2017 IEEE 23rd International Conference on Parallel and Distributed Systems

Exploring Synchronization in Cache Coherent Manycore Systems: A Case Study with Xeon Phi

Xin He, Zhiwen Chen, Jianhua Sun, Hao Chen, Dong Li†, and Zhe Quan
College of Computer Science and Electronic Engineering, Hunan University, China
†Department of Electrical Engineering and Computer Science, University of California, Merced
Email: {xinhe,zhiwenchen,jhsun,haochen}@hnu.edu.cn, †dli35@ucmerced.edu

***Abstract*—Intel Xeon Phi is a many-core architecture, featuring more than 50 cores and 200 hardware threads. Given this scale and its other distinctive architectural features, highly-concurrent applications on Xeon Phi may behave differently than on traditional multi-core systems. Yet, concurrency issues especially for synchronization intensive applications on this platform have not been thoroughly analyzed. In this paper, we conduct an extensive analysis at multiple layers, from the underlying hardware cache-coherence protocol up to the user-level applications, aiming to present the most exhaustive study of synchronization on Xeon Phi. Through a range of benchmarks, we testify the feasibility and advantage of accelerating concurrent applications with Xeon Phi. Meanwhile, we identify severe scalability issues relevant to synchronization, and solutions to these issues are discussed. We believe this work can be used as guidelines both for designing better synchronization mechanisms and in optimizing concurrent applications in order to fully exploit the capability of Xeon Phi.**

***Index Terms*—Intel Xeon Phi; Many-core; Synchronization; Scalability; Concurrent applications**

I. Introduction

Intel Xeon Phi is based on the Intel Many Integrated Core (MIC) architecture. It has over 60 cores, 30M of on-chip caches and a very fast interconnection, delivering a peak performance of 2.38 teraFLOP/s (obtained with Xeon Phi 7120P). It has been playing an significant role in supercomputers,such as STAMPEDE [1] and Tianhe-2 [2]. Despite the features it shares with many-core GPGPUs such as SIMD/SIMT, high-throughput, and high-bandwidth, the prominent advantage compared to GPGPUs is the ability of expressing algorithms with fine-grained concurrent control. Thus, leveraging xeon phi to accelerate complex concurrent applications has a promising prospect. However, in our efforts of porting highly concurrent applications to Xeon Phi, we observed scalability issues once intensive synchronization is involved even with hardware-specific optimizations. We recognize the difference of the issues relevant to synchronization between Xeon Phi and multi-core processors. Moreover, unawareness of these differences would definitely become a major impediment to the exploitation of Xeon Phi in HPC systems.

Over the last few years, a lot of research endeavors has been devoted to this new architecture. Some aim to benchmark the capability of its hardware components [3], [4], while others focus on demonstrating its performance advantages [5], [6], [1]. A portion of studies mainly regards porting and optimizing individual applications to fully utilize its hardware features [7], [8], [9]. However, existing studies do not cover synchronization issues and yet show little indication of why a given synchronization scheme scales well on a multi-core architecture but faces new challenges on Xeon Phi. Recent works [10] have conducted detailed analysis about synchronization spanning multiple layers on modern multi-processor architectures, revealing that the scalability of synchronization schemes is mainly a property of hardware. We firmly agree on this view that synchronization is exceedingly related to the low-level details of the underlying hardware. Therefore, we believe a thorough analysis of synchronization on Xeon Phi combining the underlying hardware architecture and upper-level algorithms is highly demanded.

In this paper, we perform a comprehensive study of synchronization on Xeon Phi, which ranges from the basic hardware architecture to complicated concurrent software. We design a set of micro-benchmarks targeting cache-coherence protocol, atomic primitives and locks (part of the micro-benchmarks are redesigned for Xeon Phi based on [10]. We port three representative concurrent hash tables (CHTs) [11], [12], [13] to this platform in order to dissect their synchronization behaviors. In order for comparison with traditional multi-core processor, experiments about CHTs are also performed on a two-socket Intel SandyBridge EP machine. To the best of our knowledge, this is the most thorough study of synchronization on Xeon Phi. Our results unveil some undocumented architectural features closely related to synchronization and identify several performance issues caused by improper synchronization schemes on Xeon Phi. We discuss solutions to alleviate these problems. The systematic investigation of synchronization on Xeon Phi induces the following observations.

Cross-core communication is harmful to efficient synchronization. Cross-core communication causes higher overhead than intra-core (usually by an order of magnitude) on Xeon Phi.

Access latency is dependent on the distributed tag directory but not the core distance. On Xeon Phi, a distributed tag directory system (DTD) is employed to keep all the L2 caches coherent, and a hash function based on the address of the cache line is used to determine the line-to-DTD mapping. Plus the impact of the ring bus, the distance between the core and DTD can cause variations in access latency, which renders the distance between cores less relevant to latency.

* The first two authors contributed equally to this work.

978-1-5386-2129-5/17/31.00 ©2017 IEEE
DOI 10.1109/ICPADS.2017.00040

CAS and FAI are more efficient on Xeon Phi. System designers should exploit the best performing atomic operations like CAS and FAI on Xeon Phi to implement locks and other synchronization schemes.

Locks should be chosen based on the degree of contention. While complex locks are generally more suitable for extreme contention scenarios, simple locks are preferable under low contention.

Fine-grained concurrency control is more preferable on Xeon Phi. Given the scale of this new architecture as well as its low CPU frequency, in-order execution, and other architectural features, our study reveals that fine-grained synchronization benefits scalability on Xeon Phi.

Explicit thread pinning may hurt performance on Xeon Phi. It may not be beneficial to manually place threads for concurrent applications that have random memory access pattern as on NUMA systems. Instead, the default thread scheduler provided by the operating system can often achieve better performance.

II. Related Work

In this section, we briefly discuss the work most related to our study.

Studies on Xeon Phi. A large body of work has been devoted to the study and benchmark of Xeon Phi since Intel released its first generation product in 2012 [3], [4], [8], [9]. These efforts usually focus on either evaluating the capabilities of Xeon Phi's hardware components, such as the cores, memory, on-chip and off-chip (PCIe) interconnect, or optimizing specific application on Xeon Phi to fully utilize its powerful hardware capabilities. In addition, in [14], the authors investigate the cache-coherent architecture of Xeon Phi by developing an intuitive performance model. However, existing studies on Xeon Phi have not yet mentioned synchronization issues.

Synchronization. In a recent work [10], the authors conducted a detailed analysis on a broad range of synchronization schemes, and the evaluation was performed on four mainstream representative multi-core platforms (Xeon Phi not included). One of the conclusions made in this work is that synchronization is mainly a property of the hardware. Part of our study is based on their approaches (focus on Xeon Phi). We confirm some of their observations such as that synchronization is more relevant to the underlying hardware. Moreover, we identify distinctive synchronization issues on Xeon Phi from its unique hardware architecture. Another work [15] presents an in-depth benchmarking of the impact of synchronization on concurrent algorithms. Nevertheless, they mainly focus on synchronization algorithms, and the evaluations have not been conducted on Xeon Phi. The paper [16] covers the synchronization issue of Xeon Phi by only focusing on analysis and optimization of barrier synchronization mechanism. In [17], the authors analyzed both the latency and bandwidth of atomic operations on Xeon Phi. Our work differs in that we provide a holistic view about synchronization from low-lever primitives to upper-level algorithms, which can help clarify confusing factors in designing practical systems.

Concurrent Hash Tables. Considering the widespread use of concurrent hash tables in software systems, and our intension to explore synchronization on Xeon Phi using real concurrent applications, we select three state-of-the-art concurrent hash tables including CLHT [11], Hopscotch hash [13], and concurrent Cuckoo hash [12]. All the chosen CHTs have been shown to be scalable in their original work. We port these CHTs to Xeon Phi as native applications. This is also the first work to benchmark CHTs on Xeon Phi. At the same time, we have identified severe scalability issues related to synchronization when running these CHTs on Xeon Phi, and the root causes are addressed with platform specific features.

III. Background

Designing synchronization schemes for concurrent applications on a specific platform is closely related to both hardware-level mechanisms and software-level algorithms. The cache-coherence protocol is one of such hardware-level mechanisms that can maintain the consistency when accessing shared data stored in multiple local caches. The cache-coherence protocol implements load and store operations that are fundamental for a hardware architecture. Besides, more advanced operations like atomic primitives are also provided, such as compare-and-swap and fetch-and-increment, which can be leveraged to implement locks and other synchronization mechanisms (lock-free and wait-free algorithms).

The Xeon Phi's cache-coherence protocol is implemented using a directory protocol based on MESI that uses GOLS (Globally Owned Locally Shared) to simulate an Owned state. Thus, sharing a modified line among cores is allowed. The primary goal is to avoid write-back to the main memory when another core tries to read a modified cache-line. Therefore, the shared state does not mean that the line has not been modified. Each core's cache maintains MESI states of its cache lines and the Distributed Tag Directories (DTDs) hold the global GOLS coherency state of each line. Cache lines are assigned to each DTD according to the line address instead of the core that is holding or requesting the cache line.

Software-level synchronization algorithms are typically built on top of hardware-level primitives. For instance, locks, as the most widely-used synchronization technique, are used to protect critical sections.

IV. Motivation

Considering the widespread use of concurrent hash tables and the compute capability of Xeon Phi, we ported three state-of-the-art concurrent hash tables (i.e. CLHT [11], Hopscotch [13], and Cuckoo [12]) to Xeon Phi as native applications to explore the potential performance acceleration. We employed some essential optimizations such as thread affinity control strategy in our experiments, expecting decent performance speedup.

Beyond our positive expectation, both Hopscotch and Cuckoo encounter severe performance issues (executing in native mode) especially when update operations are involved. Hopscotch scales poorly as more hardware threads are

spawned, and exhibits a huge performance degradation with read-write workload. We did not observe similar behaviors on another two-socket Xeon machine. However, under read-only workload, Hopscotch shows excellent scalability and reaches peak performance (1.210 billion ops/s) when 244 hardware threads are exploited. CLHT is the only one that always exhibits perfect scalability regardless of the types of workloads on Xeon Phi. It achieves maximized throughput of 2.2 billion ops/s with read-only workload, approximately 2 billion ops/s with 20% updates, and surprisingly more than 1.8 billion ops/s with update-only workload.

As a result, we believe that the performance issues encountered are more relevant to the underlying synchronization mechanisms. The differences in hardware architecture between the two platforms may result in different synchronization behavior and consequently performance variations, which can be explained from three aspects. First, many more cores usually encourage higher concurrency, which would incur more contentions. Second, guaranteeing cache-coherence in large scale may demand extra hardware complexities. Third, underlying synchronization instructions may not perform portably well across platforms due to different design requirements. Having a holistic view on these issues can help system designers understand pitfalls and alternatives in developing or optimizing parallel applications for Xeon Phi.

V. Experimental Environments and Configurations

In this section, we first introduce our experimental platforms. Then, the benchmarking programs used in our experiments are detailed.

A. Platforms

In order to obtain comparable results in the evaluation of concurrent hash tables, we also conduct experiments on an Intel SandyBridge EP machine.

Intel Xeon Phi 7120P. Our experiment platform is Xeon Phi 7120P that integrates 64 in-order cores on the same chip. Each core is clocked at 1.238 GHz and supports 4 hardware threads, thus having 244 hardware threads in total. All cores are connected by a high-speed on-die bidirectional bus. The main memory 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 31MB of L2 cache on the chip.

Intel SandyBridge EP. The Intel SandyBridge EP machine consists of two sockets each with 8 out-of-order cores (totaling 32 hardware threads) and 16GB memory. It operates at 1.8GHz and offers 32KB L1 cache, 256KB L2 cache, and 20MB LLC. It has 4 memory channels and 2 QPIs.

B. Design of Benchmarks

We port and redesign three benchmarks for Xeon Phi. We take advantage of the benchmark suite introduced in SSYNC [10] that was used to study synchronization on four multi-core platforms. The benchmarks can measure the cost of operations on a single cache line according to a certain line's MESI state and location in the system. Measuring the cost of individual atomic operations and several representative lock algorithms is also implemented in the benchmarks. As the binary architecture (*k1om*) of Xeon Phi is not completely compatible with x86_64, we rewrite some in-line assembly to make it compatible with the MIC instruction set. For instance, memory fences like *mfence* are not supported by Xeon Phi, so we use the alternative like *lock; addl $0,(%%rsp)* instead. The instruction *clfush* is also not supported and it had to be replaced by *clevict* for manually controlling both L1 and L2 caches. As for the *pause* instruction, we replace it with the *delay* instruction. In addition, because Xeon Phi 7120P is not a NUMA architecture, we exclude some benchmarking cases that are specifically designed for NUMA systems, and adopt new thread-to-core mappings to our benchmarks according to the topology of Xeon Phi.

For concurrent applications, we select three state-of-the-art concurrent hash tables (CHTs), namely CLHT [11], Hopscotch [13], and Cuckoo [12]. The three CHTs are ported to Xeon Phi with platform-specific optimizations, and tested under a unified framework that provides flexible parameter configuration and the collection of a wide range of performance metrics.

VI. Evaluation Results and Analysis

In this section, we first describe the impact of cache-coherence protocol on synchronization on Xeon Phi using a set of micro-benchmarks, mainly focusing on the latency of *load*, *store* operations and *atomic primitives*. Then, we present the evaluation on the scalability of atomic primitives supported by Xeon Phi in terms of throughput and latency. Finally, we present application-level analysis including lock algorithms, and concurrent hash tables.

A. Hardware Protocol Analysis

In order to better understand the following analysis, we briefly describe the testing logic. This micro-benchmark is mainly used to measure the cost of operations on a cache line according to the line's MESI state and location. It keeps a cache line in a specified state and then performs a access by either a local or a remote core. More than 30 cases are supported, such as load from modified state and compare-and-swap on shared lines. Note that at least three processes are required in all the test cases designed for the shared and owned state (others do not have this restriction).

The latency metric can be used to estimate the overhead of sharing a cache line on a shared-memory system. We collect latency values for basic operations such as load and store, as well as atomic primitives including compare-and-swap (CAS), fetch-and-increment (FAI), test-and-set (TAS), and swap (SWAP).

Local Accesses. L1 cache is directly integrated into the core to facilitate low-latency and high-speed access to the main memory. The L1 cache can hold 32KB data and its access time is 1 cycles. A cache miss in L1 would necessitate

TABLE I
LATENCIES (CYCLES) OF THE CACHE COHERENCE TO LOAD/STORE/CAS/FAI/TAS/SWAP A CACHE LINE DEPENDING ON THE MESI STATE AND THE DISTANCE. THE VALUES ARE THE AVERAGE OF 10000 REPETITIONS WITH 3% STANDARD DEVIATION.

System	Intel Xeon Phi	
Hops	same-core	other-core
loads		
Modified	25	269
Owned (GOLS)	315	26
Exclusive	25	233
Shared	30	38
stores		
Modified	37	268
Shared	250	347
Exclusive	34	244
Owned (GOLS)	37	220
atomic operations: CAS (C), FAI (F), TAS (T), SWAP (S)		
Operation	C/F/T/S	C/F/T/S
Modified	39/38/34/41	287/298/296/260
Shared	301/386/269/318	330/363/370/321

another access to the local L2 cache or the ones on remote cores via the ring interconnect. Each core's local L2 cache has the capacity of 512KB. Cache coherence is automatically maintained among the L2 caches of all the cores, effectively creating a virtual cache of 31MB. L2 cache is composed of 64-byte cache lines with 8-way associativity, and its access time is approximately 11 cycles. Accessing the L2 cache of a remote core takes longer, though Xeon Phi has no cross-socket communication overhead as in multi-socket architectures. Main memory access typically consumes hundreds of cycles.

Table I shows the results of performing load, store, and atomic operations on a cache line based on its previous state and location. Because accessing an invalid cache line is equivalent to accessing the main memory, in the following discussion, we do not consider the invalid state.

Loads. When processes are within the same core, except for the Owned state, the load operation performed on a local cache line has similar (low) latency regardless of its current state. As for the Owned state in the two scenarios (same-core and other-core), the counter-intuitive variations in latency might be caused by the interaction with DTDs [14], which usually occurs with the extended shared state (the Owned state). On the contrary, for both the Modified and Exclusive state, if the target cache line resides in a remote core (other-core), the access latency increases dramatically by more than 10-fold as compared to the same-core situation. This is due to the remote cache latency and off-chip memory access. For the Shared state, the latency remains approximately the same in both cases, because the process can get the shared cache line from another process that happens to be in the same core as the former even in the other-core scenario, avoiding remote cache access.

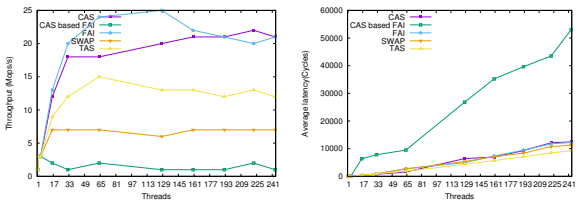
Stores. In the same-core scenario, a store has similar latency for the three states (Modified, Exclusive, and Owned) without being affected by the previous state of a cache line. For instance, the latency for the Modified, Exclusive, and Owned state is 37, 34, and 37 cycles respectively. However, a store operation on a shared cache line incurs much longer latency (250 cycles), which is induced by the invalidation of the cache line shared by a remote core. When processes are spread among different cores, much larger latency (by an order of magnitude) can be observed regardless of the cache line state. This is because the store operation introduces remote access latency for the Modified and Exclusive state, and extra cache-coherence traffic especially for the Owned and Shared state due to the maintenance of consistency for individual copies of the cache line among many cores. In addition, even with the high speed ring interconnect, communication with a remote core definitely have to pay extra overhead.

Atomic operations. Like contemporary multi-core processors, Xeon Phi also supports atomic primitives, such as CAS, FAI, TAS, and SWAP. In the same-core case, the four atomic operations have nearly the same latency with the Modified state. But the latency on the Shared state increases by a factor of 10 because of the same reason as discussed for stores. When cross-core communication is involved, we can observe similar variations for atomic primitives at the Modified state as in the load/store case. While for the Shared state, there is no noticeable difference between intra-core and inter-core latency. The invalidation traffic across sharers should be responsible for this phenomenon. Accessing remote cache lines (other-core) with the Shared state consumes a little bit more cycles than with the Modified state.

Discussion. The investigation on latency discloses some important issues that should be noticed to avoid inefficiencies in synchronization. First, because of the design of supporting four simultaneous hardware threads per physical core, the single-chip many-core architecture of Xeon Phi shares some resemblances with multi-chip systems. For instance, inter-core communication is almost 10 times expensive than intra-core. Second, within the same core, the inclusive L1 and L2 cache makes the intra-core synchronization more efficient. Third, as all the cores connected with the ring interconnect can be regarded as symmetrical peers, the distance between communicating cores has little effect on the access latency. On the contrary, the distance between the core and the DTD can cause variations in access latency. Remote access latency is not dependent on the cache line state except loading from a shared or owned state. Fourth, due to the fact that memory requests from threads residing in the same core are serialized, running threads on separate cores can help improve the utilization of memory bandwidth [3]. In addition, Xeon Phi schedules threads running on the same core in a round-robin fashion, which indicates that issuing instructions from the same thread context in back-to-back cycles is impossible. In summary, taking these factors into account, spawning two or three threads on the same core is preferred to obtain desired synchronization performance.

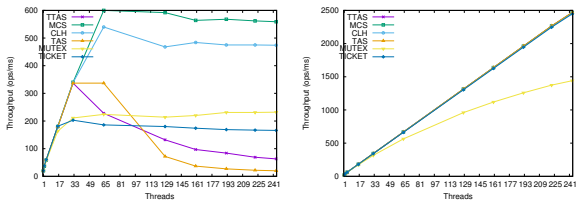
B. Stressing Atomic Operations

In this test, the logic is to let each thread try to perform an atomic operation on a single shared location repeatedly. For



(a) throughput (b) latency

Fig. 1. Throughput and latency of different atomic operations on a single memory location on Xeon Phi.



(a) 1 lock (b) 512 locks

Fig. 2. Throughput of different locking algorithms using a single lock and 512 locks.

FAI, SWAP, and CAS_FAI, these calls are always eventually successful (write to the target memory), whereas it is not the case for TAS and CAS. CAS_FAI implements an FAI operation based on CAS. This enables us to highlight both the costs of spinning until the CAS succeeds and the benefits of having an FAI instruction supported by the hardware.

We sequentially place threads onto cores. In order to reduce deviation, each experiment is repeated five times to compute the average. Figure 1(a) and Figure 1(b) present the performance results of this experiment. On the one hand, a fast increase in throughput within the same core (the first 4 threads as shown in Figure 1(a)) can be observed (see the steep slop that is almost parallel to the y axis). On the other hand, the fast increase in throughput slows down or decrease when the shared location begins to be accessed by threads from a different core (see the changing point where thread counts exceed 4). CAS based FAI exhibits dramatic decrease in performance in this stage. With more participating threads, the throughput reaches a maximum and then plateaus or declines due to increasing contentions. An appealing point worth highlighting is that both CAS and FAI (two widely used atomic operations) outperform other atomic operations. The worst performing atomic operation, CAS based FAI, illustrates the importance of having a FAI instruction supported by the hardware. TAS shows better performance in throughput and latency than SWAP, but still largely lags behind CAS and FAI.

Discussion. Xeon Phi is designed with lower CPU frequency, which results in lower single-thread performance. Furthermore, it is inevitable to suffer performance penalty from cross-core communication. However, even with these constrains, atomic operations can scale to a large number of contending threads. System designers should leverage performant atomic operations (CAS and FAI) to implement scalable and effective synchronization schemes.

C. Analysis of Locks

In this section, we explore the behaviors of several representative locking algorithms under different levels of contention. Before moving onto the details of experimental results, we first make a brief introduction to the characteristics of the selected locks except MUTEX (provided by POSIX pthread). These locks can be divided into two categories, namely simple locks and complex locks, all of which employ a busy-waiting (or called spinning) technique. Simple locks include TAS, TTAS, and TICKET, whose main advantage is small memory footprint. The disadvantage is that more cache-coherence traffic is generated with all threads spinning on the same location [18]. On the contrary, MCS and CLH are queue based locks that are more complicated and produce less cache-coherence traffic by having each thread spin on different locations, but with a larger memory footprint.

The behaviors of Locking algorithms. We study the performance of locks under extreme and very low contention because two situations are commonly met in practice. First, highly-contended locks are usually the main impediment to the scalability of lock-based synchronization. Second, many concurrent applications employ fine-grained locks in designing synchronization mechanisms, which induces low contention. We measure the throughput of lock acquisitions that is executed by each lock. Each thread randomly acquires a lock, then performs read and write operations on one corresponding cache line of data, and finally releases the lock. In the extreme contention experiment (one lock), a thread pauses after releasing the lock to guarantee that the released lock can be seen by other cores before it retries to acquire the lock. Considering that Xeon Phi is a single-socket architecture, we exclude hierarchical locks that are tailored for NUMA architectures from the evaluation.

Extreme contention. The results for the extreme contention experiment are illustrated in Figure 2(a). One core evaluation (thread count is less than 4) achieves the fastest growth in throughput (not clearly observable from the figure because of the scale). As expected, CLH and MCS outperform others under highly-contended environment where the spawned threads are up to 200+. When the number of threads ranges from 1 to 65, CLH and MCS show a linear increase in throughput, because both locking algorithms have each thread spin on a distinct location. Once a thread releases a lock, it only needs to invalidate its successor's cache, thus reducing the invalidation traffic to a minimum. As for the TAS and TTAS lock, they are not as resilient to contention as the CLH and MCS lock, a turning point can be seen when the thread count reaches 33. Even worse, we can see a steep declining trend when more threads are joined. Compared to the TAS and TTAS lock, the MUTEX and TICKET lock achieve lower peak performance. However, they are more resilient to high contentions (reach a plateau eventually), and both retain higher performance over TAS and TTAS when the thread count exceeds 100.

Very low contention. The results for low-contention experiment are shown in Figure 2(b). In general, given the complexity of queue locks, simple locks are more competitive than complicated ones when the synchronization contention is low. We can see that the differences between locks is nearly negligible except for MUTEX. The MUTEX lock performs poorly on this platform especially when the number of threads increases.

Discussion. No lock consistently outperforms the others in all cases. While complex locks are more suitable for extreme contention, simple locks often do better under low contention, especially within the same core. On Xeon Phi, optimal synchronization performance is often gained when threads are confined to a single core for both high and low contention.

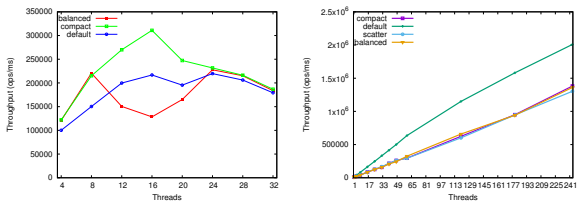
D. Case Study: Concurrent Hash Tables

In this section, we evaluate three state-of-the-art concurrent hash tables (CHTs): CLHT [11], Hopscotch [13] and Cuckoo [12], with read-only and mixed workloads. The synthesized workloads with configurable *update* intensity can help identify insights in different synchronization mechanisms. The mixed workload contains 90% *get*, 5% *put*, and 5% *remove* operations. Note that the scalability trend remains the same with other configurations that include update operations. The initial size of CHTs is set to 1024 (1024 key/value pairs).

In order to distinguish different synchronization issues from multi-core processors and underline the energy efficiency of Xeon Phi, we also benchmark the CHTs on a two-socket 16-core Intel SandyBridge EP machine. On Xeon Phi, the cores and memory controllers are connected by a bi-directional ring. Shared components like the ring stop and DTDs would be bottlenecked when multiple threads are requesting data simultaneously. Thus, employing different thread pinning strategies certainly can help to reason the impact of system components on the performance as discussed in [3].

There are four kinds of thread-to-core mapping strategies on Xeon Phi. (a) Compact - the cores are placed close to each other. (b) Scattered - the cores are divided evenly among the 61 physical cores. (c) Balanced - it is similar to *scattered* but threads with adjacent numbers are placed on the same core. (d) Default - threads are scheduled by the OS without explicit thread pinning. We use the compact, balanced and default strategy for cross-platform comparison and analysis. On Xeon Phi, the four mapping strategies are all evaluated. The following analysis is conducted from three aspects. First, we investigate the impact of different pinning strategies on the performance of CHTs on Xeon Phi. Second, a cross-platform comparison is presented. Third, we introspect the observed behaviors of CHTs relevant to synchronization on Xeon Phi, and give our explanations from the perspective of hardware architecture and algorithmic designs.

1) Thread Pinning: Figure 3 depicts the results of lock-based CLHT with an update ratio 10% and corresponding pinning strategies on the two platforms. Similar results for other CHTs are also observed, and not shown in the paper due to space constrains. On the SandyBridge machine, the throughput exhibits wide fluctuations with different strategies. The lock-based CLHT with the compact strategy achieves peak performance when the thread count is 16, and sharp declination occurs subsequently due to the cross-socket communication overhead. In comparison, the outcome is completely different on Xeon Phi. The default strategy beats all other competitors, and no noticeable difference can be observed for the explicit thread pinning strategies (see the overlapping throughput curves). Furthermore, the gap between the default strategy and others widens with the increasing number of threads. This particular phenomenon on CHTs also differs from scientific computing applications that usually benefit more from a scattered thread pinning. We explain the reasons for these discrepancies as follows.



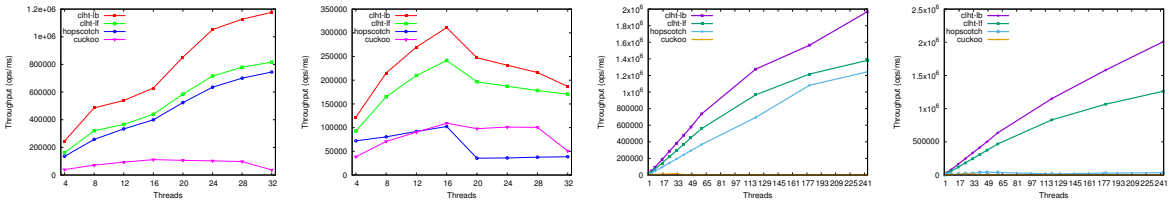
(a) CLHT-lb on SandyBridge (b) CLHT-lb on 7120P

Fig. 3. Running CLHT-lb on Intel SandyBridge EP and Xeon Phi 7120P with different thread pinning strategies (90% *get*, 5% *put*, and 5% *remove*)

As the SandyBridge machine is a typical NUMA system, remote access overhead is larger than local access. Applications using the compact thread pinning can benefit more from data locality if threads are located in the same socket. 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 communication overhead as the cross-socket overhead on NUMA systems, but the cross-core overhead is much lower, thanks to the fast bidirectional ring interconnect. In addition, given that CHTs are typical memory-bound applications with random memory access, in contrast to explicit strategy that would cause more cache misses and higher average latency of memory access 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, which is the rationale behind its sustained performance superiority.

2) Cross-Platform Behaviors of CHTs: In order to have a straightforward comparison between Xeon Phi and SandyBridge EP, we study the behaviors of CHTs under different levels of synchronization intensity. We only report experimental results with optimal pinning strategies (the default and compact strategy for Xeon Phi and SandyBridge EP respectively). Experiments are run with update ratios 0% and 10%, as illustrated in Figure 4.

In all cases, the differences between Xeon Phi and SandyBridge are prominent. Under the read-only workload, CHTs exhibit outstanding scalability on both platforms except for



(a) SandyBridge/compact (*update* 0%) (b) SandyBridge/compact (*update* 10%) (c) Phi 7120P/default (*update* 0%) (d) Phi 7120P/default (*update* 10%)

Fig. 4. Running CLHT on SandyBridge EP and Xeon Phi with optimal thread pinning strategies.

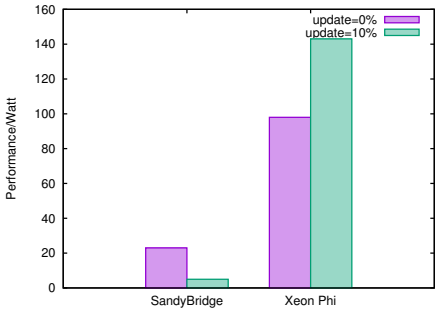


Fig. 5. Comparison of power efficiency of the best performing CHT (CLHT-lb) on two platforms.

TABLE II
ESTIMATED LATENCY IMPACT (ELI) OF CHTS MEASURED WITH INTEL VTUNE.

	CLHT-lb	CLHT-lf	Hopscotch	Cuckoo
ELI	236.5	499.5	741.9	11695.5

TABLE III
ESTIMATED LATENCY IMPACT (ELI) OF CLHT-LB WITH DIFFERENT THREAD PINNING STRATEGIES MEASURED WITH INTEL VTUNE.

	default	compact	scattered	balanced
ELI	236.5	298.0	294.3	2685.7

Cuckoo whose performance is even worse on Xeon Phi. While for workloads containing update operations, there are two striking phenomenons worth mentioning. First, two versions of CLHT (lock-free and lock-based) both scale linearly even with 10% update operations, and reach the peak performance (about 2 billion ops/s and 1.25 billion ops/s respectively) with the maximum hardware threads spawned. Second, Hopscotch scales well and peaks with 244 threads fully functioning (1.2 billion ops/s) under read-only workload. But its performance degrades significantly (nearly by three orders of magnitude) even with a small portion of update operations compared to the read-only case, and appending more threads contributes little to the performance. However, Hopscotch does not suffer such severe performance deterioration on the SandyBridge machine in the presence of updates. This observation pinpoints that weak synchronization mechanisms would exacerbate the scalability on Xeon Phi. We have also identified similar issues for other CHTs such as TBB-base and RCU-based hash tables that are not discussed in this paper.

As power efficiency has become increasingly important, we additionally present a comparison of power efficiency in terms of performance per watt. The results depicted in Figure 5 are calculated based on the peak throughput of the best performing CHT (CLHT-lb) and the corresponding power consumption. We can observe that a scalable concurrent application running on Xeon Phi is much more energy-efficient than on a multi-core platform. The advantages in high performance and power efficiency demonstrate the superiority of accelerating traditional concurrent applications with Xeon Phi.

3) Synchronization Analysis of CHTs: Next, we present a detailed analysis about the aforementioned phenomena from the perspective of hardware architectures and synchronization designs of the evaluated CHTs, based on the tool Intel VTune Amplifier. For instance, many-core systems possess the ability to reach a high concurrency level, but we can not solely depend on the hardware to mitigate contentions without considering algorithmic optimizations on synchronization. Additional complexity in enabling coherence to handle high memory bandwidth would also cause large synchronization overhead on Xeon Phi.

As pointed out in [11], the cache-coherence traffic produced by stores on shared data is the biggest impediment to the scalability of concurrent applications on multi-core systems. The situation becomes even worse on Xeon Phi given the scale of hundreds of hardware threads sharing a cache line. In addition, because 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, which in turn produces cache misses of future accesses. Worse, the high volume coherence traffic can easily saturate the ring interconnect, resulting in higher latency. We find that the *remove* call in Hopscotch is the most time-consuming operation (nearly occupy over 90% execution time) when we run it on Xeon Phi with 10% update and 244 threads. We attribute this to a shared *timestamp* field in Hopscotch that is modified in the *remove* operation to guarantee concurrent *get* calls to be failed. The functionality of *timestamp* is similar to the atomic

snapshot in CLHT. The only distinction is that CLHT does not require to store on a share variable in the *remove* method. This particular design in Hopscotch causes non-trivial coherence traffic. Turning the *remove* operation into other operations (*get* or *put*) can yield a significant increase in throughput.

We analyze how CLHT achieves near-linear scalability on Xeon Phi in the following. For CLHT, the key to its remarkable performance is that each bucket is aligned to a single cache line and the update is in-place, which greatly reduces cache-line transfers. Another benefit of aligned data is that it can avoid false sharing that is critical to the performance on Xeon Phi. Furthermore, its per-bucket lock scheme incurs low contention, and a simple spinlock implemented using FAI that has been shown to be well supported by Xeon Phi in Section VI-B, is also beneficial for its performance. On the contrary, Hopscotch uses a TTAS lock that is not scalable on Xeon Phi, and its coarse-grained locking scheme (the number of locks is equal to thread counts) generates more contentions at runtime, which together deteriorates the performance.

We can further demonstrate these issues through a metric named Estimated Latency Impact (ELI provided in VTune). ELI indicates that the majority of L1 data cache misses are not being serviced by the L2 cache. As reported in Table II, CLHT-lb has the lowest score, which means CLHT can largely benefit from the unified L2 cache without too much penalty of accessing the main memory compared to the others. Base on this metric, we can also infer that the default thread pinning strategy (in contrast to explicit thread pinning) is more preferable to CHTs featuring random memory access. As shown in Table III, the ELI value obtained when using the default strategy is much lower.

VII. Conclusion and Future Work

In this paper, we perform an extensive analysis on Xeon Phi from basic hardware cache-coherence protocol and primitives all the way up to user-level concurrent applications. By evaluating micro-benchmarks and real-world concurrent hash tables, we induce a set of key insights as follows.

In order to achieve high scalability on Xeon Phi, concurrent applications should avoid remote cache access as much as possible. The access latency is not determined by the distance between cores but the DTDs. Atomic operations CAS and FAI are more scalable and recommended to use in implementing synchronization mechanisms on Xeon Phi. Locks algorithms should be adopted according to the degree of contention. Moreover, fine-grained concurrency control is preferable, and explicitly pinning threads may hurt performance on Xeon Phi for concurrent applications that exhibit random memory access pattern.

We believe that our analysis can be used to develop better synchronization schemes or tune concurrent applications on Xeon Phi. In terms of future work, we plan to study other hardware features of Xeon Phi such as vectorization.

VIII. Acknowledgment

This research was supported in part by the National Science Foundation of China under grants 61772183, 61572179, 61272190, and 61602166.

References

[1] F. Wende and T. Steinke, "Swendsen-wang multi-cluster algorithm for the 2d/3d ising model on xeon phi and gpu," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 83.

[2] X. Liao, "Milkyway-2: back to the world top 1," *Frontiers of Computer Science Selected Publications from Chinese Universities*, vol. 8, no. 3, pp. 343–344, 2014.

[3] J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che, and C. Xu, "An empirical study of intel xeon phi," *arXiv preprint arXiv:1310.5842*, 2013.

[4] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving intel xeon phi," in *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. ACM, 2014, pp. 137–148.

[5] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1085–1097.

[6] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 273–282.

[7] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh, "Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 11, pp. 3066–3078, 2015.

[8] C. Rosales, "Porting to the intel xeon phi: Opportunities and challenges," in *Extreme Scaling Workshop (XSW), 2013*. IEEE, 2013, pp. 1–7.

[9] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh, "Improving main memory hash joins on intel xeon phi processors: An experimental approach," *Proceedings of the VLDB Endowment*, vol. 8, no. 6, pp. 642–653, 2015.

[10] T. David, R. Guerraoui, and 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*. ACM, 2013, pp. 33–48.

[11] ——, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 631–644.

[12] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 27.

[13] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing," in *Distributed Computing*. Springer, 2008, pp. 350–364.

[14] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent smp systems: a case-study with xeon phi," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 2013, pp. 97–108.

[15] V. Gramoli, "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 1–10.

[16] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján, "Effective barrier synchronization on intel xeon phi coprocessor," in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 588–600.

[17] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 445–456.

[18] M. Herlihy and N. Shavit, "The art of multiprocessor programming," in *PODC*, vol. 6, 2006, pp. 1–2.