Int J Parallel Prog
DOI 10.1007/s10766-017-0529-7

Have Your Cake and Eat it (Too): A Concurrent Hash Table with Hardware Transactions

Zhiwen Chen[1] · **Xin He**[1] · **Jianhua Sun**[1] · **Hao Chen**[1]

Received: 1 September 2017 / Accepted: 23 September 2017

Abstract Hardware Transaction Memory (HTM) opens a new way to scaling multi-core software. Its main target is to achieve high performance on multi-core systems, and at the same time simplify concurrency control and guarantee correctness. This paper presents the redesign of an existing concurrent hash table using several HTM-based synchronization mechanisms. As compared with a fine-grained lock implementation, HTM-based locking scales well on our testing platform, and its performance is higher when running large-scale workloads. In addition, HTM-based global locking consumes much less memory. In summary, several observations are made in this paper with detailed experimental analysis, which would have important implications for future research of concurrent data structures and HTM.

Keywords Hardware transactional memory · Concurrent hash table · Synchronization

1 Introduction

There is a grand challenge to develop concurrent software systems correctly and efficiently on commodity multi-core machines. To date, programmers had several

✉ Hao Chen
haochen@hnu.edu.cn

Zhiwen Chen
zhiwenchen@hnu.edu.cn

Xin He
xinhe@hnu.edu.cn

Jianhua Sun
jhsun@hnu.edu.cn

[1] College of Computer Science and Electronic Engineering, Hunan University, Changsha, China

Published online: 16 November 2017

unpleasant approaches to choose from. Coarse-grained locks provide a straightforward programming model (mutual exclusion) but can lead to poor performance under contended workload. For better performance and thread scalability, programmers have been making painstaking efforts to design fine-grained locks or lock-free schemes with atomic operations, which are often difficult to implement and error prone.

Fortunately, the emerging of hardware transactional memory (HTM) opens up another possibility of scaling multi-core software and free programmers from tedious synchronization control of concurrent data structures. HTM is built on top of the cache coherence protocol, and has the potential of approaching the performance of fine-grained locks or lock-free schemes while preserving the simplicity of programming with coarse-grained locks. There are a large number of concurrent data structures based on HTM [4,9–11], which were shown to achieve comparable or higher performance than lock-based methods under low contention.

To date, most of concurrent hash tables (CHTs) are implemented with fine-grained locks or lock-free algorithm using atomic primitives. They are widely used in Linux kernel and user-level systems [2,8]. In this paper, we redesign an cache-line based hash table (CLHT) [3] to replace its lock-based synchronization with Intel restricted transactional memory (RTM). We intend to answer several questions. Does HTM deliver its promise with a straightforward implementation of synchronization scheme in practice? Can we achieve both scalability and simplified concurrency control by exploiting HTM in designing a CHT? What are the main underlying reasons that affect the overall performance of HTM-based CHTs?

This paper tries to explore the impact of HTM on constructing concurrent hash tables. We replace the fine-grained lock of CLHT with different HTM-based global locks, and perform extensive comparison. According to our experimental results, we make the following observations. First, When the workload of the CHT exceeds the capacity of L3 cache, HTM-based global locking can achieve higher performance than fine-grained locking. However, if the workload can be contained in on-chip caches, frequent transaction abort due to data contention limit the performance of HTM. The reason for this phenomenon is that all the data are stored in the cache when the data set is small. The response time for each operation is very short, which leads to higher probability of data conflict. Second, optimizing fine-grained locks with HTM offer little help in improving performance. On the contrary, it introduces complexity in concurrency control. Third, employing software-optimized HTM global locks offers about 3X to 28X speedups compared to traditional locking methods. Some HTM-based locks are sensitive to thread binding. Fourth, the amount of aborted transactions has no clear correlation to the overall performance. However, the number of requested locks and the total number of committed transactions have much larger impact on performance.

The rest of the paper is organized as follows. Section 2 presents the background on HTM and CHT. Section 3 provides an overview of CLHT. We make an thorough analysis and evaluation in Sect. 4. Related work is given in Sect. 5. Finally, we conclude at Sect. 6.

2 Background

Transactional Memory (TM) is a concurrency control paradigm that provides atomic and isolated execution for code regions. TM is considered to be one of the most promising solution to address the problem of programming multi-core processors. Its most appealing feature is that programmers only need to reason locally about shared data accesses, and let the underlying system ensure the correct concurrent execution. This model has the potential to provide the scalability of fine-grained locking while avoiding common pitfalls of lock composition such as deadlock.

Intel Hardware Transactional Memory Today, both software transactional memory (STM) and hardware transactional memory (HTM) are well researched. The release of Haswell processor with Intel TSX [6] marks the wide availability to the market. In this paper, we only study the impact of HTM on constructing concurrent hash table. HTM is a best-effort transaction memory in two aspects. First, a transactions working set must fit in the per-core private cache. Otherwise, the transaction will be aborted. Second, a transaction may also abort due to data conflict or exceptions and faults occurring during transaction execution.

HTM provides three interfaces, *xbegin*, *xend* and *xabort*. Transactions are bracketed by *xbegin* and *xend* instruction, and use *xabort* to explicitly abort a transaction. Memory addresses read and written within a transaction constitute the read-set and write-set. If one transaction's read-set overlaps with another transaction's write-set or their write-sets overlap, there will be conflicting accesses. As a hardware mechanism, HTM provides no forward progress guarantee, programmers need to provide a fall-back handler that is often a traditional lock.

Concurrent Hash Table (CHT) is one of the important concurrent data structures that allow multiple readers and writers to access shared objects concurrently, and it is widely used in software systems. For CHTs, locks, lock-free and HTM are three typical synchronization approaches to avoiding conflicts among threads trying to read or write to the same memory address. For a lock-based CHT, its critical section is protected by a lock to ensure thread-safety. A lock-free mechanism uses atomic primitives to realize synchronization between threads instead of locks. The promising features of HTM like strong atomicity and processor-assisted conflict detection have been stimulating the use of HTM to construct concurrent data structures.

3 Cache Line Hash Table

Our performance study uses the Cache Line Hash Table (CLHT) [3]. The design of CLHT follows four asynchronized concurrency (ASCY) patterns, and the key insight is *"algorithms whose memory access to shared state best resemble those of a sequential - asynchronized - algorithm tend to achieve portable scalability"* [3]. CLHT achieves high performance benefit from *avoid cache line transfers*. The bucket size is deliberately designed to be equal to the cache line size, which guarantees that most operations are served involving at most one cache line transfer. A cache line is separated into 8 words, one for concurrency control, six for key/value pairs, and the last for a pointer

to link other buckets. Both lock-based and lock-free CLHT were implemented. In this paper, we only study the lock-based version and compare it with our redesigned HTM-based variant.

The lock-based version of CLHT deploys fine-grained concurrency control (one word in each bucket as the lock) to synchronize update operations. An update first checks whether the operation would be successful, and if it is, it grabs the corresponding lock and performs the update operation. HTM provides strong atomicity, and aims at reducing the complexity of fine-grained lock and overcoming the tedious procedure of validating the correctness on multi-core processors. Yehuda Afek [1] implemented several locking algorithms based on HTM and presented two software-improved optimization techniques, software-assisted lock removal (SLR) and software-assisted conflict management (SCM). In this paper, we redesign the lock-based version of CLHT by leveraging the optimizations for HTM, in order to show that if it is possible to not only obtain higher performance but also alleviate development efforts by using HTM compared to traditional locks.

4 Evaluation and Analysis

4.1 Experimental Platform and Configurations

Our evaluation was conducted on a Linux workstation with two Intel Xeon Broadwell EP/EN/EX processors (32 physical cores/64 logical cores) and 64 GB memory installed. The CPU clocks at 2.1 GHz and the size of the three-level caches are 64 KB, 256 KB and 40 MB respectively. The operating system was Ubuntu 16.04.

We use GCC-4.8.0 to compile the source code. Except specifically noted below, all the experiments are run without explicitly thread binding. For simplicity, keys and values are both 64-bit integers in our benchmarking, and queries including *find*, *remove*, and *add* are randomly generated to conform to pre-defined distributions. In each test duration, n threads are created to execute *find*, *remove* and *add* operations. All of the n threads spawned at the start of a test will execute the same benchmark with u% updates and 100-u% lookups (By default, the update rate is 10%). Half of updates are insert operations, the other half are removes. Other related parameters are described below. d is the time to run a benchmark in milliseconds. i is the number of elements pre-filled in a hash table, for a given initial size i, we initially fill the hash table with random elements from a domain of size $2i$.

4.2 Scalability

Throughput is one of the most intuitive metrics to evaluate the performance of concurrent hash table. In this section, we evaluate the thread scalability of CLHT with different synchronization methods, including a fine-grained locking, and a coarse-grained lock based on HTM. The former is originally implemented in [3], and we implement the latter using the techniques presented in [1]. We first study how the throughput scales with the increasing number of threads.

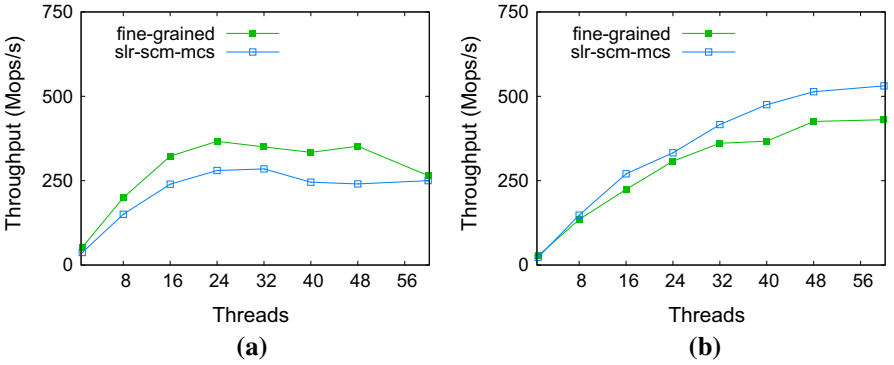

Fig. 1 Throughput of the fine-grained locking and the global MCS lock based on HTM with SLR and SCM. The update rate is 10% and the initial size is one thousand and one million respectively **a** $i = 10^3$, **b** $i = 10^6$

In this experiment, we run workloads with one thousand (can be contained in private cache) and one million (large than the capacity of L3 cache) elements initialized, and each workload has 10% update operations. Each test lasts for a duration of 5 seconds. The final results are the average of 5 runs as depicted in Fig. 1. For the workload with one million elements, both the fine-grained lock and the HTM variant show good thread scalability. The throughput increases as the number of cores. Our implementation of HTM lock obtains higher performance. Specifically, the performance under the fine-grained lock is 81% of the HTM-based lock. However, when the size of initial elements is less than the capacity of private cache, the fine-grained lock outperforms the HTM version. The reason is that the HTM global lock encounters more data conflicts that cause frequent transaction aborts.

From the perspective of complexity, HTM-based CLHT is much easier to implement than the one with fine-grained lock. It uses a single global lock to protect the critical section, while we need to pay much more attention to implement a fine-grained lock when constructing concurrent data structures. Furthermore, the fine-grained locking scheme consumes more memory. For example, CLHT take a cache line as a bucket, it split a cache line into 8 words, one for synchronization, six for key/value storage and one for pointer linking to the next bucket. If 1024 * 1024 buckets are created, we need to allocate 8 MB more memory for the storage of synchronization variables.

Implication 1. According to the experimental results, we make two observations. First, when dealing with large-scale workloads, using HTM to construct concurrent hash table would be beneficial from two aspects: (i) the performance and scalability is competitive; (ii) it achieves the goal of reducing memory consumption and simplifying programming. Second, when workloads can be fit in the on-chip cache, an HTM-based global lock may result in poor performance because of frequent transaction aborts due to data conflicts.

4.3 Fine-Grained Lock with HTM

We may ask if it is necessary to optimize the fine-grained locking with HTM when it well satisfy the performance requirement. To answer this question, we conduct a

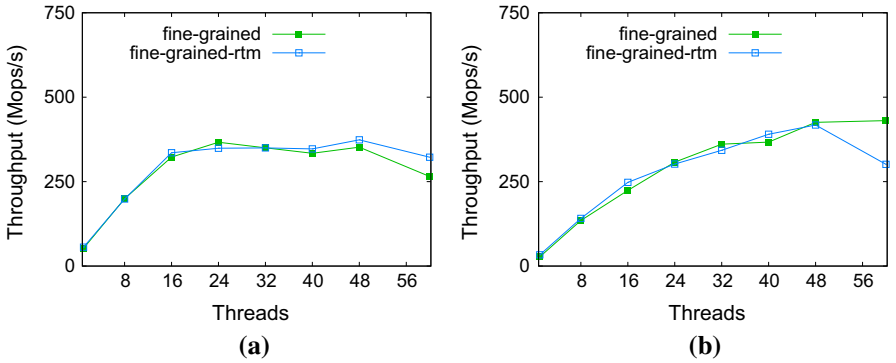

Fig. 2 Throughput of regular fine-grained lock and HTM-based fine-grained lock **a** $i = 10^3$, **b** $i = 10^6$

comparison between two versions of CLHT under fine-grained locking, one with HTM and one without. As shown in Fig. 2, the performance difference is negligible under the two settings (we get the same results when i is set to 1 million) except for the case where n is larger than 48 in the right figure. The reason is that fine-grained locks can effectively prevent multiple threads from accessing the same memory address at the same time. In such a case, it is not meaningful to adopt HTM to achieve higher performance. In Fig. 2a, when the number of threads exceeds 48, HTM-based lock exhibits better performance, because the contention increases with more threads involved, but HTM can promote parallelism if conflicts are not dominant.

Implication 2. Traditional fine-grained locks can offer good thread scalability and performance. Under these premises, using a fine-grained lock enhanced with HTM neither brings the advantage of simplifying concurrency control, nor improves the overall performance.

4.4 Comparison of Software-Improved HTM

As mentioned in Sect. 3, SLR and SCM are two software-improved methods to optimize HTM-based lock. SLR uses HTM to transactionally execute critical sections without adding the lock variable to its read-set until it is ready to commit. It reads the lock and commits if the lock is not taken; otherwise, it aborts and retries. If it fails a few times, the execution fall-backs to the non-speculative path by acquiring the lock. In SLR, a thread acquiring the lock does not automatically conflict with running transactions nor does it prevent an arriving thread from starting its transaction speculatively. Since in SLR a speculative transaction may run concurrently with a transaction that hold the lock, the speculative transaction may see an inconsistent state (which guarantees that it will fails to commit and abort).

SCM allows non-conflicting threads to continue their speculative HTM-based run without any interference from conflicting threads. By adding a serializing path to the lock implementation, an aborted thread has to acquire a distinct auxiliary lock (without using lock elision) in order to rejoin the speculative execution with other threads. Using this approach, conflicting threads are serialized among themselves and do not interfere

Algorithm 1 *Lock*() method of slr-scm-mcs scheme.

```
shared variables:
 1: lock: speculative lock
thread lock variables:
 2: retries: int
 3: my_node, my_aux_node: qnode_t
 4: function LOCK
 5:     retries ← 0
 6: speculative path:
 7:     XBEGIN(line 10)
 8:     return (0)
 9:
10: fallback path:
11:     retries ← retries + 1
12:     if retries < MAX_RETIRES then
13:         goto line 6
14:     end if
15:     if thread_handle == lock → aux_lock_owner then
16:         lock → aux_retries + +
17:     else
18:         aux_lock.lock()
19:     end if
20:     if reason&TXN_MAY_SUCCEED then
21:         if lock_aux_retries < MAX_RETRIES then
22:             goto line 6
23:         end if
24:     end if
25:     main_lock.lock()
26:     return (0)
27: end function
```

Algorithm 2 *unlock*() method of slr-scm-mcs scheme.

```
 1: function UNLOCK
 2:     if XTEST() != 0 then
 3:         if lock →! = 0 then
 4:             XABORT()
 5:         end if
 6:         XEND()
 7:         if thread_handle == lock → aux_lock_owner then
 8:             lock → aux_lock_owner ← INVALID_THREAD_HANDLE
 9:             lock → aux_retries ← 0 aux_lock.unlock()
10:         end if
11:     else
12:         main_lock.unlock()
13:     end if
14: end function
```

with other threads. Only if the thread fails due to a conflict many times it must give up and acquire the original lock. Algorithms 1 and 2 shows a brief description of an SLR-based MCS lock with conflict management.

In this section, we evaluate CLHT with different locking strategies under high contentions. We test the following six schemes: (1) Standard MCS lock (non-speculative); (2) HTM-based MCS with lock removal (slr-mcs); (3) HTM-retries; If a transaction

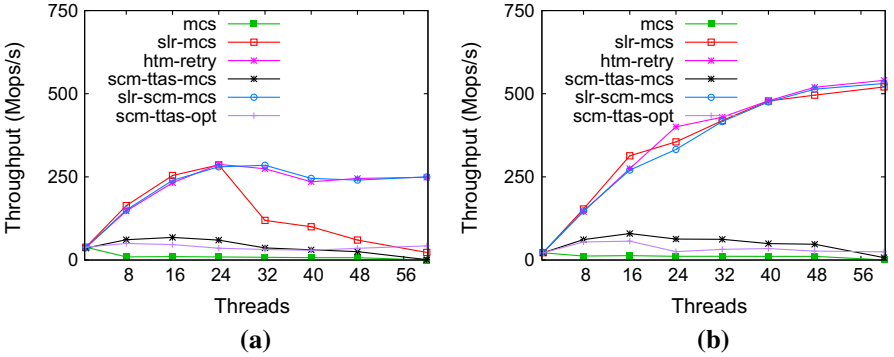

Fig. 3 Throughput with global locks. The experiments are run without explicit thread binding method. u is 10 **a** $i = 10^3$, **b** $i = 10^6$

aborts, we retry it several times as recommended by Intel [6]. (4) HTM version of ttas with conflict management; (5) SLR-based MCS lock with conflict management (slr-scm-mcs); and (6) Optimistic SCM (scm-ttas-opt), in which a thread only acquires the lock non-speculatively after retrying speculatively 10 times.

Figure 3 presents our experimental results running workloads with one thousand and one million elements initialized respectively. As expected, the traditional MCS lock exhibits poor scalability and the lowest performance. There are some differences between the two sub-figures of Fig. 3. When the workload is small, all of the locking schemes encounter a performance degradation at a certain n due to increasing data conflicts. The slr-mcs, slr-scm-mcs, HTM-retries obtain their maximum throughput at $n = 24$. However, the performance of scm-ttas-mcs and scm-ttas-opt are just marginally higher than traditional MCS lock. CLHT encounters performance degradation for small data size, because at such a scale more threads indicate large probability of incurring contentions. When the workload is larger than the capacity of L3 cache, slr-mcs, slr-scm-mcs, and HTM-retry all achieve good scalability.

Interestingly, we find that the performance of CLHT under HTM-based global lock can be affected by thread binding strategy. Figure 4 displays the throughput curves of CLHT obtained by explicitly binding threads. At first, the slr-scm-mcs, slr-mcs and HTM-retry are less sensitive to thread binding and more stable than other schemes. Secondly, scm-ttas-mcs and scm-ttas-opt scale within a single socket. While, their performance decline with the increasing of thread count. We attribute this to their weak ability to deal with cross-socket communication on a non-uniform memory access architecture.

Next, we analysis the reason caused the differences between Figs. 3 and 4 from the point of the total number of transactions, transaction abort rate, and the number of lock requests.

Implication 3. Our conclusion is threefold: (i) running a workload under high contention, traditional MCS lock has poor performance and scalability; (ii) using locks based on HTM, the performance improvement depends on the types of optimization technique employed; (iii) slr-scm-mcs and HTM-retry are more competitive than other options in scalability.

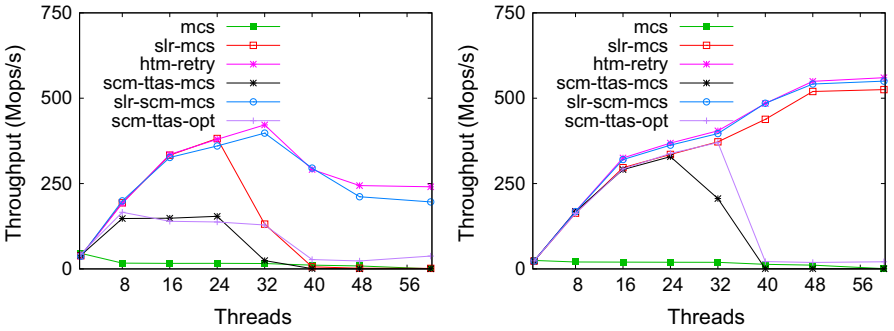

Fig. 4 Throughput with global locks. The experiments are run with explicit thread binding. u is 10. we assign successive threads to cores that are as close as possible in the topology map of the platform to get high data reuse between threads **a** $i = 10^3$, **b** $i = 10^6$

Table 1 Statistics of HTM schemes. $n = 32$ and $u = 10$

	Num. locks (million)	Total TMs (million)	Aborts (million)	Rate (%)
HTM-retry	170/160	290/360	53/67	18.3/18.5
slr-mcs	160/16	290/360	52/67	18.6/18.6
slr-scm-mcs	160/160	300/350	53/66	17.8/19.0
scm-ttas-mcs	48/12	110/130	51/62	48.6/48.4
scm-ttas-opt	110/48	220/180	53/63	23.0/35.4

The second column (c_2) represents the number of lock requests, the third column (c_3) is the total number of transactions, the fourth column (c_4) is the number of aborted transactions, and the last column is the abort rate computed by c_4/c_3. Two groups of data are presented for $i = 1$ million and $i = 1$ thousand respectively

4.5 Factors Affecting HTM Performance

Figure 3 displays the differences between the five HTM schemes. We use Intel performance counter monitor (PCM) to collect runtime metrics. Due to space constraints, we only present the most relevant data to discuss our problem. Table 1 shows the sampled data when n is 32, i is set to 1 million and 1 thousand, and u is 10. Transaction abort rate is measured by dividing the number of aborted transactions by the total number of transactions (including aborted and committed). There are few changes of the aborted transactions with different HTM schemes. The number of aborted transactions is not the main bottleneck of HTM schemes, but the number of lock requests and transactions directly impacts the overall performance. More lock requests and committed transactions result in higher throughput. For slr-mcs, scm-ttas-mcs and scm-ttas-opt, the number of lock requests and committed transactions is greater than that of the other two schemes, which can explain their superior performance in Fig. 3.

We observed that the throughput decreases as the abort rate rises. We run CLHT with different configurations to explore the relationship between abort rate and thread number and update rate and initial size. Table 2 presents the relationship between thread number and abort rate. The second and third column of Table 2 represent the total

Table 2 Variations of the abort rates with the number of threads for slr-scm-mcs scheme. $u = 10$

n	Total TMs (million)	Aborts (million)	Rate (%)
2	156/156	70/66	43.6/42.3
8	228/239	72/59	31.6/21.5
16	303/298	73/64	24.1/19.5
32	382/349	80/67	20.9/19.2
40	399/299	77/74	19.3/24.8
48	495/310	87/78	17.6/25.2
64	522/316	90/89	17.2/26.0

The second, third, and last column are the same with Table 1, and two groups of data are presented for $i = 1$ million and $i = 1$ thousand respectively

Table 3 Variations of the abort rates under different update rates for slr-scm-mcs scheme. $n = 32$

u (%)	Total TMs (million)	Aborts (million)	Rate (%)
0	346/554	73/60	21.1/10.8
10	320/352	70/66	21.9/18.8
80	282/273	74/70	26.4/25.6

The second, third, and last column are the same with Table 1, and two groups of data are presented for $i = 1$ million and $i = 1$ thousand respectively

number of transactions and the aborted transactions respectively. Both the total number of transactions and aborted transactions are increasing with thread counts. While the growth rate of aborted transactions are lower than that of the total transactions. In other words, with the increase of threads there are more committed transactions than aborted transactions. This trend is consistent with the thread scalability curve in Fig. 1.

Table 3 displays the relationship between the abort rate and percentage of update operation. We run three experiments with read-only, low update rate, and high update rate workload. For the read-only workload ($u = 0$), we get the lowest abort rate, because in the read-only scenario, the absence of write access to memory incurs little data conflicts. Higher update rates result in higher abort rates, consequently lower performance.

5 Related Work

Concurrent hash tables have been adopting fine-grained locks [5,8] or lock-free methods [7] to unleash concurrency and provide consistency. Actually, existing approaches are well designed and scales on multi-core systems. However, these synchronization mechanisms are either hard to verify the correctness, or not performance optimal, or incur complexity in development. The wide availability of HTM opens up another way for thread synchronization. Eunomia [9] is a design pattern for concurrent search tree structures under high contention. It contains several principles to reduce HTM aborts. DBX [12] is a serializable in-memory database employing restricted transactional memory. Its key idea is to use HTM to protect memory store and transactional execution. Hopscotch [4] is a concurrent hash table with fine-grained lock, and scales under

read-dominant workloads. Z. Wang et al. [11] take skiplist as an example to illustrate the pitfalls and opportunities of multi-core scaling when using HTM. Y. Afek et al. [1] provide two software mechanisms, SLR and SCM, to improve concurrency levels attained by lock-based programs using HTM-base lock elision.

6 Conclusion

This paper takes a concurrent hash table, CLHT, as an example to answer several questions regarding HTM-based concurrent hash table. We evaluate the thread scalability, compare different synchronization schemes under both low and high contention, and explore the factors related to transaction abort rate. We find that HTM-based global lock can not only achieve good scalability but also ease the design of CHTs. In future work, we plan to deploy HTM-based CHTs in practical applications and implement other concurrent data structures to further validate the findings made in this work.

Acknowledgements This research was supported in part by the National Science Foundation of China under Grants 61772183, 61572179 and 61272190.

References

1. Afek, Y., Levy, A., Morrison, A.: Software-improved hardware lock elision. pp. 212–221 (2014)
2. Arcangeli, A., Cao, M., McKenney, P.E., Sarma, D.: Using read-copy-update techniques for system v ipc in the linux 2.5 kernel. In: USENIX Annual Technical Conference, FREENIX Track, pp. 297–309 (2003)
3. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. SIGARCH Comput. Archit. News **43**(1), 631–644 (2015). https://doi.org/10.1145/2786763.2694359
4. Goel, H., Gershovitz, M.: Concurrent hopscotch hash map. http://cs.tau.ac.il
5. Herlihy, M., Shavit, N., Tzafrir, M.: Hopscotch hashing. In: International Symposium on Distributed Computing, pp. 350–364 (2008)
6. Intel, R.: Intel r 64 and ia-32 architectures. Software developers manual. (2015)
7. Liu, Y., Zhang, K., Spear, M.: Dynamic-sized nonblocking hash tables. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, pp. 242–251 (2014)
8. Metreveli, Z., Zeldovich, N., Kaashoek, M.F.: Cphash: A cache-partitioned hash table. In: ACM Sigplan Symposium on Principles and Practice of Parallel Programming, pp. 319–320 (2012)
9. Wang, X., Zhang, W., Wang, Z., Wei, Z., Chen, H., Zhao, W.: Eunomia: Scaling concurrent search trees under contention using htm. In: ACM Sigplan Symposium on Principles and Practice of Parallel Programming, pp. 385–399 (2017)
10. Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., Li, J.: Scaling multicore databases via constrained parallel execution. In: International Conference on Management of Data, pp. 1643–1658 (2016)
11. Wang, Z., Qian, H., Chen, H., Li, J.: Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In: Asia-Pacific Workshop on Systems, p. 3 (2013)
12. Wang, Z., Qian, H., Li, J., Chen, H.: Using restricted transactional memory to build a scalable in-memory database. In: European Conference on Computer Systems, pp. 1–15 (2014)