J Inf Process Syst, Vol.14, No.2, pp.499~509, April 2018
https://doi.org/10.3745/JIPS.01.0026

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Efficient Hybrid Transactional Memory Scheme using Near-optimal Retry Computation and Sophisticated Memory Management in Multi-core Environment

Yeon-Woo Jang*, Moon-Hwan Kang*, and Jae-Woo Chang*

Abstract

Recently, hybrid transactional memory (HyTM) has gained much interest from researchers because it combines the advantages of hardware transactional memory (HTM) and software transactional memory (STM). To provide the concurrency control of transactions, the existing HyTM-based studies use a bloom filter. However, they fail to overcome the typical false positive errors of a bloom filter. Though the existing studies use a global lock, the efficiency of global lock-based memory allocation is significantly low in multi-core environment. In this paper, we propose an efficient hybrid transactional memory scheme using near-optimal retry computation and sophisticated memory management in order to efficiently process transactions in multi-core environment. First, we propose a near-optimal retry computation algorithm that provides an efficient HTM configuration using machine learning algorithms, according to the characteristic of a given workload. Second, we provide an efficient concurrency control for transactions in different environments by using a sophisticated bloom filter. Third, we propose a memory management scheme being optimized for the CPU cache line, in order to provide a fast transaction processing. Finally, it is shown from our performance evaluation that our HyTM scheme achieves up to 2.5 times better performance by using the Stanford transactional applications for multi-processing (STAMP) benchmarks than the state-of-the-art algorithms.

Keywords

Bloom Filter, Concurrency Control, Hybrid Transactional Memory, Multi-core in-Memory Databases

1. Introduction

A lock is a well-known synchronization mechanism being used for a shared memory in multi-thread programing. However, since developing a software that efficiently uses a lock is notoriously challenging, transactional memory (TM) [1] has been proposed as an attractive alternative to lock-based synchronization. Unlike the lock-based approaches, which requires programmers to identify shared data and how to synchronize concurrent accesses to it, the TM-based mechanism only requires users to identify the code that have to be executed atomically [2].

Typically, TM system is classified into three categories: hardware transactional memory (HTM), software transactional memory (STM) and hybrid transactional memory (HyTM). First, HTM, i.e., Intel's TSX, has been developed to manage conflicts between transactions on multi threads with cache

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.
Manuscript received June 23, 2017, first revision August 24, 2017; accepted October 23, 2017.
Corresponding Author: Jae-Woo Chang (jwchang@jbnu.ac.kr)
* Dept. of Information and Technology, Chonbuk National University, Jeonju, Korea ({kp7050, calvin, jwchang}@jbnu.ac.kr)

Copyright© 2018 KIPS

coherence protocol. Although HTM can guarantee a similar performance to a well-designed lock-based parallel processing method, it has a critical shortcoming that the size of a hardware transaction is limited. For the Haswell architecture, it is limited to the size of the L1 cache (32 kB). This implies that it is impossible to execute a database transaction as one HTM transaction. Secondly, STM has demonstrated the simplicity of transactional programming. However, it shows poor performance with low- or medium-sized threads, compared to the fine-grained locking techniques. Finally, HyTM has been intensively studied to integrate the advantages of HTM and STM. Since HyTM provides a software fallback path in case of HTM abortion, it can integrate the HTM's performance and the scalability of STM. To provide the efficient concurrency control of transactions, the existing HyTM-based studies use a bloom filter. However, they fail to overcome the typical false positive errors of a bloom filter. On the other hand, the existing studies use a global lock to provide an efficient memory allocation of transactions. However, the efficiency of global lock based memory allocation is significantly decreased in multi-core environment.

To solve the problems, we propose an efficient hybrid transactional memory scheme using near-optimal retry computation and sophisticated memory management in order to efficiently process transactions in multi-core environment. First, we propose a near-optimal retry computation algorithm that provides an efficient HTM configuration using machine learning algorithms, according to the characteristic of a given workload. Second, we provide an efficient concurrency control for transactions in different environments by using a sophisticated bloom filter. For this, we make use of NOrec STM [3] that is well-known for its high performance among the existing STMs. Finally, we propose a memory management scheme being optimized for the CPU cache line so as to provide a fast transaction processing. That is, our hybrid transactional memory scheme assigns small-size transactions to a local cache in order to shorten the overall processing time.

The rest of this paper is organized as follows. Section 2 introduces the related work. In Section 3, we propose a memory management based hybrid transactional memory scheme. Section 4 presents our performance analysis using STAMP benchmark. Finally, Section 5 concludes the paper with future research directions.

2. Related Work

Intel has introduced their instruction set for x86 with Transactional Synchronization Extensions (TSX), which represents the first generation of mainstream and commodity HTM. TSX was released as part of the 4th generation of core processors (Haswell family). TSX has two main interfaces, called hardware lock elision (HLE) and restricted transactional memory (RTM). Especially, RTM represents the first generation of mainstream hardware transactional memory (HTM), which contributes to a blaze of publicity of HTM-based transaction processing techniques. However, one important characteristic of Haswell's RTM is its best effort nature. That is, it provides no transactional forward progress guarantees because TSX gives no guarantees as to whether transactions will ever commit in hardware due to inherent architectural limitations, even in absence of conflicts. As such, programmers must provide a software fallback path when issuing a begin instruction, in which they must decide what to do upon the abort of a hardware transaction. Thus, HTM-based transaction processing techniques should consider a software fallback handler used in conjunction with RTM, so as to ensure the completion of transaction

execution.

Since a retry count affects much on the overall processing costs for processing HTM transactions, Diegues and Romano [4] proposed the self-tuning technique, which focuses on the problem of automatically tuning the policies used to regulate the activation of the fallback path of TSX. They use the multi-armed [5] and the gradient descent exploration [6] to determine a retry count. The gradient descent exploration is used to optimize the allocation of the count of attempts while the multi-armed bandit is used to optimize the exhausting of the retry count. In case of a capacity abort, a rule is activated when the count of the attempts is exhausted based on the characteristic of transactions. That is, more attempts are explored if the upper confidence bounds belief is stubborn. Otherwise, less attempts are explored. However, the self-tuning scheme has a critical problem that it only considers the latest transaction's information, especially CPU clock, to choose a retry count. This causes an imprecise decision on the retry count for incoming transactions.

The hybrid TM (HyTM) scheme uses HTM to handle small transactions and also uses STM as an alternative path of HTM for large transactions. This combination achieves high scalability for large transactions, which HTM cannot support. One of the most promising HyTMs is Hybrid NOrec [7], which is a hybrid version of HTM and the NOrec STM [3]. The Hybrid NOrec uses a snapshot based lock-free scheme for concurrency control of HTM and STM. The Hybrid NOrec focused on the problem of a consistent concurrency control that is caused by the difference in processing speed between HTM and STM. To solve this problem, the Hybrid NOrec manages a metadata for STM to prevent unnecessary aborts of HTM. Moreover, when a conflict is occurred between HTM and STM, the Hybrid NOrec prioritizes HTM to commit and aborts STM. However, the Hybrid NOrec has the following limitations. First, it handles transactions without a global single lock, so a starvation can occur if there is a continuous conflict. Second, it uses a bloom filter for the concurrency control of HTM and STM, but does not consider the typical false positive problem of a bloom filter. Finally, it does not support an optimized memory management, so the efficiency of the algorithm is greatly decreased when the number of threads increases.

3. Efficient Hybrid Transactional Memory Scheme

3.1 Overall System Architecture

To resolve the aforementioned problems of the existing hybrid TM systems, we propose a memory management-based hybrid transactional scheme to efficiently process transactions in multi-core environment. The overall system architecture of the proposed scheme is shown in Fig. 1. Our hybrid TM scheme consists of three transaction processing components, in order to guarantee both a high commit rate and a high usability for different types of transactions. The first component is the LiteHTM, that is a light version of HTM. The remaining two components are the NOrec STM and the global single lock. When a transaction(Tcur) starts, the HTM executor in LiteHTM prepares for processing Tcur on HTM (①) and the memory allocator(MA) allocates memory for Tcur (②). When Tcur is executed using HTM, the number of optimal retries (i.e., optimal threshold) is given from (③). If any transaction is executed on STM, the Concurrency Manager performs a validation check between Tcur and the one on STM (④). Tcur is executed using HTM as many times as retry threshold (⑤).

During the execution of Tcur, the Retry handler keeps track of the number of current retires (⑥). If Tcur is not committed on HTM within the number of retries, it is forwarded to the NOrec STM as the fallback path of HTM (⑦). Also, the memory for Tcur on STM is allocated (⑧). If the Tcur on the NOrec STM again fails to commit within the retry threshold (⑨), it is processed by using the Global Single Lock that guarantees the commit of Tcur (⑩).

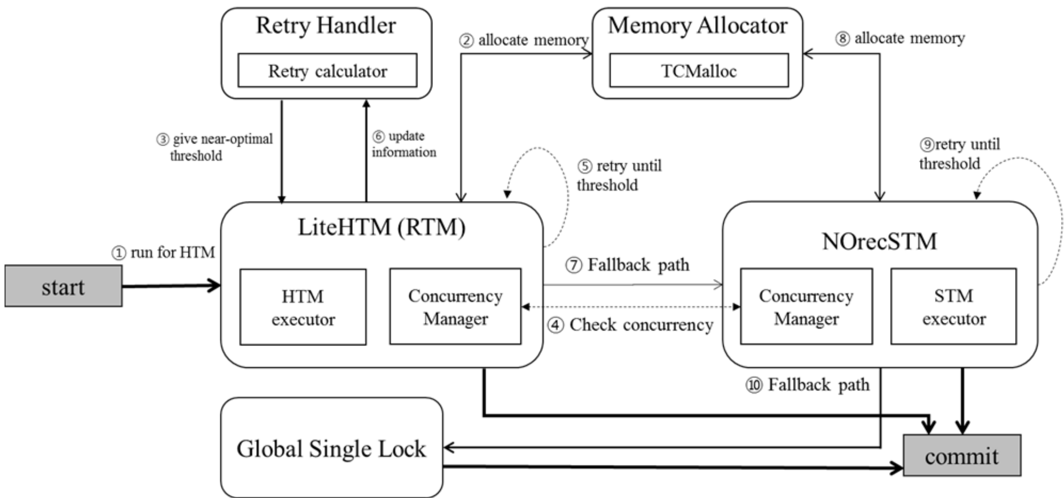

Fig. 1. System architecture of our hybrid TM scheme.

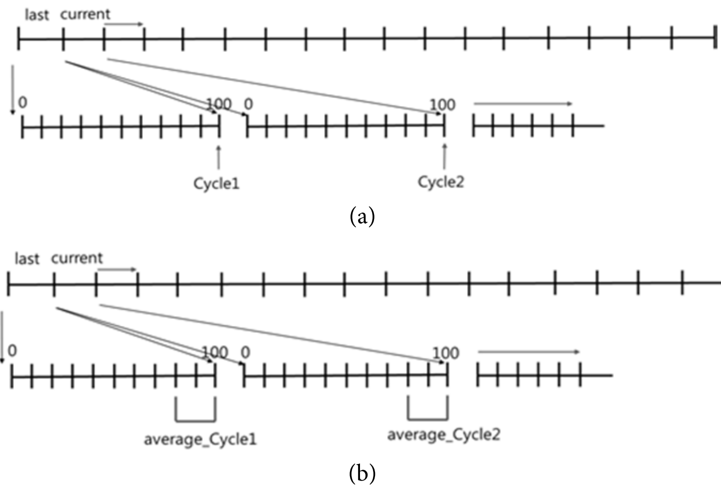

Fig. 2. Computation of the number of retries. (a) Retry computation in the existing self-tuning scheme and (b) retry computation in our adaptive algorithm.

3.2 Retry Handler

Although HTM supports efficient transaction processing, its performance degradation can be caused by unexpected aborts from OS. For this, a retry handler computes the optimal number of retries so that the failed transactions can be re-executed. Because the existing self-tuning scheme (ST-HTM) computes

the best configuration from the only last transactions, it cannot get the optimal number of retries for the incoming transaction. To resolve the problem, we propose a near-optimal retry computation algorithm that analyzes enough samples from the previously executed transactions. Fig. 2 shows the computation of the number of retries from both the existing self-tuning scheme and our near-optimal retry computation algorithm. ST-HTM newly computes the number of reties as a unit of 100 transactions by calculating the spent CPU clocks between the last transaction of the last unit and the last transaction of the current unit (Fig. 2(a)). As a result, ST-HTM may fail to find the best configuration for a new transaction because it fully depends on the last transactions of two units. On the contrary, our near-optimal retry computation algorithm does sampling from a unit of 100 transactions and computes the number of reties by calculating the spent CPU clocks between the sampled transactions of the last unit and the sampled ones of the current unit (Fig. 2(b)). For this, we set the sampling rate as 20% because we can get the best performance with that rate.

To compute the near-optimal number of retries, our retry computation algorithm maintains both the number of retries and the number of CPU clocks spent for the sampled transactions. With this information, our algorithm can calculate the near-optimal number of retries, i.e., the best threshold, by using Eq. (1), where n is the number of sampled transactions and retryNum[i] is the number of retries used for transaction i. For example, when we assume that n is 5 among 100 transactions (1 unit) and the number of retries are 2 for a transaction T3, 5 for T10, 6 for T25, 4 for T61, and 20 for T78, the best threshold is calculated as (3+5+6+4+20)/5=7.4 ≈7.

$$bestThres = \lfloor AVG(\sum_{i=0}^{n} retryNum[i]) \rfloor \quad (1)$$

Our retry computation algorithm can estimate the near-optimal number of retries by considering the characteristic of incoming transactions. Our algorithm makes use of the gradient descent algorithm that is widely used for deep learning applications [6]. The gradient descent algorithm computes a slope by differentiating data and finds the optimal solution based on the slope. Using it, our algorithm can compute the near-optimal number of retries for the next transaction by maintaining the number of retries for both the current and the previous transaction. However, the gradient descent algorithm has a critical problem that it fails to find the global minima when the standard deviation of data is high. For this case, our near-optimal retry computation algorithm adopts a momentum algorithm to remedy the weakness of the gradient descent algorithm. The momentum algorithm is a popular deep learning one that can escape from local minima by permitting a drastic change of a slope by adding a certain ratio of the previous slope [8]. However, the momentum algorithm does not find the optimal number of retries when the slope is very high because it cannot escape from local minima. In this case, our retry computation algorithm randomly selects the number of retries by computing the number of both capacity aborts (ab_cap) and other aborts (ab_other). The random number of retries (Rand_retries) is calculated by using Eq. (2). Here maxTries and curTries mean the maximum number of retries for all the previous transactions and the number of retries configured for the current transaction, respectively.

$$Rand_retries = \frac{(ab_{other})}{(ab_{cap})+(ab_{other})}(maxTries - curTries) \quad (2)$$

Our near-optimal retry computation algorithm is processed as follows (Fig. 3). First, if the last slop of the number retries between the last two transactions (i.e., last slop) is greater than their current slop

between the last transaction and the current one (i.e., current slop), it computes the number of retries by using the gradient descent algorithm (line 4–5). If the last slop is greater than the addition of the current slop and m*w(last) where m is a momentum and w(last) is the weight updated between the last two transactions, our retry computation algorithm computes the number of retries by using the momentum algorithm (line 6–7). Otherwise, it randomly selects the number of retries (line 8–9). If AvgLoss is greater than a given threshold ϴ, the algorithm sets the number of retries as bestThres (line 10–11). Here we set ϴ as 40% because we can get the best performance with 40% from our performance evaluation. The bestThres is updated continuously while transactions are executed (line 12–16).

Algorithm Adaptive retry computation algorithm

```
1:  int RetryThreshold[Max]                    // Max = Number of samples
2:  if(number of transaction % 100 ==0)     // When we meet hundred transactions, we measure the
                                                     average value of samples
3:      AvgLoss = Average Losses of the sample transactions      // 20 samples are selected
4:  if (last > current && last ≤ current+(m*w(last))
5:           Ta->retry = Gradient_Decsendent()
6:  else if(last > current && last > current+(m*w(last))
7:           Ta->retry = Momentum()
8:  else
9:     Ta->retry = Rand_retries()
10: if (AvgLoss > □)
11:      Ta->retry = bestThres
12: if(RetryThreshold[i] != Ta->retry)          // We check if there is a duplication in the array
13:       RetryThreshold[++i] = Ta->retry       // In case of no duplication, we add it into the array
14: for(i=0..Max)
15:           Sum += RetryThreshold[i]
16: bestThres = Sum/Max
```

Fig. 3. Our near-optimal retry computation algorithm.

3.3 Memory Allocator

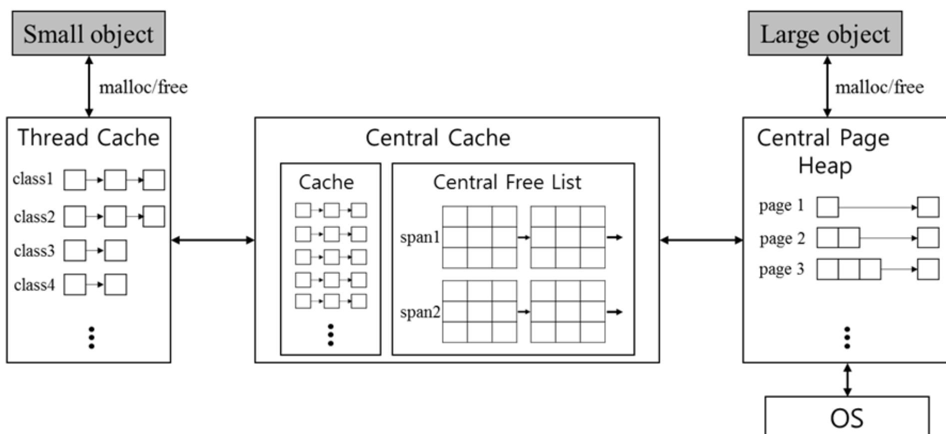

Fig. 4. Memory management of our TCMalloc-based memory allocator.

In a single-core environment, the memory manager allocates a shared memory by using one memory pool. In a multi-core environment, a lock is used to prevent data conflicts being caused from the single memory pool usage for multiple threads. However, various conflicts can occur, such as data conflict, thread latency and priority problem. Consequently, recent studies focus on an efficient memory management technique in multi-core environments [9]. The TCMalloc [10] is a lock-free memory manager that effectively manages a memory pool in a multi-core environment. The shared central cache is used for large-sized data while a local cache in each thread is used for small-sized data. Therefore, the amount of requested space is properly allocated to both local and central caches according to the size of data objects. As a result, the capacity abort caused by the cache size limit can be reduced while executing transactions using HTM. The structure of the TCMalloc is shown in Fig. 4. When an object size is less than 32 kB, our TCMalloc-based memory manager assigns the object to the local cache because a thread requires a small memory area. This can improve the proximity between a data object and a local cache. When the object size is larger than 32kb, it is assigned to the central global cache containing central free list arrays. The central page heap consists of 256 consecutive pages, each of which is divided into a series of small same-sized objects. For example, one page (4 kB) can be divided into 32 objects of 128 bytes each.

3.4 Bloom Filter Technique

A bloom filter is a probabilistic data structure being used to check whether an element belongs to a user specified set. An example of a bloom filter is shown in Fig. 5. Each element in a set{x, y, z} is hashed into a bit array as indicated with the colored arrows. On the other hand, the element w is not belong to the set{x, y, z}, because it is hashed into one of the bit-array position with 0 value. The bloom filter does not have false-negative in a result set that causes a critical reliability issue. However, since a false-positive data can be included in the result, it is important to reduce the size of false-positives.

To solve this problem, we propose a sophisticated bloom filter to reduce false-positives as shown in Eq. (3). Our method flexibly adjusts both of the size of a bloom filter and the number of hash functions by considering the user-defined error rate of the result. In Eq. (3), k is the number of hash functions, n is size of a set and m is the length of the bit-array for a bloom filter. For example, if there are 10,000 transaction operations and the error rate is set to 1%, the size of the bloom filter can be calculated as shown in Eq. (4).

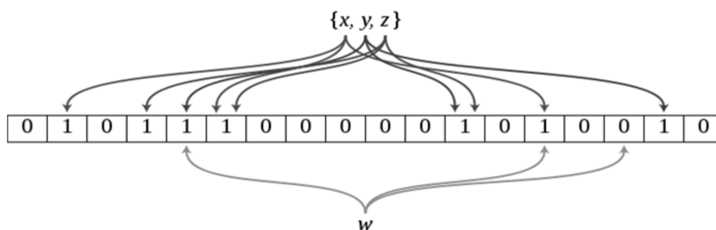

Fig. 5. An example of a bloom filter.

$$False\ Poisitive \approx \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-\frac{kn}{m}}\right)^{k}$$

$$Optimal\ of\ has\ functions\ (\mathrm{k}) \approx ln2\frac{m}{n} \approx 0.7\frac{m}{n} \tag{3}$$

$$10{,}000\,operations, 1\%\ error\ rate, bit\ size = 10$$

$$m = 10{,}000\ \times 10\ bits = 12kb\ of\ memory$$

$$k = 0.7\ \times \frac{120{,}000}{10{,}000} = 7\ hash\ functions \tag{4}$$

Our hybrid transactional memory scheme uses a bloom filter to detect the conflict of transactions that are concurrently executed. Our bloom filter-based concurrency control is performed as shown in Fig. 6. First, transactions that are currently executed on HTM or STM are recorded in the bloom filter. Second, the bloom filter can check for conflicts between transactions to provide the concurrency control of transactions. Finally, if data conflict occurs, the corresponding transactions become in the wait states and the concurrency manager of transaction memory components validates conflicts between transactions. Since our hybrid transactional memory scheme makes use of a bloom filter, it improves an overall performance by reducing a false-positive in conflict detection among transactions as well as by not using unnecessary memory space.

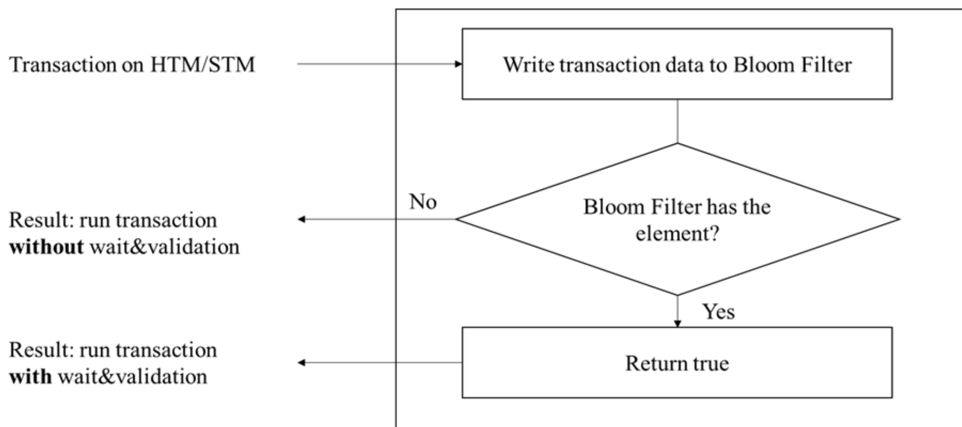

Fig. 6. Bloom filter-based concurrency control flow.

4. Performance Analysis

In this section, we present the extensive experimental analysis of our memory management-based hybrid transactional memory scheme. We compare our hybrid transactional memory scheme with the existing Hybrid NOrec, in terms of transaction processing time. We ran our experiment by using a TSX enabled Intel Haswell Xeon E3-1220v3 processor 3.10 GHz with 32 GB RAM and 8 virtual cores (4 physical cores, each one running up to 2 threads). We also ran our experiments in a machine running on Ubuntu 12.04 and the results are reported as the average value of 20 runs. For performance comparison, we make use of the standard STMAP suite, which is a popular set of TM benchmarks [11] that encompass applications and represent various domains for generating heterogeneous workload.

Fig. 7 show the speedup of both our hybrid transactional memory scheme and the Hybrid NOrec over HTM, in terms of the transaction processing time with varying number of threads. In case of 8 threads, our hybrid transactional memory scheme shows 240% better performance than HTM and 45%

better performance than Hybrid NOrec. It is noted that, in case of the SSCA2 benchmark, HTM shows better performance than the others when the number of thread is less than 4. This is because SSCA2 benchmark contains only small-sized read/write data set. However, our hybrid transactional memory scheme shows better performance as the number of threads increases. In case of KMEANS benchmark, the speed up of the Hybrid NOrec is drastically decreased, while our hybrid transactional memory scheme provides moderate performance. This is because our scheme efficiently solves the false-positive problem of a bloom filter by using the memory management-based concurrency control.

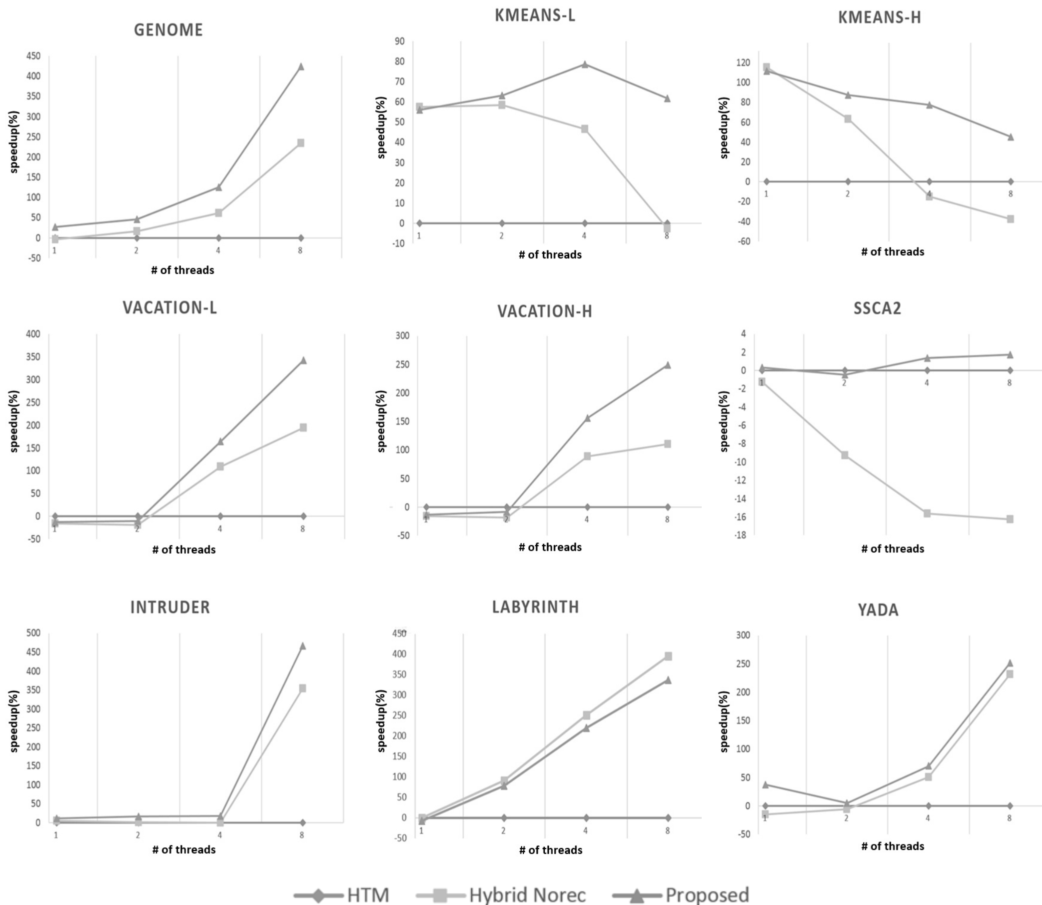

Fig. 7. Speedup over HTM using STAMP Benchmark.

5. Conclusions

In this paper, we proposed an efficient hybrid transactional memory scheme using near-optimal retry computation and sophisticated memory management so as to efficiently process transactions in multi-core environment. For this, we proposed a near-optimal retry computation algorithm providing an efficient HTM configuration using machine learning algorithms according to the characteristic of a given workload. In addition, our hybrid transactional memory scheme improves transaction processing

performance since it can minimize the number of false-positives by using a bloom filter. At last, our hybrid transactional memory scheme provides an optimized memory management for transactions in a multicore environment. We showed from our performance analysis that our hybrid transactional memory scheme outperformed the HTM by 240% and the existing Hybrid NOrec by 45%. As a future work, we plan to perform an extensive performance analysis with long-sized transactions by using more than 8 threads.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0113-16-0005, Development of an Unified Data Engineering Technology for Largescale Transaction Processing and Real-time Complex Analytics). Also, this research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant No. 2016R1D1A3B03935298).

References

[1] M. Herlihy and J. E. B. Moss, “Transactional memory: architectural support for lock-free data structures,” in *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, 1993, pp. 289-300.

[2] V. Pankratius and A. R. Adl-Tabatabai, “A study of transactional memory vs. locks in practice,” in *Proceedings of the 23rd Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, 2011, pp. 43-52.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott, “NOrec: streamlining STM by abolishing ownership records,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, 2010, pp. 67-78..

[4] N. Diegues and P. Romano, “Self-tuning intel transactional synchronization extensions,” in *Proceedings of the 11th International Conference on Autonomic Computing*, Philadelphia, PA, 2014, pp. 209-219.

[5] J. Vermorel and M. Mohri, “Multi-armed bandit algorithms and empirical evaluation,” in *European Conference on Machine Learning*. Heidelberg: Springer, 2005, pp. 437-448.

[6] L. C. Baird and A. W. Moore, “Gradient descent for general reinforcement learning,” In *Advances in Neural Information Processing Systems*, vol. 11, pp. 968-974, 1999.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, “Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 39-52, 2011.

[8] R. N. Zare, *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics*. New York, NY: Wiley, 2011.

[9] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms,” in *Proceedings of the 2014 IEEE 20th Real-Time and Embedded Technology and Applications Symposium,* Berlin, Germany, 2014, pp. 155-166.

[10] S. Ghemawat and P. Menage, “TCMalloc: thread-caching Malloc,” [Online]. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[11] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford transactional applications for multi-processing,” in *Proceedings of IEEE International Symposium on Workload Characterization,* Seattle, WA, 2008, pp. 35-46.

Yeon-Woo Jang https://orcid.org/0000-0002-2916-0759

He is a M.S. candidate in the Chonbuk National University. He received the B.S. in Chonbuk National University in 2016. His research interests include big data analysis and distributed parallel processing

Moon-Hwan Kang https://orcid.org/0000-0003-3471-5867

He is a M.S. candidate in the Chonbuk National University. He received the B.S. in Chonbuk National University in 2016. His research interests include transactional memory and distributed parallel processing

Jae-Woo Chang https://orcid.org/0000-0002-0037-6812

He is a professor in the Department of Information and Technology, Chonbuk National University, Korea from 1991. He received the B.S. degrees in Computer Engineering from Seoul National University in 1984. He received the M.S. and Ph.D. degrees in Computer Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1986 and 1991, respectively. During 1996–1997, he stayed in University of Minnesota for visiting scholar. And during 2003–2004, he worked for Penn State University (PSU) as a visiting professor. His research interests include sensor networks, spatial network database, context awareness and storage system.