Productivity and Performance: Improving Consumability of Hardware Transactional Memory through a Real-World Case Study

Huayong Wang, Yi Ge, Yanqi Wang, and Yao Zou

IBM Research - China
{huayongw,geyi,yqwang}@cn.ibm.com,frankfoxy@gmail.com

Abstract. Hardware transactional memory (HTM) is a promising technology to improve the productivity of parallel programming. However, a general agreement has not been reached on the consumability of HTM. User experiences indicate that HTM interface is not straightforward to be adopted by programmers to parallelize existing commercial applications, because of the internal limitation of HTM and the difficulties to identify shared variables hidden in the code. In this paper we demonstrate that, with well-designed encapsulation, HTM can deliver good consumability. Based on the study of a typical commercial application in supply chain simulations - GBSE, we develop a general scheduling engine that encapsulates the HTM interface. With the engine, we can convert the sequential program to multi-threaded model without changing any source code for the simulation logic. The time spent on parallelization is reduced from two months to one week, and the performance is close to the manually tuned counterpart with fine-grained locks.

Keywords: hardware transactional memory, parallel programming, discrete event simulation, consumability.

1 Introduction

Despite years of research, parallel programming is still a challenging problem in most application domains, especially for commercial applications. Transactional memory (TM), with hardware-based solutions to provide desired semantics while incurring the least runtime overhead, has been proposed to ameliorate this challenge. With hardware transactional memory (HTM), programmers simply delimit regions of code that access shared data. The hardware ensures that the execution of these regions appears atomic with respect to other threads. As a result, HTM allows programmers to enforce mutual exclusion as simple as traditional coarse-grained locks, while achieving performance close to fine-grained locks.

However, consumability of HTM programming model is still a point of controversy for commercial application developers, who have to take tradeoff between the cost of parallelization and the performance benefits. Better consumability means that applications can benefit from HTM in performance with less cost

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 163–174, 2010.

of modification and debugging. It's not easy to achieve good consumability due to two reasons. First, when parallelizing a sequential program, its source code usually contains a large quantity of variables shared by functions. Those shared variables are not indicated explicitly, nor protected by locks. To thoroughly identify these variables is a time consuming task. Second, the HTM programming model has limitations. Supporting I/O and other irrevocable operations (like system calls) inside transactions is expensive in terms of implementation complexity and performance loss. Many HTM researches adopt application kernels as benchmarks for performance evaluation. The benchmarks are designed for hardware tuning purpose and cannot reflect the consumability problem of HTM. How to improve the consumability of HTM is a practical problem at the time of HTM's imminent emergence in commercial processors.

Some research works have tried to provide friendly HTM programming interfaces [1]. By adding specific semantics into the HTM interfaces, it brings the new challenge on compatibilities of HTM implementations, which is a major concern for HTM commercial applications. It is more reasonable to solve the problem by combining two approaches: encapsulating general HTM interface by runtime libraries and taking advantage of application's particularities. In this paper a typical commercial application in supply chain simulation domain is studied as a running example. We parallelize it by two ways: HTM and traditional locks. Based on the quantitative comparison of the costs, we demonstrate that HTM can have good consumability with proper encapsulation for such kind of applications. This paper makes the following contributions:

1. It shows that our method can reduce the time of the parallelization work from two months to one week by using a new HTM encapsulation interface on the case study application. The method is also suitable to most real-world applications in the discrete event simulation domain.
2. Besides showing the improvement of productivity, we also evaluate the overall performance and analyze the factors that influence the performance. With HTM encapsulation, we can generally achieve 4.36 and 5.13 times speedup of 8 threads in two different algorithms. The result is close to the performance achieved through fine-grained locks.

The remainder of this paper is organized as follows. Section 2 provides background information on HTM and introduces the application studied in this paper. Section 3 describes the details of the parallelization work and explains how our method can help to improve the consumability. Section 4 presents and analyzes experimental results. Section 5 gives the conclusion.

2 Background

This section introduces HTM concepts as well as a case study application.

2.1 HTM Implementation and Interface

In order to conduct a fair evaluation, we choose a basic HTM implementation - Best Effort Transaction (BET), which was referred in many papers as a baseline

design [2],[3],[4]. BET uses data cache as buffers to save the data accessed by transactions. Each cache line is augmented with an atomic flag (A-flag) and an atomic color (A-color). The A-flag indicates whether the cache line has been accessed by an uncommitted transaction, and A-color indicates which transaction has accessed the cache line. When a transaction accesses a cache line, the A-flag is set; when a transaction is completed or aborted, the A-flag of each cache line that has been accessed by the transaction is cleared. When two transactions access a cache line simultaneously, and at least one of them modifies the cache line, one of the two transactions should be aborted to ensure transaction atomicity. This is called "conflict". If a transaction is aborted, all cache lines modified by the transaction need to be invalidated. Then, the execution flow jumps to a pre-defined error handler, in which the cleanup task can be performed before the transaction is re-executed.

From the programmers' perspective, this basic HTM exposes five primitives. TM_BEGIN and TM_END mark the start and end of a transaction. TM_BEGIN has a parameter "priority". It is used to avoid livelock. If a conflict happens, the transaction with lower priority is aborted. In this paper, the lower the value, the higher the priority. Therefore using time stamp as priority leads to the result that the older transaction aborts the younger in a conflict. TM_SUSPEND and TM_RESUME are two primitives used inside a transaction to temporarily stop and restart the transaction execution. In suspend state, the memory access operations are treated as regular memory access operations, except that conflicts with the thread's own suspended transaction are ignored. I/O and system calls are allowed in suspend state. If a conflict happens in suspend state and the transaction needs to be aborted, the cancelation is delayed until TM_RESUME is executed. The last primitive TM_VALIDATE is used in suspend state to check whether a conflict happens.

2.2 Discrete Event and Supply Chain Simulation

Discrete event simulation (DES) is a method to mimic the dynamics of a real system. The Parallel DES (PDES) in this paper refers to the DES parallelized by multiple threads on a shared memory multiprocessor platform, rather than the distributed DES running on clusters. The core of a parallel discrete event simulator is an event list and a thread pool processing those events. Each event has a time stamp and a handler function. Event list contains all unprocessed events, sorted by time stamps, as shown in Fig. 1. The main processing loop in a simulator repeatedly removes the oldest event from the event list and calls the handler function for the event. Thus, the process can be viewed as a sequence of event computations. When an event is being processed, it is allowed to add one or more events to the event list with time stamps in the future.

The principle of DES is to ensure that events with different time stamps are processed in time stamp order. This is worthy of extra precaution because each thread does not priori know whether a new event will be added later. Figure 1 demonstrates such a situation. Both threads fetch the event with the oldest time stamp to process. Since event B requires relatively longer processing time, thread

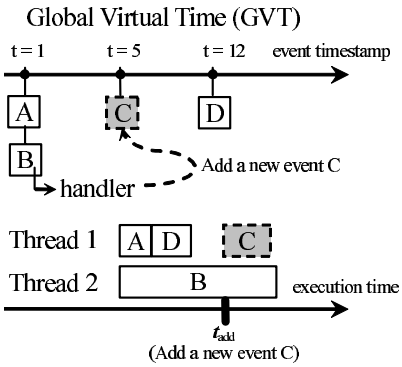

Fig. 1. Event list

1 fetches event D after it finished processing event A. However, event B adds a new event C to the event list at time t_{add}, which causes an out-of-order execution of events C and D. There are two kinds of algorithms to address the problem [5]: conservative and optimistic algorithms. Briefly, the conservative algorithm takes precautions to avoid the out-of-order processing. That is, each event is processed only when there is no event with a smaller time stamp. To achieve it, a Lower Bound on the Time Stamp (LBTS)is used in the conservative algorithm. In this paper, LBTS is the smallest time stamp in the event list. Events with time stamp equal to LBTS are safe to be processed. After all of them have been processed, LBTS is increased to the next time stamp in the event list. The optimistic algorithm, on the contrary, uses a detection and recovery approach. Events are allowed to be processed out of time stamp order. However, if the computations of two events conflicts, the event with larger time stamp must be rolled back and reprocessed.

Supply chain simulation is an application of DES. General Business Simulation Environment (GBSE) is a supply chain simulation and optimization tool developed by IBM [6]. It is a widely used commercial application and earned the 2008 Supply Chain Excellence Award, a top award in this domain. GBSE-C is the C/C++ version of GBSE. Besides the aforementioned features of DES, GBSE-C presents other properties to the parallelization work.

1. As a sequential program, it has a large amount of shared variables not protected by locks in event handlers. To parallelize it, all shared variables in the source code must be identified, which is a very time consuming task.
2. The number of events is large. Events with shared variables are probably processed at different time. Therefore the actual conflict rate between events is low.
3. For business reasons, source code of event handlers is frequently changed. Considering the code is developed by experts on supply chain, rather than experts on parallel programming, it is desirable to keep the programming style as simple as usual, i.e. writing sequential code as before, but achieving performance of parallel execution.

Using traditional lock-based technique to parallelize GBSE-C makes no sense because the business processing logic need to be modified. It's unbearable to depend on the business logic programmers, who have little knowledge about parallelization, to explicitly specify which parts should be protected by locks and which are not. On the contrary, our HTM-based approach only modifies the simulation service layer and is transparent to business logic programmers.

3 Using Transactions in GBSE-C

For the purpose of parallelization, GBSE-C can be deemed to be a two-layer program. The upper layer is the business logic, such as order handling process, inventory control process, and procurement process. These logics are implemented in the form of event handlers. The lower layer is the event scheduling engine, which is the core of the simulator. The engine consists of two modules:

1. The resource management module encapsulates commonly used functions, such as malloc, I/O, and operations to the event list. After encapsulation, those functions are made safe to be used inside transactions.
2. The scheduling management module is in charge of event scheduling and processing. Threads from a thread pool fetch suitable events from the event list and process them. The scheduling policy is either conservative or optimistic, which defines the event processing mechanism.

3.1 Resource Management

The resource management module includes a memory pool, an I/O wrapper API, and the event list interface.

Memory Pool. If a chunk of memory is allocated inside a transaction and the transaction is aborted, the allocated memory will never be released. Also memory allocation from a global pool usually requires accessing shared variables, and hence incurs conflict between transactions. To solve these problems, we implement a memory pool per thread. Inside a transaction, new functions TM_MALLOC and TM_FREE replace the original functions malloc and free. TM_MALLOC obtains a chunk of memory from the corresponding memory pool. Then it records the address and size of the memory chunk in a thread-specific table (MA table). If the transaction is aborted, the error handler releases all memory chunks recorded in the table; otherwise, the allocated memory is valid. TM_FREE returns the memory chunk to the pool and deletes the corresponding entry in the table.

I/O Wrapper. There are many disk and network I/O operations in GBSE-C. Functionally, they are used to access files, remote services, as well as for database operations. To simplify programming, these operations have already been encapsulated by helper functions. I/O operations inside a transaction are

re-executed if the transaction is re-executed. Based on whether an I/O operation can tolerate this side effect, they can be classified to two categories. The first category is idempotent, such as reading a read-only file and printing for debug purpose. It can be re-executed without harmful impact. The second category is non-idempotent, as for example appending a row of data to a database table. It cannot be executed multiple times. We have encountered a lot of cases in the first category. What we do is to add TM_SUSPEND and TM_RESUME at the start and end of those helper functions. The code in event handlers can remain unchanged if it calls the helper functions for these operations. The cases in the second category are more interesting. We have two approaches to handle these cases. First, we use I/O buffering, as described in previous work [7]. Briefly, we buffer the data of the I/O operations until the transaction finishes. If the transaction is committed, perform the I/O operations with the data in the buffer; otherwise, discard the data in the buffer. However, this method is inconvenient for some complex cases where the buffered I/O operation influences later operations in the transaction. We propose another method by adding a new flag "serial mode" to each event. If it is set, the event handler contains complex I/O operations that should be handled in traditional sequential mode, where only one event is processed at a time. After this event is processed, the scheduler resumes the parallel execution mode.

Event List Interface. Event handler may add a new event to the event list through the event list interface. The interface exposes a function ADD_EVENT for programmers. The function does not manipulate the event list immediately. Instead, it records the function call and the corresponding parameters in a thread-specific table (ELO table). After the transaction is committed, extra code after TM_END will execute the operations recorded in ELO table. If the transaction is aborted, the table is simply cleared.

3.2 Scheduling Management

In order to make TM programming transparent, the HTM interface is encapsulated in the event scheduler. Event handler programmers need not be aware of the HTM programming primitives. The engine supports both conservative and optimistic scheduling algorithms.It also has a special scheduling policy supporting conflict prediction. This policy is based on the conservative algorithm and needs change of HTM implementation.

The Conservative Algorithms. The main process loop in each thread in the conservative algorithm includes the following steps:

1. Fetch an event with the smallest time stamp in the event list.
2. If the time stamp of this event is larger than LBTS, return the event to the event list and wait until LBTS is increased. The check guarantees that an event will be processed only when there is no event with smaller time stamp.
3. Execute the event handler in the context of a transaction. After the transaction is committed, execute the delayed event list operations (if any), and clear both ELO and MA tables.

4. If all events with time stamp equal to LBTS have been processed, LBTS is increased to the next time stamp in the event list. After that, a notification about the LBTS change is sent to all other threads. The threads blocked on LBTS then wake up.

In the conservative algorithm, threads only process events with the same time stamp (LBTS). Thread pool will starve if the number of events with time stamp equal to LBTS is small. The parallelism of the conservative algorithm is limited by event density - the average number of events with each time stamp in the event list. In addition, the barrier synchronization to prevent out-of-order event processing might be costly when the event processing time is disproportional. Some threads might go idle while others are processing events that take a long execution time. Both degrade the performance of the conservative algorithm.

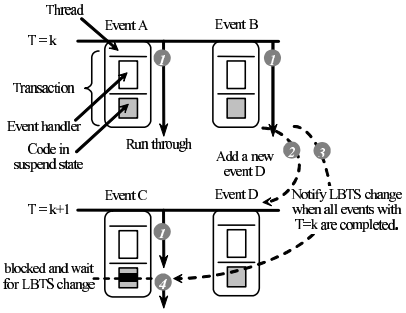

Fig. 2. An example of the optimistic algorithm

The Optimistic Algorithm. The optimistic algorithm overcomes the shortcomings of the conservative algorithm since it allows out of time stamp order execution. However, in traditional implementations, the optimistic algorithm does not necessarily bring optimized performance in a general sense, because it incurs overhead for checkpoint and rollback operations [8]. Using HTM, that overhead is minimized since checkpoint and rollback are done by hardware. The main process loop in the optimistic algorithm includes following steps:

1. Fetch an event with the smallest time stamp in the event list.
2. Execute the event handler in the context of a transaction. In this case the transaction priority is equal to the time stamp.
3. After the execution of the event handler, suspend the transaction and wait until the event's time stamp is equal to LBTS. During this period, the handler periodically wakes up and checks whether a conflict is detected. If so, abort the transaction and re-execute the event handler.
4. After the transaction is committed, execute the delayed event list operations (if any), and clear both ELO and MA tables.
5. If all events with time stamp equal to LBTS have been processed, LBTS is increased to the next time stamp in the event list. After that a notification

about LBTS change is sent to all other threads. The threads blocked on LBTS then wake up.

Figure 2 shows an example of the optimistic algorithm. At the beginning, there are three events (A, B, and C) in the event list and LBTS is equal to k. The three events are being concurrently processed by three separate threads. Since the time stamp of event C is larger than LBTS, event C is blocked and put into the suspend state. The thread releases the CPU during the blocking. Meanwhile, event B adds a new event D into the event list with time stamp k+1 by calling function ADD_EVENT. After event A and B are finished, LBTS is increased to k+1, and a notification about LBTS change is sent to event C. The execution of event C is woken up, and the corresponding transaction is going to be committed soon. Event D can also be processed whenever there is a free thread in the thread pool. If conflict happens between event A and C, event C is aborted since its priority is low. From the example, we can see that although events are processed out-of-order, they are committed strictly in order, which guarantees the correctness of the simulation. Besides the low cost of checkpoint and rollback, HTM-based optimistic algorithm has another advantage over traditional implementations: fine-grained rollback. In HTM-based implementation, only those events really affected are rolled back since each transaction has its own checkpoint.

The optimistic algorithm might suffer from overly optimistic execution, i.e., some threads may advance too far in the event list. The results are two-fold. First, conflict rate balloons with the increase of the number of events being processed. Second, overflow might happen when a lot of transactions are concurrently executed. Both results limit the parallelism of the optimistic algorithm.

Scheduler with Conflict Prediction. The performance of the case study application is limited by the high conflict rate when the thread number is large. Without appropriate event scheduling mechanisms, a large thread pool may consume more on-chip power without performance improvement, or even causing performance degradation. In our engine, the event scheduler supports a special policy to predict the conflict between events with the help of HTM. The prediction directs each thread to process suitable events to avoid unnecessary conflicts. The conflict prediction is feasible based on the following observations.

- Data locality. In PDES applications, executions of one event show data locality. The memory footprints of the previous executions give hints about the memory addresses to be accessed in the future.
- Conflict position. Some event handlers are composed with a common pattern: first read configuration data, then do some computation, and finally write back computing results. Positions of conflicts caused by shared variables accessing are usually at the beginning and end of the event handler. The time span of transaction execution between conflict position and the end of the transaction is highly relevant to the conflict probability.

In order to design the scheduler with conflict prediction, we modify the HTM implementation by adding two bloom filters in each processor to record the

transaction's read and write sets. When the transaction is committed, the contents of bloom filters called signatures are dumped into memory at pre-defined addresses. When the transaction is aborted, the bloom filters are cleared and the conflicting address is recorded. Each event maintains a conflict record table (CRT) for the conflict prediction, which contains the signatures and other transaction statistics. Some important fields of the statistics are described as follows:

- *Total Execution Cycles (TEC):* total execution cycles of a transaction between TM_BEGIN and TM_END.
- *Conflicting Addresses (CA):* conflicting addresses of a transaction.
- *Conflicting Execution Cycles (CEC):* execution cycles of a transaction between TM_BEGIN and conflict.

Before an event is going to be processed, the scheduler first checks whether the event's conflict addresses are in the signatures of any event under execution. If so, a conflict is possible to happen if this event is executed. Then the scheduler uses possible conflict time-span (PCT) as a metric to determine the conflict probability. PCT refers to the total time span within which if one event with lower priority starts to execute, it will be aborted by the other running event with high priority. The smaller the PCT is, the less probably the conflict occurs. The value of PCT between event A and event B can be computed according to Eq. 1.

$$PCT_{AB} = (TEC_A - CEC_A) + (TEC_B - CEC_B) \tag{1}$$

Fig. 3 shows three cases of transaction conflict between event A and B. In the first case, two transactions have conflicting memory accessing at the very beginning, the transaction with lower priority could be possibly aborted if it starts to execute at any time within the time span from point M to N. It has the largest PCT ($PCT_{max} \approx TEC_A + TEC_B$); in the second case, conflict position is located in the middle and conflict probability is lower than the first one with $PCT_{mid} \approx 1/2(TEC_A + TEC_B)$; while it's seldom to conflict in the third case with $PCT_{min} \approx 0$. Based on CRT and PCT, the scheduler can predict conflict and decide the scheduling policy for each event.

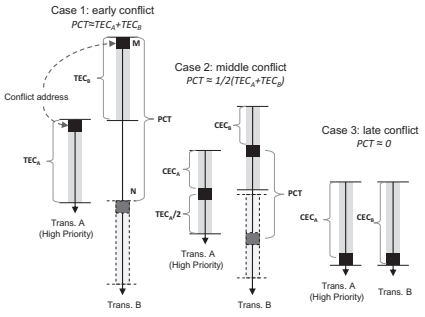

Fig. 3. Three cases with different PCT

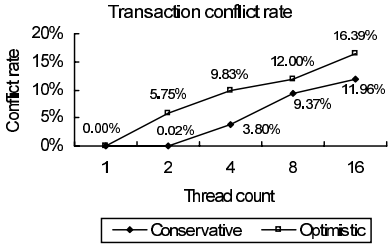

Fig. 4. Transaction conflict rate

4 Productivity and Performance Evaluation

We have parallelized GBSE-C through two approaches: the new approach based on HTM and the traditional approach based on locks. Since the work was done by the same group of developers, the time spent implementing the different approaches can be considered as a straightforward indication of productivity.

- HTM-based approach. We spent about one week to finish the parallelization work for both conservative and optimistic algorithms. Three days were used to identify I/O operations and handle them accordingly. It is not very difficult because the I/O operations are well encapsulated by helper functions. Additional time was used to encapsulate the HTM interface in the scheduling engine.
- Lock-based approach. We spent two months to complete the parallelization work for the conservative algorithm. About half of the time was used to identify the variables shared by events. This task is time-consuming because understanding the program logic in event handler requires some domain knowledge and the usage of pointers exacerbates the problem. Performance tuning and debugging took another three weeks. It includes shortening critical sections (fine-grained locks), using proper locks (pthread locks vs. spinning locks), and preventing dead locks. Since the execution order of events with same time stamp is not deterministic, some bugs were found only after running the program many times. We did not implement the optimistic algorithm using locks because doing checkpoint and rollback by software means is difficult in general.

The performance evaluation is carried out on an IBM full system simulator [9], which supports configurable cycle-accurate simulation for POWER/PowerPC based CMP and SMP. The target processor contains 4 clusters connected by an on-chip interconnect. Each cluster includes 4 processor cores and a shared L2 cache. Each core runs at a frequency of 2GHz, with an out-of-order multi-issue pipeline.

We have studied the transaction size in this application. The sizes are measured separately for the read set and the write sets. Read and write sets contain the data read and written by a transaction respectively. 88% of the read sets and 91% of the write sets are less than 4KB, indicating that most transactions are small. The sizes of the read and write sets in the largest transaction approach 512KB. Since the L2 data cache (2MB) is much larger, the execution of a single transaction dose not incur overflow.

Figure 4 shows the transaction conflict rate in conservative and optimistic algorithms. The conflict rate is defined as the number of conflict divided by the total number of transactions. Optimistic algorithm has higher conflict rate than the conservative algorithm since the optimistic algorithm tries more paralleling execution. When the number of threads in thread pool is increased, the conflict rate is also increased. Generally, the conflict rate is less than 24%. It proves that many events are actually independent and can be processed in parallel.

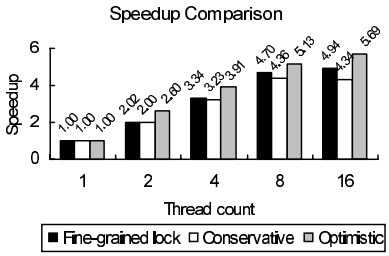

Fig. 5. Speedup from three algorithms

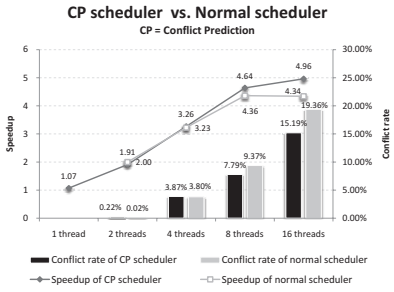

Fig. 6. Speedup of scheduler with conflict prediction

Figure 5 illustrates the observed speedup from the three approaches: the conservative algorithm with fine-grained locks, the conservative algorithm with HTM and the optimistic algorithm with HTM. The first approach represents the best performance achievable through manual optimization. The conservative algorithm with HTM achieves performance that is slightly lower than the first approach, indicating HTM is effective for the parallelization work. The optimistic algorithm with HTM is better than the previous two approaches. It gains more than 2 times speedup when the thread count is 2. This is because optimistic algorithm reduces the synchronization overhead at each time stamp and the scheduler has more chances to balance workloads among threads. Generally, through HTM, we can achieve 4.36 and 5.13 times speedup with 8 threads in the conservative and optimistic algorithms respectively, and achieve 5.69 times speedup with 16 threads in the optimistic algorithm.

Besides the performance comparison with the normal event scheduler, we also conduct an experiment to evaluate the performance of the scheduler with conflict prediction. Fig. 6 illustrates the speedup and conflict rate comparison between the two schedulers with conservative algorithm. Because of extra overhead of the conflict prediction, the scheduler with conflict prediction shows a slightly worse performance against the normal one when the thread count is low. But as the thread count increases, it gradually outperforms the normal one and the performance gap becomes larger. With 16 threads, the speedup is 14% more than the normal one. We can also see that the conflict rate is reduced by scheduler with conflict prediction.

5 Conclusion

In this paper, we demonstrate that the encapsulated HTM can have good consumability for some real-world applications. Based on these findings in the paper, we will further investigate the consumability of HTM in a broader range of applications in the future.

References

1. McDonald, A., Chung, J., Carlstrom, B.D., Minh, C.C., Chafi, H., Kozyrakis, C., Olukotun, K.: Architectural semantics for practical transactional memory. In: Proc. of the 33rd International Symposium on Computer Architecture, pp. 53–65. IEEE, Los Alamitos (2006)
2. Wang, H., Hou, R., Wang, K.: Hardware transactional memory system for parallel programming. In: Proc. of the 13th Asia-Pacific Computer System Architecture Conference, pp. 1–7. IEEE, Los Alamitos (2008)
3. Baugh, L., Neelakantam, N., Zilles, C.: Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In: Proc. of the 35th International Symposium on Computer Architecture, pp. 115–126. IEEE, Los Alamitos (2008)
4. Chung, J., Baek, W., Kozyrakis, C.: Fast memory snapshot for concurrent programming without synchronization. In: Proc. of the 23rd International Conference on Supercomputing, pp. 117–125. ACM, New York (2009)
5. Perumalla, K.: Parallel and distributed simulation: traditional techniques and recent advances. In: Proc. of the 2006 Winter Simulation Conference, pp. 84–95 (2006)
6. Wang, W., Dong, J., Ding, H., Ren, C., Qiu, M., Lee, Y., Cheng, F.: An introduction on ibm general business simulation environment. In: Proc. of the 2008 Winter Simulation Conference, pp. 2700–2707 (2008)
7. Chung, J., Chafi, H., Minh, C., McDonald, A., Carlstrom, B., Kozyrakis, C., Olukotun, K.: The common case transactional behavior of multithreaded programs. In: Proc. of the 12th International Symposium on High-Performance Computer Architecture, pp. 166–177. IEEE, Los Alamitos (2006)
8. Poplawski, A., Nicol, D.: Nops: a conservative parallel simulation engine for ted. In: Proc. of the 12th Workshop on Parallel and Distributed Simulation, pp. 180–187 (1998)
9. Bohrer, P., Peterson, J., Elnozahy, M., Rajamony, R., Gheith, A., Rochhold, R.: Mambo: a full system simulator for the powerpc architecture. ACM SIGMETRICS Performance Evaluation Review 31(4), 8–12 (2004)