Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores

Irina Calciu[1], Dave Dice[2], Tim Harris[2], Maurice Herlihy[1,2], Alex Kogan[2], Virendra Marathe[2], and Mark Moir[2]

[1] Brown University {irina,mph}@cs.brown.edu
[2] Oracle Labs
{dave.dice,timothy.l.harris,alex.kogan,virendra.marathe,mark.moir}@oracle.com

Abstract. Even for small multi-core systems, it has become harder and harder to support a simple shared memory abstraction: processors access some memory regions more quickly than others, a phenomenon called *non-uniform memory access* (NUMA). These trends have prompted researchers to investigate alternative programming abstractions based on message passing rather than cache-coherent shared memory. To advance a pragmatic understanding of these models' strengths and weaknesses, we have explored a range of different message passing and shared memory designs, for a variety of concurrent data structures, running on different multicore architectures. Our goal was to evaluate which combinations perform best, and where simple software or hardware optimizations might have the most impact. We observe that different approaches perform best in different circumstances, and that the communication overhead of message passing can often outweigh its benefits. Nonetheless, we discuss ways in which this balance may shift in the future. Overall, we conclude that, by emphasizing high-level shared data abstractions, software should be designed to be largely independent of the choice of low-level communication mechanism.

Keywords: NUMA, message passing, shared memory, delegation, locks, concurrent data structures

1 Introduction

As modern processor architectures evolve, programming abstractions are straining to keep up. The transition from single-core to increasingly multi-core architectures means that scalability, that is, the ability to exploit parallelism and manage concurrency, has become a central concern for software system design.

Even for small multi-core systems, it has become harder and harder to support a simple shared-memory abstraction. This abstraction is already starting to fail with respect to performance: processors observe that some memory regions can be accessed more quickly than others, a phenomenon called *non-uniform memory access* (NUMA). Once a concern primarily for large-scale, high-performance workloads, NUMA effects are increasingly visible to smaller, "everyday" programs. In the long term, some researchers have even suggested that cache coherence will no longer be feasible across a single multi-core chip, or that individual cores may perform better in the absence of coherence.

In reaction to these trends, researchers have investigated alternative programming abstractions in which—even within a shared-memory system—coordination is based on message passing rather than via direct access to shared-memory data structures. A key example is a design pattern we call *delegation*, in which one thread requests that another thread perform an operation on its behalf, and the request and response (if any) are sent by message passing. For instance, Barrelfish [1] runs a separate kernel on each core, and cores communicate only via a message passing interface, itself implemented in shared memory.

Advocates for delegation appeal to its simplicity: it promises to support application designs that span NUMA architectures, heterogeneous architectures, and even architectures that lack global coherence. Moving platform-specific engineering concerns—such as cache line sizes or idiosyncratic coherence protocols—out of the application and into the message passing substrate could ease porting applications from one platform to another, or from one platform to its successor.

Many of these proposals (surveyed in Section 7), however, are *ad-hoc* in nature, focusing on a specific implementation of a specific data structure, yielding little insight into where the message passing abstraction performs better than the shared-memory abstraction. Our contribution is to explore a range of message passing and shared-memory designs, on various benchmarks running on different multicore architectures, and to evaluate which combinations perform best.

This paper does not take sides in the ongoing debate about the relative merits of shared-memory versus message-passing abstractions [9]. In contrast, our contribution is to advance a pragmatic understanding of these models' strengths and weaknesses. In particular, such debates often present a false dichotomy: that we must choose between these models, and that one is superior. Instead, by emphasizing high-level abstractions, software can be designed largely independently of the choice of low-level communication mechanism. The choice itself should be based on pragmatic performance evaluations.

We will use the following terminology. Many modern large-scale multiprocessor architectures are composed of multiple *sockets* or *nodes*, each encompassing multiple *cores*. Each core has access to a local cache hierarchy and to dynamic random-access memory (DRAM), along with multiple *threads*. Such systems typically utilize a cache-coherence protocol, which creates the illusion that threads share a common memory. Nevertheless, as noted, cache coherence protocols do not hide NUMA effects in the form of differences in the times needed to communicate between local and remote memories. We use the term *NUMA domain* to indicate a set of threads with identical memory access times.

Section 2 explains our notion of delegation, and Section 3 describes alternative ways of implementing message passing on a shared-memory multicore. Section 4 describes the range of benchmarks used, and Section 5 describes the experimental results, which are discussed further in Section 6. Section 7 surveys related work, and Section 8 presents conclusions.

2 Delegation

In *delegation*, access to a data structure is mediated by one or more *server threads*, which are the only threads allowed to manipulate the data directly. Even though all threads share a common (NUMA) memory, client and server threads communicate by a message passing protocol whose implementation is optimized to take advantage of the underlying shared memory.

When a client thread needs to apply an operation to a data structure, it *delegates* that operation by sending a request message to the server thread. When the server thread receives the message, it carries out the operation directly on the data structure, and stores the result value, if any, in a client-allocated buffer.

Delegation is attractive for several reasons. First, the server thread can operate directly on the data structure—without synchronizing its accesses with other threads, in which case programmers need not worry about synchronization. Furthermore, a server thread may encounter fewer cache misses and generate less coherence traffic than threads operating on the data structure directly. Advocates of delegation often suggest that delegation can produce more robust software designs: to be cost-effective, applications must be designed to work over a wide range of parallel platforms, making it difficult to optimize shared data structures for any specific platform. Delegation introduces an abstraction layer, allowing implementations to be optimized for different platforms with no changes to applications. This abstraction layer allows applications to be more easily scaled out from multicores to multiple machines by replacing the shared-memory communication protocol with one that operates over a distributed system.

Nevertheless, delegation also has its pitfalls. From the point of view of an individual operation on a data structure, a central question is how the time spent operating directly on a shared data structure compares with the cost of (i) sending and receiving messages, (ii) queuing time of a message at a server thread, and (iii) execution time at the server thread. Server threads are statically assigned to cores, so they may be idle some of the time. If there are enough cores, however, this static assignment removes the need for complex mechanisms to enable server threads to be quickly identified and dispatched (as with active messages [17]). Server threads may become a bottleneck, so the underlying data structure may need to be partitioned and delegated to multiple servers, a problem similar to moving from coarse to fine-grained locking. Finally, delegation imposes some inconvenience on programmers, as operation requests and responses must be marshalled into messages before being sent and unmarshalled upon receipt.

In short, while delegation has some attractive properties, it does not follow that delegation-based data structures are inherently preferable.

3 Communication

Message Passing Clients communicate with servers via a message-passing protocol implemented in shared memory. Although we consider several different mechanisms for communication, the messages themselves are similar across schemes.

Each message contains an *opcode* identifying the requested operation (e.g., add a key-value pair to a table), one or more arguments (the values to add), a pointer to a buffer where the call's result is to be stored (e.g., the value returned by a get() call), and a `ready` flag that the server sets when the result is ready. We follow the convention that the client manages the memory occupied by messages (most messages are allocated on the client's stack).

In our experiments, when a client issues a request, it blocks until the response is available. Straightfoward alternative approaches could allow clients to issue requests for multiple operations to be performed in parallel.

We evaluate three communication mechanisms: MPSCChannel, InletQueue, and DNCInletQueue, each making different synchronization trade-offs.

The MPSCChannel (multiple producer, single consumer), based on the "Multilane" structure of Dice and Otenko [6], uses an array of request slots. A shared variable PutCursor indicates the next available slot. A client uses a *compare-and-swap* (CAS) instruction to increment PutCursor atomically, and uses the previous value (modulo the size of the array) to choose a slot. Because that slot might still be in use, the client repeatedly calls CAS to swap null with a pointer to its request message. The server uses a private variable TakeCursor to cycle around slots, waiting for each one to contain a non-null pointer to a request. It then reads the opcode and arguments from the request, resets the array slot to null (making it available to other clients), performs the operation, stores the result in the buffer provided, and finally sets the `ready` flag.

In NUMA architectures, memory accesses not satisfied by local cache are substantially slower when applied to remote memory than to local memory. A disadvantage of MPSCChannel is that it requires threads to repeatedly apply CAS to remote PutCursor locations, and, more rarely, to remote slots.

The InletQueue channel provides one slot per NUMA domain. Each client uses CAS to attempt to replace the slot's null value with a pointer to its message. When the server reads the request, it resets the slot to null to make it available again, performs the operation, copies the result (if any) to the client's buffer, and sets the message's `ready` flag.

The DNCInletQueue channel ("direct, no CAS") uses only load and store operations to access remotely share variables, and a lock for synchronization among threads on a single node. In this channel, the node's slot contains the message itself, not just a pointer to the message. When a client thread acquires the lock for its node's slot, it copies the request message into the slot, including a pointer to the client buffer where the result is to be stored.

The motivation for DNCInletQueue is to ensure that the mechanism used for actual inter-socket communication is as simple as possible (simple stores by the client and simple loads by the server): synchronization such as acquiring the lock that protects the slot is performed only among threads on the same node. (With InletQueue, although only clients on the same node attempt to modify the slot, slots are still shared remotely with the server reading them.) We believe this approach creates the best opportunity for potential future hardware optimizations to reduce communication overhead. Even without such optimizations, the "direct" aspect of DNCInletQueue ensures that, when a server reads a slot written by a client, it already knows the operation to perform. In contrast, methods that send a pointer to the message require the server to initiate another round of inter-socket communication to fetch the message contents.

Shared memory For shared-memory mechanisms, we consider lock-based structures employing the following kinds of locks: a simple spin lock, the MCS lock [11], and fair and unfair versions of a NUMA-aware "cohort" lock C−TKT−MCS [5] that uses MCS for synchronization between threads on the same socket, and a global ticket lock to explicitly manage when the lock is handed off to a thread on another socket. Handing off the lock preferentially within a socket can reduce lock handoff time, and increase cache locality for data accessed in the critical section. However, doing so blindly can result in "gross unfairness", in which high throughput is achieved, but some threads are essentially starved. Perhaps surprisingly, depending on the architecture, this phenomenon can occur even with simple locks that do not explicitly seek to keep the lock within a socket. Thus, it is important to manage such pitfalls. We therefore include "fair" and "unfair" variants of C−TKT−MCS (denoted as C−TKT−MCS-fair and C−TKT−MCS-unfair, respectively). The fair version imposes a limit on how many times the lock can be handed off within a socket, avoiding grossly unfair behavior.

4 Benchmarks

In this paper, we restrict our analysis to two representative cases among the data structures we explored. Both implement a *map* interface, storing key-value pairs with standard insert(), remove() and get() operations.

4.1 Concurrent Hash Maps

The hash map is partitioned into multiple pieces; with delegation, each is managed by a server thread. Each partition has a preconfigured number of *buckets*, where each bucket is a linked list of *chunks*. Each chunk is a fixed-size, cache-line-aligned structure that holds a set of key-value pairs whose keys lie within a fixed range. Chunk size is a multiple of 64 bytes (the unit of cache coherence). To speed searches, adjacent chunks' key ranges do not overlap, and each chunk records the maximum key that it stores. Chunks in a bucket are sorted by their maximum stored keys, but key-value pairs within a chunk are unordered.

Each bucket is a linked list of cache-aligned chunks, instead of the more traditional list of key-value pairs, because loading each chunk brings in multiple key-value pairs, reducing cache coherence traffic. This structure should benefit both shared-memory and delegation-based methods. However, it is likely to favor shared-memory more because delegation ensures that all accesses to this data are from the same NUMA node, resulting in more effective use of lower level caches, and more ability to place data in memory near where it will be accessed.

To store a key-value pair in a partition, the key is hashed to identify the bucket where the pair will be stored. The bucket's list of chunks is then scanned to identify which chunk should contain the given key, skipping chunks whose maximum key is smaller than it. The target chunk is then scanned linearly for the given key. If found, the value is updated. If not, but the chunk is full, the chunk is split and half of its elements are moved to a new chunk, making space for the new pair. Chunk size is subject to a trade-off: smaller chunks are better for cache locality, but larger chunks reduce the frequency of splitting.

Although many other possibilities exist, we have chosen shared-memory and delegation-based implementations that each exploit a key advantage they have over the other. For delegation, by having a single server thread manage each partition, it can do so without additional synchronization. For shared memory, we have chosen an example in which fine-grained locking is straightforward: a fixed-size hash map implemented using a single lock for each bucket, allowing one thread per bucket to access the hash map concurrently.

While multiple server threads could also use this technique to collectively manage a partition, this would impose overhead on each operation, introduce issues such as how clients balance requests over these multiple servers, and require additional hardware threads to be reserved for the additional severs. In contrast, in the lock-based hash map, as long as the total number of buckets (and therefore locks) remains the same, the actual number of partitions has almost no effect on performance or on the number of hardware threads required.

4.2 Concurrent Linked Lists

The concurrent linked list is a degenerate hash map, where each partition consists of a single bucket. In particular, each bucket is a linked list of chunks, as explained in Section 4.1. One important point is that the whole partition/bucket is protected by a single lock, so the concurrency achievable in the lock-based linked list is bounded by the number of partitions, much as with delegation.

4.3 Workloads

We used both *small* and *large* workloads. The small (large) hash map has 500 (50,000) buckets per partition. In the small (large) workload, the hash map is initialized by storing a key-value pair with a randomly-chosen key 1000 (100,000) times. For the linked list, the small (large) workload initializes the list by storing a key-value pair with a randomly-chosen key 1000 (100,000) times. Thus, the small workload has better cache locality than the large workload. After some experimentation, we sized chunks to accommodate 64 key-value pairs.

We experimented with three *mixes* of operations: *read-only*, consisting entirely of get() calls, *write-only*, consisting of 50% insert () and 50% remove() calls, and *read-write*, a mixture of 50% get(), 25% insert (), and 25% remove() calls. There are too many combinations of data structures, architectures, and workloads to present them all, so we focus here on the most interesting cases.

Results were qualitatively similar for the three operation mixes (read-only, write-only, and read-write); for brevity, we present only the read-write results.

5 Performance Results

The experiments were conducted on two systems with different architectures. The first is an 8-socket Nehalem system [13] ("X4800"), each socket containing a Xeon X7560 processor chip with 8 hyperthreaded cores running at a 2.26Ghz clock frequency, with a total of 128 hardware threads. The second system is an Oracle T4-4 [14] ("SPARC T4-4"), which consists of 4 T4 SPARC sockets, each socket containing 8 cores, and each core containing 8 hardware thread contexts, for a total of 256 hardware thread contexts, running at a 3 GHz clock frequency.

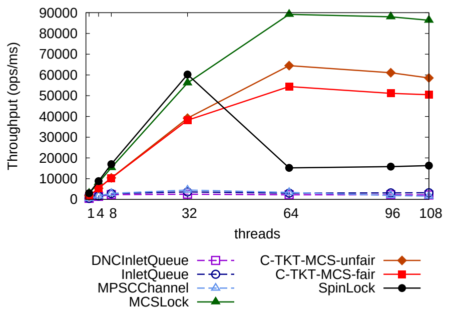
For the delegation-based implementations, server threads were placed uniformly among the sockets (see Section 5.4 for additional details). Placement of client threads was controlled by the OS in all cases. In each experiment, each thread repeatedly chooses at random whether to insert or delete an item, and performs the operation. No "external" work is performed between operations. We measure the total number of operations completed by all threads over a measurement period of ten seconds, and report throughput as the number of operations performed by all client threads per millisecond. Each experiment was repeated 6 times, and the average throughput for each configuration is reported.

5.1 Hash Map

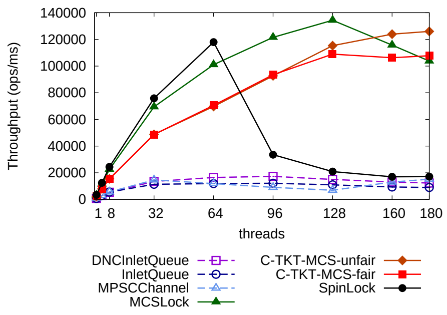
The first set of experiments was conducted on the concurrent hash map data structure of Section 4.1. We experimented with both small (Figures 1(a) and 1(b)), and large (Figure 1(c) and 1(d)) workloads. Unless stated otherwise, the number of partitions (as well as the number of server threads in the case of delegation) is constant at 8 (which is equal to or a small multiple of the number of sockets), and we use the read-write operation mix.

Figure 1 shows that, for the hash map benchmark, shared-memory mechanisms with any of locks performs much better than delegation for any channel type. This difference is because the fine-grained locking employed in our hash map implementation allows many threads to manipulate the shared hash map data structure concurrently. On the other hand, concurrency is limited by the number of servers in the case of delegation. Indeed, the performance of delegation scales only up to 32 threads (which is more than 8, the number of servers, because clients perform work such as choosing a random key and determining which server thread will perform the operation before sending the request).

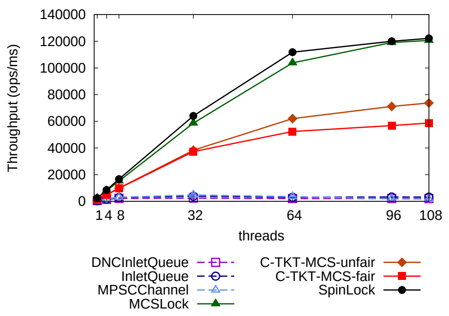
For small hash maps (Figures 1 (a) and (b)), all locks eventually stop scaling (and most perform worse) as the number of threads increases. This is due to contention on the (relatively) small number of buckets/locks. The performance of delegation, though worse than that of locking, is less sensitive to this contention because it is limited primarily by the sequential server threads, whose



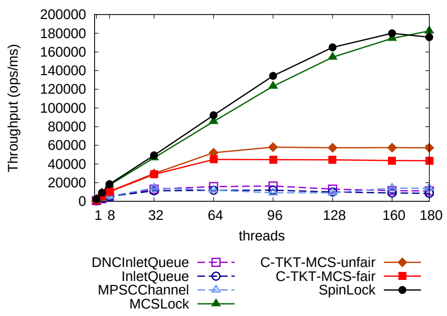
(a) Small RW Hash Map (X4800)



(b) Small RW Hash Map (SPARC T4-4)



(c) Large RW Hash Map (X4800)



(d) Large RW Hash Map (SPARC T4-4)

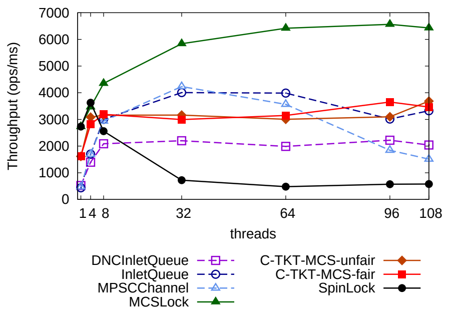
Fig. 1: Hash Map experiment

performance is largely insensitive to contention on message queues. (Nonetheless, MPSCChannel's centralized PutCursor makes it more sensitive to contention than the other message queues.)

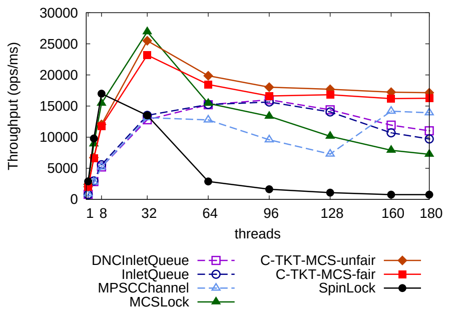
The simple MCS lock is typically the best-performing lock at low contention. However, MCS's performance degrades under heavy contention. By contrast, NUMA-aware locks perform better under high contention because there is an increased likelihood that locks can be handed off to threads on the same socket. The unfair C−TKT−MCS variant provides better high-contention performance than the fair variant because it permits more consecutive, local hand-offs. We return to this point in Section 5.3.

5.2 Linked List

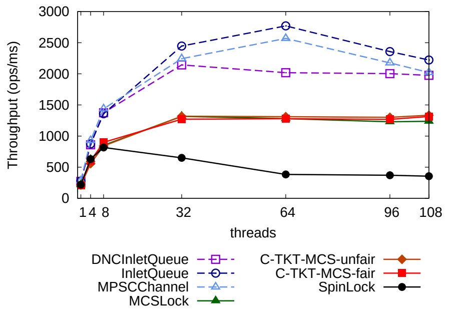
Figure 2 summarizes results for the linked list benchmark. As noted in Section 4.2, each partition contains just one bucket protected by a lock. Furthermore, each operation performs more memory accesses with the linked list than with the hash map, as all key-value pairs of a partition are stored in one bucket. A larger number of memory accesses per operation favors delegation if better



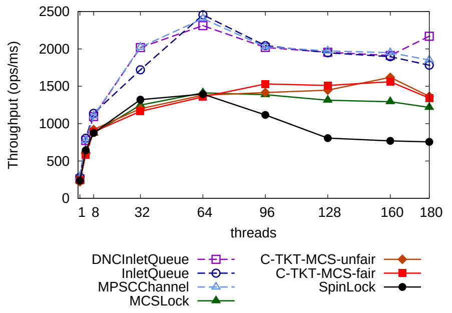
(a) Small RW Linked List (X4800)



(b) Small RW Linked List (SPARC T4-4)



(c) Large RW Linked List (X4800)



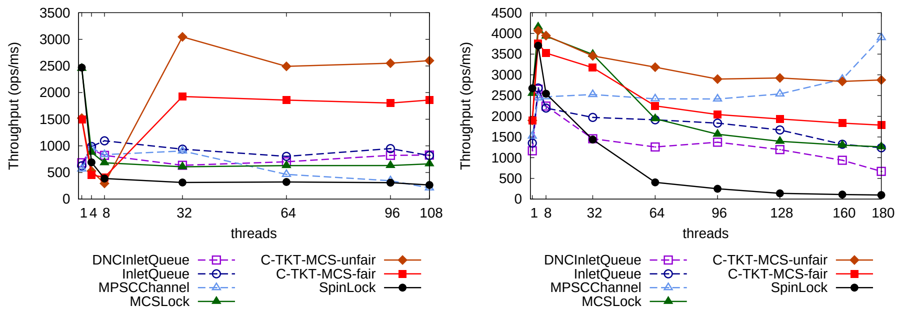
(d) Large RW Linked List (SPARC T4-4)

Fig. 2: Linked List experiment

server cache locality outweighs the cost of client-server communication. Indeed, the delegation methods performed considerably better than all locking schemes for large linked lists (Figures 2(c) and 2(d)), where operations access a large number of memory locations during list traversals. For small linked lists (Figures 2(a) and 2(b)), delegation provided competitive performance, losing only to MCS on X4800, and to the C−TKT−MCS variants on SPARC T4-4.

Although the simple MCS algorithm [11] provides superior performance in many cases, its performance degrades severely in some cases. There are two reasons for this. First, when contention increases, MCS has no facility to encourage consecutive lock handoffs within the same socket. As a result, the `Tail` variable that is modified by every lock acquisition "bounces" around the system frequently. This in turn causes data accessed in the critical section to similarly bounce around the system. NUMA-aware locks are able to avoid this effect and thus outperform MCS in this case (Figure 2(b)).

To evaluate these mechanisms in less balanced workloads, we repeated the experiment using only one partition, representing a partition that receives a disproportionate fraction of the requests, or alternatively a configuration in which there are not enough partitions, so all partitions may be overloaded.



(a) Small RW Linked List (X4800) (b) Small RW Linked List (SPARC T4-4)

Fig. 3: Linked List experiment with single partition

Results are shown in Figure 3. (We omit results for large linked lists for this case, as sequential execution of operations dominates performance. Thus, the synchronization mechanism used has little bearing on performance.)

Although InletQueue usually outperforms DNCInletQueue, recall that DNCInletQueue was specifically designed to be more amenable to hypothetical future hardware enhancements (Section 3). Interestingly, while MPSCChannel's performance often degrades going from low to medium thread counts, it *improves* at even higher threading levels on SPARC T4-4. We believe that this is because, with more client threads, there is more contention for slots, thus reducing contention on PutCursor, the primary bottleneck. We have not yet evaluated sensitivity to the number of slots, which would shed some light on this issue.

Superficially, C−TKT−MCS-unfair seems to significantly outperform all other methods and—to a lesser extent—both C−TKT−MCS-fair and MPSCChannel also stand out. However, some caution is needed in interpreting these results. As discussed in Section 3, some methods provide deceptively high throughput by "gross unfairness": they provide high throughput to some threads, while other threads receive much lower throughput or even starve completely. If this issue is overlooked, it is easy to conclude that a method that would be unacceptable in practice delivers the best results. We discuss this issue in more detail next.

5.3 Fairness

As a crude indicator of unfairness, we use *spread*, defined as the maximum per-thread count divided by the minimum per-thread count (plus one to avoid divide by zero). If the throughput of all threads is approximately equal, the spread will be close to 1. Methods that are grossly unfair—particularly those that starve some threads completely—exhibit very high spread.

In Figure 3(b), C−TKT−MCS-unfair consistently delivers the highest or nearly the highest throughput, but exhibits a spread value of over 560,000 at 32 threads. Its fair counterpart typically exhibits a spread value close to 1 (we occasionally see values of up to 4.5), but delivers significantly lower throughput in most cases. Similarly, on X4800, C−TKT−MCS-unfair exhibits spread over 1,000,000 in the highest contention case (single partition, 108 threads), while C−TKT−MCS-fair almost always yields spread very close to 1 (with rare outliers not exceeding 40).

The delegation methods also exhibited high spread values (for example, on SPARC T4-4, up to 2,100 for InletQueue, 670,000 for DNCInletQueue and 360,000 for MPSCChannel; the situation is not as bad on X4800, but still we occasionally see spread values up to 1,200).

Next we describe a preliminary exploration of how the fairness of the delegation methods might be improved. MPSCChannel suffers from CAS contention on the remotely-shared PutCursor variable. InletQueue applies CAS on the message slot to swap in a pointer to the message, and DNCInletQueue uses a simple spinlock to acquire ownership of the message slot. When threads compete in this manner, unfairness can result because a thread that releases the message slot has the corresponding synchronization variable in cache and is therefore likely to be able to acquire the slot again before another thread can.

To address this issue, we experimented with simple backoff mechanisms whereby, if a thread experiences too many consecutive CAS failures, it sets a flag causing all threads accessing that channel to pause before retrying, conditional on a function of their thread IDs and the number of times the slot lock has been acquired. This reduces contention and gives "priority" to different threads over time. This eliminated the gross unfairness on SPARC T4-4 without impacting throughput, but we still observed spread values of up to 2000 for DNCInletQueue and 1150 for InletQueue, indicating that there is still considerable room for improvement. We found that parameters controlling the threshold and backoff could be tuned to different points in a tradeoff between spread and throughput. We are still experimenting to improve our results here.

Unlike SPARC T4-4, X4800 yielded spread values at worst in the low hundreds even before these optimizations, which were less effective on X4800, although we have not yet tuned them for this platform.

5.4 Hardware-related details

In earlier experiments, InletQueue and DNCInletQueue degraded significantly at higher thread counts. After some investigation, we hypothesized that this was due to "sibling rivalry": client threads executing on the same core as a server thread would compete with the server thread for resources, thus indirectly slowing client threads making requests to that server. To address this issue, when placing a server thread on a core, we reserve all other hardware threads on that core so that they are not used by clients. This resulted in a significant improvement, allowing the delegation methods to outperform all others across the threading range for large linked lists on both platforms, for example Figures 2(c) and 2(d). Although this dedicates more hardware threads to delegation, these threads could potentially be used to benefit the server, rather than interfering with it. We leave investigation of this direction for future work.

This experience highlights one potential downside of delegation. Apart from using cores that might otherwise be used by additional application threads, reserving sibling threads requires server threads to be "pinned" to a specific hardware thread, which can be a mixed blessing. First, overriding the operating system's thread placement policy prevents it from choosing the best placement based on the current workload. This is clearly demonstrated in Figure 2(b): at low thread counts, the lock-based methods have a significant advantage because the operating system is able to place all threads on the same socket.

On the other hand, a fixed relationship between data and the hardware threads that access it can be exploited in some contexts. To illustrate this point we performed an experiment (not shown), in which we controlled the placement of these structures so that each delegation message queue was allocated on the same NUMA node as the corresponding server thread. In contrast, these structures are usually allocated by a single thread at initialization and are thus all allocated in physical memory of the same NUMA node.

This simple placement optimization substantially improved the performance of delegation on X4800, especially for InletQueue and DNCInletQueue; the latter improved by more than 2x in most cases. This may be counterintuitive given that these structures are likely to remain in cache. However, on X4800, each memory access requires communication with the location's "home node" (see [4] for a detailed explanation). Thus, locating each communication structure near the server thread that accesses it most often improves performance.

The substantial performance gains achieved by even this modest optimization reinforces our belief that significantly more could be achieved if hardware were explicitly optimized for such communication patterns.

Reducing coherence traffic between nodes can reduce consumption of inter-socket bandwidth, which may in turn avoid a system-wide bottleneck that may indirectly reduce performance [3]. The delegation methods we have presented were in large part motivated by similar concerns. Using hardware performance counters, we have found that the delegation methods typically generate a small number of remote cache misses per operation (typically around 4-5, although we sometimes observe significantly higher rates in high-contention cases). Software techniques—such as discussed in [2,8], and hardware optimizations tailored for these communication patterns could both significantly reduce this number.

However, recent progress in building NUMA-aware locks [5] has changed the landscape. By limiting how often locks (and therefore associated data) migrate between sockets—while avoiding gross unfairness exhibited by locks that do so "accidentally", such locks can reduce the per-operation remote cache miss rate almost to zero by performing large numbers of operations protected by a lock on one socket before allowing the lock to migrate to another. This depends on sufficient demand for a lock within a socket, suggesting that such techniques are excellent for avoiding performance disasters due to lock contention, but may not be as effective in scalable applications with little lock contention.

6 Discussion

Our results show that delegation can sometimes outperform direct shared-memory approaches, particularly when operations access enough data to ensure that the benefits of delegation outweigh its communication costs. Nonetheless, the best shared-memory mechanisms often performed about the same as or substantially better than the delegation mechanisms. Synchronization granularity is a key issue, for both locking and delegation. For easily partitionable data structures, like those considered in this paper, fine-grained locking is straightforward. For delegation, it is similarly straightforward to partition the data structure, allowing multiple server threads to service requests from client threads in parallel, but finer granularity requires additional hardware threads to be used.

While granularity affects both approaches in similar ways, there are interesting differences. Suppose, for example, that we want to make our hash map resizeable. Resizing is straightforward in the case of delegation, because operations need not synchronize with each other. In contrast, resizing a hash map implemented with per-bucket locks is more challenging, as the resizing must be coordinated with threads accessing the partition using these locks.

Different challenges and opportunities exist when workloads face contention. NUMA-aware locks such as the C−TKT−MCS variants can help limit the performance degradation of lock-based approaches, although these locks impose overhead in the hopefully more common case in which there is no lock contention.

With delegation, an overloaded server thread can become a bottleneck. Client-side techniques that combine multiple requests into one equivalent one, thus reducing the communication costs and the demand on the server thread, may improve performance. Elimination [8] can be used to complete operations without communicating with the server at all [2].

Server-side techniques may help too. For example, a server thread experiencing high demand could repartition its own partition and create an additional server thread to manage it. This may be effective if the execution of operations is the bottleneck. If, however, the communication channel for requests is the bottleneck, simple repartitioning will not help, and more ambitious techniques would be required in which client threads also become aware of the repartitioning.

7 Related Work

Lozi *et al.* [10] propose structuring a client-server system so that one or more cores are dedicated to server threads that execute critical sections on behalf of client threads. Client and server threads communicate through an array of contexts, one per client. A client's context includes the lock address, the critical section's private variables, and a function that encapsulates the critical section's code. (Note that some effort is required to encapsulate critical sections in this way.) Clients and servers use atomic operations on shared variables to signal when a request starts and completes. The authors observe that their scheme improves lock access contention and cache locality, but do not explore alternative signaling or communication structures.

Suleman *et al.* [16] consider an asymmetric multicore architecture encompassing a small number of high-performance cores and many smaller, less powerful cores. The paper examines architectural support for delegating critical sections to the high-performance cores; evaluation is via an in-house simulator.

Metreveli *et al.* [12] describe *CPHASH*, a concurrent hash map that uses a form of delegation to enhance cache locality. They show that delegation can outperform locking for one data structure on one platform configuration. Our goal, in contrast, is to characterize the relative merits of delegation and direct shared-memory mechanisms in a range of data structures, communication mechanisms, workloads, and platform configurations.

Hendler *et al.* [7] and Oyama *et al.* [15] propose mechanisms in which threads execute operations on behalf of others while holding a lock. (Again, critical sections must be encapsulated as self-contained functions.) This approach resembles delegation: a single thread serially executes multiple operations. But that thread is determined dynamically, not statically as in delegation schemes.

8 Conclusions

Delegation works well when the data structure can be partitioned so that it fits in the servers' collective caches. Delegation also works well when critical sections encompass many memory accesses, as in the case of the linked lists, because the communication overhead is outweighed by the savings in cache misses and coherence traffic. These savings are more substantial when the cost of remote memory access is high, allowing delegation to beat efficient NUMA-aware locks.

However, delegation is often outperformed by the best locking implementations. In particular, when critical sections are short, and especially in "small" workloads in which data accessed in the critical section is likely to be cached, locking approaches require little or no remote communication, while delegation still pays in communication overhead but delivers less benefit.

Nevertheless, as the number of sockets in multicore machines grows, so will the cost of remote memory access. Furthermore, techniques not explored in this paper (such as elimination and combining), as well as potential hardware improvements, may make delegation more attractive in the future.

Our experience has shown that low-level hardware details can make a considerable difference to the behavior of synchronization algorithms. Thus, we conclude that multicore applications should be designed around high-level data abstractions, hiding the low-level communication details, so that one mechanism can be replaced by another as workloads and platforms change.

Acknowledgments We are grateful to Nir Shavit for useful input and feedback and to Bill Bridge and Garret Swart who suggested this research direction.

References

1. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scal-

able multicore systems. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

2. I. Calciu, J. E. Gottschlich, and M. Herlihy. Using elimination and delegation to implement a scalable NUMA-friendly stack. In *Proc. Usenix Workshop on Hot Topics in Parallelism (HotPar)*, 2013.
3. M. Dashti, F. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lachaize, V. Quema, and M. Quema. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proc. Conf. on Arch. Support for Prog. Lang. and Op. Systems (ASPLOS)*, pages 381–394, 2013.
4. D. Dice. NUMA-aware placement of communication variables, November 2012. blogs.oracle.com/dave/entry/numa_aware_placement_of_communication1.
5. D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proc. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 247–256, 2012.
6. D. Dice and O. Otenko. Brief announcement: multilane - a concurrent blocking multiset. In *Proc. ACM SPAA*, pages 313–314, 2011.
7. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat-combining and the synchronization parallelism tradeoff. In *Proceedings of the Twenty Third ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, June 2010.
8. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215, 2004.
9. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, Apr. 1979.
10. J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proc. USENIX Annual Technical Conference (ATC)*, 2012.
11. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.
12. Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: a cache-partitioned hash table. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 319–320, New York, NY, USA, 2012. ACM.
13. Oracle Corporation. Oracle's Sun Fire X4800 Server Architecture, 2010. www.oracle.com/technetwork/articles/systems-hardware-architecture/sf4800g5-architecture-163848.pdf.
14. Oracle Corporation. Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B Server Architecture, 2012. www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o11-090-sparc-t4-arch-496245.pdf.
15. Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, 1999.
16. M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proc. Conf. on Arch. Support for Prog. Lang. and Op. Systems (ASPLOS)*, pages 253–264, 2009.
17. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. Int. Symposium on Computer Architecture (ISCA)*, pages 256–266, 1992.