Parallelization of IBM Mambo System Simulator in Functional Modes

Kun Wang
IBM China Research Lab
wangkun@cn.ibm.com

Yu Zhang
IBM China Research Lab
zhyu@cn.ibm.com

Huayong Wang
IBM China Research Lab
huayongw@cn.ibm.com

Xiaowei Shen
IBM China Research Lab
xwshen@us.ibm.com

ABSTRACT

Mambo [4] is IBM's full-system simulator which models PowerPC systems, and provides a complete set of simulation tools to help IBM and its partners in pre-hardware development and performance evaluation for future systems. Currently Mambo simulates target systems on a single host thread. When the number of cores increases in a target system, Mambo's simulation performance for each core goes down. As the so-called "multi-core era" approaches, both target and host systems will have more and more cores. It is very important for Mambo to efficiently simulate a multi-core target system on a multi-core host system. Parallelization is a natural method to speed up Mambo under this situation.

Parallel Mambo (P-Mambo) is a multi-threaded implementation of Mambo. Mambo's simulation engine is implemented as a user-level thread-scheduler. We propose a multi-scheduler method to adapt Mambo's simulation engine to multi-threaded execution. Based on this method a core-based module partition is proposed to achieve both high inter-scheduler parallelism and low inter-scheduler dependency. Protection of shared resources is crucial to both correctness and performance of P-Mambo. Since there are two tiers of threads in P-Mambo, protecting shared resources by only OS-level locks possibly introduces deadlocks due to user-level context switch. We propose a new lock mechanism to handle this problem. Since Mambo is an on-going project with many modules currently under development, co-existence with new modules is also important to P-Mambo. We propose a global-lock-based method to guarantee compatibility of P-Mambo with future Mambo modules.

We have implemented the first version of P-Mambo in functional modes. The performance of P-Mambo has been evaluated on the OpenMP implementation of NAS Parallel Benchmark (NPB) 3.2 [12]. Preliminary experimental results show that P-Mambo achieves an average speedup of 3.4 on a 4-core host machine.

Keywords

architectural simulation, parallel simulation, dynamic binary translation

1. INTRODUCTION

Both industry and academia have relied on architectural simulators [5, 13, 10, 3, 7, 16, 11] in system architecture design, software development and performance evaluation during past decades. Mambo [4] is IBM's full-system simulator which models the PowerPC-based [8] systems, and provides a complete set of simulation tools to help IBM and its partners in pre-hardware development and performance evaluation for future systems [15, 1, 6, 14]. Mambo supports both functional and cycle-accurate simulation modes:

- Functional modes emulate the PowerPC Instruction Set Architecture [8] and devices necessary for executing the operation systems. The basic implementation of functional modes is called "simple mode" , which is based on per-instruction interpretation. An enhancement called "turbo mode" has been made to significantly improve the performance of simple mode by the so-called "dynamic binary translation(DBT)" technology [17, 9, 2]. Major ideas of DBT are to dynamically translate target binary codes to host binary codes, and directly execute the host binary codes on a host machine. Currently turbo modes support x86, x86_64, POWER and PowerPC host systems;

- Cycle-accurate modes provide the capability to gather detailed information of target systems, such as cycle numbers, cache miss ratios and bus traffic. Cycle-accurate modes require a detailed modeling of the PowerPC processor micro-architecture, such as out-of-order execution pipelines and memory-hierarchy with coherent caches. A typical simulation of cycle-accurate modes is 2∼3 orders of magnitude slower than turbo modes.

Currently Mambo simulates target systems on a single host thread. When the number of cores increases in a target system, Mambo's simulation performance for each core goes down. As the so-called "multi-core era" approaches, both target and host systems will have more and more processor cores. It is crucial for Mambo to efficiently simulate a multi-core target system on a multi-core host system. Parallelization is a natural method to speed up Mambo under this situation. However, there are challenges in parallelizing Mambo:

1. Protecting shared resources from concurrent accesses is critical to parallelizing a sequential program. The simulation engine of Mambo is a user-level thread-scheduler. A Mambo module can be regarded as a user-level thread. There are 2 tiers of threads in P-Mambo. One is OS-level, the other is user-level. Protection of share resources should be carefully designed to avoid deadlocks and minimize overhead;

2. Parallelizing turbo modes of Mambo introduces some unique challenges related to dynamic binary translation. For example, the PowerPC architecture supports self-modifying codes, i.e. target binary codes are possibly generated on-the-fly. It is critical to both correctness and performance of P-Mambo to detect the modified codes and keep coherency of translation caches among multiple host threads; and

3. Mambo is an on-going project. Many teams are developing new modules for Mambo to support new target processors and devices. Although the existing modules can be modified to guarantee their correctness in parallel execution, a mechanism is needed to ensure that P-Mambo can co-exist with new modules, which are possibly not thread-safe in a parallel environment.

We have implemented the first version of P-Mambo in functional modes. Some benchmarks have been tested to evaluate its performance. The host machine is an IBM Blade Center LS21, which has two dual-core AMD Opteron 275 processors and 8GB memory. The target machine is a 4-core PowerPC machine with 6GB memory. The benchmark set is the OpenMP implementation of NAS Parallel Benchmark (NPB) 3.2 [12]. The experimental results show that the maximum and average speedups of P-Mambo are 3.8 and 3.4 respectively.

The reminder of the paper is organized as follows. Section 2 presents our design and implementation of P-Mambo. Section 3 shows the experimental results of P-Mambo on NPB benchmark set. Finally, Section 4 concludes the paper.

2. DESIGN AND IMPLEMENTATION

In this section, we first present our design on simulation engine enhancement and module partion in P-Mambo. Then we describe implementation details to attack challenges mentioned in section 1, which are shared resource protection, co-existence with new modules and handling of DBT-related issues in P-Mambo.

2.1 Simulation Engine Enhancement

Mambo is a discrete event-driven simulator. A scheduler (named *tsim*) sorts and schedules all modules (named *jobs*) by their trigger time in Mambo. Figure 1 illustrates a job queue of a tsim. To simplify the development effort while still precisely modeling hardware events, tsim is implemented as a user-level thread-scheduler [4]. Each job is regarded as a user-level non-preemptive thread on tsim. Below is an example job simulating a core.

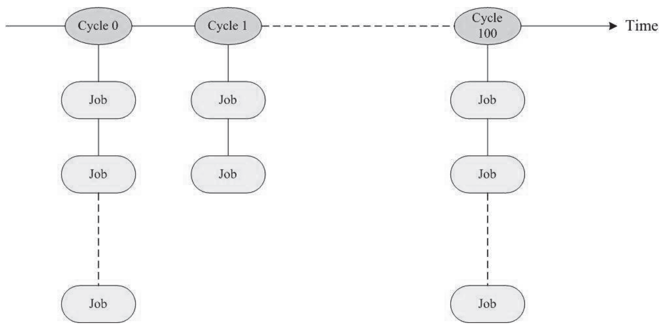

Figure 1: Ready queue of a tsim

```
while (! core->stop)
{
    Fetch_Instruction(&core, &inst);
    Decode_Instruction(&core, &inst);
    Execute_Instruction(&core, &inst);
    Check_Interrupt_Exception(&core, &inst);
    do_delay(inst->delay);
}
```

After invoking `do_delay(inst->delay)`, the job is switched out and re-activated `inst->delay` cycles later. By this way, multiple cores can be simulated in an interleaving manner. Tsim also provides primitives for inter-job synchronization. For example, a producer job can wake up a consumer job by a counter [4] when data becomes ready.

As a user-level thread-scheduler, tsim supports only one active job at any time. Mambo is implemented as a sequential simulator because there is only one tsim to schedule Mambo's modules. We have two choices to parallelize Mambo:

1. creating multiple tsims, each of which runs on a dedicated host thread; and

2. enhancing tsim to support the execution of multiple active jobs on multiple host threads.

Both choices faces the same problem of what group of jobs can be executed simultaneously. We prefer the first choice, because the second introduces more complexity and overhead in protection of tsim internal data structure. There are three challenges once we have multiple concurrent tsims:

1. how to partition jobs among tsims to allow maximum parallelism;

2. how to support job interactions across tsims with minimum overhead; and

3. how to protect shared resources accessed by multiple jobs with minimum overhead.

2.2 Job Partition

Job partition determines parallelism and interaction between different tsims, and thus is crucial to performance. We choose core-based job partition due to following reasons: 1) in functional modes, most workloads are from the simulation of processor cores; 2) there exists natural parallelism between different cores; and 3) the data dependency between different cores is low.

The general rules to create jobs on tsims are as follows:

1. jobs belonging to the same target processor should be created on the same tsim;
2. jobs belonging to different target processors can be created on different tsims.

The number of tsims created can be customized in P-Mambo. Users can create as many host threads as they want. By this way, users can simulate a target machine with more processor cores than the host machine.

2.3 Job Interactions

There are two types of job interactions in tsim: 1) one job wakes up another job; 2) one job blocks anther job. Both of them are implemented by ready or waiting queue manipulations. Thread-safety should be guaranteed for inter-tsim job interactions. One possible implementation is to simply serialize accesses to shared queues by locks. However, there is only one ready queue in a tsim. Each time a tsim switches from one job to another, the ready queue is accessed. Obtaining and releasing lock for each ready queue access introduce too much overhead. We improve the implementation by adding another ready queue named *external ready queue* to tsim. A job is inserted into the external ready queue when it is waken up by a job on another tsim. The ready queue access in job switch is implemented as follows:

```
//Getting next ready job
if (tsim->external_readyq == NULL)
{
    get_next_ready_job(tsim->internal_readyq);
}
else
{
    lock(tsim->external_readyq);
    get_next_ready_job(tsim->external_readyq);
    unlock(tsim->external_readyq);
}
```

Since the external ready queue is empty in most cases, the overhead of obtaining and releasing lock is low.

2.4 Shared Resource Protection

Protecting shared resources from concurrent accesses is critical to both correctness and performance of parallelizing a sequential program. Lock is a common technique to serialize all accesses to shared resources. However there are 2 tiers of threads in P-Mambo. One is OS-level (inter tsim), the other is user-level (intra tsim). Due to these two tiers of threads, deadlocks in P-Mambo are different from that in traditional multi-threaded programming. Competition for one type of shared resource introduces no deadlock in traditional multi-threaded programming, however may result in deadlocks in P-Mambo.

Here is an example of deadlock competing only one type of shared resource in P-Mambo. Load Reserve (larx) and Store Conditional (stcx) instructions are used in implementing atomic operations in the PowerPC architecture [8]. A larx instruction creates a reservation for an address. A stcx instruction successfully performs a store to the address only if the reservation created by the previous larx is not cleared by another processor or mechanism. Therefore, stcx instructions should be serialized to guarantee the correctness in P-Mambo. Below is the code segment of handling stcx instructions using OS-level lock to guarantee thread-safety.

```
// handling Store Conditional instructions
stcx(Address addr, Value v)
{
    os_level_lock(reservation(addr));

    clearing_reservation(reservation(addr));
    if (core->reservation_addr != addr)
    {
        os_level_unlock(reservation(addr));
        return (RESERVATION_LOST);
    }
    else
        write_memory_with_latency(addr, v);

    os_level_unlock(reservation);
    return (SUCCESS);
}
```

Consider a scenario of simulating four cores on two tsims. There are core job 1 and 2 on tsim 1. Deadlock occurs if the events happen in the following sequence:

- job 1 obtains lock for a reservation;
- job 1 is switched out by `write_memory_with_latency`;
- job 2 is switched in and tries to obtain lock for the same reservation.

This case shows that accesses to shared resource with tsim-level context switch can not be protected by OS-level locks. We propose job-level lock to attack the problem. The major idea is that failure in competing job-level lock results in only hanging up current job, instead of blocking host thread of current tsim. Below are the pseudo codes of obtaining and releasing a job-level lock.

```
job_level_lock(job_lock* l)
{
    os_level_lock(l->os_lock);

    if (l->counter > 0) {
        add_to_wait_queue(l->wait, current_job);
        os_level_unlock(l->os_lock);
        switch_to_next_ready_job();
    }
    else
        l->counter = 1;

    os_level_unlock(l->os_lock);
}

job_level_unlock(job_lock* l)
{
    os_level_lock(l->os_lock);

    l->counter --;
    wake_up_the_first_job(l->wait);

    os_level_unlock(l->os_lock);
}
```

2.5 Co-existence with New Modules

Mambo is an on-going project. Many teams are developing new modules to support new target processors and devices. Although existing jobs can be modified to guarantee their correctness in parallel execution, the new jobs are possibly not thread-safe. It would be a disaster that P-Mambo is broken each time a new job is introduced.

We use a tsim-level-lock to solve this problem. A tsim-level-lock can be regarded as a global lock among all tsims. Once it is obtained by a tsim, all other tsims should wait until it is released. Furthermore, an attribute *thread-safety* is added to all jobs. Each job is set as *thread-unsafe* by default. A job can be set as *thread-safe* only if its thread-safety can be guaranteed. And two new scheduling rules are also needed:

1. before switching to a thread-unsafe job, a tsim should ask all others to wait by requiring a tsim-level lock; and
2. after leaving a thread-unsafe job, a tsim should wake up all others by releasing the tsim-level lock.

By the rules above, P-Mambo can ensure that no thread-unsafe job intervenes with each other, i.e., execution of all thread-unsafe jobs is serialized. Therefore, the correctness can be guaranteed.

2.6 Handling of DBT Issues

Mambo turbo mode is based on dynamic binary translation. It includes two major components: binary translator and translation cache. Binary translator is responsible for generating host binary codes based on target codes, so it can be shared by multiple tsims in nature. Translation cache contains the generated host codes for direct execution, and then can not be shared by multiple tsims without any protection. There are three cases that translation cache is modified:

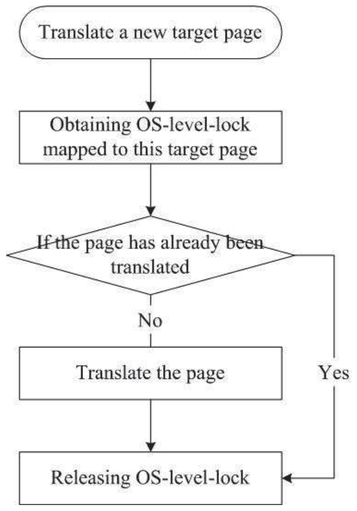
Translate a new target page
Obtaining OS-level-lock mapped to this target page
If the page has already been translated
No
Yes
Translate the page
Releasing OS-level-lock

Figure 2: Translation of a new target page

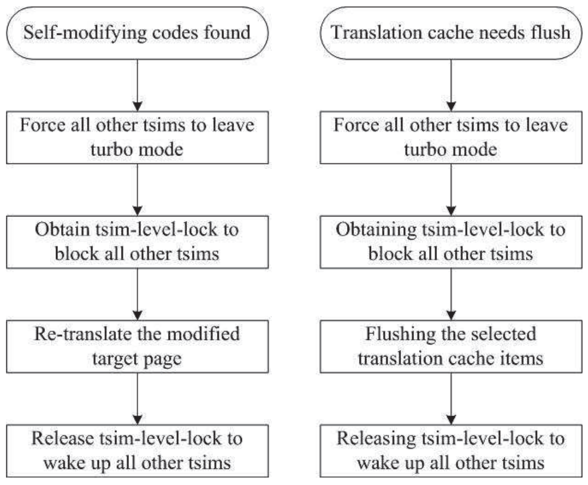
Self-modifying codes found
Force all other tsims to leave turbo mode
Obtain tsim-level-lock to block all other tsims
Re-translate the modified target page
Release tsim-level-lock to wake up all other tsims
Translation cache needs flush
Force all other tsims to leave turbo mode
Obtaining tsim-level-lock to block all other tsims
Flushing the selected translation cache items
Releasing tsim-level-lock to wake up all other tsims

Figure 3: Handling of self-modifying codes and translation cache flush

1. adding a new cache item by translating a new target page;
2. flushing a cache item due to some reason, such as translation cache overflow; and
3. modifying a cache item due to self-modifying codes.

For the first case, it is impossible that when a tsim is going to translate a new page, some others are executing the un-translated codes. Therefore, OS-level lock is enough to handle this case. Figure 2 illustrates the basic idea to handle the translation of a new target page in P-Mambo.

For the second and third cases, it is possible that some other tsims are executing the codes of a cache item when a tsim wants to modify or invalidate it. A tsim can only modify or invalidate a cache item when there is no other executing the codes of the item. Figure 3 shows the steps to handle self-modifying codes and translation cache flush in P-Mambo.

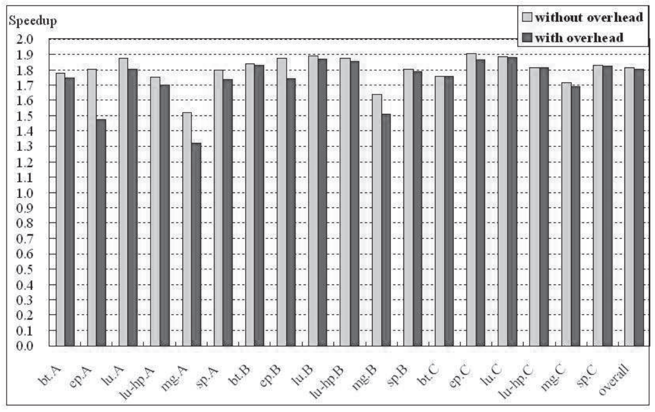

Figure 4: Speedups on 2 host threads

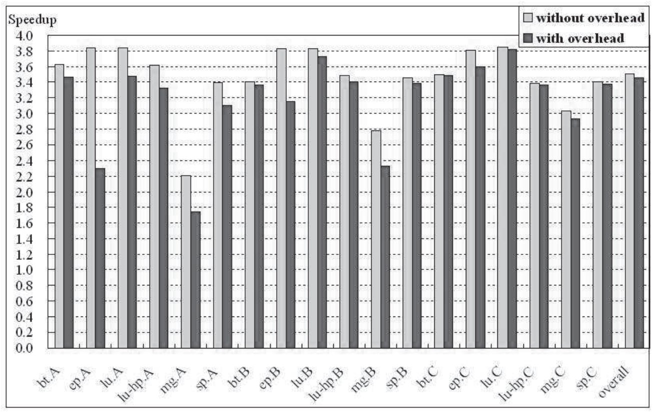

Figure 5: Speedups on 4 host threads

3. PERFORMANCE EVALUATION

We have implemented the first version of P-Mambo in functional modes. Some benchmarks have been tested to evaluate the performance of P-Mambo. The benchmark set is the OpenMP implementation of NAS Parallel Benchmark (NPB) 3.2 [12]. The host machine is an IBM Blade Center LS21, which has two dual-core AMD Opteron 275 processors and 8GB memory. The target machine is a 4-core PowerPC machine with 6GB memory. The target OS is linux 2.6.16(ppc64), while the host OS is linux-2.6.18(x64_64).

We evaluate the performance of P-Mambo under two cases: one is creating two host threads, the other is creating four host threads. Figure 4 summarizes the speedup of P-Mambo in the case of creating two host threads. Please note that "bt.A" means benchmark bt with size of Class A. Since P-Mambo is a full-system simulator, the whole simulation time of a benchmark includes overhead of booting OS. The speedup with overhead is calculated by the whole simulation time of a benchmark, while the speedup without overhead is calculated by the pure workload simulation time of a benchmark. The results show that P-Mambo achieves the maximum and average speedups (without overhead) of 1.9 and 1.8 respectively when running on two host threads.

Figure 5 summarizes the speedup of P-Mambo in the case of creating four host threads. P-Mambo shows its scalability and achieves the maximum and average speedups (without overhead) of 3.8 and 3.4 respectively when running on four host threads.

4. CONCLUSIONS

Mambo is IBM's full-system simulator which supports both functional and cycle-accurate simulation modes, and hence helps IBM and its partners in both pre-hardware development and performance evaluation for future systems. However, Mambo is a sequential program, its performance goes down as the number of cores increases in a target system. Thereby how to benefit from a multi-core host system in simulating a multi-core target system is critical to Mambo, when the so-called "multi-core era" approaches. Parallelization is a natural way to speedup Mambo under this situation. We have implemented the first version of P-Mambo in functional modes. The performance evaluation on NAS Parallel Benchmark (NPB) 3.2 shows promising speedups: P-Mambo achieves the maximum and average speedups of 3.8 and 3.4 respectively, in the case of simulating a 4-core PowerPC machine on a 4-core AMD Opteron machine.

Our next step is parallelizing Mambo in cycle-accurate modes. It is much more difficult to parallelize cycle-accurate modes than functional modes, because there are more dependencies existing in cycle-accurate modes than functional modes. There are possibly two critical issues of parallelizing cycle-accurate Mambo: 1) how to efficiently exploit system-level parallelism among different cores, as well as micro-architecture-level parallelism among different core components; and 2) how to achieve a reasonable trade-off between performance and accuracy.

5. ACKNOWLEDGMENTS

We would like to thank Patrick Bohrer, Ahmed Gheith, Ronald Rockhold, Lixin Zhang and James Peterson from IBM Austin Research Laboratory for their help and support on Mambo.

POWER and PowerPC are the trademarks of IBM Corp. X86 is a trademark of Intel Corp. Opteron is a trademark of Advanced Micro Devices, Inc. We acknowledge all trademarks referenced herein to be the property of their owners.

6. REFERENCES

[1] L. R. Bachega, J. R. Brunheroto, L. DeRose, P. Mindlin, and J. E. Moreira. The BlueGene/L Pseudo Cycle-accurate Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2004.

[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN)*, September 2005.

[4] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold,

H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo – A Full System Simulator for the PowerPC Architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[6] L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Cascaval, J. G. Castanos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full Circle: Simulating Linux Clusters on Linux Clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, June 2003.

[7] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[8] I. Corporation. *The PowerPC Architecture: A Specification for a New Family of Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[9] K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of 24th Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[11] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. ATLAS: A Chip-Multiprocessor with Transactional Memory Support. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, 2007.

[12] NPB. NAS Parallel Benchmarks. *http://www.nas.nasa.gov/Resources/Software/npb.html*.

[13] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[14] H. Shafi, P. J. Bohrer, J. Phelan, C. A. Rusu, and J. L. Peterson. Design and validation of a performance and power simulator for PowerPC systems. *IBM Journal of Research and Development*, 47(5-6):641–651, September 2003.

[15] T. B. Team. An Overview of the BlueGene/L Supercomputer. In *Proceedings of ACM/IEEE Conference on Supercomputing*, November 2002.

[16] S. Wee, J. Casper, N. Njoroge, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. A Practical FPGA-based Framework for Novel CMP Research.

[17] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1996.