Hardware Transactional Memory System for Parallel Programming

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

Hou Rui
IBM China Research Lab
hourui@cn.ibm.com

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

Abstract

Hardware transactional memory (HTM) is an attractive research topic in recent years. It has great potential to simplify parallel programming on the soon-to-be-ubiquitous multi-core systems. In this paper, a HTM design is proposed, and overall performance is evaluated. This HTM design distinguishes itself from others by its best effort philosophy. The hardware makes best effort to complete each transaction and software handles those transactions that cannot be completed by hardware. This design seeks a balance between application performance and hardware implementation complexity, and tries to answer the question: what should be done by hardware and what should be done by software. The overall performance of benchmarks is also evaluated by simulation

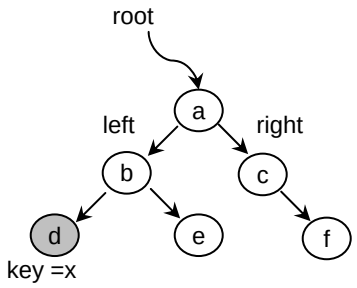
1. Introduction

Parallel programming remains a challenging problem in most application domains despite years of research. As chip-multiprocessors become ubiquitous, this problem becomes more severe. Providing architectural support for massive parallel programming is now critical. With this background, hardware transactional memory (HTM) systems have been proposed to ameliorate this challenge.

HTM provides a programming model that makes parallel programming easier. A programmer delimits regions of code that access shared data and the hardware executes these regions atomically and in isolation, buffering the results of individual instructions, and retrying execution if isolation is violated. Generally, TM allows programs to use a programming style that is close to coarse-grained locking to achieve performance that is close to fine-grained locking.

Figure 1 shows a program using transactions, where the begin and end of a transaction are marked in the source code by TM_BEGIN and TM_END. The transaction accesses the shared variables in a balanced binary tree. Using transactions, multiple threads can access the tree simultaneously, which shows potentials for performance improvement.

```
TM_BEGIN {
  p = root;
  while (TRUE) {
    if (x < p->key) {
      p = p->left;
    } else if (x > p->key) {
      p = p->right;
    } else { break; }
  }
  do read/write/deletion/insertion here;
} TM_END;
```



Balanced binary tree operation
(read, write, delete and insert)

Figure 1. A transaction example

This paper proposes a HTM design for Power/PowerPC based CMP with best effort philosophy. That is, the hardware makes best effort to complete each transaction without the guarantee of completing all transactions. The software handles those transactions that cannot be completed by hardware. This design philosophy seeks to balance the

application performance and hardware implementation complexity, which is crucial for industries to implement HTM in real products.

The remaining sections are arranged as following. Section 2 introduces the related work. Section 3 gives our HTM proposal and explains our design philosophy. Section 4 introduces the benchmark and methodology. Section 5 gives the experiment result of overall performance evaluation. Section 6 concludes the paper.

2. Related work

The original transaction concept stems from early research in database management systems, e.g. IMS [1] and System R [2], and has been widely adopted and studied since then. A number of researchers proposed to adopt the transaction principle outside database systems, e.g. for parallelizing the execution of mostly functional programs [3] or for the management of concurrent access to virtual memory [4]. The HTM mentioned in this paper was firstly proposed in 1993 by [5] and concurrently by [6]. However, the HTM becomes a popular research topic only in recent years after multi-core system is agreed to be the trend of chip design. Currently, HTM attracts the interests from both research communities and industries. Some HTM designs were proposed, such as LogTM [7], UTM [8], VTM [9], PTM [10] and TCC [11]. More and more papers on HTM are published in research conferences.

HTM proposals from academia often adopt fancy ideas to solve the typical problems in HTM design, such as large transaction and transaction nesting. However, these proposals are hard to be implemented in real products by industries because of the implementation complexity and cost. Recently, Sun announced the first processor that supports HTM [12], named Rock. Rock maintains the isolation and atomicity of transactions by buffering all stores in the store queue, keeping them invisible from the outside until the transaction commits, and by using the per-thread, per-L1-cache-line S-bits (speculative bits) to track locations read by transactions. These bits are cleared and the transactional stores buffered in the store queue are discarded if the transaction fails. Rock also adopts best effort philosophy to simplify the design and implementation. We believe that how to limit the resource applied to HTM is a valuable research topic.

An alternative to implement transactional memory is by pure software, named software transactional memory (STM). The typical examples include DSTM [13], WSTM [14], OSTM [15] and so on. STM has good flexibility and small implementation cost, but the performance is generally worse than HTM. In this paper, we only focus on HTM.

3. Architecture design

To minimize the hardware complexity, in our design, the HTM makes best effort to complete each transaction. For transactions that may not finish in transactional execution (e.g., due to hardware resource limitation), it uses a software handler to force serialization of transactions so that they can be completed in regular non-transactional execution (refer to section 3.3).

To balance performance and complexity, our design approach is to optimize for common cases while handles rare cases in a slower but acceptable way. On the one hand, it is difficult for hardware to support all kinds of transactions. There are many fundamental limitations, such as area, power and verification cost. On the other hand, most transactions (under evaluation) actually show common characteristics, such as small write set.

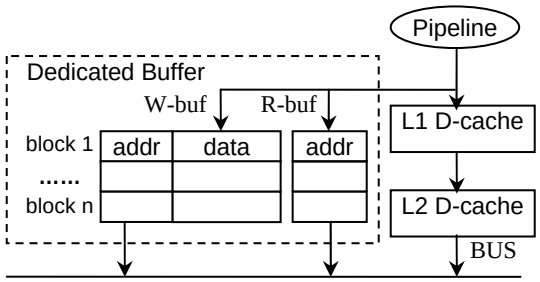


Figure 2. Dedicated buffer

The HTM proposed in this paper requires the following extensions to the hardware. As shown in Figure 2, a dedicated buffer is added to each processor. Each dedicated buffer contains two parts: write buffer (W-buf) and read buffer (R-buf). When the execution enters a transaction, all memory stores are saved in the write buffer (W-buf) instead of the conventional memory hierarchy, and the addresses of all memory loads are recorded in the read buffer (R-buf). The data read or written by transactions are called speculative data.

The dedicated buffers and the cache are connected together by bus, and read/write messages are broadcast and monitored by dedicated buffers to detect conflicts among concurrent transactions. The granularity for conflict detection is a block. That is, a conflict is detected if two transactions access the same buffer block (not necessarily the same address) and at least one of them is a write. An immediate result is HTM

may have false sharing. False sharing refers to the situation that two transactions access the different portions in a same buffer block, and at least one of them tries to write. Since the granularity for conflict detection is a block, the above situation has to be deemed as a conflict although it is actually not.

If no conflict is detected, upon reaching the end of the transaction, stores in W-buf are committed into cache coherent memory. This is called transaction commit. Otherwise, a conflict resolution handler determines which of the two conflicting transactions to be aborted. The aborted transaction has to rollback (by clearing out W-buf and R-buf) and retry at some time later.

As an important design choice, the logging of load addresses and buffering of stores in transactions can be done by adding dedicated buffers, or by augmenting the cache. Unlike many other HTM proposals [7,8,9], this design does not use cache to save speculative data because of three reasons. The first, cache-based designs modify the existing cache system. Since the cache verification is expensive, industries are reluctant to accept cache-based designs. The second, current TM benchmarks [16] and related research work [17] all show that a 4KB W-buf can hold the data written by most transactions. And R-buf capacity is not an urgent problem since it only records addresses, rather than data. Therefore, buffer overflow would be a rare case. It is believed that the cost and complexity of dedicated buffer are acceptable. The third, large granularity for conflict detection actually hurts the program concurrency because of false sharing. Cache-based design naturally uses cache line as the granularity for conflict detection. Modern computers generally have large cache line size. For example, Power architecture has 128B cache line. Buffer-based design enables the designers to reduce false sharing by choosing smaller block size.

3.1. Data path of the HTM design

The load/store operation inside a transaction is called transactional load/store. The transactional load/store has different semantics from normal load/store. Figure 3 illustrates the data path of the HTM design in this paper. For simplicity, only one processor core is shown. In actual system, multiple cores share one L2 cache.

Transactional store saves data into W-buf.

1) If it hits W-buf, the data is written directly into W-buf.

2) If it misses W-buf, do conflict detection.

2.1) If there is no conflict, the store saves data into W-buf with write allocation.

2.2) If there is a conflict, one of the conflicted transactions has to abort. Which one to abort depends on the contention management policy.

3) If W-buf overflows, the transaction aborts.

Transactional load reads data from both L1 and W-buf, and checks whether the address hits R-buf.

1) If the load operation hits W-buf, return the data from W-buf and ignore the data from cache.

2) If the load operation misses W-buf but hits R-buf, return the data from cache.

3) If the load operation misses both W-buf and R-buf, do conflict detection.

3.1) If there is no conflict, read the data again from cache, and insert the address into R-buf.

3.2) If there is a conflict, one of the conflicted transactions has to abort. Which one to abort depends on the contention management policy.

4) If the R-buf overflows, the transaction aborts.

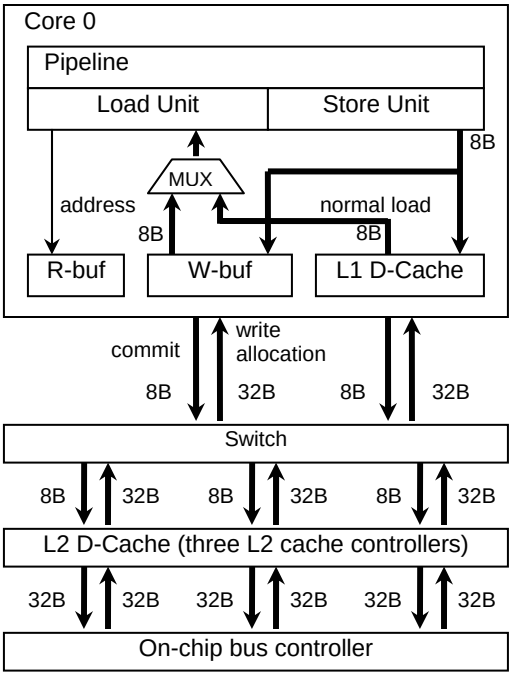


Figure 3. Data path of the HTM design

Dedicated buffers, L1 data cache are connected with L2 data cache by a crossbar switch. The L2 is implemented as three separate and autonomous cache controllers. Each L2 controller can operate concurrently and feed 32B of data per cycle. The switch also accepts stores from the processor core and sequences them to the L2 controllers. This is an IBM Power4-like architecture [18].

3.2. Runtime design

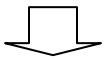
```
Define TM_BEGIN as following:
while (1) {
        int _ret;
        get thread local jmp_buf(allocated by tm_init);
        _ret = setjmp(jmp_buf);
        if (_ret == 0) {
           tbegin default_handler;
```

```
Define TM_END as following:
              tcommit();
              break;
      }
}
```

```
default_hanlder {
   get thread local jmp_buf;
     trollback();
     long_jmp(jmp_buf, 1);
}
```

Figure 4. Runtime of the HTM system

```
TM_BEGIN {
   // do something;
} TM_END;
```



```
while (1) {
   int _ret;
   get thread local jmp_buf;
   _ret = setjmp(jmp_buf);
   if (_ret == 0) {
      xbegin default_handler;
         {
            // do something;
         }
         xcommit();    // success
         break;
   }
}
```

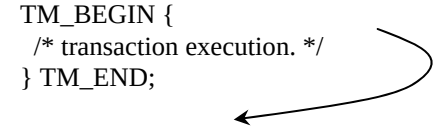
Figure 5. Transaction runtime macros

The runtime provides the basic software support for transactions, as well as overflow handling, contention management, and retry management. Fig. 4 gives a simplified implementation of the runtime (overflow handling are omitted). The runtime provides two macros: TM_BEGIN and TM_END. TM_BEGIN uses setjmp to save registers, and executes tbegin instruction to start transaction. TM_END executes tcommit instruction to commit a transaction. tbegin instruction has an argument "default_handler", which is the address of a TM handler provided by runtime. When conflict happens, the execution flow jumps into default_handler. long_jmp is called by default_handler to rollback transaction. Figure 5 shows a transaction after macro expansion.

3.3. Overflow handling

Buffer overflow is a rare case. The basic idea of overflow handling is to serialize the execution of transactions when overflow happens. Since, under this mode, there are no concurrent transactions, the transaction can be executed in a non-transactional manner and access only the regular memory hierarchy.

To implement this overflow handling mechanism, each transaction tests a global unique token before its execution. If the token is taken, it waits until the token is returned. If a transaction overflows, the execution jumps to an overflow handler, shown as Figure 6. The handler fetches the global unique token, and waits on all outstanding transactions to finish. After that, the overflowed transaction re-executes. During the re-execution, the data is directly written to cache coherent memory without conflict detection. After the re-execution finishes, the token is returned. The re-execution of the overflowed transaction is the only one transaction in the system, which is the key point to guarantee atomicity.



```
 TM_BEGIN {
   /* transaction execution. */
 } TM_END;

overflow_handler:
 do {
    // atomically fetch global unique token
  got_token = try_fetch_token();
 } while (got_token == FALSE);
 do {
   // wait until existing transactions finish
   no_trans = check_trans();
 } while (no_trans == FALSE);

 /* re-execute the transaction */
 return_token();
```

Figure 6. An overflow transaction

4. Benchmark and methodology

The benchmarks used in the paper come from several resources: STAMP, SPLASH-2 and self-developed application kernels. STAMP [16] is a TM benchmark published by Stanford University, including genome, vacation, rbtest and kmeans. genome is a bioinformatics application, performing gene sequencing from a very large pool of gene segments. vacation is a travel reservation system powered by an in-memory database. rbtest implements a red-black tree. kmeans is an clustering algorithm for data mining workload. SPLASH-2 is a well-studied benchmark for parallel computing. Three programs, barnes, ocean and raytrace, are ported to BET platform. Besides, a b+tree program is developed on BET platform. Many database system uses b+tree as kernel data structure.

In order to characterize those benchmarks and perform evaluation, we implemented two statistics.

1) Overflow rate is calculated by formula (1).

$$overflow\ rate = \frac{aborted\ trans\ by\ overflow}{committed\ trans} \times 100\% \quad (1)$$

Overflow rate actually reflects the transaction size. By observing the overflow rates at different buffer sizes, the information about transaction size is revealed. In this paper, the overflow rate is only accounted for W-buf since R-buf only records addresses. Some signature-based method can extend R-buf capacity to be reasonable large. So, the capacity of R-buf is not big problem.

2) Conflict rate is calculated by formula (2).

$$conflict\ rate = \frac{aborted\ trans\ by\ conflict}{committed\ trans} \times 100\% \quad (2)$$

A large number of conflicts can indicate either that application lacks inherent concurrency, or that certain design choices are causing unnecessary conflicts. In the above formula, if a transaction is aborted *n* times by conflict, it is counted as *n* in numerator.

The evaluation methodology is simulation. We modify the simulator mambo [19] to support HTM. Mambo is an IBM simulator for Power architecture. Figure 7 shows the target system to be simulated. The CMP contains two groups, connected by on-chip bus. Each group has 4 processor cores and a shared L2 cache. The corresponding configuration parameters are listed in table 1.

5. Design evaluation

This section gives the evaluation to the design in this paper, including the W-buf size selection, R-buf evaluation and performance comparison.

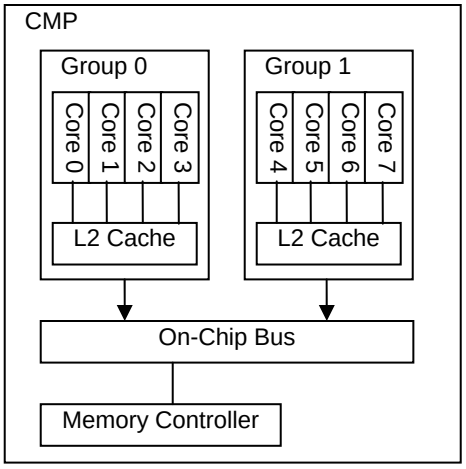


Figure 7. Target system

Table 1. System configuration

	L1	L2	Memory
size	32KB	2MB	128MB
associativity	4	8	NA
line size	128B	128B	NA
access latency (cycles)	2	load: 6 store: 20	254
type	write through	inclusive	NA

W-buf size selection is actually a tradeoff between performance and cost. Large W-buf reduces the chance to overflow, but requires more chip area and power. Since overflow has great impact to the performance, the W-buf size has to be well-selected to suit current benchmarks.

Table 2 gives the overflow rate of different benchmarks at different block size and W-buf size. Benchmarks kmeans, raytrace, barnes and ocean are not shown in this table because their transactions are very small (a 512B W-buf is quite enough for them). Generally, when W-bus size is small, the overflow rate is high, which significantly hurts performance. Take b+tree as example, experiments show that 35% ~ 48% execution time is used for overflow handling when W-buf size is 1KB, while the ratio falls to 1% ~ 6% when W-buf size is 4KB. An immediate conclusion is that small W-buf is unacceptable in performance. In our design, 4KB is chosen to be W-buf size since it has lowest overflow rate.

Figure 8 shows the scalability of the design in this paper. Different benchmarks are run with 1, 2, 4 and 8 threads. And the sequential version is a single-threaded program without any locks. Generally, single-threaded benchmarks have slightly slowdown if compared to

sequential version, but 8-threaded benchmarks may have 3~6 times speedup. The major source of the speedup comes from the concurrency of transactions. When thread number increases, more and more transactions run concurrently.

Figure 9 shows the performance comparison between lock-based programs and TM-based programs. Both run with 8 threads. W-buf size is 4KB and block size is 32B. Generally, TM-based programs run 2~5 times faster. SPLASH-2 has very small transactions, both in terms of transaction number and transaction size. So, it does not show good performance gain.

6. Conclusions

This paper proposes a HTM design for Power/PowerPC based CMP. The design seeks balance between performance and complexity. The application performance in HTM context is evaluated. Generally, HTM is able to achieve good performance gain when it is compared with lock-based systems.

Table 2. Overflow rate (8 threads)

block	32B			64B			128B		
W-buf	1KB	2KB	4KB	1KB	2KB	4KB	1KB	2KB	4KB
b+tree	3.04%	0.25%	0.001%	5.01%	0.25%	0.008%	5.02%	0.25%	0.25%
vacation	1.13%	1.06%	0%	6.80%	1.07%	0%	/	3.52%	0.12%
genome	0.14%	0%	0%	0.34%	0%	0%	0.51%	0%	0%
rbtest	0%	0%	0%	0.41%	0%	0%	5.44%	0.22%	0%

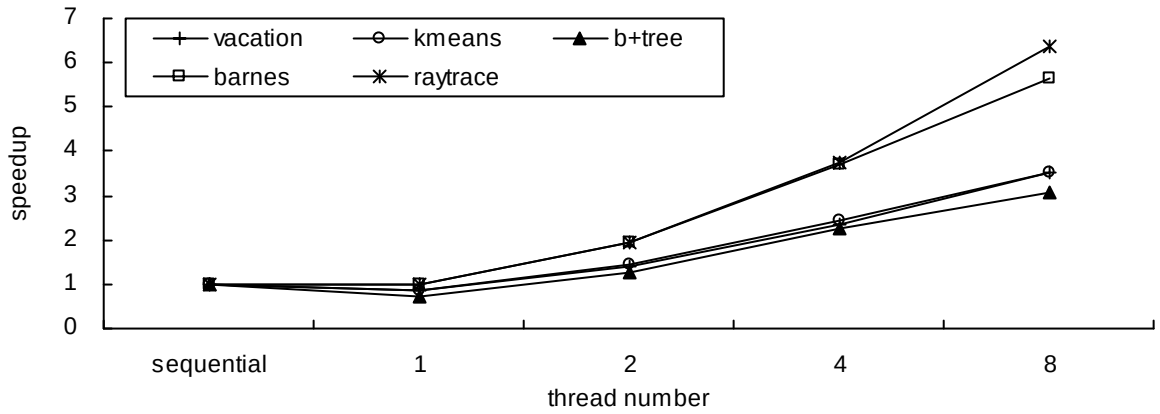


Figure 8. The scalability of the HTM

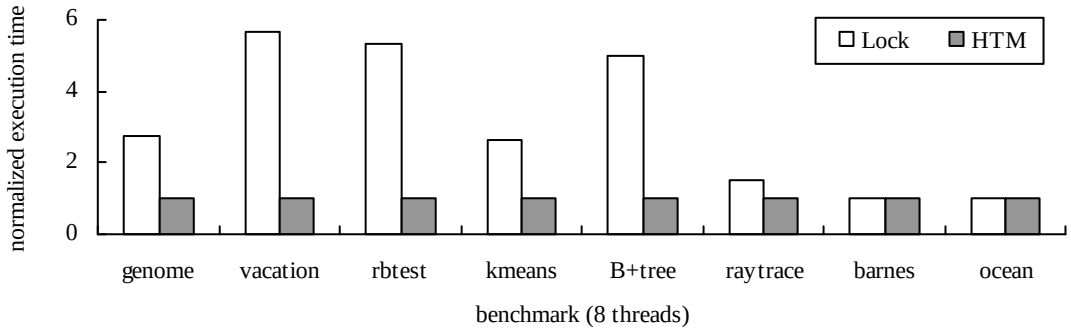


Figure 9. Performance comparison (4KB W-buf, 32B block size)

7. References

[1] W.C. McGee, "The information management system imv/vs", *IMB System Journal*, vol. 2, IBM, 1977, pp. 84-168.

[2] M. Astrahan, M. Blasgen, D. Chamberlin and et al, "System r: Relational approach to database management", *Transactions on Database Systems*, vol. 1, ACM, 1976, pp. 97-137.

[3] T.F. Knigh, "An architecture for mostly functional languages", *Proceedings of AMC Lisp and Functional Programming Conference*, ACM, 1986, pp. 500-519.

[4] A. Chang and M.F. Mergen, "801 storage: Architecture and programming", *ACM Transactions on Computer System (TOCS)*, vol. 6, ACM, 1988, pp. 28-50.

[5] M. Herlihy and J.E.B. Moss, "Transactional memory: Architectural support for lock-free data structures", *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, IEEE Computer Society, 1993, pp. 289-300.

[6] J.M. Stone, H.S. Stone, P. Heidelberger and et al, "Multiple reservations and the Oklahoma update", *IEEE Parallel and Distributed Technology*, 1(4), IEEE Computer Society, 1993, pp. 58-71.

[7] K.E. Moore, J. Bobba, M.J. Moravan and et al, "LogTM: Log-based transactional memory", *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA)*, IEEE Computer Society, 2006, pp. 254-256.

[8] C.S. Ananian, K. Asanovic, B.C. Kuszmaul and et al, "Unbounded transactional memory", *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE Computer Society, 2005, pp. 316-327.

[9] R. Rajwar, M. Herlihy and K. Lai, "Virtualizing transactional memory", *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, IEEE Computer Society, 2005, pp. 494-505.

[10] W. Chuang, S. Narayanasamy, G. Venkatesh and et al, "Unbounded page-based transactional memory", *ACM SIGPLAN Notices*, 41(11), ACM, 2006, pp. 347-358.

[11] L. Hammond, B.D. Carlstrom, V. Wong and et al, "Transactional coherence and consistency: Simplifying parallel hardware and software", *IEEE Micro*, 24(6), IEEE Computer Society, 2004, pp. 92-103.

[12] M. Moir, K. Moore and D. Nussbaum, "The adaptive transactional memory test platform: A tool for experimenting with transactional code for rock", *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT2008)*, ACM, 2008.

[13] M. Herlihy, V. Luchangco, M. Moir and et al, "Software transactional memory for dynamic-sized data structures", *Proceedings of the 22nd AMC Symposium on Principles of Distributed Computing*, ACM, 2003, pp. 92-101.

[14] T. Harris, "Exceptions and side-effects in atomic blocks", *Proceedings of the 2004 Workshop on Concurrency and Synchronization in Java programs*, 2004, pp. 46-53.

[15] K. Fraser and T. Harris, "Concurrent programming without locks", *Technical report*, Microsoft Research, 2003.

[16] Stanford Transactional Applications for Multi-Processing (STAMP), http://stamp.stanford.edu/

[17] J.W. Chung, H. Chafi, C.C. Minh and et al, "The common case transactional behavior of multithreaded programs", *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, IEEE Computer Society, 2006, pp. 266-277.

[18] J.M. Tendler, J.S. Dodson, J.S. Fields, and et al, "POWER4 system microarchitecture", *IBM Journal of Research and Development*, 46(1), 2002, pp. 5-25.

[19] Patrick Bohrer, Mootaz Blnozahy, Ahmed Gheith, et al, "Mambo: a full system simulator for the PowerPC architecture", *ACM SIGMETRICS Performance Evaluation Review*, 31(4), ACM, 2004, pp. 8-12.