any combination of heterogeneous workloads.

3. MRP IMPLEMENTATION AND VALIDATION

3.1. Validation Methodology

To validate the effectiveness of our MRP method, we implement a four-core CMP simulator on a dual Vertex-6 verification board and use the corresponding tape-out chip as the standard comparison system. We configure the hardware and software parameters of the two systems as close as possible and use the same benchmark to measure the relevance between the MRP system and the real chip. We define relevance as the degree of performance fidelity of MRP compared to the original simulated CMP system.

Our comparison chip is a four-core symmetric multicore processor whose architecture is the same to that illustrated in Figure 1. The hardware parameters are listed in Table 1.

The same RTL source code of the tape-out chip is used to develop the MRP simulator on FPGA platform. The MRP simulator has the same architecture as the real chip except that the real cores 1~3 are replaced with the pseudo cores and the last level cache shrinks to 512KB. To make it equivalent to the FPGA platform, the L2 cache of the real chip is configured to use only one of the four banks which is 1MB.

TABLE 1. TARGET SYSTEM HARDWARE PARAMETERS

Microarchitecture	4-issue out-of-order superscalar, 10-stage pipeline
L1 Cache of Each Core	Private, 64K 4-way set associative I-Cache and 64K 4-way set associative D-Cache
Victim Cache	Private, 128K 8-way set associative
Last Level Cache (L2 cache)	Shared, 4M 4-way set associative, 4 Bank 1M each
On-Chip Network	8x8 Crossbar
Off-Chip DRAM	DDR3 UDIMM, 2GB, 1Rank

3.2. Design and Implementation

In this section, we give our design of the pseudo core and the trace recording and replaying engines in detail. The hardware platform is also introduced briefly.

For pseudo core that uses the real core's trace as its input, it transforms the core ID, destination address to the pseudo core's own version and then sends the faked requests to the on-chip network. For example, if the real core 0 sends a read request to address A, then the pseudo core 1 will also send a read request but to address A' which is located in the pre-allocated address space for pseudo core 1. For write requests, the situation is a little more complicated. As our cache hierarchy uses inclusion strategy between L1 and L2 caches, the pseudo cores cannot write arbitrary address into the next level L2 cache. So a faked L1 cache is added in the pseudo core. The faked L1 cache has a similar RAM organization as the real L1 cache except that it does not store data segment. If a write request occurs, the write budget for the corresponding cache set is increased by one. And the faked L1 cache records every read address (tag) it has sent out and if a cache line conflict occurs, some old record must be replaced. If the current write budget is not zero, the replacement translates into a write request and the write budget is decreased, else no write request is generated. The faked L1 cache is also in charge of dealing with the cache coherence requests issued by the next level L2 cache. For an invalidation request, the faked L1 cache simply removes the corresponding record in the RAM. And for a write-back request, the faked L1 cache writes all 0 to the next level L2 cache without touching the content of RAM. When the real core's real-time trace is feed into the pseudo core, the pseudo core behaves the same as the real core. We call this type of pseudo core as "mirror core".

As to pseudo cores that generate requests by predefined patterns, we currently only implement a simple prototype. Our faked core sends memory requests at fixed time interval with linearly increasing address. Both the time interval and address range can be configured by parameters. We name this type of pseudo core as "*stream* core".

For our "trace recording and replaying" mechanism, main memory is used to store the trace. The trace recorder collects, compresses the real core's trace and writes the trace into main memory. The trace replayer reads trace out of main memory, decompresses and feeds it into the pseudo cores. The recorded trace includes the read/write type, address and timestamp. The timestamp records the number of clock cycles between two consecutive accesses. The trace recorder and replayer access main memory directly, bypassing the last level cache. In our design of the trace recorder, we use two buffers to store the trace data temporarily. When one buffer becomes full, the other buffer is switched on to store the incoming trace continuously, and the last buffer starts dumping all the data into main memory in sequential and burst mode. The trace replayer works in a similar way with opposite operations. Currently, we set the buffer size at 1KB. As the SDRAM memory system prefers sequential access, this manner of burst read/write can minimize the influence of the trace recording and replaying to the original system. When the pseudo core replays the recorded trace, it repeats the process of the last program running. We note this type of pseudo core as "trace core".

We use the TAI Logic Module (LM) from S2C Corporation as our FPGA board. The LM provides two Xilinx Virtex-6 FPGAs to the user as programmable logic and also provides standard DDR2/3 SODIMM interfaces to connect DIMMs. Our custom board provides North/South bridges and IO interfaces. As the whole CMP is too big to fit into a single Virtex-6, we partition the original design into two parts. The real core is put in one of the FPGAs (U2) alone, and the rest parts are put in the other FPGA (U1). The U1 and U2 are connected by shared IO pins. The pseudo cores are also put in the U1 side to reduce the number of interconnection pins. When we select one type of pseudo core for a specific simulation, the corresponding pseudo cores are added into the FPGA. The FPGA resource utilization is given in Table 2.

TABLE 2. FPGA RESOURCE UTILIZATION

FPGA Resource Utilization	Registers	Lookup Tables	BlockRAMs
U2	6%	38%	21%
U1 without pseudo cores	11%	27%	22%
U1 with three pseudo "mirror cores"	13%	31%	26%
U1 with three pseudo "*stream* cores"	13%	32%	25%
U1 with one pseudo "trace core"	13%	34%	26%

Currently, we do not do any performance optimization specific for FPGA implementation, and the RTL code is seldom modified for FPGA synthesis except the removal of some un-synthesizable modules and the replacement of some customized RAMs. We achieve 50MHz clock frequency of core clock domain which includes the processor core, private caches, on-chip network and last level cache. As the core to DDR frequency ratio of the real chip is about 2:1, we run the memory controller of the FPGA platform at 25MHz to make the simulation system and the real chip have the same core to DDR frequency ratio.

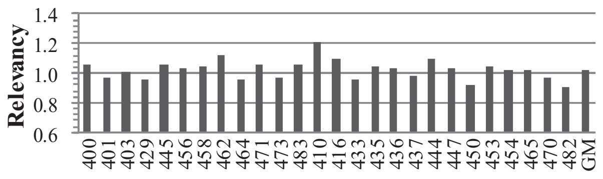
3.3. Experimental Setup and Results

To validate the MRP, we boot the unmodified Linux 2.6.36 kernel, and use the standard CPU performance evaluation benchmark SPEC CPU2006 (use SPEC for short after) and memory intensive program *stream* as our benchmarks. The *stream* benchmark is a simple program that measures sustainable memory bandwidth [7]. The SPEC benchmarks are compiled using GCC 4.4.3. To complement the relatively low frequency of FPGA board, the OS clock interrupt frequency is decreased compared with that of the real chip system.

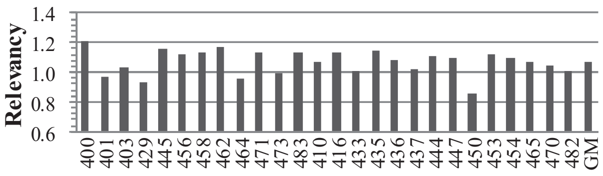
For both the real chip and MRP platforms, the single core and multicore workloads are executed separately. For the MRP platform, the real core is used to run the SPEC program, and its execution time is used as the final performance metric. We divide the SPEC execution time in multicore environment by the time in the single-core execution to calculate the slowdown of the multicore workloads. Then we calculate the ratio of the FPGA system's slowdown to the real chip system's slowdown to measure the relevance between the two platforms. The closer to 1 the ratio is, the more similar the two systems are, in other words, the more effective MRP is.

3.3.1 Simulate Homogeneous Workload

To simulate the execution of homogeneous workload, "mirror cores" are added into the MRP platform. We evaluate the relevance between MRP and real chip system on 2-core and 4-core SPEC workloads separately. The experimental results are shown in Figure 3. We use the benchmark number to stand for each of the SPEC benchmark tests for X axis, and the Y axis shows the relevance of each benchmark. The "GM" is short for geometric mean which is used to calculate the average score. Figure 4 and 5 use the same notion.



a) Mix one SPEC process with one "mirror core"



b) Mix one SPEC process with three "mirror cores"

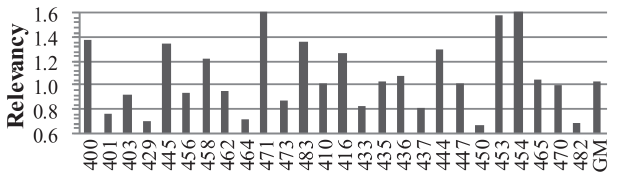
Figure 3. Relevance between the MRP system and the real chip system for homogeneous workload

Figures 3(a) and (b) show that the relevance score of the two systems is very close to 1 when using MRP to simulate a multicore homogeneous workload. Most of the SPEC benchmarks have a relevance score between 0.9 and 1.2, and the averaged relevance score of all the benchmarks on 2-core and 4-core simulation reaches 1.02 and 1.07 respectively. The standard deviations of all the relevance scores are 6.4% and 7.4% respectively. This degree of variance proves that manipulating the real core's online trace to simulate a pseudo core is very effective for the simulation where multiple cores are running the same application.

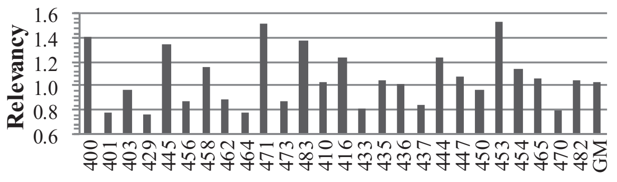
3.3.2 Simulate Heterogeneous Workloads

1) Use program statistical characteristics to fake a core

To fake a *stream* process, the statistical method is used. First, we use OProfile [8] count the averaged L1 miss interval, read/write ratio of the *stream* program when it runs alone on the real chip. Then the pseudo core is configured to generate memory accesses with the same characteristic. We also evaluate the scenarios where one or three pseudo "*stream* cores" are added into MRP platform separately. The results are given in Figure 4.



a) Mix one SPEC process with one *"stream* core"



b) Mix one SPEC process with three "*stream* cores"

Figure 4. Relevance between the MRP system and the real chip system for heterogeneous workloads using statistical characteristics

Figures 4(a) and (b) show that when using program statistical characteristic to fake a *stream* process, some SPEC benchmarks show big deviations, the maximum and minimum values reach 1.6 and 0.67 respectively. The standard deviations of all the relevance scores are 23% and 21.8% for 2-core and 4-core workloads respectively. Interestingly, even though some of the benchmarks show big deviations, the overall averaged relevance score is very close to 1 (1.03 and 0.99 for 2-core and 4-core workloads). The reduced accuracy is relevant to our simple *stream* trace generating strategy. It seems like this approach does not faithfully reflect the microarchitecture characteristics of our system with enough details.

2) Use trace recording and replaying to fake a core

To overcome the shortcoming of the "*stream* core", we also implement the "trace recording and replaying" mechanism on our FPGA platform. Currently we only add one "trace core" in the FPGA system to simulate the co-running of two different programs. We allocate 1 GB memory space for the trace recorder and replayer and reserve 32 MB address space for the pseudo core. As our trace entry for one request takes about five bytes (1 bit access type, 11-bit timestamp, 28-bit address and 2-bit extra information, the address information excludes the cache line byte offset), the 1 GB storage space can store about 200 million accesses. The trace recorder stops recording if the allocated memory becomes full. The trace replayer keeps playing the recorded trace data repeatedly. Although we have taken steps to minimize the influence of the trace recorder and replayer, both of them incur extra traffic on the memory bus. To measure this influence, we run *stream* and SPEC benchmarks with and without these two trace engines separately. The results (which are omitted for page limit) show that the influence is negligible. We simulate combinations of *stream* and SPEC benchmarks and the mixtures of different SPEC benchmarks. For mix of *stream* and SPEC benchmarks, we record the *stream*'s trace first, and then replay this trace during the execution of SPEC benchmarks. For mix of different SPEC benchmarks, we do the same process twice, each for one target SPEC benchmark. We select five INT and five FP SPEC benchmarks to do the simulation. The simulation results are given in Figure 5.

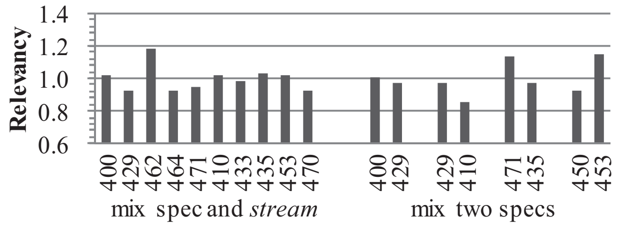


Figure 5. Relevance between the MRP system and the real chip system for heterogeneous workloads using trace recording and replaying mechanism

Figure 5 demonstrates that the "trace recording and replaying" mechanism can improve the relevance score of the simulation of heterogeneous workloads significantly. Most benchmarks have a relevance score close to 1. The standard deviation of mixing SPEC benchmark and *stream* is 7.5%, and the same value of mixing two SPEC benchmarks is 9.4%, the overall deviation is 8.3%. The decreased relevance of mix two SPEC benchmarks is part because of the limitation of trace storage space. The trace size of *stream* program in our system is about 400MB which is smaller than our trace storage size. As the SPEC benchmarks run much longer and generate more accesses, the allocated storage space is not enough to store all the trace data. In our experiment, we start recording trace from the beginning of the execution of SPEC benchmarks (skip the benchmarks compiling and setup stage) and abandon the left trace if the trace storage is used up.

In summary, by duplicating the real core's real-time trace, MRP can achieve a relative error of 6.4% for homogeneous workload. And by employing the trace recording and replaying mechanism, MRP can achieve a relative error of 8.3% for heterogeneous workloads. Based on the experimental results, we conclude that MRP is effective for the FPGA-based chip-multiprocessor simulation.

4. Performance Evaluation

To evaluate the performance of MRP, we select two SPEC benchmarks randomly to do software-based and commercial emulator-based simulations separately. We run the gem5 [3] simulator on an Intel Xeon E5606 8-core processor with 48 GB of DRAM. We select the OOO CPU model and the Alpha ISA to do a full system simulation. As the gem5's multiprocessor model is not based on shared memory system as our target CMP, we only simulate the single user SPEC test. As to commercial emulator, we use the EVE Zebu platform [5]. Zebu is an extremely high capacity system emulator that requires no RTL modification and has up to 1 billion ASIC gates. We run the 4-core homogeneous workload on the MRP and Zebu platforms, and the single core workload on the gem5. We use the train input set. The execution times are shown in Figure 6.

To run the *bzip2* and *hmmer* benchmarks using train input set, it takes about 2.8 and 6 hours respectively on MRP platform, while it takes about 4.6 and 10 days on Zebu, and 3 and 5 months on gem5 (even only a uniprocessor simulation) approximately. The speedup of MRP over Zebu is about 40 times, and about 800 times over gem5.

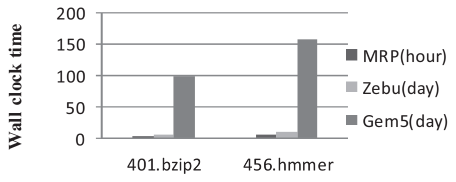


Figure 6. Wall clock time of MRP, Zebu and gem5 simulators running SPEC CPU2006 benchmarks. Note that we use different time units for MRP and Zebu/gem5. As the gem5's simulation is too long to finish, we only run some samples of the benchmarks and estimate the overall time.

5. Related Work

To overcome the capacity constraint of FPGA-based simulation, researchers have proposed several techniques.

FAST [9] splits the functional and timing model. Only the timing model is implemented on FPGA, the functional model is still running by software. Although the split model can reduce the FPGA resource requirement and engineer efforts, the long communication latency between the split components limits the simulation performance.

HAsim [10] implement only one physical core on FPGA and use the time-division multiplexing to virtualize a multicore environment for the software. With this technique, multiple FPGA cycles have to be spent to simulate one *target* cycle which makes the simulation speed decrease linearly with the simulated core number.

[11, 12, 13] partition the target chip into multiple FPGAs to construct the whole system. Partition is the most straightforward approach to overcome the FPGA capacity problem, and it is the most accurate method. However, partition also suffers from performance loss because of the communication latency between separated FPGA chips and the degraded IO speed when using IO multiplexing technique. For example, as reported by Intel [11], the simulation speed is reduced from 100 MHz *host* frequency to 520 KHz *target* frequency because of IO multiplexing and inter-FPGAs delay.

RAMP Gold [14] also splits the functional and timing model, but both parts are implemented on FPGA, and it employs multithreading technique to simulate a 64-core processor. However, their target processor core is in-order and single-issue which differs from our out-of-order superscalar core, and the simulator's fidelity is not verified. Tan et al. [15] summarize commonly-used FPGA-based simulation techniques and provide taxonomy of these approaches.

MRP differs from the above approaches in that it combines both the advantages of FPGA-based prototype and trace-based simulation. And our method is orthogonal to the above FPGA-based simulation techniques.

6. Discussion and Conclusion

Currently, for heterogeneous workloads, we only add one "trace core" in our FPGA platform to do a two-core simulation. With the number of simulated cores increases, more storage room will be needed for the trace data. To deal with this problem, one way is to increase the storage memory directly. Adding more DRAM is the cheapest solution. After the DRAM capacity reaches its limit, exploring the disk storage may provide another option. The other way is employing sampling to decrease the storage requirement by selectively recording the representative trace segments. We leave these explorations for the future work.

When facing the speed challenge of multicore simulation, FPGA-based simulations are increasingly used. To tackle with the FPGA capacity problem, MRP provides a promising solution with low cost. We implement an instance of MRP and validate its effectiveness under both homogeneous and heterogeneous workloads. The experimental results show that MRP has a relative error rate of about 8%, but achieves a speedup of about 800 over a popular software simulator.

References

[1] Austin, T., E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," Computer, 2002. 35(2): p. 59-67.

[2] Rosenblum M, Bugnion E, Devine S, et al., "Using the SimOS machine simulator to study complex computer systems," ACM Transactions on Modeling and Computer Simulation, 1997. 7(1): p. 78-103.

[3] Binkert, N., Beckmann, B., Black, G., et al., "The gem5 simulator," ACM SIGARCH Computer Architecture News, 2011. 39(2): p. 1-7.

[4] Wawrzynek J, Patterson D, Oskin M, et al., "RAMP: Research accelerator for multiple processors," Micro, IEEE, 2007. 27(2): p. 46-57.

[5] ZeBu Server-3, http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/zebu-server-asic-emulator.aspx.

[6] Uhlig, Richard A., and Trevor N. Mudge. "Trace-driven memory simulation: A survey," ACM Computing Surveys 29.2 (1997): 128-170.

[7] http://www.cs.virginia.edu/stream/ref.html.

[8] http://oprofile.sourceforge.net/news/

[9] Derek Chiou, Dam Sunwoo, Joonsoo Kim, et al. "FPGA-accelerated simulation technologies (FAST): Fast, Full-system, Cycle-accurate Simulators," in Proc. of the 40th Annual IEEE/ACM international Symposium on Microarchitecture. 2007.

[10] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in Proc. of the 17th International Symposium on HPCA, 2011.

[11] G. Schelle, J. Collins, E. Schuchman,et al. "Intel nehalem processor core made FPGA synthesizable," in Proc. of the 18th ACM/SIGDA international symposium on Field Programmable Gate Arrays. 2010.

[12] Sameh Asaad, Ralph Bellofatto, Bernard Brezzo, et al. "A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multicore processor simulation," in Proc. of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. 2012.

[13] Wee S., Casper J., Njoroge N., Tesylar, Y., Ge, D., et al,. "A practical FPGA-based framework for novel CMP research," in Proc. of the ACM/SIGDA 15th international symposium on FPGA, 2007.

[14] Z Tan, A Waterman, R Avizienis, Yunsup Lee, Henry Cook, et al. "RAMP gold: an FPGA-based architecture simulator for multiprocessors," in Proceedings of the 47th DAC. 2010.

[15] Zhangxi Tan, Andrew Waterman, Henry Cook, et al. "A case for FAME: FPGA architecture model execution," in Proc. of the 37th ACM/IEEE International Symposium on Computer Architecture (ISCA), 2010.