Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask

Tudor David, Rachid Guerraoui, Vasileios Trigonakis*

School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

{tudor.david, rachid.guerraoui, vasileios.trigonakis}@epfl.ch

Abstract

This paper presents the most exhaustive study of synchronization to date. We span multiple layers, from hardware cache-coherence protocols up to high-level concurrent software. We do so on different types of architectures, from single-socket – uniform and non-uniform – to multi-socket – directory and broadcast-based – many-cores. We draw a set of observations that, roughly speaking, imply that scalability of synchronization is mainly a property of the hardware.

1 Introduction

Scaling software systems to many-core architectures is one of the most important challenges in computing today. A major impediment to scalability is *synchronization*. From the Greek "syn", i.e., *with*, and "khronos", i.e., *time*, synchronization denotes the act of coordinating the timeline of a set of processes. Synchronization basically translates into cores slowing each other, sometimes affecting performance to the point of annihilating the overall purpose of increasing their number [3, 21].

A synchronization scheme is said to scale if its performance does not degrade as the number of cores increases. Ideally, acquiring a lock should for example take the same time regardless of the number of cores sharing that lock. In the last few decades, a large body of work has been devoted to the design, implementation, evaluation, and application of synchronization schemes [1, 4, 6, 7, 10, 11, 14, 15, 26–29, 38–40, 43–45]. Yet, the designer of a concurrent system still has little indication, a priori, of whether a given synchronization scheme will scale on a given modern many-core architecture and, a posteriori, about exactly why a given scheme did, or did not, scale.

depth	breadth
concurrent software hash table, Memcached, STM	**single-socket** uniform (Sun Niagara 2) non-uniform (Tilera TILE-Gx36)
primitives locks, message passing	
atomic operations CAS, FAI, TAS, SWAP	**multi-socket** directory-based (AMD Opteron) broadcast-based (Intel Xeon)
cache coherence loads, stores	

Figure 1: Analysis method.

One of the reasons for this state of affairs is that there are no results on modern architectures connecting the low-level details of the underlying hardware, e.g., the cache-coherence protocol, with synchronization at the software level. Recent work that evaluated the scalability of synchronization on modern hardware, e.g., [11, 26], was typically put in a specific application and architecture context, making the evaluation hard to generalize. In most cases, when scalability issues are faced, it is not clear if they are due to the underlying hardware, to the synchronization algorithm itself, to its usage of specific atomic operations, to the application context, or to the workload.

Of course, getting the complete picture of how synchronization schemes behave, in every single context, is very difficult. Nevertheless, in an attempt to shed some light on such a picture, we present the most exhaustive study of synchronization on many-cores to date. Our analysis seeks completeness in two directions (Figure 1).

1. We consider multiple synchronization layers, from basic many-core hardware up to complex concurrent software. First, we dissect the latencies of cache-coherence protocols. Then, we study the performance of various atomic operations, e.g., compare-and-swap, test-and-set, fetch-and-increment. Next, we proceed with locking and message passing techniques. Finally, we examine a concurrent hash table, an in-memory key-value store, and a software transactional memory.

*Authors appear in alphabetical order.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522714

2. We vary a set of important architectural attributes to better understand their effect on synchronization. We explore both single-socket (chip multi-processor) and multi-socket (multi-processor) many-cores. In the former category, we consider uniform (e.g., Sun Niagara 2) and non-uniform (e.g, Tilera TILE-Gx36) designs. In the latter category, we consider platforms that implement coherence based on a directory (e.g., AMD Opteron) or broadcast (e.g., Intel Xeon).

Our set of experiments, of what we believe constitute the most commonly used synchronization schemes and hardware architectures today, induces the following set of **observations**.

Crossing sockets is a killer. The latency of performing any operation on a cache line, e.g., a store or a compare-and-swap, simply does not scale across sockets. Our results indicate an increase from 2 to 7.5 times compared to intra-socket latencies, even under no contention. These differences amplify with contention at all synchronization layers and suggest that cross-socket sharing should be avoided.

Sharing within a socket is necessary but not sufficient. If threads are not explicitly placed on the same socket, the operating system might try to load balance them across sockets, inducing expensive communication. But, surprisingly, even with explicit placement within the same socket, an incomplete cache directory, combined with a non-inclusive last-level cache (LLC), might still induce cross-socket communication. On the Opteron for instance, this phenomenon entails a 3-fold increase compared to the actual intra-socket latencies. We discuss one way to alleviate this problem by circumventing certain access patterns.

Intra-socket (non-)uniformity does matter. Within a socket, the fact that the distance from the cores to the LLC is the same, or differs among cores, even only slightly, impacts the scalability of synchronization. For instance, under high contention, the Niagara (uniform) enables approximately 1.7 times higher scalability than the Tilera (non-uniform) for all locking schemes. The developer of a concurrent system should thus be aware that highly contended data pose a higher threat in the presence of even the slightest non-uniformity, e.g., non-uniformity inside a socket.

Loads and stores can be as expensive as atomic operations. In the context of synchronization, where memory operations are often accompanied with memory fences, loads and stores are generally not significantly cheaper than atomic operations with higher consensus numbers [19]. Even without fences, on data that are not locally cached, a compare-and-swap is roughly only 1.35 (on the Opteron) and 1.15 (on the Xeon) times more expensive than a load.

Message passing shines when contention is very high. Structuring an application with message passing reduces sharing and proves beneficial when a large number of threads contend for a few data. However, under low contention and/or a small number of cores, locks perform better on higher-layer concurrent testbeds, e.g., a hash table and a software transactional memory, even when message passing is provided in hardware (e.g., Tilera). This suggests the exclusive use of message passing for optimizing certain highly contended parts of a system.

Every locking scheme has its fifteen minutes of fame. None of the nine locking schemes we consider consistently outperforms any other one, on all target architectures or workloads. Strictly speaking, to seek optimality, a lock algorithm should thus be selected based on the hardware platform and the expected workload.

Simple locks are powerful. Overall, an efficient implementation of a ticket lock is the best performing synchronization scheme in most low contention workloads. Even under rather high contention, the ticket lock performs comparably to more complex locks, in particular within a socket. Consequently, given their small memory footprint, ticket locks should be preferred, unless it is sure that a specific lock will be very highly contended.

A high-level **ramification** of many of these observations is that the scalability of synchronization appears, first and above all, to be a property of the hardware, in the following sense. Basically, in order to be able to scale, synchronization should better be confined to a single socket, ideally a uniform one. On certain platforms (e.g., Opteron), this is simply impossible. Within a socket, sophisticated synchronization schemes are generally not worthwhile. Even if, strictly speaking, no size fits all, a proper implementation of a simple ticket lock seems enough.

In summary, our main **contribution** is the most exhaustive study of synchronization to date. Results of this study can be used to help predict the cost of a synchronization scheme, explain its behavior, design better schemes, as well as possibly improve future hardware design. SSYNC, the cross-platform synchronization suite we built to perform the study is, we believe, a contribution of independent interest. SSYNC abstracts various lock algorithms behind a common interface: it not only includes most state-of-the-art algorithms, but also provides platform specific optimizations with substantial performance improvements. SSYNC also contains a library that abstracts message passing on various platforms, and a set of microbenchmarks for measuring the latencies of the cache-coherence protocols, the locks, and the message passing. In addition, we provide implementations of a portable software transac-

Name	**Opteron**	**Xeon**	**Niagara**	**Tilera**
System	AMD Magny Cours	Intel Westmere-EX	SUN SPARC-T5120	Tilera TILE-Gx36
Processors	4× AMD Opteron 6172	8× Intel Xeon E7-8867L	SUN UltraSPARC-T2	TILE-Gx CPU
# Cores	48	80 (no hyper-threading)	8 (64 hardware threads)	36
Core clock	2.1 GHz	2.13 GHz	1.2 GHz	1.2 GHz
L1 Cache	64/64 KiB I/D	32/32 KiB I/D	16/8 KiB I/D	32/32 KiB I/D
L2 Cache	512 KiB	256 KiB		256 KiB
Last-level Cache	2×6 MiB (shared per die)	30 MiB (shared)	4 MiB (shared)	9 MiB Distributed
Interconnect	6.4 GT/s HyperTransport (HT) 3.0	6.4 GT/s QuickPath Interconnect (QPI)	Niagara2 Crossbar	Tilera iMesh
Memory #Channels / #Nodes	128 GiB DDR3-1333 4 per socket / 8	192 GiB Sync DDR3-1067 4 per socket / 8	32 GiB FB-DIMM-400 8 / 1	16 GiB DDR3-800 4 / 2
OS	Ubuntu 12.04.2 / 3.4.2	Red Hat EL 6.3 / 2.6.32	Solaris 10 u7	Tilera EL 6.3 / 2.6.40

Table 1: The hardware and the OS characteristics of the target platforms.

tional memory, a concurrent hash table, and a key-value store (Memcached [30]), all built using the libraries of SSYNC. SSYNC is available at:

`http://lpd.epfl.ch/site/ssync`

The rest of the paper is organized as follows. We recall in Section 2 some background notions and present in Section 3 our target platforms. We describe SSYNC in Section 4. We present our analyses of synchronization from the hardware and software perspectives, in Sections 5 and 6, respectively. We discuss related work in Section 7. In Section 8, we summarize our contributions, discuss some experimental results that we omit from the paper because of space limitations, and highlight some opportunities for future work.

2 Background and Context

Hardware-based Synchronization. Cache coherence is the hardware protocol responsible for maintaining the consistency of data in the caches. The cache-coherence protocol implements the two fundamental operations of an architecture: load (read) and store (write). In addition to these two operations, other, more sophisticated operations, i.e., atomic, are typically also provided: compare-and-swap, fetch-and-increment, etc.

Most contemporary processors use the MESI [37] cache-coherence protocol, or a variant of it. MESI supports the following states for a cache line: **M**odified: the data are stale in the memory and no other cache has a copy of this line; **E**xclusive: the data are up-to-date in the memory and no other cache has a copy of this line; **S**hared: the data are up-to-date in the memory and other caches might have copies of the line; **I**nvalid: the data are invalid.

Coherence is usually implemented by either *snooping* or using a *directory*. By snooping, the individual caches monitor any traffic on the addresses they hold in order to ensure coherence. A directory keeps approximate or precise information of which caches hold copies of a memory location. An operation has to consult the directory, enforce coherence, and then update the directory.

Software-based Synchronization. The most popular software abstractions are *locks*, used to ensure mutual exclusion. Locks can be implemented using various techniques. The simplest are spin locks [4, 20], in which processes spin on a common memory location until they acquire the lock. Spin locks are generally considered to scale poorly because they involve high contention on a single cache line [4], an issue which is addressed by queue locks [29, 43]. To acquire a queue lock, a thread adds an entry to a queue and spins until the previous holder hands the lock. Hierarchical locks [14, 27] are tailored to today's non-uniform architectures by using node-local data structures and minimizing accesses to remote data. Whereas the aforementioned locks employ busy-waiting techniques, other implementations are cooperative. For example, in case of contention, the commonly used Pthread Mutex adds the core to a wait queue and is then suspended until it can acquire the lock.

An alternative to locks we consider in our study is to partition the system resources between processes. In this view, synchronization is achieved through message passing, which is either provided by the hardware or implemented in software [5]. Software implementations are generally built over cache-coherence protocols and impose a single-writer and a single-reader for the used cache lines.

3 Target Platforms

This section describes the four platforms considered in our experiments. Each is representative of a specific type of many-core architecture. We consider two large-scale multi-socket multi-cores, henceforth called the *multi-sockets*[1], and two large-scale chip multi-processors (CMPs), henceforth called the *single-sockets*. The multi-sockets are a 4-socket AMD Opteron (*Opteron*) and an 8-socket Intel Xeon (*Xeon*), whereas the CMPs are an 8-core Sun Niagara 2 (*Niagara*) and a 36-core Tilera TILE-Gx36 (*Tilera*). The characteristics of the four platforms are detailed in Table 1.

All platforms have a single die per socket, aside from the Opteron, that has two. Given that these two dies are actually organized in a 2-socket topology, we use the term *socket* to refer to a single die for simplicity.

[1]We also test a 2-socket Intel Xeon and a 2-socket AMD Opteron.

3.1 Multi-socket – Directory-based: Opteron

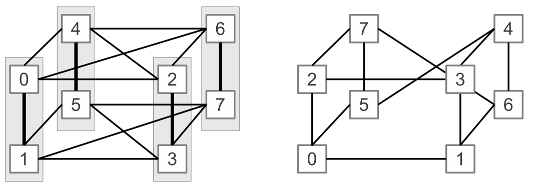
The 48-core AMD Opteron contains four multi-chip modules (MCMs). Each MCM has two 6-core dies with independent memory controllers. Hence, the system comprises, overall, eight memory nodes. The topology of the system is depicted in Figure 2(a). The maximum distance between two dies is two hops. The dies of an MCM are situated at a 1-hop distance, but they share more bandwidth than two dies of different MCMs.

The caches of the Opteron are write-back and non-inclusive [13]. Nevertheless, the hierarchy is not strictly exclusive; on an LLC hit the data are pulled in the L1 but may or may not be removed from the LLC (decided by the hardware [2]). The Opteron uses the MOESI protocol for cache coherence. The 'O' stands for the *owned* state, which indicates that this cache line has been modified (by the owner) but there might be more shared copies on other cores. This state allows a core to load a modified line of another core without the need to invalidate the modified line. The modified cache line simply changes to *owned* and the new core receives the line in shared state. Cache coherence is implemented with a broadcast-based protocol, assisted by what is called the HyperTransport Assist (also known as the probe filter) [13]. The probe filter is, essentially, a directory residing in the LLC[2]. An entry in the directory holds the owner of the cache line, which can be used to directly probe or invalidate the copy in the local caches of the owner core.

3.2 Multi-socket – Broadcast-based: Xeon

The 80-core Intel Xeon consists of eight sockets of 10-cores. These form a twisted hypercube, as depicted in Figure 2(b), maximizing the distance between two nodes to two hops. The Xeon uses inclusive caches [23], i.e., every new cache-line fill occurs in all the three levels of the hierarchy. The LLC is write-back; the data are written to the memory only upon an eviction of a modified line due to space or coherence. Within a socket, the Xeon implements coherence by snooping. Across sockets, it broadcasts snoop requests to the other sockets. Within the socket, the LLC keeps track of which cores might have a copy of a cache line. Additionally, the Xeon extends the MESI protocol with the *forward* state [22]. This state is a special form of the shared state and indicates the only cache that will respond to a load request for that line (thus reducing bandwidth usage).

[2]Typically, the probe filter occupies 1MiB of the LLC.



(a) AMD Opteron (b) Intel Xeon

Figure 2: The system topologies.

3.3 Single-socket – Uniform: Niagara

The Sun Niagara 2 is a single-die processor that incorporates 8 cores. It is based on the chip multi-threading architecture; it provides 8 hardware threads per core, totaling 64 hardware threads. Each L1 cache is shared among the 8 hardware threads of a core and is write-through to the LLC. The 8 cores communicate with the shared LLC through a crossbar [36], which means that each core is equidistant from the LLC (*uniform*). The cache-coherence implementation is directory-based and uses duplicate tags [32], i.e., the LLC cache holds a directory of all the L1 lines.

3.4 Single-socket – Non-uniform: Tilera

The Tilera TILE-Gx36 [41] is a 36-core chip multiprocessor. The cores, also called tiles, are allocated on a 2-dimensional mesh and are connected with Tilera's iMesh on-chip network. iMesh handles the coherence of the data and also provides hardware message passing to the applications. The Tilera implements the Dynamic Distributed Cache technology [41]. All L2 caches are accessible by every core on the chip, thus, the L2s are used as a 9 MiB coherent LLC. The hardware uses a distributed directory to implement coherence. Each cache line has a *home tile*, i.e., the actual L2 cache where the data reside if cached by the distributed LLC. Consider, for example, the case of core *x* loading an address homed on tile *y*. If the data are not cached in the local L1 and L2 caches of *x*, a request for the data is sent to the L2 cache of *y*, which plays the role of the LLC for this address. Clearly, the latency of accessing the LLC depends on the distance between *x* and *y*, hence the Tilera is a *non-uniform* cache architecture.

4 SSYNC

SSYNC is our cross-platform synchronization suite; it works on x86_64, SPARC, and Tilera processors. SSYNC contains `libslock`, a library that abstracts lock algorithms behind a common interface and `libssmp`, a library with fine-tuned implementations of message passing for each of the four platforms. SSYNC also includes microbenchmarks for measuring the latencies of the cache coherence, the locks, and the message passing, as well as `ssht`, i.e., a cache efficient hash table and TM^{2}C, i.e., a highly portable transactional memory.

4.1 Libraries

libslock. This library contains a common interface and optimized implementations of a number of widely used locks. `libslock` includes three spin locks, namely the test-and-set lock, the test-and-test-and-set lock with exponential back-off [4, 20], and the ticket lock [29]. The queue locks are the MCS lock [29] and the CLH lock [43]. We also employ an array-based lock [20]. `libslock` also contains hierarchical locks, such as the hierarchical CLH lock [27] and the hierarchical ticket lock (hticket) [14][3]. Finally, `libslock` abstracts the Pthread Mutex interface. `libslock` also contains a cross-platform interface for atomic instructions and other architecture dependent operations, such as fences, thread and memory placement functions.

libssmp. `libssmp` is our implementation of message passing over cache coherence[4] (similar to the one in Barrelfish [6]). It uses cache line-sized buffers (messages) in order to complete message transmissions with single cache-line transfers. Each buffer is one-directional and includes a byte flag to designate whether the buffer is empty or contains a message. For client-server communication, `libssmp` implements functions for receiving from any other, or from a specific subset of the threads. Even though the design of `libssmp` is identical on all platforms, we leverage the results of Section 5 to tailor `libssmp` to the specifics of each platform individually.

4.2 Microbenchmarks

ccbench. `ccbench` is a tool for measuring the cost of operations on a cache line, depending on the line's MESI state and placement in the system. `ccbench` brings the cache line in the desired state and then accesses it from either a local or a remote core. `ccbench` supports 30 cases, such as store on modified and test-and-set on shared lines.

stress tests. SSYNC provides tests for the primitives in `libslock` and `libssmp`. These tests can be used to measure the primitives' latency or throughput under various conditions, e.g., number and placement of threads, level of contention.

4.3 Concurrent Software

Hash Table (ssht). `ssht` is a concurrent hash table that exports three operations: `put`, `get`, and `remove`. It is designed to place the data as efficiently as possible in the caches in order to (i) allow for efficient prefetching and (ii) avoid false sharing. `ssht` can be configured to use any of the locks of `libslock` or the message passing of `libssmp`.

Transactional Memory (TM^{2}C). TM^{2}C [16] is a message passing-based software transactional memory system for many-cores. TM^{2}C is implemented using `libssmp`. TM^{2}C also has a shared memory version built with the spin locks of `libslock`.

[3]In fact, based on the results of Section 5 and without being aware of [14], we designed and implemented the hticket algorithm.

[4]On the Tilera, it is an interface to the hardware message passing.

5 Hardware-Level Analysis

In this section, we report on the latencies incurred by the hardware cache-coherence protocols and discuss how to reduce them in certain cases. These latencies constitute a good estimation of the cost of sharing a cache line in a many-core platform and have a significant impact on the scalability of any synchronization scheme. We use `ccbench` to measure basic operations such as load and store, as well as compare-and-swap (CAS), fetch-and-increment (FAI), test-and-set (TAS), and swap (SWAP).

5.1 Local Accesses

Table 3 contains the latencies for accessing the local caches of a core. In the context of synchronization, the values for the LLCs are worth highlighting. On the Xeon, the 44 cycles is the local latency to the LLC, but also corresponds to the fastest communication between two cores on the same socket. The LLC plays the same role for the single-socket platforms, however, it is directly accessible by all the cores of the system. On the Opteron, the non-inclusive LLC holds both data and the cache directory, so the 40 cycles is the latency to both. However, the LLC is filled with data only upon an eviction from the L2, hence the access to the directory is more relevant to synchronization.

5.2 Remote Accesses

Table 2 contains the latencies to load, store, or perform an atomic operation on a cache line based on its previous state and location. Notice that the accesses to an invalid line are accesses to the main memory. In the following, we discuss all cache-coherence states, except for the invalid. We do not explicitly measure the effects of the forward state of the Xeon: there is no direct way to bring a cache line to this state. Its effects are included in the load from shared case.

Loads. On the Opteron, a load has basically the same latency regardless of the previous state of the line; essentially, the steps taken by the cache-coherence protocol are always the same. Interestingly, although the two dies of an MCM are tightly coupled, the benefits are rather small. The latencies between two dies in an MCM and two dies that are simply directly connected differ by roughly 12 cycles. One extra hop adds an additional overhead of 80 cycles. Overall, an access over two hops is approximately 3 times more expensive than an access within a die.

The results in Table 2 represent the best-case scenario

	Opteron	Xeon	Niagara	Tilera
L1	3	5	3	2
L2	15	11		11
LLC	40	44	24	45
RAM	136	355	176	118

Table 3: Local caches and memory latencies (cycles).

System	Opteron				Xeon			Niagara		Tilera	
State \ Hops	same die	same MCM	one hop	two hops	same die	one hop	two hops	same core	other core	one hop	max hops
loads											
Modified	81	161	172	252	109	289	400	3	24	45	65
Owned	83	163	175	254	-	-	-	-	-	-	-
Exclusive	83	163	175	253	92	273	383	3	24	45	65
Shared	83	164	176	254	44	223	334	3	24	45	65
Invalid	136	237	247	327	355	492	601	176	176	118	162
stores											
Modified	83	172	191	273	115	320	431	24	24	57	77
Owned	244	255	286	291	-	-	-	-	-	-	-
Exclusive	83	171	191	271	115	315	425	24	24	57	77
Shared	246	255	286	296	116	318	428	24	24	86	106
atomic operations: CAS (C), FAI (F), TAS (T), SWAP (S)											
Operation	all	all	all	all	all	all	all	C/F/T/S	C/F/T/S	C/F/T/S	C/F/T/S
Modified	110	197	216	296	120	324	430	71/108/64/95	66/99/55/90	77/51/70/63	98/71/89/84
Shared	272	283	312	332	113	312	423	76/99/67/93	66/99/55/90	124/82/121/95	142/102/141/115

Table 2: 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.

for the Opteron: at least one of the involved cores resides on the memory node of the directory. If the directory is remote to both cores, the latencies increase proportionally to the distance. In the worst case, where two cores are on different nodes, both 2-hops away from the directory, the latencies are 312 cycles. Even worse, even if both cores reside on the same node, they still have to access the remote directory, wasting any locality benefits.

In contrast, the Xeon does not have the locality issues of the Opteron. If the data are present within the socket, a load can be completed locally due to the inclusive LLC. Loading from the shared state is particularly interesting, because the LLC can directly serve the data without needing to probe the local caches of the previous holder (unlike the modified and exclusive states). However, the overhead of going off-socket on the Xeon is very high. For instance, loading from the shared state is 7.5 times more expensive over two hops than loading within the socket.

Unlike the large variability of the multi-sockets, the results are more stable on the single-sockets. On the Niagara, a load costs either an L1 or an L2 access, depending on whether the two threads reside on the same core. On the Tilera, the LLC is distributed, hence the latencies depend on the distance of the requesting core from the home tile of the cache line. The cost for two adjacent cores is 45 cycles, whereas for the two most remote cores[5], it is 20 cycles higher (2 cycles per hop).

Stores. On the Opteron, both loads and stores on a modified or an exclusive cache line have similar latencies (no write-back to memory). However, a store on a shared line is different[6]. Every store on a shared or owned cache line incurs a broadcast invalidation to all nodes. This happens because the cache directory is incomplete (it does not keep track of the sharers) and does not in any way detect whether sharing is limited within the node[7]. Therefore, even if all sharers reside on the same node, a store needs to pay the overhead of a broadcast, thus increasing the cost from around 83 to 244 cycles. Obviously, the problem is aggravated if the directory is not local to any of the cores involved in the store. Finally, the scenario of storing on a cache line shared by all 48 cores costs 296 cycles.

Again, the Xeon has the advantage of being able to locally complete an operation that involves solely cores of a single node. In general, stores behave similarly regardless of the previous state of the cache line. Finally, storing on a cache line shared by all 80 cores on the Xeon costs 445 cycles.

Similarly to a load, the results for a store exhibit much lower variability on the single-sockets. A store on the Niagara has essentially the latency of the L2, regardless of the previous state of the cache line and the number of sharers. On the Tilera, stores on a shared line are a bit more expensive due to the invalidation of the cache lines of the sharers. The cost of a store reaches a maximum of 200 cycles when all 36 cores share that line.

Atomic Operations. On the multi-sockets, CAS, TAS, FAI, and SWAP have essentially the same latencies. These latencies are similar to a store followed by a memory barrier. On the single-sockets, some operations clearly have different hardware implementations. For instance, on the Tilera, the FAI operation is faster than the others. Another interesting point is the latencies for performing an operation when the line is shared by all the cores of the system. On all platforms, the latencies follow the exact same trends as a store in the same scenario.

[5]10 hops distance on the 6-by-6 2-dimensional mesh of the Tilera.

[6]For the store on shared test, we place two different sharers on the indicated distance from a third core that performs the store.

[7]On the Xeon, the inclusive LLC is able to detect if there is sharing solely within the socket.

Implications. The latencies of cache coherence reveal some important issues that should be addressed in order to implement efficient synchronization. Cross-socket communication is 2 to 7.5 times more expensive than intra-socket communication. The problem on the broadcast-based design of Xeon is larger than on the Opteron. However, within the socket, the inclusive LLC of the Xeon provides strong locality, which in turn translates into efficient intra-socket synchronization. In terms of locality, the incomplete directory of the Opteron is problematic in two ways. First, a read-write pattern of sharing will cause stores on owned and shared cache lines to exhibit the latency of a cross-socket operation, even if all sharers reside on the same socket. We thus expect intra-socket synchronization to behave similarly to the cross-socket. Second, the location of the directory is crucial: if the cores that use some memory are remote to the directory, they pay the remote access overhead. To achieve good synchronization performance, the data have to originate from the local memory node (or to be migrated to the local one). Overall, an Opteron MCM should be treated as a two-node platform.

The single-sockets exhibit quite a different behavior: they both use their LLCs for sharing. The latencies (to the LLC) on the Niagara are uniform, i.e., they are affected by neither the distance nor the number of the involved cores. We expect this uniformity to translate to synchronization that is not prone to contention. The non-uniform Tilera is affected both by the distance and the number of involved cores, therefore we expect scalability to be affected by contention. Regarding the atomic operations, both single-sockets have faster implementations for some of the operations (see Table 2). These should be preferred to achieve the best performance.

5.3 Enforcing Locality

A store to a shared or owned cache line on the Opteron induces an unnecessary broadcast of invalidations, even if all the involved cores reside on the same node (see Table 2). This results in a 3-fold increase of the latency of the store operation. In fact, to avoid this issue, we propose to explicitly maintain the cache line to the modified state. This can be easily achieved by calling the `prefetchw` x86 instruction before any load reference to that line. Of course, this optimization should be used with care because it disallows two cores to simultaneously hold a copy of the line.

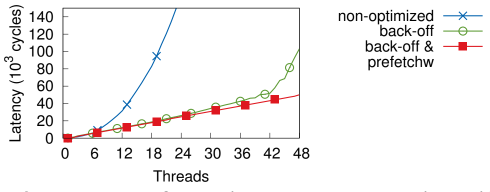


Figure 3: Latency of acquire and release using different implementations of a ticket lock on the Opteron.

To illustrate the potential of this optimization, we engineer an efficient implementation of a ticket lock. A ticket lock consists of two counters: the *next* and the *current*. To acquire the lock, a thread atomically fetches and increases the *next* counter, i.e., it obtains a *ticket*. If the *ticket* equals the *current*, the thread has acquired the lock, otherwise, it spins until this becomes true. To release the lock, the thread increases the value of the *current* counter.

A particularly appealing characteristic of the ticket lock is the fact that the *ticket*, subtracted by the *current* counter, is the number of threads queued before the current thread. Accordingly, it is intuitive to spin with a back-off proportional to the number of threads queued in front [29]. We use this back-off technique with and without the `prefetchw` optimization and compare the results with a non-optimized implementation of the ticket lock. Figure 3 depicts the latencies for acquiring and immediately releasing a single lock. Obviously, the non-optimized version scales terribly, delivering a latency of 720K cycles on 48 cores. In contrast, the versions with the proportional back-off scale significantly better. The `prefetchw` gives an extra performance boost, performing up to 2 times better on 48 cores.

SSYNC uses the aforementioned optimization wherever possible. For example, the message passing implementation on the Opteron with this technique is up to 2.5 times faster than without it.

5.4 Stressing Atomic Operations

In this test, we stress the atomic operations. Each thread repeatedly tries to perform an atomic operation on a single shared location. For FAI, SWAP, and CAS_FAI these calls are always eventually successful, i.e., they write to the target memory, whereas for TAS and CAS they are not. CAS_FAI implements a FAI operation based on CAS. This enables us to highlight both the costs of spinning until the CAS is successful and the benefits of having a FAI instruction supported by the hardware. After completing a call, the thread pauses for a sufficient number of cycles to prevent the same thread from completing consecutive operations locally (long runs [31])[8].

On the multi-sockets, we allocate threads on the same socket and continue on the next socket once all cores of the previous one have been used. On the Niagara, we divide the threads evenly among the eight physical cores. On all platforms, we ensure that each thread allocates its local data from the local memory node. We repeat each experiment five times and show the average value.

[8]The delay is proportional to the maximum latency across the involved cores and does not affect the total throughput in a way other than the intended.

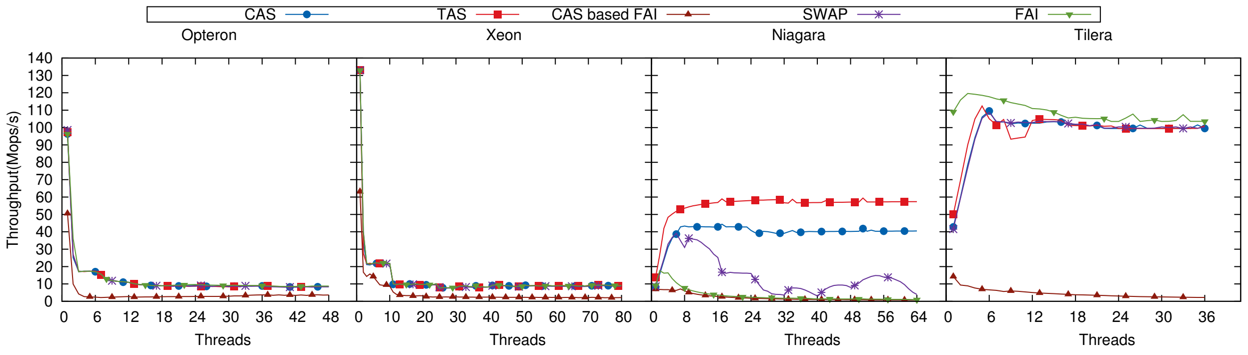


Figure 4: Throughput of different atomic operations on a single memory location.

Figure 4 shows the results of this experiment. The multi-sockets exhibit a very steep decrease in the throughput once the location is accessed by more than one core. The latency of the operations increases from approximately 20 to 120 cycles. In contrast, the single-sockets generally show an increase in the throughput on the first few cores. This can be attributed to the cost of the local parts of the benchmark (e.g., a while loop) that consume time comparable to the latency of the operations. For more than six cores, however, the results stabilize (with a few exceptions).

Both the Opteron and the Xeon exhibit a stable throughput close to 20 Mops/s within a socket, which drops once there are cores on a second socket. Not surprisingly (see Table 2), the drop on the Xeon is larger than on the Opteron. The throughput on these platforms is dictated by the cache-coherence latencies, given that an atomic operation actually brings the data in its local cache. In contrast, on the single-sockets the throughput converges to a maximum value and exhibits no subsequent decrease. Some further interesting points worth highlighting are as follows. First, the Niagara (SPARC architecture) does not provide an atomic increment or swap instruction. Their implementations are based on CAS, therefore the behavior of FAI and CAS_FAI are practically identical. SWAP shows some fluctuations on the Niagara, which we believe are caused by the scheduling of the hardware threads. However, SPARC provides a hardware TAS implementation that proves to be highly efficient. Likewise, the FAI implementation on the Tilera slightly outperforms the other operations.

Implications. Both multi-sockets have a very fast single-thread performance, that drops on two or more cores and decreases further when there is cross-socket communication. Contrarily, both single-sockets have a lower single-thread throughput, but scale to a maximum value, that is subsequently maintained regardless of the number of cores. This behavior indicates that globally stressing a cache line with atomic operations will introduce performance bottlenecks on the multi-sockets, while being somewhat less of a problem on the single-sockets. Finally, a system designer should take advantage of the best performing atomic operations available on each platform, like the TAS on the Niagara.

6 Software-Level Analysis

This section describes the software-oriented part of this study. We start by analyzing the behavior of locks under different levels of contention and continue with message passing. We use the same methodology as in Section 5.4. In addition, the globally shared data are allocated from the first participating memory node. We finally report on our findings on higher-level concurrent software.

6.1 Locks

We evaluate the locks in SSYNC under various degrees of contention on our platforms.

6.1.1 Uncontested Locking

In this experiment we measure the latency to acquire a lock based on the location of the previous holder. Although in a number of cases acquiring a lock does involve contention, a large portion of acquisitions in applications are uncontested, hence they have a similar behavior to this experiment.

Initially, we place a single thread that repeatedly acquires and releases the lock. We then add a second thread, as close as possible to the first one, and pin it

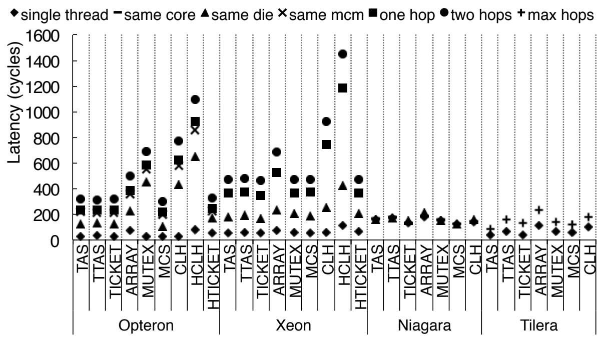


Figure 6: Uncontested lock acquisition latency based on the location of the previous owner of the lock.

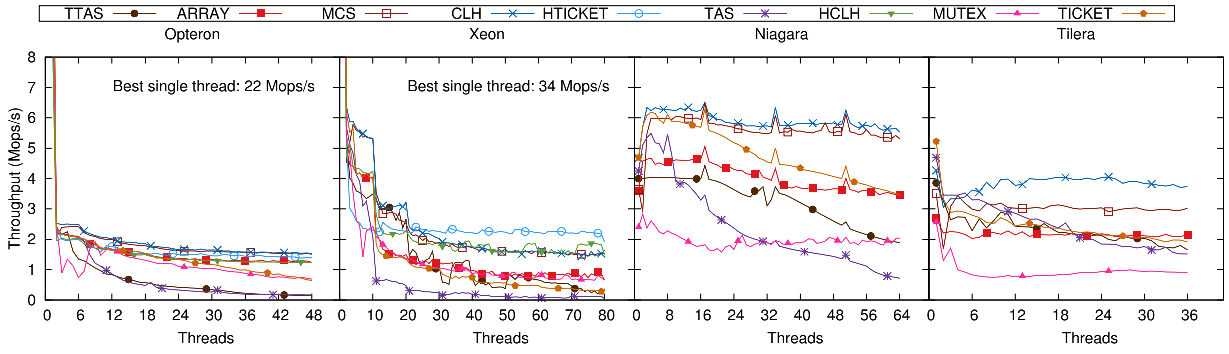


Figure 5: Throughput of different lock algorithms using a single lock.

further in subsequent runs. Figure 6 contains the latencies of the different locks when the previous holder is at various distances. Latencies suffer important increases on the multi-sockets as the second thread moves further from the first. In general, acquisitions that need to transfer data across sockets have a high cost. Remote acquisitions can be up to 12.5 and 11 times more expensive than local ones on the Opteron and the Xeon respectively. In contrast, due to the shared and distributed LLCs, the Niagara and the Tilera suffer no and slight performance decrease, respectively, as the location of the second thread changes. The latencies of the locks are in accordance with the cache-coherence latencies presented in Table 2.

Moreover, the differences in the latencies are significantly larger between locks on the multi-sockets than on the single-sockets, making the choice of the lock algorithm in an uncontested scenario paramount to performance. More precisely, while spin locks closely follow the cache-coherence latencies, more complex locks generally introduce some additional overhead.

Implications. Using a lock, even if no contention is involved, is up to one order of magnitude more expensive when crossing sockets. The 350-450 cycles on a multi-socket, and the roughly 200 cycles on a single-socket, are not negligible, especially if the critical sections are short. Moreover, the penalties induced when crossing sockets in terms of latency tend to be higher for complex locks than for simple locks. Therefore, regardless of the platform, simple locks should be preferred, when contention is very low.

6.1.2 Lock Algorithm Behavior

We study the behavior of locks under extreme and very low contention. On the one hand, highly contended locks are often the main scalability bottleneck. On the other hand, a large number of systems use locking strategies, such as fine-grained locks, that induce low contention. Therefore, good performance in these two scenarios is essential. We measure the total throughput of lock acquisitions that can be performed using each of the locks. Each thread acquires a random lock, reads and writes one corresponding cache line of data, and releases the lock. Similarly to the atomic operations stress test (Section 5.4) in the extreme contention experiment (one lock), a thread pauses after it releases the lock, in order to ensure that the release becomes visible to the other cores before retrying to acquire the lock. Given the uniform structure of the platforms, we do not use hierarchical locks on the single-socket machines.

Extreme contention. The results of the maximum contention experiment (one lock) are depicted in Figure 5. As we described in Section 5, the Xeon exhibits very strong locality within a socket. Accordingly, the hierarchical locks, i.e., hticket and HCLH, perform the best by taking advantage of that. Although there is a very big drop from one to two cores on the multi-sockets, within the socket both the Opteron and the Xeon manage to keep a rather stable performance. However, once a second socket is involved the throughput decreases again.

Not surprisingly, the CLH and the MCS locks are the most resilient to contention. They both guarantee that a single thread is spinning on each cache line and use the globally shared data only to enqueue for acquiring the lock. The ticket lock proves to be the best spin lock on this workload. Overall, the throughput on two or more cores on the multi-sockets is an order of magnitude lower than the single-core performance. In contrast, the single-sockets maintain a comparable performance on multiple cores.

Very low contention. The very low contention results (512 locks) are shown in Figure 7. Once again, one can observe the strong intra-socket locality of the Xeon. In general, simple locks match or even outperform the more complex queue locks. While on the Xeon the differences between locks become insignificant for a large number of cores, it is generally the ticket lock that performs the best on the Opteron, the Niagara, and the Tilera. On a low-contention scenario it is thus difficult to justify the memory requirements that complex lock algorithms have. It should be noted that, aside from the acquisitions and releases, the load and the store on the

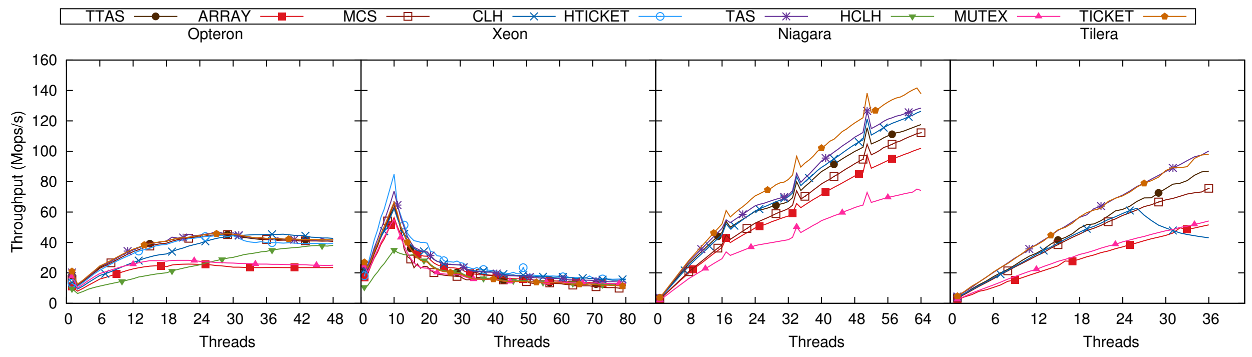


Figure 7: Throughput of different lock algorithms using 512 locks.

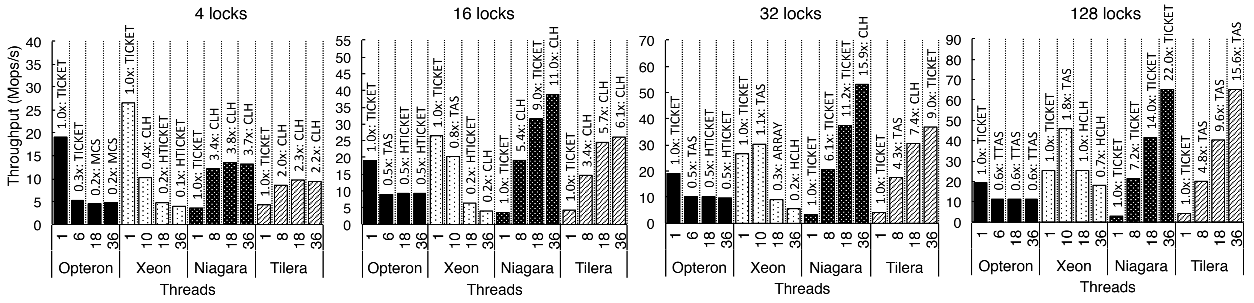


Figure 8: Throughput and scalability of locks depending on the number of locks. The "X : Y" labels on top of each bar indicate the best-performing lock (Y) and the scalability over the single-thread execution (X).

protected data also contribute to the lack of scalability of multi-sockets, for the reasons pointed out in Section 5.

Implications. None of the locks is consistently the best on all platforms. Moreover, no lock is consistently the best within a platform. While complex locks are generally the best under extreme contention, simple locks perform better under low contention. Under high contention, hierarchical locks should be used on multi-sockets with strong intra-socket locality, such as the Xeon. The Opteron, due to the previously discussed locality issues, and the single-sockets favor queue locks. In case of low contention, simple locks are better than complex implementations within a socket. Under extreme contention, while not as good as more complex locks, a ticket lock can avoid performance collapse within a socket. On the Xeon, the best performance is achieved when all threads run on the same socket, both for high and for low contention. Therefore, synchronization between sockets should be limited to the absolute minimum on such platforms. Finally, we observe that when each core is dedicated to a single thread there is no scenario in which Pthread Mutexes perform the best. Mutexes are however useful when threads contend for a core. Therefore, unless multiple threads run on the same core, alternative implementations should be preferred.

6.1.3 Cross-Platform Lock Behavior

In this experiment, we compare lock behavior under various degrees of contention across architectures. In order to have a straightforward cross-platform comparison, we run the tests on up to 36 cores. Having already explored the lock behavior of different algorithms, we only report the highest throughput achieved by any of the locks on each platform. We vary the contention by running experiments with 4, 16, 32, and 128 locks, thus examining high, intermediate, and low degrees of contention.

The results are shown in Figure 8. In all cases, the differences between the single and multi-sockets are noticeable. Under high contention, single-sockets prevent performance collapse from one thread to two or more, whereas in the lower contention cases these platforms scale well. As noted in Section 5, stores and atomic operations are affected by contention on the Tilera, resulting in slightly less scalability than on the Niagara: on high contention workloads, the uniformity of the Niagara delivers up to 1.7 times more scalability than the Tilera, i.e., the rate at which performance increases. In contrast, multi-sockets exhibit a significantly lower throughput for high contention, when compared to single-thread performance. Multi-sockets provide limited scalability even on the low contention scenarios. The direct cause of this contrasting behavior is the higher latencies for the cache-coherence transitions on multi-sockets, as well as the differences in the throughput of the atomic operations. It is worth noticing that the Xeon scales well when all the threads are within a socket. Performance, however, severely degrades even

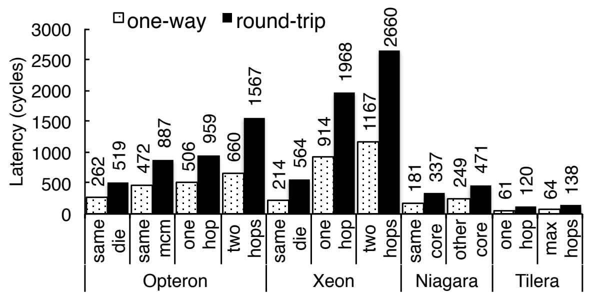


Figure 9: One-to-one communication latencies of message passing depending on the distance between the two cores.

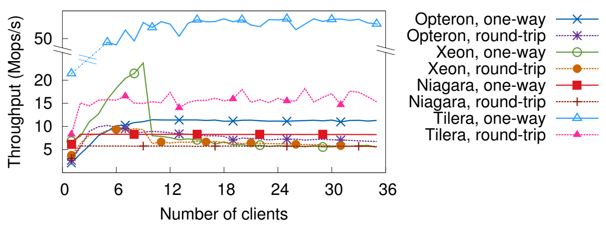


Figure 10: Total throughput of client-server communication.

with one thread on a remote socket. In contrast, the Opteron shows poor scalability regardless of the number of threads. The reason for this difference is the limited locality of the Opteron we discussed in Section 5.

Implications. There is a significant difference in scalability trends between multi and single-sockets across various degrees of contention. Moreover, even a small degree of non-uniformity can have an impact on scalability. As contention drops, simple locks should be used in order to achieve high throughput on all architectures. Overall, we argue that synchronization intensive systems should favor platforms that provide locality, i.e., they can prevent cross-socket communication.

6.2 Message Passing

We evaluate the message passing implementations of SSYNC. To capture the most prominent communication patterns of a message passing application we evaluate both one-to-one and client-server communication. The size of a message is 64 bytes (a cache line).

One-to-One Communication. Figure 9 depicts the latencies of two cores that exchange one-way and round-trip messages. As expected, the Tilera's hardware message passing performs the best. Not surprisingly, a one-way message over cache coherence costs roughly twice the latency of transferring a cache line. Once a core x receives a message, it brings the receive buffer (i.e., a cache line) to its local caches. Consequently, the second core y has to fetch the buffer (first cache-line transfer) in order to write a new message. Afterwards, x has to re-fetch the buffer (second transfer) to get the message. Accordingly, the round-trip case takes approximately four times the cost of a cache-line transfer. The reasoning is exactly the same with one-way messages, but applies to both ways: send and then receive.

Client-Server Communication. Figure 10 depicts the one-way and round-trip throughput for a client-server pattern with a single server. Again, the hardware message passing of the Tilera performs the best. With 35 clients, one server delivers up to 16 Mops/s (round-trip) on the Tilera (less on the other platforms). In this benchmark, the server does not perform any computation between messages, therefore the 16 Mops constitutes an upper bound on the performance of a single server. It is interesting to note that if we reduce the size of the message to a single word, the throughput on the Tilera is 27 Mops/s for round-trip and more than 100 Mops/s for one-way messages, respectively.

Two additional observations are worth mentioning. First, the Xeon performs very well within a socket, especially for one-way messages. The inclusive LLC cache plays the role of the buffer for exchanging messages. However, even with a single client on a remote socket, the throughput drops from 25 to 8 Mops/s. The second point is that as the number of cores increases, the round-trip throughput becomes higher than the one-way on the Xeon. We also observe this effect on the Opteron, once the length of the local computation of the server increases (not shown in the graph). This happens because the request-response model enables the server to efficiently prefetch the incoming messages. On one-way messages, the clients keep trying to send messages, saturating the incoming queues of the server. This leads to the clients busy-waiting on cache lines that already contain a message. Therefore, even if the server prefetches a message, the client will soon bring the cache line to its own caches (or transform it to shared), making the consequent operations of the server more expensive.

Implications. Message passing can achieve latencies similar to transferring a cache line from one core to another. This behavior is slightly affected by contention, because each pair of cores uses individual cache lines for communication. The previous applies both to one-to-one and client-server communication. However, a single server has a rather low upper bound on the throughput it can achieve, even when not executing any computation. In a sense, we have to trade performance for scalability.

6.3 Hash Table (`ssht`)

We evaluate `ssht`, i.e., the concurrent hash table implementation of SSYNC, under low (512 buckets) and high (12 buckets) contention, as well as short (12 elements) and long (48 elements) buckets. We use 80% `get`, 10% `put`, and 10% `remove` operations, so as to keep the size of the hash table constant. We configure

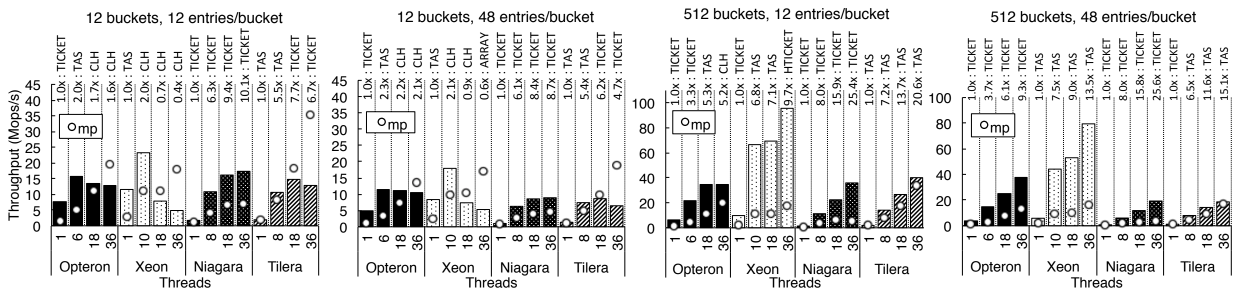


Figure 11: Throughput and scalability of the hash table (`ssht`) on different configurations. The "X : Y" labels on top of each bar indicate the best-performing lock (Y) and the scalability over the single-thread execution (X).

`ssht` so that each bucket is protected by a single lock, the keys are 64 bit integers, and the payload size is 64 bytes. The trends on scalability pertain on other configurations as well. Finally, we configure the message passing (mp) version to use (i) one server per three cores[9] and (ii) round-trip operations, i.e., all operations block, waiting for a response from the server. It should be noted that dedicating some threads as servers reduces the contention induced on the shared data of the application. Figure 11 depicts the results on the four target platforms on the aforementioned scenarios[10].

Low Contention. Increasing the length of the critical sections increases the scalability of the lock-based `ssht` on all platforms, except for the Tilera. The multi-sockets benefit from the efficient prefetching of the data of a bucket. All three systems benefit from the lower single-thread performance, which leads to higher scalability ratios. On the Tilera, the local data contend with the shared data for the L2 cache space, reducing scalability. On this workload, the message passing implementation is strictly slower than the lock-based ones, even on the Tilera. It is interesting to note that the Xeon scales slightly even outside the 10 cores of a socket, thus delivering the highest throughput among all platforms. Finally, the best performance in this scenario is achieved by simple spin locks.

High Contention. The situation is radically different for high contention. First, the message passing version not only outperforms the lock-based ones on three out of the four platforms (for high core counts), but it also delivers by far the highest throughput. The hardware threads of the Niagara do not favor client-server solutions; the servers are delayed due to the sharing of the core's resources with other threads. However, the Niagara achieves a 10-fold performance increase on 36 threads, which is the best scalability among the lock-based versions and approaches the optimal 12-fold scalability. It is worth mentioning that if we do not explicitly pin the threads on cores, the multi-sockets deliver 4 to 6 times lower maximum throughput on this workload.

Summary. These experiments illustrate two major points. First, increasing the length of a critical section can partially hide the costs of synchronization under low contention. This, of course, assumes that the data accessed in the critical section are mostly being read (so they can be shared) and follow a specific access pattern (so they can be prefetched). Second, the results illustrate how message passing can provide better scalability and performance than locking under extreme contention.

6.4 Key-Value Store (Memcached)

Memcached (v. 1.4.15) [30] is an in-memory key-value store, based on a hash table. The hash table has a large number of buckets and is protected by fine-grain locks. However, during certain rebalancing and maintenance tasks, it dynamically switches to a global lock for short periods of time. Since we are interested in the effect of synchronization on performance and scalability, we replace the default Pthread Mutexes that protect the hash table, as well as the global locks, with the interface provided by `libslock`. In order to stress Memcached, we use the *memslap* tool from the *libmemcached* library [25] (v. 1.0.15). We deploy *memslap* on a remote server and use its default configuration. We use 500 client threads and run a get-only and a set-only test.

Get. The *get* test does not cause any switches to global locks. Due to the essentially non-existent contention, the lock algorithm has little effect in this test. In fact, even completely removing the locks of the hash table does not result in any performance difference. This indicates that there are bottlenecks other than synchronization.

Set. A write-intensive workload however stresses a number of global locks, which introduces contention. In the *set* test the differences in lock behavior translate in a difference in the performance of the application as well. Figure 12 shows the throughput on various platforms using different locks. We do not present the results on more than 18 cores, since none of the platforms scales further. Changing the Mutexes to ticket, MCS, or TAS locks achieves speedups between 29% and 50%

[9]This configuration achieves the highest throughput.

[10]The single-thread throughput for message passing is actually a result of a one server / one client execution.

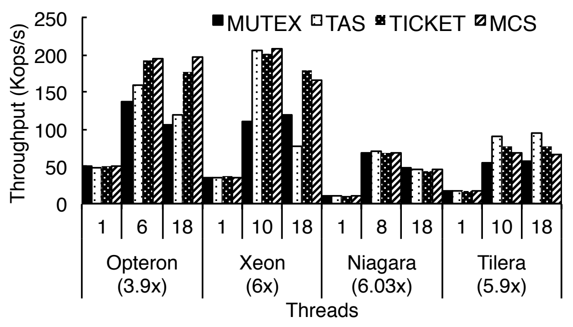


Figure 12: Throughput of Memcached using a set-only test. The maximum speed-up vs. single thread is indicated under the platform names.

on three of the four platforms[11]. Moreover, the cache-coherence implementation of the Opteron proves again problematic. Due to the periodic accesses to global locks, the previously presented issues strongly manifest, resulting in a maximum speedup of 3.9. On the Xeon, the throughput increases while all threads are running within a socket, after which it starts to decrease. Finally, thread scheduling has an important impact on performance. Not allocating threads to the appropriate cores decreases performance by 20% on the multi-sockets.

Summary. Even in an application where the main limitations to performance are networking and the main memory, when contention is involved, the impact of the cache coherence and synchronization primitives is still important. When there is no contention and the data is either prefetched or read from the main memory, synchronization is less of an issue.

7 Related Work

Our study of synchronization is, to date, the most exhaustive in that it covers a wide range of schemes, spanning different layers, and different many-core architectures. In the following, we discuss specific related work that concerns the main layers of our study.

Leveraging Cache Coherence. The characteristics of cache-coherence protocols on x86 multi-sockets have been investigated by a number of studies [17, 33], whose focus is on bandwidth limitations. The cache-coherence latencies are measured from the point of view of loading data. We extend these studies by also measuring stores and atomic operations and analyzing the impact on higher-level concurrent software. Molka et al. [33] consider the effect of the AMD and Intel memory hierarchy characteristics on various workloads from SPEC OMPM2001 and conclude that throughput is mainly dictated by memory limitations. The results are thus of limited relevance for systems involving high contention. Moses et al. [35] use simulations to show that increasing non-uniformity entails a decrease in the performance of the TTAS lock under high contention. However, the conclusions are limited to spin locks and one specific hardware model. We generalize and quantify such observations on commonly used architectures and synchronization schemes, while also analyzing their implications.

Scaling Locks. Mellor-Crumney et al. [29] and Anderson [4] point out the lack of scalability with traditional spin locks. They introduce and test several alternatives, such as queue locks. Their evaluation is performed on large scale multi-processors, on which the memory latency distribution is significantly different than on today's many-cores. Luchangco et. al [27] study a NUMA-aware hierarchical CLH lock and compare its performance with a number of well-known locks. Aside from a Niagara processor, no other modern many-core is used. Our analysis extends their evaluation with more locks, more hardware platforms, and more layers.

Other studies focus on the Linux kernel [12] and conclude that the default ticket lock implementation causes important performance bottlenecks in the OS on a multi-core. Performance is improved in a number of different scenarios by replacing the ticket locks with complex locks. We confirm that plain spin locks do not scale across sockets and present some optimizations that alleviate the issue.

Various techniques have been proposed in order to improve the performance of highly contended locks, especially on multi-sockets. For example, combining [18] is an approach in which a thread can execute critical sections on behalf of others. In particular, RCL [26] replaces the "lock, execute, and unlock" pattern with remote procedure calls to a dedicated server core. For highly contended critical sections this approach hides the contention behind messages and enables the server to locally access the protected data. However, as the RCL paper mentions, the scope of this solution is limited to high contention and a large number of cores, as our results on message passing confirm.

Scaling Systems on Many-Cores. In order to improve OS scalability on many-cores, a number of approaches deviate from traditional kernel designs. The OS is typically restructured to either improve locality (e.g., Tornado [15]), limit sharing (e.g., Corey [10]), or avoid resource sharing altogether by using message passing (e.g., Barrelfish [6], fos [45]). Boyd-Wickizer et al. [11] aim at verifying whether these scalability issues are indeed inherent to the Linux kernel design. The authors show how optimizing, using various concurrent programming techniques, removes several scalability issues from both the kernel and the applications. By doing so, they conclude that it is not necessary to give up the traditional kernel structure just yet. Our study confirms these papers' observation that synchronization can

[11]The bottleneck on the Niagara is due to network and OS issues.

be an important bottleneck. We also go a step further: we study the roots of the problem and observe that many of the detected issues are in fact hardware-related and would not necessarily manifest on different platforms.

8 Concluding Remarks

Summary. This paper dissects the cost of synchronization and studies its scalability along different directions. Our analysis extends from basic hardware synchronization protocols and primitives all the way to complex concurrent software. We also consider different representative hardware architectures. The results of our experiments and our cross-platform synchronization suite, SSYNC, can be used to evaluate the potential for scaling synchronization on different platforms and to develop concurrent applications and systems.

Observations & Ramifications. Our experimentation induces various observations about synchronization on many-cores. The first obvious one is that crossing sockets significantly impacts synchronization, regardless of the layer, e.g., cache coherence, atomic operations, locks. Synchronization scales much better within a single socket, irrespective of the contention level. Systems with heavy sharing should reduce cross-socket synchronization to the minimum. As we pointed out, this is not always possible (e.g., on a multi-socket AMD Opteron), for hardware can still induce cross-socket traffic, even if sharing is explicitly restricted within a socket. Message passing can be viewed as a way to reduce sharing as it enforces partitioning of the shared data. However, it comes at the cost of lower performance (than locks) on a few cores or low contention.

Another observation is that non-uniformity affects scalability even within a single-socket many-core, i.e., synchronization on a Sun Niagara 2 scales better than on a Tilera TILE-Gx36. Consequently, even on a single-socket many-core such as the TILE-Gx36, a system should reduce the amount of highly contended data to avoid performance degradation (due to the hardware).

We also notice that each of the nine state-of-the-art lock algorithms we evaluate performs the best on at least one workload/platform combination. Nevertheless, if we reduce the context of synchronization to a single socket (either one socket of a multi-socket, or a single-socket many-core), then our results indicate that spin locks should be preferred over more complex locks. Complex locks have a lower uncontested performance, a larger memory footprint, and only outperform spin locks under relatively high contention.

Finally, regarding hardware, we highlight the fact that implementing multi-socket coherence using broadcast or an incomplete directory (as on the Opteron) is not favorable to synchronization. One way to cope with this is to employ a directory combined with an inclusive LLC, both for intra-socket sharing and for detecting whether a cache line is shared outside the socket.

Miscellaneous. Due to the lack of space, we had to omit some rather interesting results. In particular, we conducted our analysis on small-scale multi-sockets, i.e., a 2-socket AMD Opteron 2384 and a 2-socket Intel Xeon X5660. Overall, the scalability trends on these platforms are practically identical to the large-scale multi-sockets. For instance, the cross-socket cache-coherence latencies are roughly 1.6 and 2.7 higher than the intra-socket on the 2-socket Opteron and Xeon, respectively. In addition, we did not include the results of the software transactional memory. In brief, these results are in accordance with the results of the hash table (Section 6.3), both for locks and message passing. Furthermore, we also considered the MonetDB [34] column-store. In short, the behavior of the TPC-H benchmarks [42] on MonetDB is similar to the *get* workload of Memcached (Section 6.4): synchronization is not a bottleneck.

Limitations & Future Work. We cover what we believe to be the "standard" synchronization schemes and many-core architectures used today. Nevertheless, we do not study lock-free [19] techniques, an appealing way of designing mutual exclusion-free data structures. There is also a recent body of work focusing on serializing critical sections over message passing by sending requests to a single server [18, 26]. It would be interesting to extend our experiments to those schemes as well.

Moreover, since the beginning of the multi-core revolution, power consumption has become increasingly important [8, 9]. It would be interesting to compare different platforms and synchronization schemes in terms of performance per watt. Finally, to facilitate synchronization, Intel has introduced the Transactional Synchronization Extensions (TSX) [24] in its Haswell micro-architecture. We will experiment with TSX in SSYNC.

Acknowledgements

We wish to thank our shepherd, Sam King, and the anonymous reviewers for their helpful comments. We would also like to thank Christos Kozyrakis for the fruitful discussions at the incipient stages of this work, as well as, Babak Falsafi, Tim Harris, Luis Ceze, Rodrigo Rodrigues, Edouard Bugnion, and Mark Moir for their comments on an early draft of this paper. Finally, we wish to thank the EPFL DIAS lab and EcoCloud for providing us with some of the platforms employed in this study. Part of this work was funded from the European Union Seventh Framework Programme (FP7) under grant agreement 248465, the S(o)OS project and the Swiss National Science Foundation project 200021_143915, Reusable Concurrent Data Types.

References

[1] J. Abellan, J. Fernandez, and M. Acacio. GLocks: Efficient support for highly-contended locks in Many-Core CMPs. IPDPS 2011, pages 893–905.

[2] AMD. Software optimization guide for AMD family 10h and 12h processors. 2011.

[3] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS 1967 (Spring), pages 483–485.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE TPDS*, 1(1):6–16, 1990.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. PODC 1990, pages 363–375.

[6] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP 2009, pages 29–44.

[7] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *J. Parallel Distrib. Comput.*, 21(2):246–254, 1994.

[8] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, 1999.

[9] S. Borkar and A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. OSDI 2008, pages 43–57.

[11] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *OSDI 2010*, pages 1–16.

[12] S. Boyd-Wickizer, M. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, 2012.

[13] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro, IEEE*, 30(2):16 –29, 2010.

[14] D. Dice, V. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. PPoPP 2012, pages 247–256.

[15] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. OSDI 1999, pages 87–100.

[16] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: a software transactional memory for many-cores. EuroSys 2012, pages 351–364.

[17] D. Hackenberg, D. Molka, and W. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. MICRO 2009, pages 413–422.

[18] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA 2010, pages 355–364. ACM.

[19] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[20] M. Herlihy and N. Shavit. *The art of multiprocessor programming, revised first edition*. 2012.

[21] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, 2008.

[22] Intel. An introduction to the Intel QuickPath interconnect. 2009.

[23] Intel. Intel 64 and IA-32 architectures software developer's manual. 2013.

[24] Intel. Transactional Synchronization Extensions Overview. 2013.

[25] libmemcached. `http://libmemcached.org/libMemcached.html`.

[26] J. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. USENIX ATC 2012.

[27] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. ICPP 2006, pages 801–810.

[28] J. Mellor-Crummey and M. Scott. Synchronization without contention. ASPLOS 1991, pages 269–278.

[29] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 1991.

[30] Memcached. `http://www.memcached.org`.

[31] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. PODC 1996, pages 267–275.

[32] S. Microsystems. UltraSPARC T2 supplement to the UltraSPARC architecture. 2007.

[33] D. Molka, R. Schöne, D. Hackenberg, and M. Müller. Memory performance and SPEC

OpenMP scalability on quad-socket x86 64 systems. ICA3PP 2011, pages 170–181.

[34] MonetDB. `http://www.monetdb.org/`.

[35] J. Moses, R. Illikkal, L. Zhao, S. Makineni, and D. Newell. Effects of locking and synchronization on future large scale CMP platforms. CAECW 2006.

[36] U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-Core, 64-thread, power-efficient SPARC server on a chip. *Solid-State Circuits, IEEE Journal of*, 43(1):6 –20, 2008.

[37] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. ISCA 1984, pages 348–354.

[38] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. HPCA 2007, pages 13–24.

[39] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *SOSP 1989*, pages 83–90.

[40] M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. PPoPP 2001, pages 44–52.

[41] Tilera tile-gx. `http://www.tilera.com/products/processors/TILE-Gx_Family`.

[42] TPC-H. `http://www.tpc.org/tpch/`.

[43] C. T.S. Building FIFO and priority-queuing spin locks from atomic swap. Technical report, 1993.

[44] J. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue, and T. Nakatani. Performance studies of commercial workloads on a multi-core system. IISWC 2007, pages 57–65.

[45] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *OSR*, 43(2):76–85, 2009.