Concurrent Hash Tables on Multicore Machines: Comparison, Evaluation and Implications

Zhiwen Chen[a], Xin He[a], Jianhua Sun[a], Hao Chen[a], Ligang He[b]

[a]*College of Computer Science and Electronic Engineering, Hunan University, Changsha, China.*
[b]*Department of Computer Science, University of Warwick, Coventry CV47AL, United Kingdom.*

Abstract

Concurrent hash table has been an area of active research in recent years, and a wide variety of fast and efficient concurrent hash tables (CHTs) have been proposed to exploit the advantages of modern parallel computer architectures such as today's mainstream multi-core systems. As one of the fundamental data structures widely used in software systems, existing works on CHTs focus on either algorithmic improvements, or hardware-oriented optimizations, or application-specific designs. However, there is a lack of a comprehensive and comparative study on different implementations. In this paper, we conduct an experimental study on the state-of-the-art, and our goal is to critically review existing CHTs from wider aspects and with more detailed analysis. Concretely, we have conducted extensive evaluations of five CHTs using a unified testing framework on four multi-core hardware platforms, and implemented our HTM-based concurrent hash table. A variety of metrics such as throughput scalability, latency, impact of memory hierarchy, thread pinning strategies, synchronization mechanisms, and memory consumption, are measured in order to obtain the deep insights about performance impediments and good design choices. With this study, we hope to identify potential issues and pinpoint promising directions for future research of CHTs.

Keywords: Concurrent hash table, Multiprocessors, Multi-core, Synchronization, NUMA, Hardware Transactional Memory

1. Introduction

As the number of cores has been increasing on modern computer architectures in recent years, challenges in inventing new or enhancing existent concurrency data structures to fully leverage the hardware advancements are emerging. Hash table is a well-known data structure, which provides simple interfaces to access the elements. *lookup*, *insert* and *delete* are the three main operations provided by hash tables. As it can offer constant time lookup and update operation, it is widely used in most software systems [1, 2]. In spite of the fact that the study on sequential hash tables is relatively mature, the research on concurrent hash tables (CHTs) has attracted a lot of efforts in recent years, due to the promising performance, hardware advancement, and diversified requirements in different usage scenarios.

Ideally, CHTs should achieve high performance and scalability under different workloads and hardware settings. However, designing and implementing such CHTs is very challenging [3, 4]. Hardware-conscious CHTs leveraging platform-specific features are often ineffective in obtaining portable performance [5]. On the other hand, hardware-oblivious CHTs often fails to achieve highest possible performance. Similarly, a CHT optimized for a specific type of workloads may exhibit poor performance under a slightly different workload. For example, the Read-Copy-Update (RCU) based hash table is one of the workload sensitive CHTs. It obtains high throughput and shows good scalability when dealing with read-only workloads. However, for workloads with a small fraction of update operation, it exhibits significant performance degradation. In [5], the authors present a complete picture of how synchronization schemes behave in concurrent algorithms. In most cases, when a scalability issue is encountered, it is not straightforward to identify the root cause that may be the underlying hardware, synchronization algorithm, usage of specific atomic primitives, application context, or workloads.

Although both the industry and academia have proposed a range of CHTs with different target such as Threading Building Blocks (TBB) [6] and ConcurrentHashMap in Java [7], the performance of CHTs not only is related to the application requirements but also relies on the exploitation of lower-level hardware characteristics. When profiling CHTs, we need to perform the analysis by integrating many relevant elements and considering at different levels rather than evaluating based only on intuitive metrics such as throughput and latency. Furthermore, with a unified testing framework and a set of common performance metrics, it seems that no single CHT can outperform others in all aspects when handling a diversified set of workloads. On the other hand, from the perspective of users, the effective way to adopt a CHT is to clearly recognize all the potential performance obstacles. Unfortunately, these practical concerns are rarely mentioned in previous studies. Lacking a unified benchmark to evaluate CHTs, it is hard for the users to make a decision about which CHTs to employ in their software systems in order to attain desired performance. In summary, a comprehensive and in-depth analysis is of significance in using, designing, and optimizing CHTs. Inspired by the practices, we choose five state-of-the-art CHTs to conduct a comprehensive evaluation and analysis across a wide set of metrics. The five CHTs taken from the literature are listed in Table 1 with brief description.

CHTs in this study are written in C/C++, and they are evaluated on 4 multi-core platforms: AMD Opteron, Intel Xeon Phi 7120P (a many-core platform based on Intel MIC architecture), Intel Xeon E5-2630, and Intel Xeon E7-4850. To the best of our knowledge, it is the most comprehensive evaluation of concurrent hash tables to date. We make the following contributions in this paper.

- First, we present a framework, named **CHT-bench**, which provides a fair testing environment and unified interface for the experiments by hiding the discrepancies of hardware platforms, synthesized workloads, concurrency models, and compiler configurations. The source code of this work can be found at https://github.com/Gwinel/CHT-bench. In this way, we can guarantee the experimental results generated from our framework are fairly comparable between different CHTs.

- Second, the evaluations are explored from a wide range of perspectives including thread scalability, throughput, latency, memory hierarchy impact, low-level synchronization primitives, and memory usage. The inter-correlations between relevant metrics are also discussed when necessary. The experiments are conducted on four major hardware platforms including Intel MIC and three representative NUMA systems. We ported CHTs to the MIC platform, and to our knowledge, this is the first extensive study of concurrent hash tables on Intel MIC architecture.

- Third, implications about pitfalls, design trade-offs, and desirable optimizations are summarized for each evaluated metric, which can serve as guidelines for future research and practical development of CHTs.

The rest of the paper is organized as follows. Section 2 provides brief background on CHTs and modern computer hardware features. We present the experimental platforms and parameter configurations in Section 3. A comprehensive analysis of CHTs are made in Section 4. The related work is presented in Section 5. Section 6 concludes this paper.

2. Background

The explosive growth of commercial multiprocessor machines has brought about a revolution in the art of concurrent programming. The shared-memory programming model enforced by the underlying hardware and programming languages/runtime systems imposes much greater challenges in designing and verifying concurrent data structures than their sequential counterparts. In this section, we first introduce basic concepts and operations of concurrent hash tables and common metrics used to evaluate them. Inherent hardware features that have non-trivial impact on the performance of concurrent data structures are then presented, such as the intricacies of cache coherence on NUMA systems and the interplay between the cache coherent protocol and synchronization primitives.

A hash table (hash map) is a data structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Hash collisions are unavoidable when hashing a random subset of a large set of possible keys. The chained and open addressing hashing are two common strategies to avoid hash collisions.

A chained hash table indexes into an array of pointers to the heads of linked lists. Each linked list cell has the key for which it was allocated and the value which was inserted for that key. To lookup a particular element from its key, the key's hash is used to work out which linked list to follow, and then that particular list is traversed to find the element. The disadvantage of chained hashing is that following pointers to search the linked list consumes more memory. The advantage is that the

Table 1: List of Concurrent hash tables.

No.	Algorithm	Description	Links	Language
1	Cache Line Hash Table (CLHT)	Minimizes cache line transfers	[8]	C
2	Hopscotch Hashing (Hopscotch)	Combines the features of cuckoo, linear probing and chaining	[9]	C++
3	Concurrent Cuckoo Hashing (Cuckoo)	A concurrent cuckoo hashing supports multi-reader/multi-wirter	[10]	C++
4	User-Level Read-copy Update (URCU)	lock-free, trades update performance for read-side performance	[11]	C
5	Threading Building Block (TBB)	Based on separate chaining, scales well for read-heavy workload	[6]	C++

chained hash is only linearly slower as the load factor (the ratio of elements in the hash table to the length of the bucket array) increases, even if it is large than 1.

An open-addressing hash table indexes into an array of pointers to pairs of key/value. If there are hash collisions in the hash table, certain schemes are needed to find another slot instead. Open-addressing is usually faster than chained hashing when the load factor is low because it does not need to follow pointers between list nodes. However, it will become slower as the load factor is close to 1. In addition, the load factor of open addressing is always less than 1.

A **Concurrent Hash Table** (CHT) is a hash table that allows multiple readers and writers (or multiple readers and single writer) to access shared objects concurrently. Like its sequential counterpart, a CHT not only offers the same set of APIs, but can exert the performance of multiprocessors more efficiently. Arbitrating concurrent accesses is a necessity for all concurrent data structures. Lock-based and lock-free synchronization are two commonly used concurrency programming model. For lock-based CHTs, critical sections are protected by locks to ensure thread-safety. Coarse-grained locking is relatively easy to implement, while preventing more efficient utilization of computing resources. With fine-grained locking, multiple threads are allowed to operate on different partitions of the data concurrently. Finer granularity is beneficial to improve the overall performance but at the cost of more implementation endeavors. Lock-free is another concurrency programming paradigm without using explicit locks. Lock-free CHTs are also widely proposed [12, 13].

In pursuing high performance concurrent data structures, hardware support for synchronization is also a main challenge. In multi-processor multi-core environments, to maintain data consistency, hardware cache-coherence is often needed to ensure the consistency of accessing shared data from different cores. A cache-coherence protocol maintains state transitions on load, store, and atomic instructions (i.e., CAS and FAI). For example, a protocol may choose different update and invalidation transitions such as update-on-read, update-on-write, invalidate-on-read, or invalidate-on-write. State transitions can affect the amount of inter-cache traffic, and consequently the available cache bandwidth for actual work. MESI cache-coherence protocol and its variants are commonly used in modern processors.

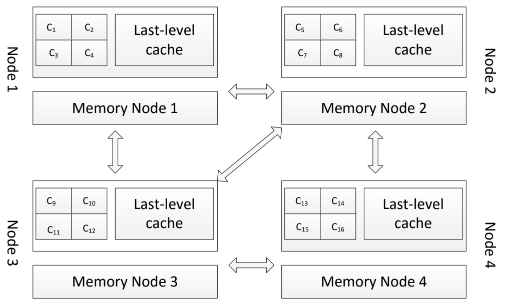


Figure 1: A typical NUMA system architecture with four nodes each containing four physical cores.

Modern servers are often equipped with multiple nodes each containing a multi-core CPU and a local DRAM served by one or more memory controllers (see Figure 1). The nodes are organized into a single cache-coherent system with high-speed interconnects. Accesses to a local node go through a local memory controller, while requests to remote nodes need to traverse the interconnect and access a remote controller. Remote accesses typically take longer than local ones, which is often called the *Non-Uniform Memory Access* (NUMA). Data consistency on NUMA systems is more complicated than on single-node multi-core machines.

3. Experimental Environments and Configurations

In this section, we first introduce our experimental platforms, and then describe the five CHTs in detail. Finally, parameter configurations are presented.

3.1. Platforms

In order to produce experimental results as general as possible, we take AMD Opteron, Intel Xeon E5-2630, Xeon E7-4850, and Intel Xeon Phi 7120P as our target platforms. The operating system is Ubuntu 14.04 LTS.

AMD Opteron. There are four Opteron 6172 multi-chip modules (MCMs) in the 48-core AMD Opteron machine with 128 GB memory. Its max memory bandwidth is 42.7 GB/s. Each MCM has two 6-core dies with independent memory controllers. There are 8 memory nodes in this system. The distance between two dies in the same socket is 1-hop. Two dies from different socket are situated at 2-hop distance. The overhead between two dies is lower than the overhead between sockets. The CPU clocks at 2.1 GHz, and the size of L1, L2, and LLC cache is 64KB, 512KB, and 4MB (per die) respectively. The Opteron platform uses an MOESI protocol for cache coherence. The 'O' stands for the *owned* state, which indicates that this cache line has been modified but there might be shared copies on other cores. The caches in Opteron are write-back and non-inclusive. The hierarchy is not strictly exclusive. Data hit in LLC are pulled into L1 but may or may not be deleted from the LLC.

Intel Xeon E5-2630. The 16-core Intel Xeon E5-2630 machine consists of two sockets each with 8 cores (16 hardware threads) and 16 GB memory. It operates at 2.4 GHz and provides 32 KB L1 cache, 256 KB L2 cache, and 20 MB LLC. It has 4 memory channels and 2 QPIs. The max memory bandwidth of E5-2630 is 51.2 GB/s. This processor adopts an extended MESI protocol, and its caches are inclusive (every new cache-line fill occurs in all the three levels of the hierarchy).

Intel Xeon E7-4850. This platform consists of four sockets each with 12 cores (24 hardware threads) and 128 GB memory and is clocked at 2.3 GHz. The size of L1, L2, and LLC cache is 32 KB, 256 KB and 24 MB respectively. It has 4 memory channels and 3 QPIs. The max memory bandwidth of E7-4850 is 68 GB/s. This processor adopts an extended MESI protocol with inclusive cache.

Intel Xeon Phi 7120P. It's a many-core machine based on Intel MIC architecture. It integrates 64 in-order cores on the same chip. Each core is clocked at 1.23 GHz and supports 4 hardware threads. So the total number of hardware threads is up to 244. The memory hierarchy of Xeon Phi is similar to a conventional multi-core system. The memory on Xeon Phi is shared among and accessible to all cores, and it is 16 GB in size. Each core has a 32 KB L1 data cache and 32 KB L1 instruction cache, and a private 512 KB L2 cache, thus presenting a total 31 MB of L2 cache on the chip. Its max memory bandwidth is 352 GB/s. Xeon Phi implements an extended MESI protocol, which is differentiated in that the shared state is extended with a directory-based cache coherence protocol named GOLS (Globally Owned, Locally Shared), which enables the sharing of a modified cache line and can avoid broadcast storms on the address buses. Each cache determines the state of a line via consulting the GOLS protocol. The global coherence is maintained by the distributed tag directories (DTDs) that record the coherence state of each cache line. The address of each line is mapped to a DTD by a hash function, leading to an even load distribution.

3.2. Concurrent Hash Tables Overview

In the following, we present an overview of the five CHTs evaluated in this paper.

Frequent cache line transfers is a disaster to concurrency. The cache-line hash table (CLHT) can reduce the number of cache-line transfers as many as possible by using cache lines as buckets [3]. CLHT is a chaining based hash table, which uses pointers to link other buckets. Additionally, the granularity of coherence is a cache-line that is 64 bytes on most modern multi-core systems. Therefore, a 64-byte bucket is split into eight words, six to store key/value pair, one for concurrency control and one for linking buckets. Based on this bucket structure, it is straightforward to design a lock-based hash table. Intuitively, when an update operation is completed (e.g., a new key/value pair is inserted into the hash table), at least one write on shared state must be performed.

However, it is suggested that search operations should not include any stores. Consequently, the search operation of CLHT needs to parse the keys of the buckets and return without any synchronizations. In order to support in-place updates, parsing the bucket does not simply traverse the keys, but obtains an atomic snapshot of each key/value pair. The atomic snapshot guarantees that if a search finds the target key, the value that will be returned corresponds to that key but not to a concurrent modification.

In this paper, we evaluate both lock based (**CLHT-lb**) and lock-free (**CLHT-lf**) variants of CLHT. CLHT-lb uses the concurrency control word in a cache line as a lock. Search operations traverse the key/value pairs and return the value if it is matched. Updates first perform a search to check whether the key exists. If there is not enough space for an insertion, the operation either links a new bucket by using the next pointer, or resizes the hash table. As for CLHT-lf, in order to keep the atomicity of key/value pair insertions, a *snapshot_t* object is

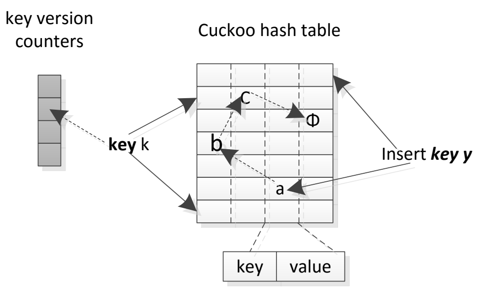


Figure 2: A four way Cuckoo hash table. Each key is mapped into two buckets by hash functions with an associated version counter.

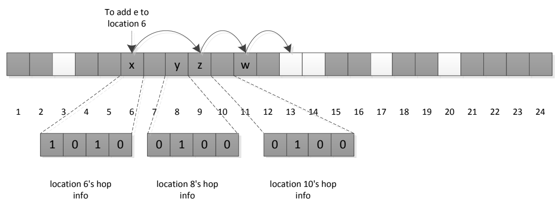


Figure 3: An *insert* operation of Hopscotch. To insert the item e with hash value 6, a linear probe finds the nearest empty entry 13 (because entry 6 is occupied), but it is too far to move x into entry 13 because H is 4. So Hopscotch moves w to entry 13, z to entry 11, x to entry 9, and finally empties entry 6.

devised. The size of a snapshot_t is 8 bytes, containing a 4-byte version number and an array of 4 bytes (map). The snapshot_t provides an interface to atomically get or set the value of an index in the map. The version number is used to enable sets/gets to do atomic changes with respect to the other spots in the map. In short, atomicity is implemented by reading the value of the snapshot_t object before the atomic section and by using the version number to get/set the target index in the map using a CAS on the whole object. For instance, if another concurrent insertion has already been completed, the current operation will fail the CAS, because the version number will be different. We then use the fields of the map as flags that indicate whether a given key/value pair is valid, invalid, or is being inserted.

Contrary to CLHT, *Cuckoo hashing* is an open addressing hash table design. All items are stored in a large array, without pointers or linked lists. Two techniques are used to mitigate hash collision. First, items can be stored in one of two buckets in the array, and they can be moved to the other location if one is full. Second, the hash buckets are multi-way set associative, i.e., each bucket has B "slots" for items. A lookup for a key proceeds by computing two hashes of the key to find buckets b_1 and b_2 that could be used to store the key, and examining all the slots within each of those buckets to determine if the key is present. A basic 2,4-cuckoo hash table (two hash functions, four slots per bucket) is shown in Figure 2. A consequence of this design is that lookup operations are both fast and predictable, always checking 2B keys. To insert a new key into the table, if either of the two buckets has an empty slot, it is then inserted in one bucket; if neither bucket has space, a randomly chosen key from one candidate bucket is displaced by the new item. The displaced item is then relocated to its own alternate location, possibly displacing another item, and so on, until a maximum number of displacements is reached. If no vacant slot is found, the hash table is considered too full to insert, and an expansion process is scheduled. The sequence of displaced keys in an *insert* operation is defined as a cuckoo path. Write performance of cuckoo hashing degrades as the table occupancy increases, since the cuckoo path length will increase, and more random reads/writes are needed for each *insert*.

There are several variants of Cuckoo hashing [14, 4, 15] . In our work, we use the **M**ulti-**R**eader/**M**ulti-**Writer** Cuckoo hash (MRMW Cuckoo). MRMW Cuckoo uses algorithmic engineering of Cuckoo hashing, combined with architectural tuning in the form of effective prefetching, and an optimistic design that minimizes the size of the locked critical section during updates. In order to reduce the number of item displacements and the size of critical sections, MRMW Cuckoo hash uses a breadth-first search instead of depth-first search. Each *insert* optimistically searches for a cuckoo path, displacing items along the path with the protection of striped fine-grained spinlocks. Execution terminates at the end of the path or if the path becomes invalid.

Hopscotch hashing [16] is a scheme for resolving hash collisions in a hash table using open addressing. It combines the characteristics of cuckoo hashing, linear probing, and chaining in a novel way. Hopscotch hash table consists of an array of buckets. The key notion in Hopscotch is the neighborhood of buckets around any specific bucket. This neighborhood has the property that the cost of finding a desired item in any of the buckets in the neighborhood is the same or very close to the overhead of finding it in the bucket itself. This property is achieved and maintained in the *insert* process.

Figure 3 illustrates an *insert* operation. The item hashed to an entry will always be found either in that entry, or in one of the next $H-1$ neighboring entries, where H is a constant (H could be 32/64, the standard machine word size). Each entry includes a hop-information word, an H-bit bitmap that indicates which of the next $H-1$ entries contains items that are hashed to the current entry's neighborhood bucket. In this way, an item can be found quickly by looking at the word to see which entries belong to the bucket, and then scanning through a constant number of entries (on most machines this requires at most two cache-line transfers).

In summary, Hopscotch moves the empty slot towards the desired bucket but cuckoo hashing moves an item out of the desired bucket and tries to find a new place for it. Both of the sequential and concurrent Hopscotch are presented in [16], we use the concurrent version in this work.

Read-copy update (RCU) [17] is a concurrency model originally proposed to optimize the shared data access in the Linux kernel, instead of merely a concurrent hash table design. For the shared data structure protected by RCU, readers can access it without a lock, but writers need to copy this data structure before modifying it. When the modification is finished, the pointer (pointing to the old data) is modified to point to new data (copied from the old) through a callback function appropriately. The key notions of RCU are *quiescent state* and *grace period*. RCU trades the update performance for read-side performance. We use the quiescent state based implementation of URCU (version-0.8.8) from *liburcu* to facilitate the integration into our testing framework and the comparison with other CHTs.

Intel's *Threading Building Blocks* (TBB) [6] is a task-based parallel programming paradigm for multi-core platforms. It provides a *concurrent_hash_map* that allows multiple threads to access concurrently. The concurrent_hash_map maps keys to values in a way that permits multiple threads to concurrently access values. The keys are unordered. For each key, there is at most one element in a concurrent_hash_map, but the key may have other elements in flight. Member classes *const_accessor* and *accessor* are called accessors. Accessors allow multiple threads to concurrently access pairs in a shared concurrent_hash_map. An accessor acts as a smart pointer to a pair. It holds an implicit lock on a pair until the instance is destroyed or the method release is called on the accessor. It is based on the classic separate-chaining, where keys are hashed into a bucket that contains a linked list of entries. TBB inherits all of the advantages and disadvantages of chaining. It scales very well for read-dominant workloads. In this work, we integrate TBB version 4.2 into our testing framework.

Resizing a hash table is expensive. In order to simplify the task of moving elements among buckets during a *resize* operation, Y. Liu et al. [13] apply a freezable set abstraction in the nonblocking hash table that supports resizing in both directions, shrinking and growing. Approaches to improving cache locality is also proposed. We tested their implementations in our machines and found that *lock-free list* (LFList) achieved the highest throughput. The advantage of LFList is its stable increasing of throughput under different parameter configurations and hardware platforms. According to our test, the performances of the dynamic-sized nonblocking hash tables are roughly similar to TBB. But it is written in Java and the CHTBench framework is based on C/C++. Thus, we do not include it for comparison in this work.

3.3. Experimental Configuration

CHTs proposed in the literature often exhibit large diversities in the design (synchronization model, such as lock-based or lock-free), implementation (hardware-specific optimizations), and evaluation (synthesized datasets and testing methodologies). This makes it hard for end users to have an intuitive view about the performance impact from various aspects. To this end, we designed a benchmarking framework, CHTBench, for our evaluation. The framework provides a small set of APIs, and integrates the original CHT implementations offered by the authors. Keys and values are both 64-bit integers.

In CHTBench, operations including *lookup*, *remove*, and *insert* are randomly generated to conform to uniform distributions. It creates an empty table and n threads at first. Then, using the thread pining strategies described in Section 4.5 to bind each thread with a core. The hash table is initialized with n concurrent threads. The internal work flow of each thread is illustrated in Figure 4.

All of CHTs are compiled using gcc (version 4.8.2) with the '-O3' flag. In each test duration, n threads are spawned to execute *lookup*, *remove*, and *insert* operations. A random number c ranging from 1 to 100 is generated to control the ratio of the three different operations. In this way, we make sure that each thread performs a certain proportion of updates and lookups. Unless otherwise stated, the filling rate of the hash table is 0.5. We do not show the results with different filling rates because the filling rate of chaining-based CLHT may be large than 1, and the filling rate of other open addressing CHTs should be strictly less than 1. Our test runs a while loop during d milliseconds that executes *lookup*, *remove*, and *insert* operations determined by a

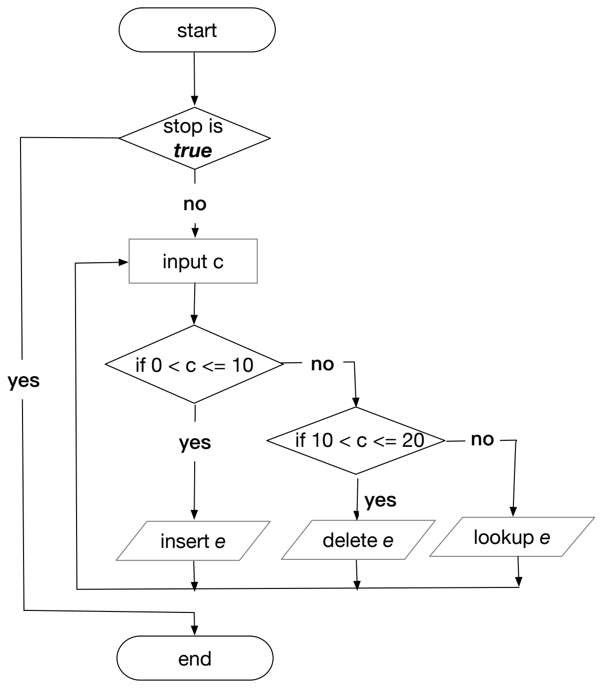


Figure 4: The execution flow of a single thread in our testing framework.

set of parameters. All of the n threads created at the start of a test will execute the same benchmark with $u\%$ updates and $(100 - u)\%$ lookups. Half of updates are *insert* operations, the other half are *delete*s. The benchmarking is configured by parameters described below. d is the time to run a benchmark in milliseconds. n is the number of threads created when running a test. The maximum value of this parameter is 244, 96, 48 and 32 for Xeon Phi 7120, Xeon E7-4850, Opteron 6172, and Xeon E5-2630 respectively. i is the number of elements pre-filled in the hash table. f is the filling rate of the hash table, and it is the result of i divided by the total number of buckets. r is a value in the range $[1...2i]$, which represents the position of a randomly generated key. This can guarantee half of the operations are successful and their structure size remains close to i. u represents the proportion of updates among all operations.

4. Evaluation and Analysis

This section presents a comprehensive analysis on CHTs. The analysis is conducted from the following seven aspects: scalability, impact of update, memory hierarchy, latency, thread pinning strategy, synchronization mechanism, and memory consumption. In most cases, we report results obtained from the four platforms. But for cases where performance results are similar, we only present the results on the Intel E5-2630 machine for space constraints.

4.1. Scalability

Throughput is a common performance metric, and in this section, we observe how throughput scales with the increasing number of threads. In this experiment, we employ a compact strategy (detailed in Section 4.5) to pin threads to cores, and each run lasts for a duration of 5 seconds. The final results are the average of 5 runs as depicted in Figure 5.

From Figure 5, we can observe that the throughput curves of URCU almost overlap with the x axis when the workload contains 10% update operations. Benefiting from less cache line transfers and better synchronization mechanism, CLHT exhibits the best performance on four platforms (the difference between its two versions is marginal). As for Hopscotch, the throughput peaks with 16 threads on the E5-2630 machine and 24 threads on the E7-4850 machine. Noticed that 16 and 24 are the maximum number of threads supported by a socket on the two platforms. While for the AMD platform its peak performance occurs when the number of threads reaches 6 (this happens to be the maximum number of threads in the same die). Combining the observations from the three NUMA platforms, we can infer that Hopscotch is less capable of reducing the overheads in cross memory-node communication, and it may only perform well in single-socket environments. In other words, further optimizations are needed to exploit the features in NUMA systems. We will present detailed arguments in Section 4.5.

Considering that in Figure 5(a), an inflection point can be observed when n is 16, we break the curves into two stages that are separately discussed as follows.

Stage 1: The number of threads created ranges from 1 to 16 and hyper-threading is disabled. Obviously, CLHT shows the best growth rate, and its throughput increases by about 15 Mops/s with an additional thread. The lock-based CLHT performs better than the lock-free version. For Hopscotch, adding one thread can speed up the throughput by about 5.5 Mops/s. The performance of Cuckoo is worse than TBB at this stage.

Stage 2: At this stage, threads are distributed into two sockets and hyper-threading is involved in computation. CLHT has a minor increase in throughput, which may be due to the influence of memory bandwidth utilization. As reported in Table 2, the memory bandwidth of CLHT-lb remains unchanged at this stage. While the sharp decrease of Hopscotch's performance

Table 2: Memory bandwidth usage of CLHT-lb on Intel Xeon E5-2630.

Threads	1	4	8	12	16	20	24	28	32
MBW (in GB/s)	1.5	5.8	11.2	12.4	12.8	12.8	12.8	12.8	12.8

implies that cross-socket communication overhead outweighs the gains of more threads. In addition, TBB and Cuckoo still maintain good growth rate.

As depicted in Figure 5(b), the throughput curves on E7-4850 show similar trends as on E5-2630 except CLHT. On E5-2630, limited by the memory bandwidth utilization, the throughput increase of CLHT is marginal, while we can see steady growth on E7-4850 (the maximum bandwidth of E7-4850 is large than that of E5-2630).

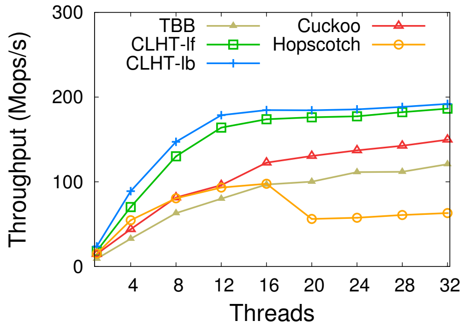
Figure 5(c) depicts the throughputs on the AMD machine. TBB achieves better scalability than Cuckoo, Hopscotch, and URCU on this four-socket platform (although the overall throughput is still lower than CLHT). Hopscotch shows optimal performance only when all threads reside in a single die (less than 6 threads). More threads do not necessarily benefit the overall performance. Instead, negative results can be observed for CLHT and Cuckoo when the thread count exceeds a certain threshold, due to the available memory bandwidth and resource contention.

The results on Xeon Phi 7120P are shown in Figure 5(d). Surprisingly, both versions of CLHT achieve linear scalability. For CLHT-lb, adding one thread can speed up the throughput by about 2.5 Mops/s. The reason for this linear scalability is two folds. First, each bucket is aligned to a single cache line and the update is in-place, which greatly reduces cache-line transfers. And aligned data can avoid false sharing that is critical to the performance on Xeon Phi. Second, its fine-grained lock scheme incurs low contention. However, not all CHTs can deliver portable scalability like CLHT, the performance of Cuckoo and TBB are very poor, and spawning more threads contributes little to the performance. Detailed explanations are given in Section 4.2 and 4.6.

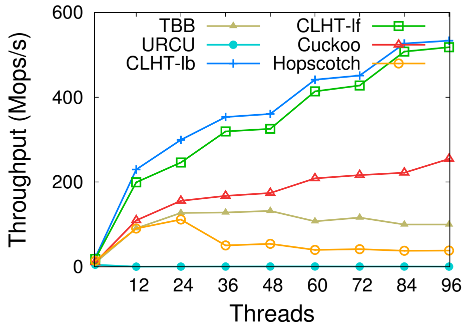
Impact of Data Distributions. In previous experiments, the key-value pairs are randomly generated. However, in the real applications, perfect uniform distributions are typically not always available, so the performance consequences with different data distribution are of interest. In this subsection, we study the performance impact of workloads following the *zipf* distribution. The configurations such as filling rate, update rate, and the number of elements are the same as in the evaluation of scalability. Figure 6 shows the throughput for workloads with *zipf* distribution (other platforms exhibit similar results that are not shown in Figure 6). The throughput of Cuckoo, CLHT, Hopscotch, and TBB decreases by 44%, 51%, 53%, and 30% respectively, as compared with the results from random distribution (see Figure 5(a)). The *zipf* distribution makes the data access more skewed, and a majority of operations accesses a small part of data. This intensifies the contention among threads accessing the same key, which stresses the on-chip interconnects and incurs more cache-coherent traffic due to synchronization.

Hash Table Resizing. The time that an insert operation takes will increase as the filling rate of hash table increases. In the worst case, insert operation will fail because it could not find an appropriate position after too many retries. Undoubtedly, this has a great impact on performance. In order to address this issue, a resize operation is adopted by some CHTs. With this operation, CHTs can create a new hash table whose capacity is larger than the old table, and all the elements in the old table are copied into the new table, which inevitably incurs additional overhead. In order to evaluate the overhead of resizing CHTs, we conduct an experiment, in which the table is configured as 90% full and the update rate is 40% (35% insert and 5% remove). Hopscotch selected in our study does not implement the resize operation. Experimental results show that Cuckoo has a lower probability to trigger the resize operation than CLHT and TBB. We own this to Cuckoo's optimization in finding a cuckoo path. The throughput is decreased by 5% compared to that without triggering resize.

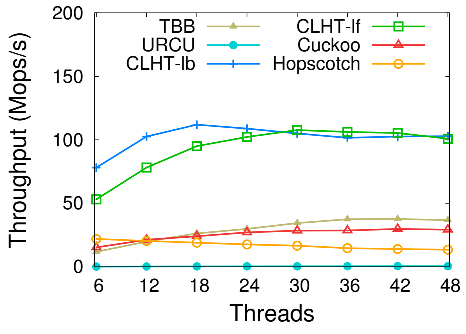
Implication 1. Spawning more threads does not necessarily lead to higher throughput. On the one hand, adding more threads may saturate the memory subsystem, which either makes the performance reach a plateau or even degrades the overall performance. On the other hand, higher concurrency without proper arbitration may incur interconnect contentions on NUMA systems, resulting in suboptimal performance. Many-core systems like Xeon Phi show non-trivial discrepancies in achieving scalable CHTs as compared to conventional multicore systems. Designing concurrent hash tables for this new platform should concern about its architectural features. Skewed data access would incur more cache-coherent traffic, and intelligent synchronization algorithms are desired to alleviate perfor-



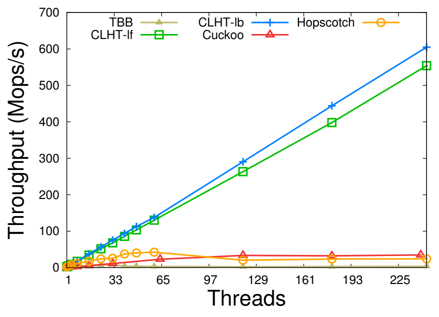
(a) E5-2630



(b) E7-4850

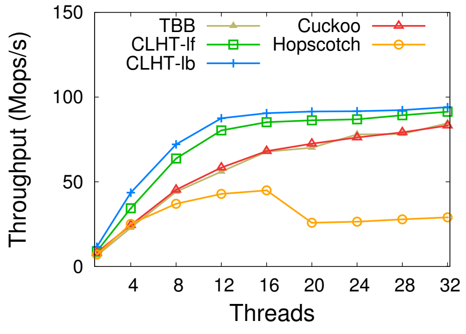


(c) Opteron



(d) Phi 7120P

Figure 5: Throughput of CHTs on four platforms, with *f* 50%, update rate 10%, and 1 million elements initialized.



(a) E5-2630

Figure 6: Throughput with *zipf* distribution.

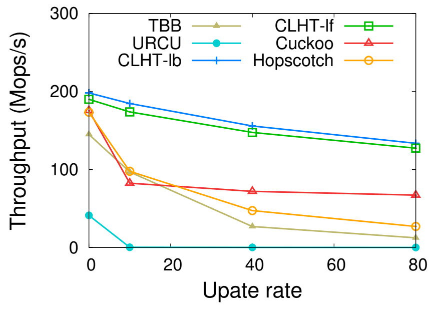
mance degradation.

4.2. *Impact of Update Rate*

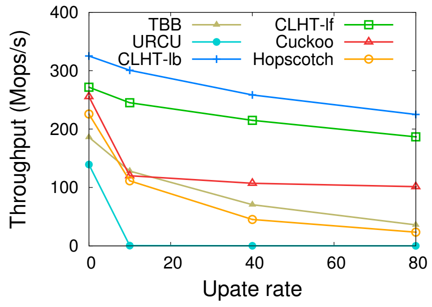
Like the sequential counterparts, CHTs have demonstrated the superiority for read-dominant workloads. In this section, we examine the behaviors of CHTs by varying the percentage of *update* operations. The initial size of hash tables is configured as 1 million.

As shown in Figure 7, all of the CHTs achieve the highest throughput when running with a read-only workload. For the three multi-socket platforms, we set *n* to the max number of threads. With the compact strategy to pin threads to cores, we can avoid the interference from cross-socket traffic. When update operations are involved, the performance declines sharply. URCU is very sensitive to updates, by changing the update rate from 0% to 10%, the throughput of URCU decreases by 270x. Hopscotch also suffers a steep decrease in throughput compared to the read-only case. This phenomenon also illustrates the severe scalability issue that Hopscotch encounters due to the high overhead of synchronization (other CHTs, such as TBB and URCU, have the same problem). Cuckoo exhibits the lowest decrease rate among the five CHTs when the update rate varies from 10% to 80%, which indicates that Cuckoo is more tolerable to workloads with high update rate.

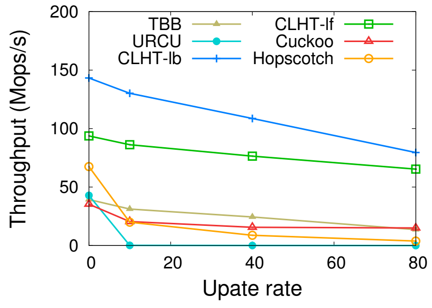
We use cache misses per operation as the indicator to explain the performance degradation when update operations are involved. The data cache misses are measured using *VTune Amplifier*. The results (with 16 threads) are listed in Table 3, which shows that the update rate and the cache misses per operation are positively correlated. CLHT works well under both low and high update rate, which can be inferred from its slight value variance (the cache misses with the 80% update rate workload is only 30% higher than that of the read-only workload). How-



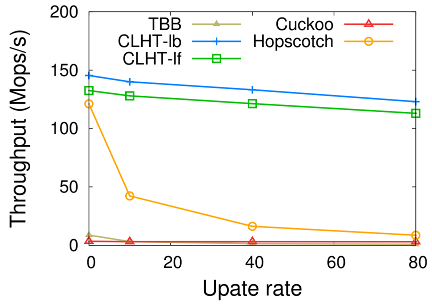
(a) E5-2630, n = 16



(b) E7-4850, n = 24



(c) Opteron, n = 12



(d) Phi 7120P, n = 60

Figure 7: Throughput with different update rates on different platforms, with 1 million initialized elements

Table 3: Average data cache misses per operation with different update rates on E5-2630.

u	TBB	URCU	CLHT-lb	CLHT-lf	Cuckoo	Hopscotch
0%	28.6	85.2	23.6	31.7	33.5	22.9
10%	46.3	1169.1	24.6	34.9	44.2	28.1
40%	99.2	4046	27.0	37.3	57.2	62.7
80%	159.1	10788	30.7	40.6	66.2	115.5

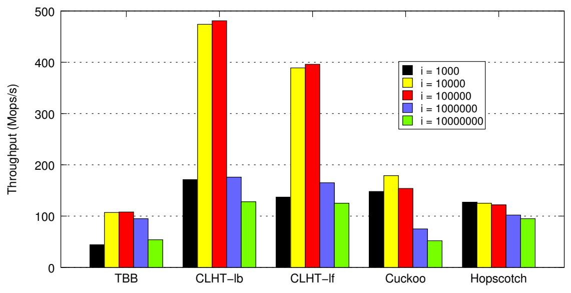
ever, URCU faces a 14x and 127x increase of cache misses when u changes from 0 to 10% and 0 to 80% respectively. Worse, the CPU utilization of URCU is very low (only 3%) when running the workload with 10% updates, compared with the 50% CPU utilization of other CHTs. Spawning more threads does not noticeably increase cache misses, indicating that the increase of update operations is the main reason for the sharp decrease of throughput.

Implication 2. Frequent cache line transfers are the enemy of update performance. Update operations invalidate local cache lines, which results in not only write traffic to the local memory node, but also cross-socket messages that are enforced by cache-coherence protocols and transmitted via on-chip interconnects such as Intel QPI and AMD HyperTransport. Some concurrent hash tables are designed for read-dominant workload, so their performance degrades significantly even when involving only a small amount of update operations. Write-friendly hash tables often precisely control the layout of critical data in the cache such as shared variables.

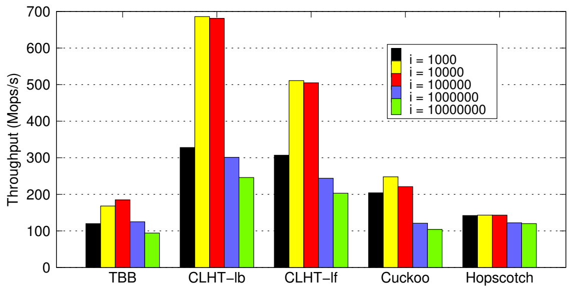
4.3. Cache and Main Memory

Considering the facts that the intensive memory access is one of the prominent features of hash tables and the memory bandwidth is easy to be a bottleneck on such systems [18], in this part, we investigate the impact of the memory hierarchy on performance. The histograms in Figure 8 illustrate how the throughput varies with the different number of elements initialized .

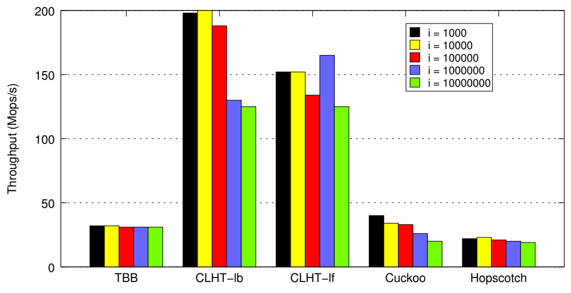
From Figure 8(a) - (c), we can see that when the working sets completely reside in the cache, all CHTs reach the highest throughput. We validate this by monitoring the bandwidth and traffic between the LLC and main memory using a tool *likwid-perfctr* [19]. Once the working set size exceeds the LLC capacity, the performance degrades significantly. In addition, the in-cache performance of CLHT is much better than the others due to its well-designed cache exploitation mechanism. We also observed remarkable contentions in high-level caches with the increasing number of threads (not shown in Figure 8). For example, when i is 1000, the performance of CHTs decreases by increasing thread count. The reason is that more threads would increase



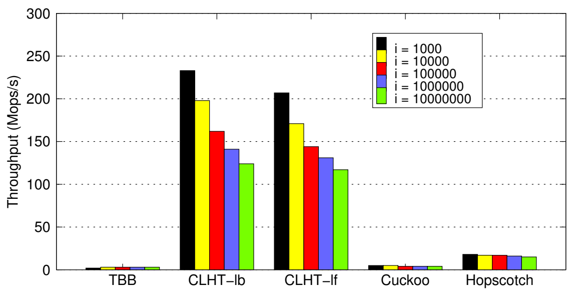
(a) E5-2630, n = 16



(b) E7-4850, n = 24



(c) AMD Opteron, n = 12



(d) Phi 7120P, n = 60

Figure 8: Histograms with different initial sizes, the u is fixed at 10%.

the possibility of different threads operating on shared data, which incurs synchronization overhead.

Hopscotch's throughput remains relatively invariable with different initial sizes. With further investigation, we found that the original implementation of Hopscotch hard-coded a fixed-size memory allocation. This approach sacrifices flexibility but guarantees stable performance. For highly-concurrent applications, both lock-based and lock-free CHTs internally rely on an efficient mechanism for memory management, which is typically built on top of a third-party or system-provided dynamic memory allocator. Further evaluation on dynamic memory allocators is left as our future work. In addition, Hopscotch failed to run larger benchmarks (exceeding 100 million elements). This may be attributed to its weak memory management mechanisms. TBB also exhibits little variance on AMD Opteron as compared to other platforms.

As aforementioned, the memory hierarchy of Xeon Phi 7120P is different from main stream multi-core architectures. There are only two level of caches, and the cores and memory controllers are connected by a bi-directional ring. As reported in Figure 8(d), we can observe lower throughputs for larger initialized hash tables. Cuckoo and TBB both performs poorly no matter the size of CHTs.

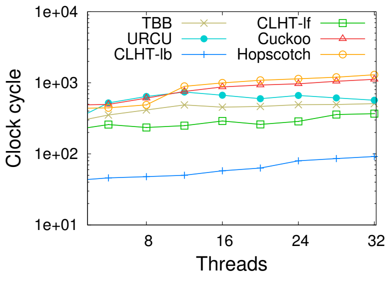
We also tested large workloads on the order of magnitude of gigabytes (the results are not shown in Figure 8). The performance curves show similar trends with running workloads of 1 million elements, but lower absolute throughputs.

Implication 3. Caches have important implications in accelerating performance across different architectures. Approaches for fine-grained control over caches are desired to obtain predictable results, such as scheduling threads for cache-resident working set to avoid excessive synchronizations. Static memory allocation is not preferable, and the impact of dynamic memory allocators needs further investigations especially for large hash tables.

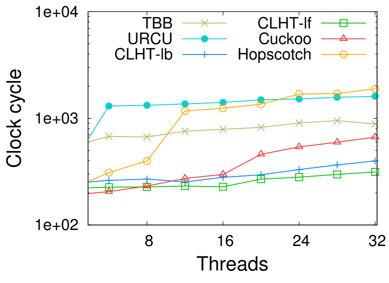
4.4. Latency

Having presented the throughput of CHTs from a macroscopic view, in this section, we explore the latency experienced by individual operations from a microscopic perspective. Understanding the impact of different algorithmic designs on latency variation is especially important for latency-sensitive applications.

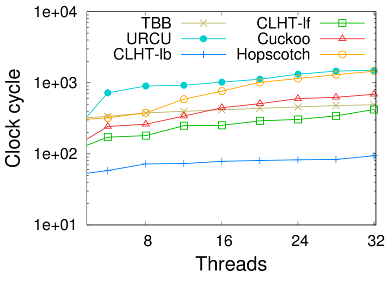
We measure the timings of hash table operations under the granularity of a single CPU cycle. In this experiment, the update rate is configured as 10% and initially the hash tables are filled with 1 million elements. An operation is successful if its expected objective is achieved. Otherwise, we call it a failed operation. Therefore, we have 6 types of operations including *get-suc*, *get-fail*, *put-suc*, *put-fail*, *rem-suc* and *rem-fail*. Clock cycles are collected using a simple profiler *sspfd*



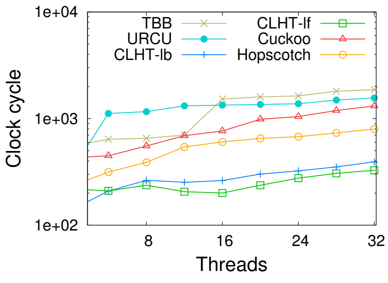
(a) get-fail



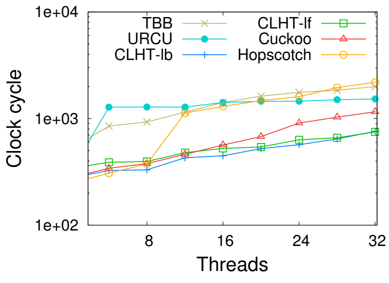
(b) put-fail



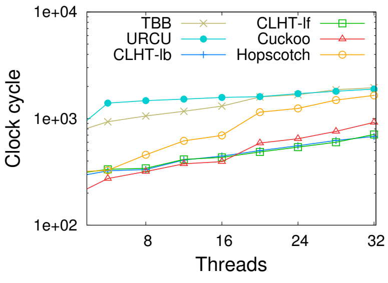
(c) rem-fail



(d) get-suc



(e) put-suc



(f) rem-suc

Figure 9: Latency of *search*, *insert* and *remove* operations on Xeon E5-2630.

[20], and a uniform sampling is used to calculate the average latency. We only report the results of the E5-2630 because of space constraints.

As shown in Figure 9, with more threads involved, resource contentions (memory bandwidth, caches, and memory controller) and synchronization overheads are surfacing, which inevitably cause latency increases. In comparison, CLHT (both versions) outperforms other CHTs in most cases, and this superiority can be mainly attributed to the four ASCY patterns [3] collectively employed in its design. Excessive cache line transfers necessitated by the cache-coherence protocol significantly increase the latency of operations. URCU performs worst because of its copy-on-write mechanism that involves waiting for a RCU grace period. For Hopscotch, its latency is sensitive to inter-socket communication (see the steep curve beyond 8 threads). Moreover, a timestamp field in Hopscotch will be modified during a *remove* operation in order to ensure that concurrent *lookup* operations will fail, similar to the atomic snapshot of CLHT. The difference between Hopscotch and CLHT is that the former needs to store shared variables which results in extra cache-coherence traffic. In addition, the search phase of Hopscotch's update operation involves waiting as a lock is held at the beginning of this phase. Both of these designs violate the second pattern of ASCY, which advocates *the search phase of an update operation should not perform any stores other than for cleaning-up purposes and should not involve waiting, or retries*. Even worse, as shown in Figure 9(b), (e), its latency exceeds URCU in the cases of *put-fail* and *put-suc*.

For Cuckoo, search operations counter-intuitively take more clock cycles than update operations (including *insert* and *remove*). We attribute this to the overhead of looking up elements in long Cuckoo paths. For the *put-fail* and *rem-suc* case, Cuckoo performs best under low concurrency levels. Hopscotch achieves lowest latency for *put-suc* when thread count is less than 6. TBB's latency in fail cases remains relatively stable, while in success cases it is worse or in par with URCU.

Implication 4. The asynchronized concurrency (ASCY) patterns proposed in [3] are really helpful in reaching portable scalability as demonstrated by the superior performance of CLHT in our analysis. In particular, specific patterns in ASCY can help identify potential pitfalls in implementations of CHTs. For instance, as in the analysis of Hopscotch, the shared variable modification in the remove operation and the wait in the parse phase of update operation are exact validations of a ASCY pattern. Turning the remove operation into other operations (get or put) may yield a significant increase in throughput. As latency-critical workloads are common and user-facing applications need low tail la-

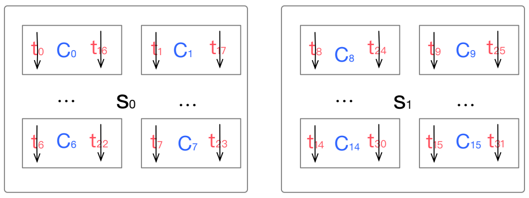


Figure 10: The hardware thread topology of Intel Xeon E5-2630.

tencies [21], principled approaches are demanded when designing low-latency concurrent hash tables.

4.5. Thread Pinning Strategies

On multi-core systems, the mapping of threads onto cores, called *thread pinning*, is of high importance [22]. In this section, we use the E5-2630 machine as an example to explain how three thread pinning strategies work. Figure 10 shows the topology of this machine. We use *likwid-topology* in the *likwid* toolkit to obtain the hardware thread topology. There are 2 sockets on it, and each socket consists of 8 cores with each containing 2 hardware threads.

- **Default strategy**. Threads are assigned to cores by the OS scheduler rather than manually. The OS scheduler tries to improve load-balance among cores. This strategy allows thread migration between cores during the execution.

- **Compact strategy**. This strategy assigns successive threads to cores that are as close as possible in the topology map of the platform. For example, if we want to create 24 threads, 16 threads are mapped to s_0 (from c_0 to c_7), and remaining threads are mapped to s_1 (from c_8 to c_{11}). Compact strategy is beneficial for high data reuse between threads.

- **Balanced strategy**. Although the compact strategy can take advantage of shared caches in the same socket, it causes load imbalance. Thus, in this strategy, we distribute threads to cores relatively evenly. For example, if we need to create 16 threads, we assign each core from c_0 to c_{15} with a single thread. If the number of threads exceeds the number of cores, hyper-thread will be used to map threads to cores with the same strategy. Figure 10 illustrates a case where 32 threads are assigned to cores using the balanced strategy. The advantage of balanced strategy is twofold. First, load balancing is maintained. Second, if the number of threads is less than the core counts, we can avoid the interference caused by hyper-threading.

Figure 11 reports the results of three CHTs (URCU is excluded, because its results are too low to be displayed on the given scale).

Under the three strategies, CLHT exhibits small variations in throughput, while for Hopscotch and Cuckoo, wide fluctuations can be observed from the figure. For E5-2630, using the default strategy as the baseline, Hopscotch obtains 100% improvement with the balanced and compact strategy when the thread count is 8. And with the compact strategy it obtains 200% improvement when the thread count reaches 16. The reason for the difference between the balanced and compact strategy is that the former involves cross-socket communication, but it is not the case for the later. E7-4850 shows similar trends.

The noteworthy speedup of Hopscotch is due to its static memory allocation. Hopscotch preallocates and initializes memory on startup, which causes the physical memory to reside on just one NUMA node because of the *first touch* allocation strategy of the Linux kernel (we use *libnuma* to evenly allocate physical memory across NUMA nodes for other CHTs expect for Hopscotch). However, Hopscotch and Cuckoo (under the compact strategy) experience a radical performance decline at the interval where the threads are incremented from 16 to 20 on E5-2630 and from 24 to 36 on E7-4850, because of the fact that cross-socket communication overhead outweighs the gain of more participating threads.

In comparison, on the AMD machine, CHTs seem not very sensitive to the pinning strategies, which can be explained from two aspects. First, the strong locality offered by the inclusive LLC of Intel Xeon makes intra-socket communication efficient. Second, the incomplete directory protocol of AMD Opteron incurs cross-socket invalidation traffic when stores are performed on owned and shared cache lines even if all sharers are in the same socket. Thus, the intra-socket performance behaves similarly to the cross-socket. Moreover, Hopscotch performs better with the default and balanced strategy when threads are spread to multiple sockets. This benefits from the balanced workload distribution given the cross-socket insensitivity on this platform.

On Phi 7120P, CLHT performs exceptionally well under all strategies (linear increase in throughput). No noticeable disparity in throughput can be observed for Cuckoo (see the almost overlapping curves). In contrast, Cuckoo obtains better performance under the default strategy on the other three NUMA systems (with high thread counts). As for Hopscotch, the compact strategy achieves better performance when the thread count ranges from 1 to around 150. The default strat-

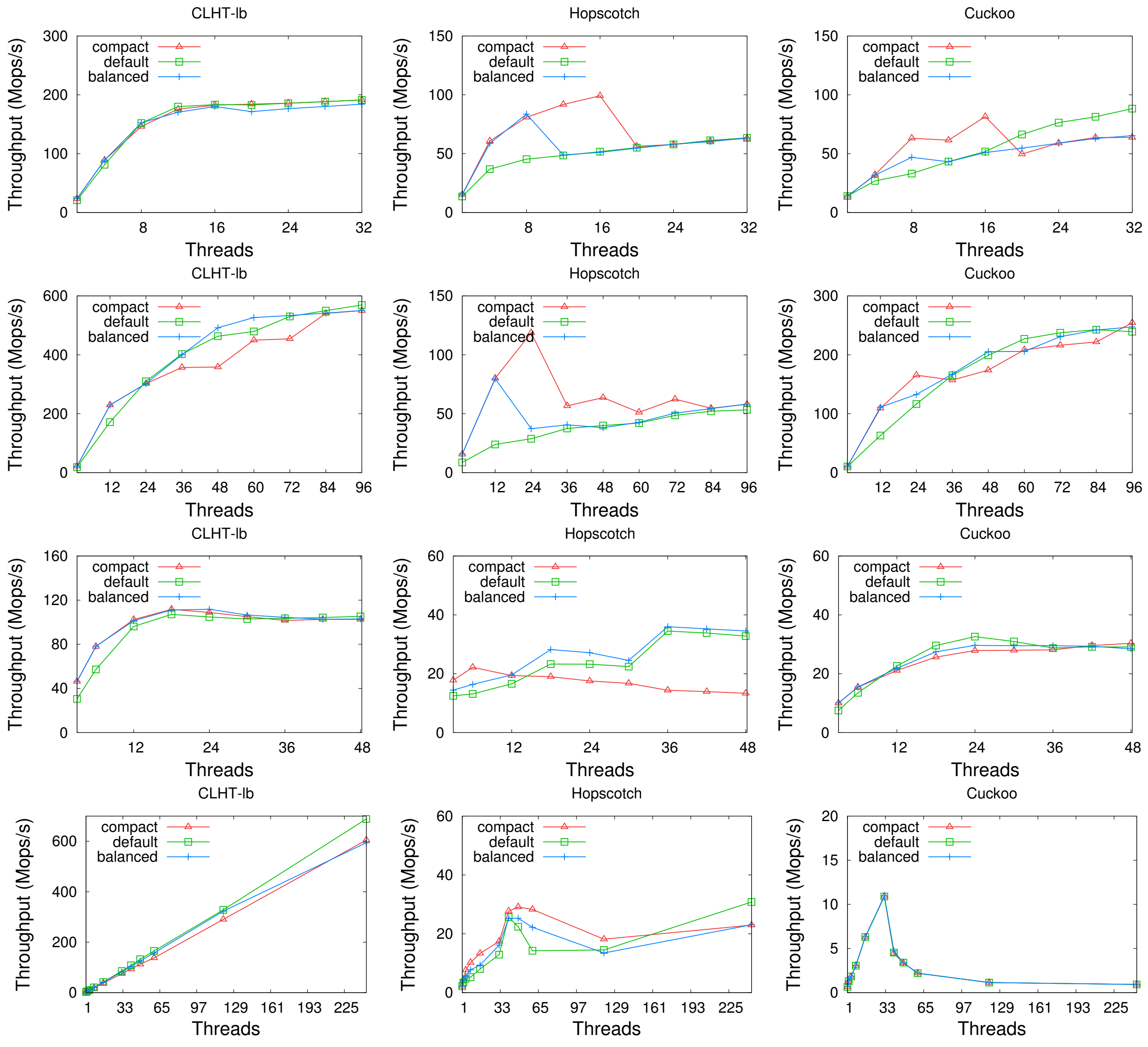


Figure 11: Throughputs measured using different pinning strategies on four platforms, Xeon E5-2630, Xeon E7-4850, AMD Opteron 6172, and Xeon Phi 7120P from top to bottom respectively.

egy prevails over others eventually for both CLHT and Hopscotch. In contrast to explicit pinning strategies that would cause more cache misses and higher memory access latency due to the small per-core L2 cache, the default strategy that relies on the OS scheduler to dynamically migrate threads across cores, can alleviate resource contentions and better utilize the cache subsystem of Xeon Phi. In addition, Xeon Phi acts like a symmetric multiprocessing (SMP) system, and the cores have the same distance to the main memory. Placing threads in different physical cores incurs similar overhead as the cross-socket overhead on NUMA systems, but the cross-core overhead is much lower, thanks to the fast bidirectional ring interconnect.

Implication 5. No universal conclusions can be made in comparing the three thread pinning strategies, because no single strategy performs portably well over different CHTs and hardwares. For example, on Intel Phi, CLHT performs better with the default strategy but differently on other platforms. And, Cuckoo's throughput *shows little variance under the three strategies on Intel Phi. Thus, to reason about abnormalities, it is important to understand the peculiarity caused by the combination of certain CHT designs, pinning strategies, thread count, and hardware-specific features. This poses non-trivial burdens on developers, and the ideal solution could be that the runtime switches different strategies adaptively by monitoring the change of hardware and workload.*

4.6. Synchronization

Synchronization is an essential part in designing CHTs, and it guarantees coordinated access to shared objects among concurrent reads and writes. If not dealt properly, it would become a major impediment to scalability. In this part, we discuss synchronization mechanisms employed in the CHTs and their impact on performance.

As we can see from Figure 7, CLHT performs better than all other CHTs especially on Phi 7120P. Besides the well-designed data structure, we attribute this scalability to its underlying synchronization mechanism. CLHT-lb uses an atomic snapshot technique to synchronize readers and writers, which incurs low synchronization overhead. Between writer threads, it use a very simple per-bucket spinlock implemented with the Fetch-and-Increment (FAI that is well supported by Xeon Phi) atomic operation. As a result, even under high concurrency, the fine-grained lock causes very low contention. Simple lock plus low contention deliver ideal scalability [5]. The lock-free version of CLHT uses a snapshot structure, which is 8 bytes in size and can be loaded, stored, or CASed with a single operation (i.e. atomically). A version number in its bucket is used for synchronizing concurrent writers, and a map is used to indicate three states (i.e. valid, invalid, or being inserted). These design choices make CLHT's performance outstanding.

Under read-only workload, Hopscotch even outperforms CLHT (not shown in the figure). But once the update operation is involved, the performance decrease significantly. The situation deteriorates even further as the percentage of update grows. This illustrates the significance of the synchronization scheme. Hopscotch use a TTAS Lock to coordinate writer threads. In order to alleviate synchronization overhead, a timestamp is used between readers and writers. The synchronization overhead between writers is much higher. The number of locks in Hopscotch is configured to be equal to its thread count, which causes more contentions as compared to CLHT. Worse, the TTAS Lock is not scalable. These reasons explain the severe performance degradation.

Cuckoo deploys a striped fine-grained spinlock based on Compare-And-Swap (CAS) atomic primitive for two kinds of synchronization (i.e., reader and writer, writer and writer). TBB uses a fine-grained per-bucket lock similar to lock-based CLHT. The difference is that TBB's per-bucket lock controls more items than CLHT, while in CLHT only 3 key-value pairs are guarded by a lock.

As shown in Figure 5(d) and Figure 7(d), on Xeon Phi, CHTs behaves remarkably different from the other three NUMA systems. Xeon Phi employs an extended MESI cache-coherence protocol that uses GOLS (Globally Owned Locally Shared) to simulate an owned state to permit sharing a modified line, a store to a cache line in GOLS state (besides the share state) also induces invalidation traffic. High coherence traffic can easily saturate its ring interconnect. CLHT's design fits pretty well with the architecture of Xeon Phi. Cuckoo exhibits poor performance on Xeon Phi (even under read-only workload), because in each lookup operation, the buckets associated with a given hash value are locked. This severely limits the concurrency, and in turn degrades its performance on this platform.

Implication 6. Synchronization plays a critical role in achieving high performance CHTs. However, designing an efficient synchronization scheme is challenging given a wide variety of factors ranging from low-level atomic primitives and architectural features to high-level lock algorithms and concurrency models. Obtaining portable performance is even more challenging as shown in our analysis, such as the abnormal low performance on Xeon Phi except for CLHT. In practice, to reason about the performance of synchronization mechanisms, we should have a holistic understanding of the multi-faceted influence of these factors. For example, we should know how to choose a preferable atomic primitive on a specific platform when designing locks that in turn depends on the cache coherence to achieve better performance. Furthermore, some locks behave optimally under high contention, but others are more competitive when the contention is low. So, it would be desirable to implement adaptive locks to take advantage of the merits of different locks.

4.7. Memory Consumption

For memory-intensive applications like CHTs, besides the performance requirement, memory consumption is also important especially for scenarios where physical memory capacity is constrained. Below, we dissect the memory usage except for Hopscotch (as mentioned, the implementation of Hopscotch in our test uses a fixed memory allocation strategy).

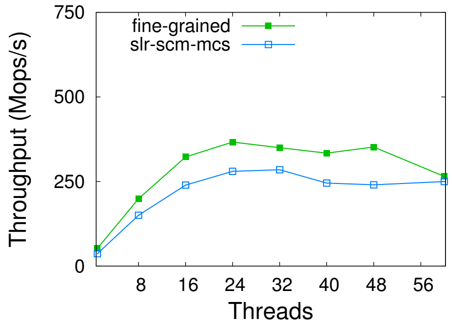
Table 4: Memory usage (MiB) on the E5-2630 machine.

i	TBB	CLHT	URCU	Cuckoo	Hopscotch
10^3	0.6	1	**0.6**	63	608.6
10^4	2.2	15.4	**1.8**	63	608.6
10^5	14.1	34.4	**7.8**	63	608.6
10^6	80	78.4	**43.9**	101	608.6
10^7	857	1024	645	**633**	608.6
10^8	5.5 GiB	8 GiB	5 GiB	**2.3** GiB	-

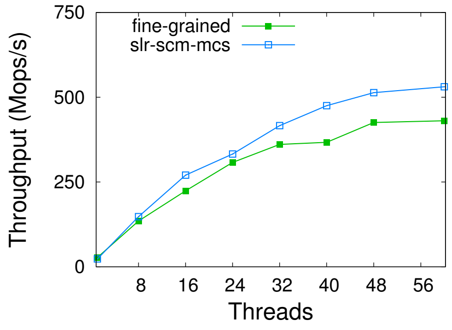
Table 4 shows the memory usage of CHTs configured with 16 threads, 10% update, and different initial sizes. The memory consumption is measured using the system monitoring tool. With the same amount of elements, Cuckoo is the most memory efficient CHT for large workloads, and URCU is second to Cuckoo and is very memory-efficient for small workloads. CLHT and TBB consume several times more memory than Cuckoo and URCU. Next, we analyze the reason behind the discrepancy of memory usage.

Cuckoo's memory efficiency mainly comes from two design choices. The first is that Cuckoo is designed with a higher set-associativity that greatly improves space utilization, and the second is the use of version counter instead of pointer to connect buckets, which improves memory efficiency especially small for key/value pairs. Both CLHT and TBB use a fine-grained per-bucket lock mechanism based on the classic separate chaining. While often faster, pointers cost more memory for workloads containing a large amount of elements. With fine-grained locking, we trade space for high performance, while the drawback is that less items can be stored with the same amount of physical memory.

Implication 7. On the one hand, lower memory consumption allows larger workloads to be served, and reduces the possibility of page swapping in the OS, which would decrease the performance impact of execution environment. On the other hand, memory-efficiency depends on internal data management mechanisms that may also affect performance to some extent. Chaining hash consumes more memory because of an extra next pointer, especially for CLHT that requires a pointer for each cache line, while Cuckoo hash is more memory-friendly due to its high capacity design. Fine-grained synchronization also contributes to memory consumption, for example, CLHT has a lock for each cache line. In summary, optimized data organization and synchronization granularity like CLHT achieve high performance but at the cost of larger memory consumption.



(a) $i = 10^3$



(b) $i = 10^6$

Figure 12: Throughput of the fine-grained locking and the global HTM-based lock on a Linux workstation with to Intel Broadwell EP/EN/EX processor (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 32 KB, 256 KB and 40 MB respectively. The update rate is 10% and the initial size is one thousand and one million respectively.

4.8. CHT with Hardvare Transactions

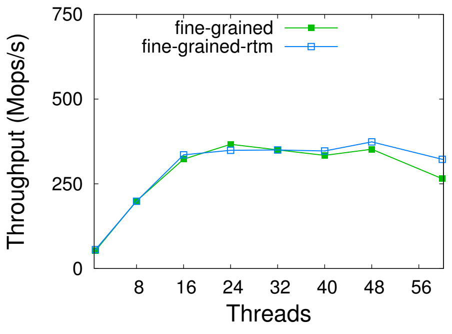
Transactional Memory (TM) is a concurrency control paradigm that provides atomic and isolated execution for code regions. It is considered to be one of the most promising solution to address the problem of programming multi-core processors. This model has the potential to provide the scalability of fine-grained locking while avoiding common pitfalls of lock composition such as deadlock. Today, both software and hardware TM methods are well researched.

In this section, we study the impact of hardware transactional memory (HTM) on constructing concurrent hash tables (only the Intel Restricted Transactional Memory is evaluated). Our HTM-based CHTs follows CLHT's design but using a different synchronization mechanism. At first, we compare a coarse-grained lock based on HTM using the optimization techniques presented in [23] with a fine-grained lock using traditional lock method. We run workloads with one thousand (less than the capacity of private cache) and one million (large than the capacity of L3 cache) initialized elements respectively, and each workload with 10% update operations. The final results are the average of 5 runs as depicted in Figure 12. For the workload with one million elements, both the fine-grained lock version and HTM variants 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 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 HTM global lock encounters more data conflicts which cause frequent transaction abort.

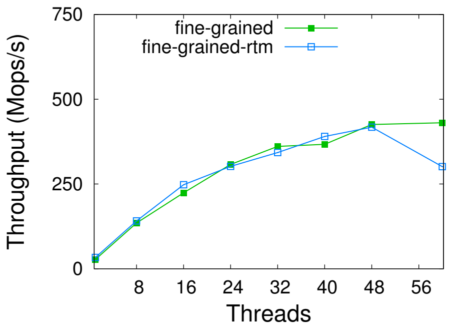
From the perspective of the complexity, HTM-based CLHT is much easier to implement than the one with fine-grained lock. It uses a single global lock to protect the critical section, while we need to pay much more attention to implement a fine-grained lock when constructing concurrent data structures. Furthermore, the fine-grained locking scheme consume 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 created 1024*1024 buckets, we need pay 8 MB more memory for the storage of synchronization variables.

On the other hand, one may ask if it is necessary to optimize the fine-grained locking with HTM when it satisfies the performance requirement very well. To answer this question, we conduct a comparison between two versions of CLHT under fine-grained locking, one with HTM and one without. As shown in Figure 13, 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 Figure 13(a), 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 8. According to the experimental results, we make three observations. Firstly, when dealing with a 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. Secondly, the workloads can 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. Lastly, traditional fine-grained locks can offer good thread scalability and performance. Under this premises, using a fine-grained lock enhanced with HTM neither brings the advantage of simplifying concurrency control, nor improves the overall performance.



(a) $i = 10^3$



(b) $i = 10^6$

Figure 13: Throughput of the traditional fine-grained lock and HTM-based fine-grained lock on Intel Broadwell. The update rate is 10% and the initial size is one thousand and one million respectively.

5. Related Work

While sequential hash tables have been studied for decades, the research on CHTs [24, 17, 25, 26, 13, 15, 27, 28, 29, 30] has been gaining popularity in recent years. In the following, we review the most relevant work to concurrent hash tables.

Synchronization. As pointed by Attiya et al., due to the constraints of cache coherence protocol, expensive synchronization in concurrent algorithms cannot be eliminated [31]. What can people do is to design good synchronization mechanisms to alleviate the synchronization overheads on multi-core systems. David et al. [5] present a thorough study of synchronization schemes on four representative multi-core systems. In a recent work, the same group of authors suggest [3] that a concurrent search data structure (CSDS) will perform better if it is designed in resemblance to sequential implementations. With four ASCY patterns in mind, they designed a concurrent hash table, which achieves high throughput and low latency due to its optimized cache line transfer. Gramoli [32] evaluated 5 different synchronization techniques using a set of 31 concurrent algorithms and developed a new micro-benchmark suite, *Synchrobench*, to measure the impact of synchronization on concurrent algorithms.

RCU [17] is a concurrent programming technique that is widely used in the Linux Kernel. As discussed in [12], RCU eases lock-based programming when locks are dynamically created and destroyed, which occurs frequently in concurrent programs. Desnoyers et al. [12] proposed the design of an user-level RCU called URCU that aims at improving read-side performance. [33] presents a predicate RCU to scale the performance under concurrent update. According to our evaluation, the write performance of the predicate RCU is still very limited.

CHTs for memcached-like systems. Memcached [2] is an in-memory key/value store for Web applications. In order to improve the memory efficiency and throughput, B. Fan et al. [15] implemented the first concurrent Cuckoo hashing algorithm that supports multiple-reader single-writer access to shared data. In [4], a concurrent cuckoo hashing that supports multiple writers was proposed. CPHash [27] is a cache-partitioned hash table. It splits the hash table into partitions and assign each partition to the L1/L2 cache of a particular core. Instead of running the lookup/insert operation locally, CPHash uses a message passing mechanism to batch queries, which can not only increase parallelism but also reduce the number of cache line transfers.

NUMA architecture and heterogeneous systems. Intel announced NUMA compatibility for its X86 and Itanium servers in the late 2007 with the Nehalem and Tukwila CPUs [34], and AMD implemented NUMA with its Opteron processor using HyperTransport [35]. Li et al. made comprehensive tests over modern hosts to show the importance of NUMA-avareness [36]. Severe shared cache misses and remote memory accesses are two main challenges for achieving high performance on NUMA machines. Locality-Aware Work-stealing (LAWS) scheduler [37] was designed to solve this problem. In LAWS, tasks are evenly partitioned by a task allocator and the data set is replicated to all memory nodes. And Zoltan Majo [38] et al. designed a scheduling algorithm, *N-MASS*, which reduces data locality to avoid the performance degradation caused by cache contention on NUMA systems. Designing hash tables and other algorithms [39, 40, 41] for heterogeneous systems such as GPUs are also gaining popularity, we plan to conduct performance evaluations on these systems in future work.

6. Conclusion

Concurrent hash table is a crucial component in modern software systems. A range of CHTs has been proposed to tackle practical problems with hardware-specific optimizations and intricate algorithm designs. However, to better understand the pros and cons of existing CHTs, an in-depth and comprehensive analysis is needed. To this end, we develop a testing framework to evaluate 5 state-of-the-art CHTs in a unified way. The measurements and comparisons are conducted over a wide spectrum of metrics, spanning from macroscopic throughput to microscopic latency, from on-chip caches and to main memory, from complex synchronization mechanisms to portable optimizations, etc. To our knowledge, this study is the most extensive evaluation on CHTs to date. We believe the implications made in this paper are valuable for both future research and algorithm development in practice.

References

[1] A. Arcangeli, M. Cao, P. E. McKenney, D. Sarma, Using read-copy-update techniques for system v ipc in the linux 2.5 kernel., in: USENIX Annual Technical Conference, FREENIX Track, 2003, pp. 297–309.

[2] Memcached, Memcached, Website, `https://www.memcached.org`.

[3] T. David, R. Guerraoui, V. Trigonakis, Asynchronized concurrency: The secret to scaling concurrent search data structures, SIGARCH Comput. Archit. News 43 (1) (2015) 631–644. doi:10.1145/2786763.2694359.

[4] X. Li, D. G. Andersen, M. Kaminsky, M. J. Freedman, Algorithmic improvements for fast concurrent cuckoo hashing, in: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, ACM, New York, NY, USA, 2014, pp. 27:1–27:14. doi:10.1145/2592798.2592820.
URL `http://doi.acm.org/10.1145/2592798.2592820`

[5] T. David, R. Guerraoui, V. Trigonakis, Everything you always wanted to know about synchronization but were afraid to ask, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, ACM, New York, NY, USA, 2013, pp. 33–48. doi:10.1145/2517349.2522714.
URL `http://doi.acm.org/10.1145/2517349.2522714`

[6] Intel, Threading building blocks, Website, https://www.threadingbuildingblocks.org.

[7] Oracle, Java 7 ee, Website, https://www.oracle.com.

[8] LPD-EPFL, Cache-line hash table, Website, https://github.com/LPD-EPFL/CLHT.

[9] Hopscotch hashing, Website, https://sites.google.com/site/cconcurrencypackage.

[10] X. L. Bin Fan, Andersen, Cuckoo hashing, Website, https://github.com/efficient/libcuckoo.

[11] liburcu, User-level read-copy update, Website, http://liburcu.org.

[12] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, J. Walpole, User-level implementations of read-copy update, Parallel and Distributed Systems, IEEE Transactions on 23 (2) (2012) 375–382.

[13] Y. Liu, K. Zhang, M. Spear, Dynamic-sized nonblocking hash tables, in: Proceedings of the 2014 ACM symposium on Principles of distributed computing, ACM, 2014, pp. 242–251.

[14] R.Pagh, E.F.Rodler, Cuckoo hashing, Journal of Algorithms 51 (2) (2004) 122–144.

[15] B. Fan, D. G. Andersen, M. Kaminsky, Memc3: Compact and concurrent memcache with dumber caching and smarter hashing., in: NSDI, Vol. 13, 2013, pp. 385–398.

[16] M. Herlihy, N. Shavit, M. Tzafrir, Hopscotch hashing, in: Distributed Computing, Springer, 2008, pp. 350–364.

[17] P. E. McKenney, J. D. Slingwine, Read-copy update: Using execution history to solve concurrency problems, in: Parallel and Distributed Computing and Systems, 1998, pp. 509–518.

[18] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Communications of the ACM 52 (4) (2009) 65–76.

[19] LIKWID, Likwid-perfctr, Website, https://code.google.com/p/likwid/wiki/LikwidPerfCtr.

[20] V. Trigonakis, Sspfd, Website, https://github.com/trigonak/sspfd.

[21] D. Jeffrey, B. Luiz Andr, The tail at scale, Communications of the ACM 56 (2) (2013) 74–80.

[22] A. Mazouz, S.-A.-A. Touati, D. Barthou, Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures, in: High Performance Computing and Simulation (HPCS), 2011 International Conference on, IEEE, 2011, pp. 273–279.

[23] Y. Afek, A. Levy, A. Morrison, Software-improved hardware lock elision, 2014, pp. 212–221.

[24] O. Shalev, N. Shavit, Split-ordered lists: Lock-free extensible hash tables, Journal of the ACM (JACM) 53 (3) (2006) 379–405.

[25] M. M. Michael, High performance dynamic lock-free hash tables and list-based sets, in: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, ACM, 2002, pp. 73–82.

[26] N. Nguyen, P. Tsigas, Lock-free cuckoo hashing, in: Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, IEEE, 2014, pp. 627–636.

[27] Z. Metreveli, N. Zeldovich, M. F. Kaashoek, Cphash: A cache-partitioned hash table, in: ACM SIGPLAN Notices, Vol. 47, ACM, 2012, pp. 319–320.

[28] J. Triplett, P. E. McKenney, J. Walpole, Resizable, scalable, concurrent hash tables via relativistic programming., in: USENIX Annual Technical Conference, 2011, p. 11.

[29] C. Click, A lock-free wait-free hash table, work presented as invited speaker at Stanford.

[30] H. Gao, J. F. Groote, W. H. Hesselink, Almost wait-free resizable hashtables, in: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, 2004, p. 50.

[31] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, M. Vechev, Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated, in: ACM Sigplan-Sigact Symposium on Principles of Programming Languages, 2011, pp. 487–498.

[32] V. Gramoli, More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, ACM, New York, NY, USA, 2015, pp. 1–10. doi:10.1145/2688500.2688501.
URL http://doi.acm.org/10.1145/2688500.2688501

[33] M. Arbel, A. Morrison, Predicate rcu: an rcu for scalable concurrent updates, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2015, pp. 21–30.

[34] R. Intel, Quickpath architecture: A new system architecture for unleashing the performace of future generations of intel r© multi-core microprocessors, a white paper by intel (2008).

[35] H. Consortium, et al., Hypertransport i/o technology overview: An optimized, low-latency board-level architecture, The HyperTransport Consortium, www.hypertransport.com.

[36] T. Li, Y. Ren, D. Yu, S. Jin, Analysis of numa effects in modern multicore systems for the design of high-performance data transfer applications, Future Generation Computer Systems 74 (2017) 41–50.

[37] Q. Chen, M. Guo, H. Guan, Laws: locality-aware work-stealing for multi-socket multi-core architectures, in: Proceedings of the 28th ACM international conference on Supercomputing, ACM, 2014, pp. 3–12.

[38] Z. Majo, T. R. Gross, Memory management in numa multicore systems: trapped between cache contention and interconnect overhead, Acm Sigplan Notices 46 (11) (2011) 11–20.

[39] Z. Choudhury, S. Purini, Heterogeneous (cpu+gpu) working-set hash tables, in: Proceedings of the 9th International Workshop on Programmability and Architectures for Heterogeneous Multicores, MULTIPROG '16, 2016.

[40] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, N. Amenta, Real-time parallel hashing on the gpu, in: SIGGRAPH Asia '09, SIGGRAPH Asia '09, ACM, New York, NY, USA, 2009, pp. 154:1–154:9.

[41] W. Zhong, J. Sun, H. Chen, J. Xiao, Z. Chen, C. Chang, X. Shi, Optimizing graph processing on gpus, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2017) 1149–1162.