On the Performance of Delegation over Cache-Coherent Shared Memory

Darko Petrović Thomas Ropars André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract

Delegation is a thread synchronization technique where access to shared data is performed through a dedicated *server thread*. When a *client thread* requires shared data access, it makes a request to a server and waits for a response. This paper studies delegation implementation over cache-coherent shared memory, with the goal of optimizing it for high throughput. Whereas client-server communication naturally fits message-passing systems, efficient implementation over cache-coherent shared memory requires careful optimization. We demonstrate optimizations that significantly improve delegation performance on two modern x86 processors (the Intel Xeon *Westmere* and the AMD Opteron *Magny-Cours*), enabling us to come up with counter, stack and queue implementations that outperform the best known alternatives in a large number of cases. Our optimized delegation solution achieves 1.4x (resp. 2x) higher throughput compared to the most efficient state-of-the-art delegation solution on the Intel Xeon (resp. AMD Opteron).

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Cache-coherent shared memory, mutual exclusion, delegation

1. Introduction

The emergence of multi- and manycore processors has urged researchers to design algorithms that can scale to a large number of cores. Mutual exclusion is one of the basic concepts in shared-memory programming. Delegation is a well-known technique to efficiently implement mutual exclusion on shared objects [5, 6, 9, 10, 14, 19]. In delegation, a *server* thread, typically pinned to a processor core to improve data locality, sequentially executes requests sent by other, *client* threads. Thus, clients delegate their work to a server, instead of executing it on their own. Since requests for shared data access are serially executed by the server, and the server is the only thread accessing them, there are no concurrency issues. Such a solution efficiently replaces classic lock-based solutions where a thread takes a lock, directly accesses the shared data, and then releases the lock. It has been shown that delegation-based critical section execution can significantly outperform even the most efficient known classic locks under high contention [10].

Clearly, request throughput achievable using delegation is limited on the server side, *i.e.*, it is limited by the speed at which the server can receive requests, process them, and respond. Delegation is most naturally implemented using message passing, if the hardware at hand provides such a feature [16, 19]. This can result in optimal performance, as the server can be fully prevented from stalling [16]. However, many contemporary processors, including the dominant x86 architecture, still do not provide hardware support for message passing, which makes the implementation of delegation much more challenging. This is because on such processors, delegation has to be implemented over cache-coherent shared memory: Client-server communication is implemented by reading and writing shared cache lines. As such, the performance directly depends on details of the processor's cache coherence protocol, which are usually undocumented.

In this paper, we study how the performance of delegation over cache-coherent shared memory can be improved by taking into account the subtleties of the underlying cache coherence protocol. We show that a significant throughput increase is achievable by employing simple, but inobvious and even counterintuitive optimizations. This enables us to present implementations of basic concurrent objects that perform better under high contention than many known alternatives. More precisely, our contributions are the following:

- We present two optimizations for delegation over shared memory (Section 3). The first one, *local-spin backoff*, consists of introducing a well-tuned backoff in the client's local-spinning loop. This leads to alleviating the collision between local spinning and hardware prefetching, which we have identified as an important source of overhead on real-world processors. The second one is the use of *streaming stores* instead of ordinary store instructions. Their specific cache management and weak ordering enables us to significantly reduce the server's overhead associated with every request.
- We provide a detailed experimental analysis of these two optimizations using two multi-socket x86 processors, an 80-core Intel Xeon Westmere E7-L8867 and a 48-core AMD Opteron Magny-Cours 6176 (Section 4).
- We evaluate ubiquitous shared objects (counters, queues, stacks) implemented with our optimized delegation solution using the two previously mentioned processor architectures. We provide a performance comparison with state-of-the-art delegation techniques, including NUMA-aware techniques, and the best known stack and queue algorithms (Section 4).

Our results show that our optimized delegation solution outperforms state-of-the-art NUMA-oblivious delegation techniques by

up to 4.9x (resp. 2x) on the Intel Xeon (resp. AMD Opteron) at high concurrency levels. Even when compared with NUMA-aware techniques, which are a good fit for the Intel processor, our solution is still 1.4x faster. Finally, our simple linearizable queue implementation using a single server manages to achieve similar or even better throughput compared to the best known blocking and nonblocking alternatives.

2. Background

Before describing our contributions in more detail, we present some basic assumptions and an overview of related work.

2.1 The cache-coherent shared memory model

In the cache-coherent (CC) shared-memory model, threads operate on cached copies of shared variables. We assume a model adapted from the one by Sorin *et al* [18]. A processor chip is composed of single-threaded cores. Each core has its local, private data cache. All cores have access to a globally shared memory through an interconnection network. The cache coherence protocol maintains the *single-writer-multiple-reader* invariant: At any given time, either a single core has read-write access to a cached variable, or some cores have read-only access [18]. *Remote Memory References* (RMRs) are accesses to shared variables that involve communication on the interconnection network. In this model and assuming write-back caches, reading a shared variable generates an RMR if the core does not hold a copy of the variable in either mode. Writing a shared variable generates an RMR if the core does not hold a copy of the variable in read-write mode. RMRs are typically orders of magnitude more expensive than an access to the local cache. Hence, algorithms targeting CC processors should try to minimize the number of RMRs.

This simple model is sufficient to explain the problems studied in this paper. Note that real processors can have features that are not in the model's strictest scope, such as *Simultaneous Multithreading* (SMT), but they do not fundamentally change our considerations (our evaluation includes a processor with SMT support). Note also that in NUMA architectures, where a multicore machine contains multiple nodes (sockets) each provided with local memory and caches, all RMRs do not have the same cost: Bringing a variable from a remote cache of the local node is much less costly than bringing it from the cache of a remote node. Our experimental evaluation also includes comparisons with solutions tailored to NUMA architectures.

2.2 Related work

Delegation is a well-known technique for improving the performance of critical section (CS) execution. A thread is temporarily or permanently dedicated to executing CSes on a shared object on behalf of other program threads. This improves performance compared to a traditional lock-based solution because, assuming that the shared object is not accessed outside CSes, it remains in the cache of the delegated thread. In a solution based on locks, the thread acquiring the lock has to bring the object from the cache of the core that just released the lock to its own cache, potentially generating several RMRs.

The most extreme version of the delegation technique is the server approach. One non-application thread, the server, is permanently delegated to the execution of CSes on a contended shared object. The application threads are clients that can send requests to the server to execute CSes. Remote Core Locking (RCL) [10] is an efficient implementation of the server approach over shared memory. Cleary *et al* [3] take a similar approach, but apply it to *asymmetric synchronization*, where one thread executes the CS much more often then the others. Suleman *et al* [19] propose delegation over dedicated hardware and evaluate how much chip real estate should be used for the server core.

Combining [5, 6, 9, 14] avoids dedicating a core to CS execution statically. Instead, when a thread gets the lock associated with a shared object, it can execute CSes on behalf of other threads in addition to its own. To prevent this thread from starving, it can only execute a predefined number of CSes before releasing the lock and handing over the combiner role to another thread. The different combining approaches mainly differ in the way pending requests are managed. Additionally, Klaftenegger *et al* [9] propose an optimization that enables a client thread to return without waiting for the combiner to execute its request (if the request has no return value). In such cases, one RMR can be removed from the critical path of the server compared to other combining approaches. We propose complementary optimizations that also work when clients need to wait for the result of their operations.

Elsewhere, we have studied delegation in the context of a processor provided with support for message passing in hardware [16]. The results show that this feature can be used to significantly improve performance, mainly because RMRs on the server's critical path can be fully avoided. In this paper, we complement that work by studying how the effect of RMRs at the server can be alleviated on prevalent processors without hardware messaging support.

Experimental comparisons of delegation and locking techniques over CC shared memory have been conducted [2, 4]. Results show that for data structures where fine-grained locking can be efficiently applied (*e.g.*, hash tables with large number of buckets), state-of-the-art locking solutions remain most efficient under high contention. In other cases, delegation is shown to perform better. The optimizations we propose can further increase the performance of delegation compared to classic locking techniques.

Finally, note that delegation is not used only in the context of mutual exclusion. For instance, several studies propose delegation as a solution to design scalable operating systems [1, 21]. The results we present could also be of interest in this context.

3. Optimizing delegation over CC shared memory

In this section, we first describe the server-based delegation algorithm that is used as a starting point for our work. Then, we explain its main bottleneck and detail how it can be optimized for execution over CC shared memory by taking into account characteristics of modern processors.

3.1 Baseline algorithm

In the description of the baseline algorithm, we make the following assumptions: (a) Participating threads are known in advance; (b) Data exchanged between clients and servers can fit into one cache line. Assumption (a) allows us to pre-allocate per-client buffers and thus eliminate the cost of synchronization on a shared buffer[1]. Since work to delegate is usually encapsulated inside a function, considering the typical case of 64-byte cache lines, one cache line can store a function pointer, a flag, and several arguments, which justifies assumption (b).

The code is given in Algorithm 1. It uses an array of cache-line-sized slots, one per client thread. Every slot contains a flag with two possible values, $REQUEST$ and $RESPONSE$. To make a request, a client writes the function pointer and arguments in the corresponding slot, and then sets the flag. The server repeatedly scans the client slots, and if there is a request, it is immediately

[1] If participating threads are not known in advance, one solution is to assign communication buffers to cores, and to use a simple locking algorithm to arbitrate between threads that would execute on the same core.

Algorithm 1 Baseline delegation algorithm

```
1:  CacheLine channel[0..n−1]   {channel[i]: communication
    channel between client i and the server}
2:  channel[0..n − 1].flag ← RESPONSE
3:  channel[0..n − 1].msg ← NULL

    {Server code}
4:  function run_server()
5:      client_id ← 0
6:      while true do
7:          if channel[client_id].flag = REQUEST then
8:              {func, args} ← channel[client_id].msg
9:              channel[client_id].msg ← func(args)
10:             channel[client_id].flag ← RESPONSE
11:         client_id ← (client_id + 1) mod n

    {Code of client i}
12: function delegate(func_ptr, args)
13:     channel[i].msg ← {func_ptr, args}
14:     channel[i].flag ← REQUEST
15:     while channel[i].flag ≠ RESPONSE do
16:         nop
17:     return channel[i].msg
```

executed, and a response is sent to the client by writing it in the slot and appropriately setting the flag. Algorithm 1 is essentially a stripped-down version of RCL [10]. RCL is more complex because it provides additional features that are not central to this paper such as support for nested critical sections or the possibility to have one server managing CSes on several shared objects.

3.2 Opportunities for optimization

The server in Algorithm 1 experiences at least two RMRs for each client request. This is illustrated in Figure 1. This figure shows accesses to the shared cache line `channel` during the communication between a client and the server for the execution of one CS. The figure also shows the cache line status during the execution: state `M` (*modifed*) corresponds to read-write mode; state `S` (*shared*) corresponds to read-only mode. When the client wants to execute a CS, it writes its request to the cache line `channel`, and then spins on that cache line until it receives a reply from the server. The server first reads the request from `channel`. Since the last access to the cache line was from the client writing the request, this read triggers an RMR. Then, the server executes the critical section. Finally, it writes to `channel` to inform the client that the request has been processed. This write triggers invalidation of the client's copy of the cache line, which represents a second RMR. Obviously, when the server is under high load, these two RMRs are the main performance impediment. Their net effect is very dependent on the processor's characteristics: The different RMRs might partially overlap, depending on the memory consistency model of the processor at hand, resulting in fewer CPU stalls. Nevertheless, they remain an important source of overhead even on a processor with weak memory consistency [16]. In the following we study different techniques to minimize their impact on the server performance.

3.3 Proposed optimizations

By carefully analyzing hardware-level details of executing the presented algorithm on a typical multisocket multicore processor, we identified two optimizations that can considerably improve its performance: backoff in local-spin loops and streaming (non-temporal) stores. We now discuss each of them in more detail.

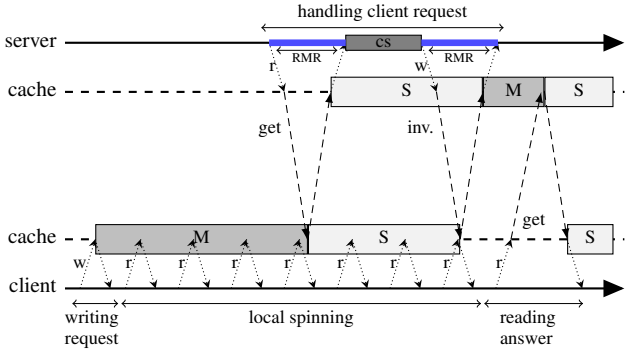


Figure 1: Communication between the server and a client – Baseline algorithm (2 RMRs on the server per request).

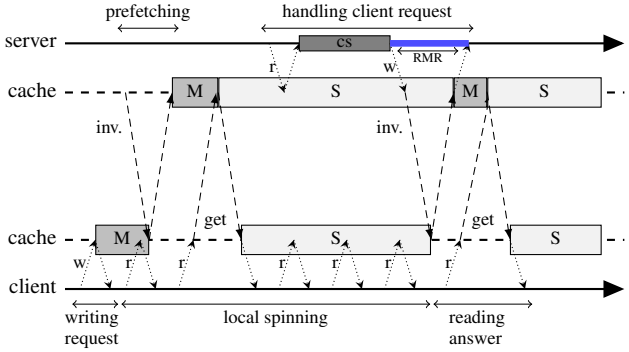


Figure 2: Communication between the server and a client – Spinning-prefetching collision (one RMR on the server per request).

3.3.1 Backoff with local spinning

Contemporary processors usually have automatic prefetchers, which detect regular data access patterns and proactively bring data closer to the processor core before it is referenced, thus hiding access latency. In Algorithm 1, the server repeatedly iterates over consecutive cache lines, which results in a very regular cache line access pattern, likely to trigger the prefetcher. In our case, prefetching a cache line in read-only mode could hide the latency of reading the client's request, but an RMR would still be generated to upgrade the cache line to read-write mode, at the time the server writes to the channel. Prefetchers are actually able to detect write-access patterns and bring the cache line to the server cache directly in read-write mode. However, the cache line will get downgraded to read-only mode immediately as illustrated by Figure 2, since the client is spinning on that cache line, waiting for a response. Therefore, even local spinning, usually considered to be the first condition for a concurrent algorithm's scalability [11], can be detrimental to performance, since it hinders the automatic prefetcher. We will refer to this problem as the *spinning-prefetching collision*.

A way to avoid this collision is to introduce a well-tuned backoff in the client's spin loop. Instead of constantly checking the flag in a loop, the client introduces a fixed waiting time between consecutive checks. Ideally, the backoff should be such that there is only one check, right after the server has written the response, as shown in Figure 3. If the waiting time is too short, the spinning might conflict with the prefetcher; If it is too long, the client will unnecessarily keep waiting even though the response is already available. The right value depends on many factors, such as the current load,

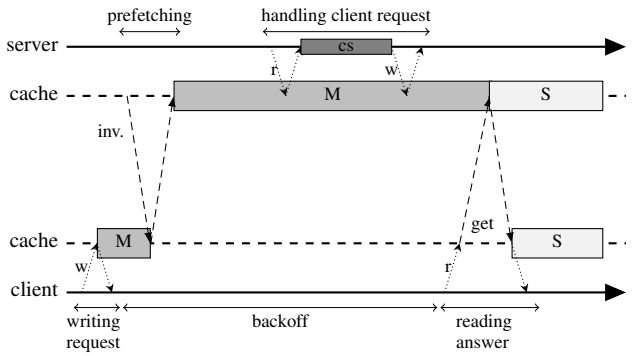


Figure 3: Communication between the server and a client – Backoff with local spinning (no RMRs on the server).

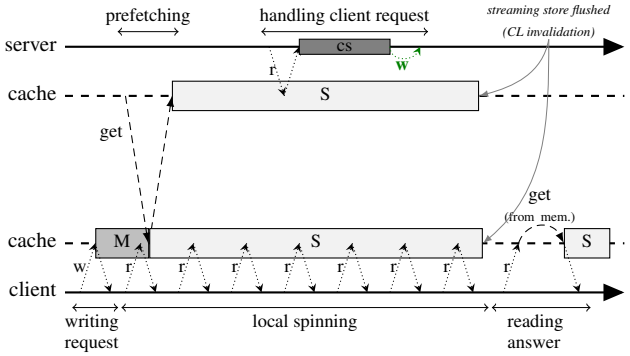


Figure 4: Communication between the server and a client – Streaming stores (no RMRs on the server, streaming stores in green).

the way prefetching works etc. In the paper, we tune the backoff manually, i.e., we measure performance with different fixed backoff values, but it would be interesting to study how the waiting time can be re-calculated and updated at runtime.

It is important to stress that, although backoff is a well-known technique in concurrent programming, it is most often used to deal with a completely different problem. Namely backoff is usually used to reduce contention on a shared variable that is concurently accessed by an arbitrary number of threads [7, 11]. In our case shared variables are not contended since only one client and the server can access the same cache line concurently, but introducing backoff in the spinning loop of the client allows avoiding interference with prefetching on the server side.

Another way to prevent the spinning-prefetching collision from happening would be to use the MONITOR/MWAIT instructions, supported by x86 processors. With these instructions, a thread can switch to a low-power state and get notified when a memory location changes, instead of spinning on it. Although this is conceived as an energy-saving feature, it might also have visible performance benefits in our case, since spinning is avoided. However, the MONITOR/MWAIT instuction pair is available only in kernel mode on the processors we could get access to. This is not a problem per se, because MONITOR/MWAIT can be exposed to userspace applications via a special piece of kernel code – a loadable kernel module in case of Linux. Even though we have written such a kernel module, as a simple character device, it turned out to be of little use, because the kernel itself becomes the bottleneck in contended scenarios. This is so even if a separate kernel module is used for every core, and we used a very recent kernel (Ubuntu's Linux 3.2.0-64-generic from June 2014). We speculate that this is due to concurrent access to the kernel data structures for managing character devices. Still, the possibility of introducing userspace access to MONITOR/MWAIT is left open [8], which would make it an interesting alternative to study.

3.3.2 Streaming stores

To make the implementation of Algorithm 1 more efficient, we explore an alternative store instruction, a *streaming store*, also referred to as *non-temporal store*. Streaming stores differ from ordinary stores in two aspects: (a) They are weakly ordered and (b) they do not bring the data to the core's cache for writing, but write directly to memory instead. Algorithm execution using streaming stores is illustrated in Figure 4. Hence, (a) allows a server store operation to be asynchronously completed and to be overlapped with subsequent requests' handling. Note also that (b) implies that the spinning-prefetching collision described in the previous subsection is not a concern in this case as the prefetcher will try to fetch cache lines in read-only mode (since the server does not issue read-write access requests anymore).

The server's stores still cannot become visible to other cores in a fully arbitrary order: Program order needs to be preserved between stores belonging to the same operation (i.e., the flag must not be written before the actual data). An obvious way to ensure this is to put a memory fence between writing the data and the flag, but such a fence at the server side would force the write buffers to be flushed, incurring overhead that defeats the purpose of using streaming stores. To avoid this, one can take advantage of the fact that the server only sends a function's return value (if there is one) back to the client, so the data and the one-bit flag can fit a variable that can be atomically read and written, thus ensuring that the flag is never updated before the data. There might be other platform-specific ways to ensure this.

In spite of potential performance benefits, it should be noted that streaming stores cannot be applied in all cases because of their weak ordering semantics. Namely, a server implemented with streaming stores can only be used if the data accessed by the server is never accessed by any other thread. This is the case when the server is used to replace a coarse-grained lock on a concurrent object (since threads do not access the object outside the lock). Also, the constraint is satisfied by algorithms based on fine-grained locking, as long as the different locks protect disjoint data sets (*e.g.*, hash tables). However, if data sets are not disjoint, the streaming store that acknowledges request handling can become visible to other cores before the stores that changed the object. An example is the Michael and Scott blocking queue algorithm [12] that we adapt in Section 4 to use delegation. The original algorithm uses two locks, one for enqueue the other for dequeue operations, that we replace by two servers. Streaming stores cannot be used in this case since data enqueued by one server are eventually dequeued by the second one, breaking the above constraint. Of course, falling back to one lock ensures correctness.

4. Evaluation

The goal of this section is twofold: to examine the effectiveness of the proposed optimizations when delegation is implemented on real-world processors (Section 4.2) and to compare the performance of optimized delegation with that of most relevant related approaches (Section 4.3). Before presenting experimental results, we describe our setup.

4.1 Experimental setup

We use two x86 machines throughout this section: a Supermicro SuperServer 5086B-TRF consisting of eight 10-core Intel Xeon Westmere E7-L8867 (2.13 GHz) chips with 2-way SMT (Hyper-

threading), *i.e.* 160 hardware threads in total, and an IBM x3755-M3 with four 12-core AMD Opteron Magny-Cours 6176 (2.3 GHz) packages without SMT, for a total of 48 hardware threads. The Xeon runs Red Hat Enterprise Linux Server 6.4 with Linux 2.6.32-358.6.2.el6.x86_64, and the Opteron runs SUSE Linux Enterprise Server 11 with Linux 2.6.32.46-0.3-default. All of the implementations are written in C, carefully optimized and compiled with the O3 flag (maximum optimization level) using GCC 4.4.7 (resp. 4.7.2) on the Xeon (resp. Opteron).

Besides the optimized server-based solutions that implement Algorithm 1, we also evaluate CC-Synch [5], as a representative of combining approaches, as well as H-Synch, its NUMA-aware version. H-Synch follows the general idea of grouping operations originating from the same node and executing them together in batches, thus incurring fewer cross-socket cache line transfers and significantly increasing throughput. Even though the Opteron is a multisocket NUMA platform, we do not present H-Synch results on it, since its internal characteristic incur cross-socket communication even if only cores from one socket are involved, thus making typical NUMA-aware strategies unsuccessful [4]. Our experiments have confirmed this.

To evaluate the performance of an algorithm, we use it to implement a concurrent object and stress-test it using a varying number of threads. Each thread repeatedly executes operations on the concurrent object, with a short pause of random duration (up to 1000 CPU cycles) between two consecutive requests. We increase the number of clients and measure aggregate throughput, *i.e.* the total number of executed operations by all threads in a unit of time. Every point in the graphs is an average over 10 one-second runs. To avoid OS scheduler interference, we explicitly pin threads to respective cores and run at most one thread on each core. When increasing the number of clients, we pin them to cores from different sockets in a round robin fashion, in order to uniformly distribute threads across the sockets[2]. In server-based implementations, the server is pinned to hardware thread 0. If two servers are used, the second server is pinned to thread 1. On the Xeon, whenever a server thread is used, we do not pin any thread to the other hardware thread that belongs to the same physical core as the thread running the server. This is to avoid undesirable interference with the server, which can impact performance and thus render result analysis significantly more difficult. Note that this is unnecessary on the Opteron since it does not have SMT support.

Unless otherwise stated, the client-server communication slots in implementations of Algorithm 1 on the Opteron are allocated as a contiguous array of cache lines, to maximize automatic prefetching. The slots are homed at the server's socket. On the Xeon, instead of using consecutive cache lines, every second cache line is used. We do so because of the *adjacent line prefetcher*, which on every cache miss prefetches the first neighbouring cache line, thus making cache lines always move in pairs [8]. This turned out to result in unfavorable interference in our experiments, which we avoid by skipping every second cache line when allocating client slots. In experiments where memory management is needed (stacks and queues), cache-aligned memory chunks are allocated and deallocated using per-thread pools (we use the implementation provided by the authors of CC-Synch [5]).

In all delegation implementations, clients pass pointers to functions that the servicing thread should execute. An alternative is to pass a *token* (usually an integer) the server can use to decide what to execute [3]. This avoids function pointers and thus enables the compiler to optimize away the function call for every request, but this did not show performance benefits in our experiments because the other synchronization overheads on the tested processors dominate the overhead of a function call. Moreover, we have observed that the function call is mostly "absorbed" by the surrounding code, *i.e.* it is executed in cycles that would otherwise remain idle.

[2] We have also done single-socket experiments on the Xeon, but our optimizations are not a good fit for that case, because intra-socket cache coherence has very different characteristics, such as relying on the inclusive L3 cache, and very short communication latencies. The Opteron has only 6 cores per socket, which is not enough parallelism to make strong conclusions in this case.

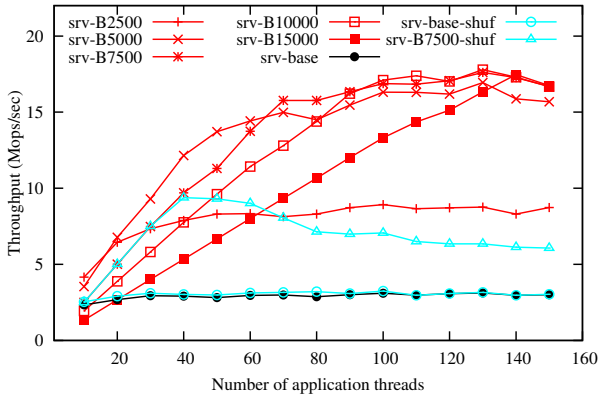
4.2 Analysis of the optimization performance

We present the performance of Algorithm 1 with and without the optimizations proposed in Section 3. To do so, we implement a concurrent counter, which supports only one operation, $fetch\_and\_add$ (atomically increment the counter and returns its previous value).

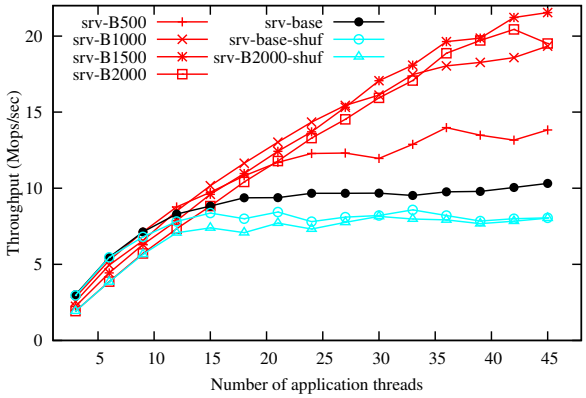
First we evaluate the impact of local backoff in Figure 5. We can see that it significantly improves throughput in most concurrency levels on both processors. The performance increase is up to 6x (2x) on the Xeon (Opteron). Increasing the backoff duration above a certain value does not increase the throughput further, most likely because the backoff is sufficient to fully avoid collision with the prefetcher. To confirm that the performance increase comes from minimizing the spinning-prefetching collision, we include an implementation where the server does not access client slots sequentially, but randomly. This results in an irregular access pattern at the server, which is harder to track by the prefetcher. As can be seen in the figure, such shuffling of client slots greatly reduces performance when backoff is used, which is due to less prefetching. On the other hand, shuffling has little or no effect when backoff is not employed (see *srv-base* vs *srv-base-shuf*): Due to the spinning-prefetching collision, every response written by the server still causes a cache miss, so the bottleneck stays the same as without prefetching.

Figure 6 shows the impact of using streaming instead of ordinary stores. There is a visible throughput increase of 3.5x (1.7x) on the Xeon (Opteron) with respect to the baseline performance, which confirms that streaming stores are a good choice for throughput optimization. Further, we examine local backoff effectiveness in this case. The results are different on the two tested processors. On the Opteron, backoff on top of streaming stores does not result in a further performance increase, meaning that applying either backoff or streaming stores in isolation is already enough for attaining the highest throughput. This is not a surprise, since the spinning-prefetching collision is not expected when streaming stores are used (cf. Section 3). However, the result on the Xeon does not follow this logic – adding local backoff helps even on top of streaming stores. Because the implementation of streaming stores is not documented in detail, we have done additional experiments to get a better understanding of this behavior. These experiments indicate that there is a conflict: If there is an outstanding streaming store to a cache line from core A, its performance is significantly impaired by core B spinning on the same cache line. The pending streaming store invalidates the copy on core B, which immediately issues another read request, since it is spinning. This newly generated read request apparently obstructs the streaming store, causing it to take about 3x more time to complete. This obstruction is avoided by adding backoff. Higher backoff values help because such conflicts become less probable. This is a strong hint that the spinning-prefetching collision is not the only reason why local spinning can hamper performance: Other characteristics of the machine at hand may incur it as well. The conflict was irreproducible with both normal and streaming stores on the Opteron, and with normal stores on the Xeon.

In the above experiments, we can see that there is a tradeoff involved in choosing the best backoff duration. Increasing it improves throughput (to some extent), but at the expense of worsening low-concurrency performance. Choosing the right value depents on

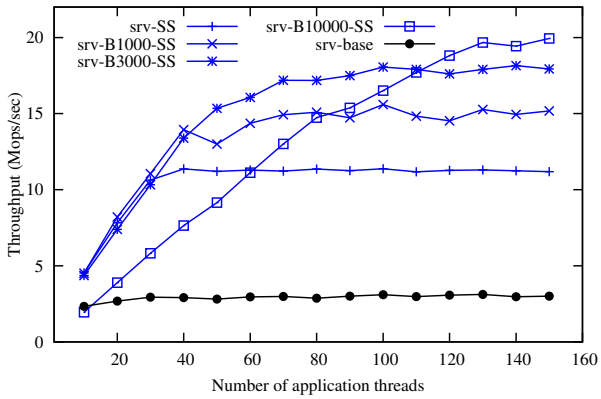


(a) Xeon E7-L8867

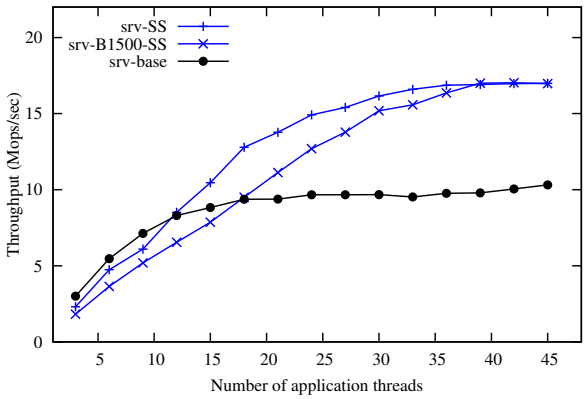


(b) Opteron 6176

Figure 5: Impact of local backoff on delegation throughput. *srv-base* is the implementation of Algorithm 1 before our optimizations. Suffix *Bx* corresponds to an implementation with a backoff of x CPU cycles. Suffix *shuf* denotes cache line shuffling (cache lines are not sequentially, but randomly read by the server).



(a) Xeon E7-L8867



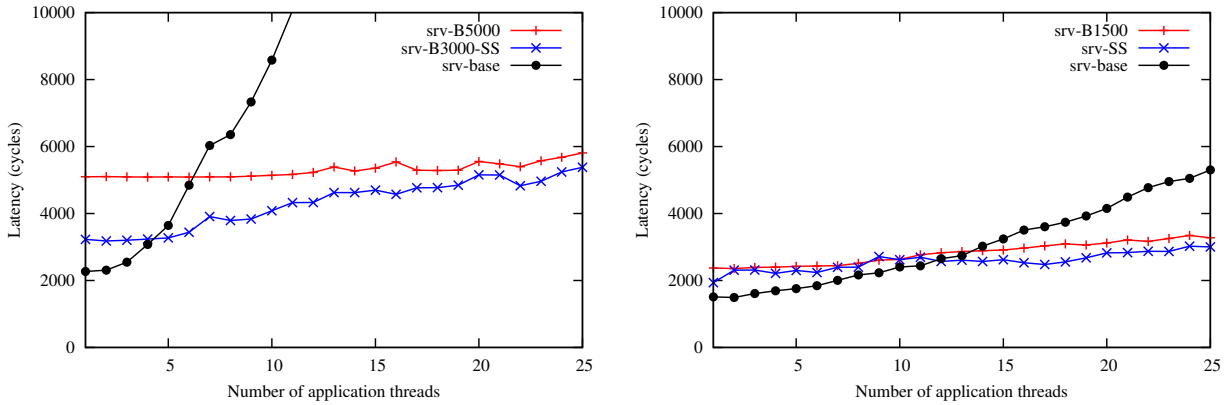
(b) Opteron 6176

Figure 6: Impact of streaming stores on delegation throughput (*SS* denotes that streaming stores are used). *srv-base* is the implementation of Algorithm 1 before our optimizations. Local backoff is also evaluated: suffix *Bx* corresponds to an implementation with a backoff of x CPU cycles.

the targeted application. In the rest of this section, we have chosen values that attain high throughput without unreasonably increasing latency in low concurrency levels (*srv-B5000* and *srv-B3000-SS* on the Xeon, *srv-B1500* and *srv-SS* on the Opteron).

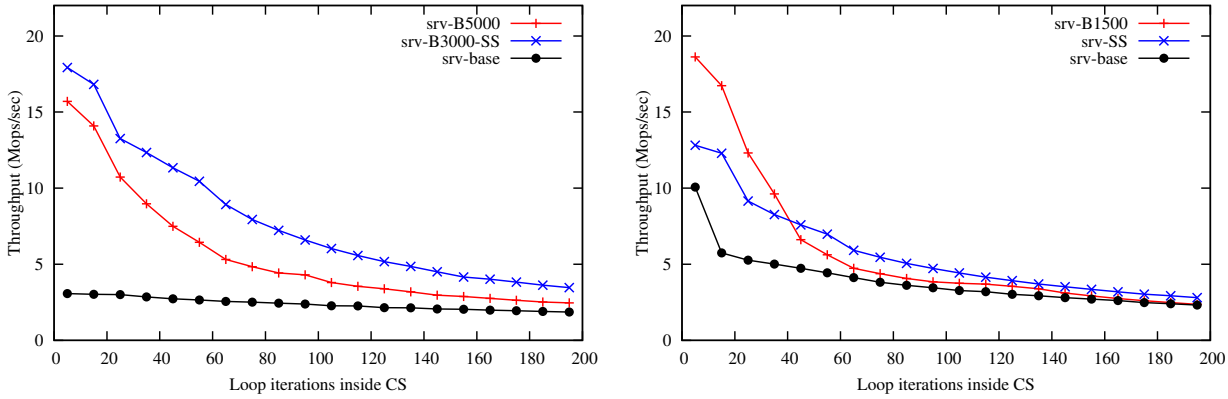
We quantify more precisely the impact of local backoff and streaming stores in cases of little or no concurrency, by observing average request latency in Figure 7. Not surprisingly, the baseline implementation performs best in the lowest concurrency levels. The latency of backoff-based implementations is mostly dependant on the chosen backoff duration, which only adds overhead in case of few active threads. However, even the backoff values in the figure, chosen for high throughput, do not lead to excessively high latency. With the exception of *srv-B5000*, they are within 1.6x of the latency of *srv-base* even with only one client thread. Note that the small diference is partly due to the test configuration, which stresses the general case of cross-socket communication: Delegation within a socket, when possible, would exhibit lower latencies. Overall, backoff and streaming stores are not the best fit for low-concurrency cases, but as the level of concurrency increases, they become a more and more appealing alternative. This is expected, because both optimizations deliberately trade low-concurrency for high-concurrency performance.

Now we measure performance with a longer critical section. Instead of one counter increment as in previous experiments, we allocate an array of 64 integers and the critical section consists of incrementing each integer sequentially (modulo 64) in a loop. The number of loop iterations varies. We stress the server with the maximum number of clients (158 on the Xeon and 47 on the Opteron) executing this CS, and we plot the result in Figure 8. As the critical section size increases, it starts dominating the synchronization overhead and optimizations become less and less relevant. However, even at 200 loop iterations there are still visible benefits: the



(a) Xeon E7-L8867 (b) Opteron 6176

Figure 7: Latency evaluation with local backoff and streaming stores. Suffix *SS* – streaming stores are used; suffix *Bx* – backoff of x CPU cycles is used.



(a) Xeon E7-L8867 (b) Opteron 6176

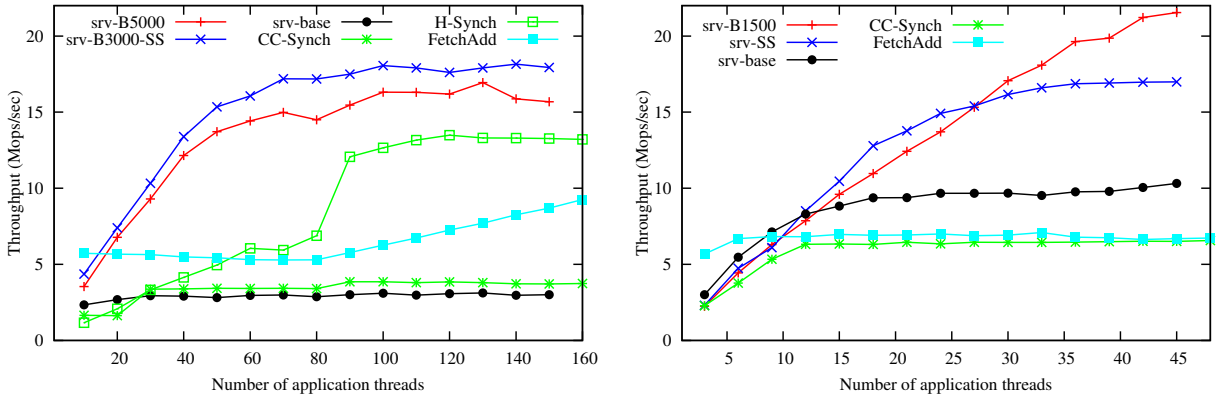
Figure 8: Maximum throughput with long critical sections. Inside the CS, elements of an array of 64 integers are incremented in a loop, one increment per loop iteration. Suffix *SS* – streaming stores are used; Suffix *Bx* – backoff of x CPU cycles is used.

version optimized using streaming stores outperforms the baseline implementation by 1.84x (1.19x) on the Xeon (Opteron). Still, it should be noted that this experiment serves only as a rough estimate of what happens with longer critical sections, as it does not simulate many things that a real-life critical section might do, such as cache and TLB misses, floating-point operations, etc. Thus, actual performance impact should be evaluated on a case-by-case basis, which we do next, by evaluating concurrent objects that should benefit the most from the proposed optimizations.

4.3 Concurrent data structures

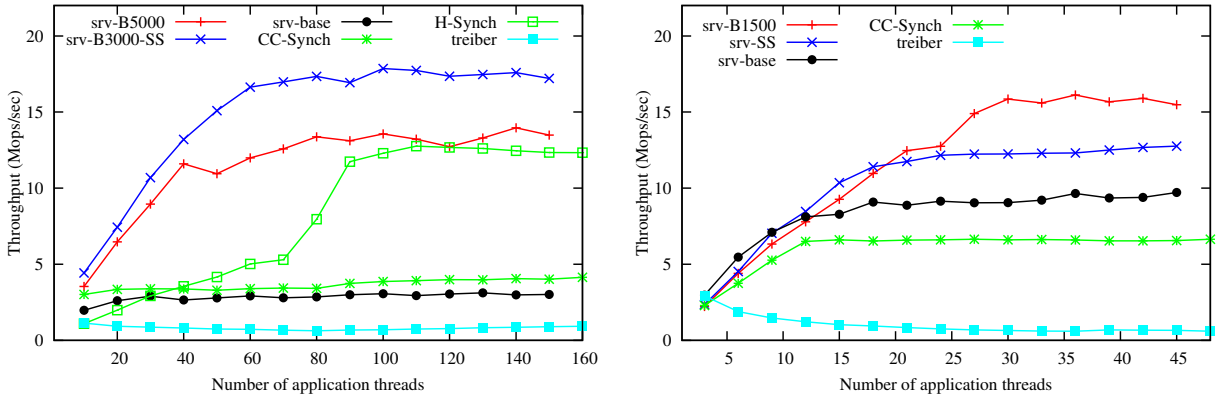
Here we use delegation to come up with efficient implementations of some ubiquitous concurrent objects, counters, stacks and queues, and we compare them with well-known existing implementations.

Figure 9 gives the performance of different concurrent counters. Besides the server-based implementations, CC-Synch, and H-Synch, we also include a concurrent counter trivially implemented using the atomic *fetch-and-add* instruction. In high concurrency levels, our optimized *srv* implementations consistently outperform all other counters. CC-Synch achieves performance similar to that of *srv-base*, which is not surprising, given that the servicing threads in both implementations have a similar communication pattern – two cache misses at the server (combiner) per operation and no further optimizations. On the Xeon, H-Synch gives a significant performance improvement over CC-Synch because of its NUMA-awareness, indicating a striking difference in inter- and intra-socket communication costs. Still, optimized *srv* performs even better in most concurrency levels, although it does not take into account the processor's NUMA characteristics. This shows that cross-socket communication does not necessarily need to be eliminated to achieve high throughput: Identifying important latencies and removing them from the critical path, as we do here, can yield even better results. Perhaps surprisingly, even the *fetch-and-add* counter reaches far lower throughput than srv. This is mostly because every core has to bring the counter to the local cache in order to increment it, so the cache line containing the counter bounces between opera-



(a) Xeon E7-L8867 (b) Opteron 6176

Figure 9: Performance of concurrent counters. *srv-** – server-based implementations; *CC-Synch, H-Synch* – combining implementations [5]; *FetchAdd* – hardware fetch-and-add instruction



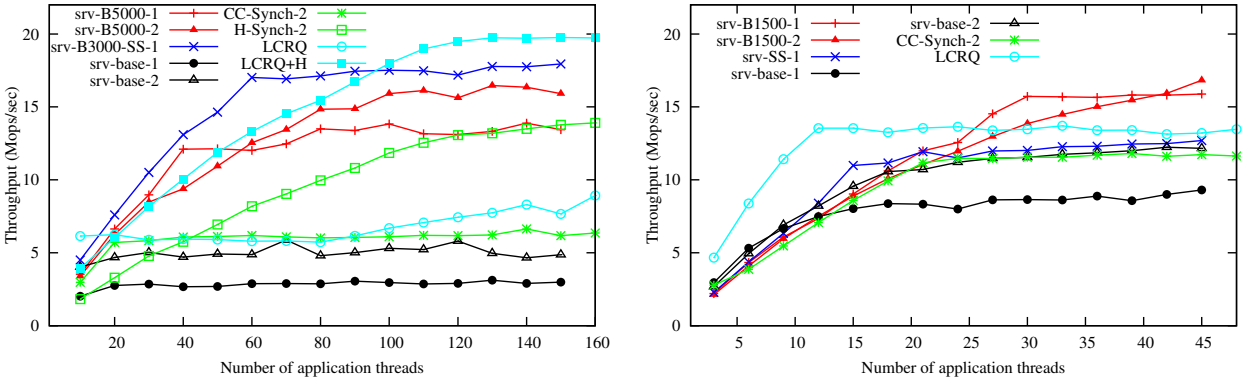
(a) Xeon E7-L8867 (b) Opteron 6176

Figure 10: Performance of concurrent stacks (initially empty) under balanced load (every thread alternates between *push* and *pop*). *srv-** – server-based implementations; *CC-Synch, H-Synch* – combining implementations [5]; *treiber* – Treiber's nonblocking stack [20]

tions, which often includes a cross-socket transfer. On the Xeon, we can also see that *fetch-and-add* performance, after a period of stability, suddenly grows again with more than 80 threads. This is because each newly added thread is co-located with an existing thread on the same physical core (because of Hyperthreading). When the counter's cache line is brough to a core's cache, increments of both threads sharing that core are often executed together, which avoids cache misses and thus improves performance.

Now we examine implementations of concurrent stacks (Figure 10). They are implemented straightforwardly from the sequential specification of a stack, by putting the code of push and pop operations inside a critical section. It is well known that the scalability of a concurrent stack can be further improved using *elimination* [17], *i.e.*, combining push and pop operations that cancel each other out, without even accessing the stack. We do not include an elimination layer in our implementations, as it is orthogonal to the main topic of this work. Stacks based on a server, CC-Synch, H-Synch, as well as the nonblocking Treiber stack [20] are presented. The results with CC-Synch, H-Synch, and a server are very similar to those in Figure 9 (counter), which is not a surprise because both counters and stacks are implemented with only one server. There is only a small difference in the peak throughput: Since the stack's push and pop operations include more work than incrementing a counter (data allocation/deallocation and a short linked list manipulation), critical sections are longer and the server can execute fewer operations in a unit of time. The performance drop is more visible on the Opteron: a possible reason is that the Xeon's more complex prefetching logic is able to hide the latency of the cache miss that happens when a new stack element is allocated.

Finally, we compare concurrent FIFO queue implementations. In contrast to the implemented counters and stacks, where the object is coarsely locked, queues allow a certain degree of fine-grained locking. The well-known Michael and Scott blocking queue (MS-Queue), for instance, uses two separate locks, operating at opposite ends of the queue (one for enqueue, and the other for dequeue operations) [12]. We implement MS-Queue using two



(a) Xeon E7-L8867

(b) Opteron 6176

Figure 11: Performance of concurrent queues (initially empty) under balanced load (every thread alternates between *enqueue* and *dequeue*). *srv-** – server-based implementations of blocking MS-Queue [12]; *CC-Synch, H-Synch* – combining implementations [5] of blocking MS-Queue [12]; *LCRQ, LCRQ+H* – nonblocking queues for x86 [13]; suffix *-x* is the number of locks used in MS-Queue implementations

servers, as well as using two CC-Synch/H-Synch combiners. Besides two-lock implementations, we also evaluate one-lock ones, for two reasons. First, that enables us to directly quantify the benefit from introducing fine-grained locking. Second, the streaming-store optimization is not applicable to fine-grained MS-Queue, as explained in Section 3. In addition to these different implementations of MS-Queue, nonblocking LCRQ [13], as well as its hierarchical version, LCRQ+H, are also included (with the ring size of 2^{17}, and for LCRQ+H, timeout set to 400 Kcycles). LCRQ is specifically designed with the rich set of atomic instructions supported on x86 processors in mind, and is therefore expected to perform well.

The results are shown in Figure 11. First we discuss the MS-Queue implementations. Fine-grained locking significantly improves the performance of the *srv-base*, CC-Synch and H-Synch queues implemented using a single lock (*CC-Synch-1* and *H-Synch-1* not shown to avoid clutter), but has a much less pronounced impact on the optimized server implementations, especially on the Opteron, where it does not give any tangible benefits over the coarse-grained version. We believe one reason is the hardware prefetcher, which has a more complex task in this case. When there is a coarse lock on an object, there is only one server executing all critical sections, so cache misses mostly originate from client-server communication, since the data structure itself, once allocated, stays in the server's cache. In case of fine-grained locking, however, data locality is suboptimal because the queue is directly accessed by two server threads and the data needs to move – typically, when a dequeuing server dequeues an item, it incurs a cache miss, since the item is in the enqueuing server's cache. With more cache misses coming from data accesses, the pattern observed by the prefetcher becomes less regular and the performance drops.

Nevertheless, our optimized implementations still provide competitive performance (even those without fine-grained locking): In high concurrency levels, they reach the highest throughput on the Opteron, and are only outperformed by LCRQ+H on the Xeon. However, it should be noted that the NUMA-awareness strategy used by LCRQ+H trades performance for fairness. In the presented experiment, the *fairness ratio* of LCRQ+H, *i.e.*, the ratio between the highest and the lowest number of operations executed by some thread during a time interval, was typically 1.4x. At the same time, the server and combiner-based implementations exhibit almost perfect fairness (every thread executed nearly the same number of operations). In more detail, with LCRQ+H, at every point there is one *active* NUMA socket – any operations from other sockets are paused for a certain amount of time, and then they try to make their socket active [13]. The duration of this pause is a tradeoff – higher values give a better NUMA locality and thus higher throughput, but some nodes are increasingly likely to starve. The result shown in Figure 11 is for the pause of 400 Kcycles. With a 1 Mcycle pause, maximum throughput grows over 30 Mops/sec, significantly outperforming the other queues, but with lower fairness – a typical fairness ratio in high concurrency was 4.

5. Discussion

The above experiments show that local backoff and streaming stores can dramatically improve delegation performance in many cases. It turns out that simple hardware-aware optimizations play a key role in optimizing concurrent code, which corroborates recent results, stating that synchronization performance is mainly a hardware property [4].

It is also noteworthy that there is a number of other details at the level of cache coherence protocols that can affect delegation performance. For example, we have experimented with different placements of client communication slots across sockets (recall that they are allocated at the server's socket in Section 4). This turned out to have a surprisingly big performance impact, most likely because of different work distributions between coherence agents. However, exploring this in more details is hard without knowing the inner workings of the cache coherence protocol.

Even with the proposed optimizations, there is still ample space for further improvement. In terms of throughput, we can see that the best result is about 20 Mops/sec on both processors for a concurrent counter, which means that the server takes about 100 CPU cycles to process every request. Since the critical section itself is very short in this case (only a couple of cycles to access a variable in the L1 cache), we can conclude that the rest is pure synchronization overhead. Indeed, hardware event counters indicate that the server core is stalled most of the time, even after applying our optimizations. We believe this is mostly due to unsuccessful or partial prefetching, as well as to the cost of flushing the streaming-store buffer.

In [16], we showed that on a processor provided with hardware message-passing support, such stalls on the server can be fully avoided. Reaching the same result in a cache-coherent shared-

memory system would probably require being able to specify the cache of a remote core where data should be placed, as proposed in [15]. Such a solution would allow a client to specify that its request should be moved to the server cache, avoiding the need to rely on hardware prefetchers to transfer cache lines in time to avoid stalls. Our experiments also show that the solutions that optimize throughput are detrimental to latency in low concurrency. On the contrary, hardware message-passing [16] or location-aware cache [15] allow achieving both high throughput and low latency. Considering the relatively low performance our optimized technique achieves compared to a solution based on hardware message passing, and the huge number of experiments we had to conduct to understand how these optimizations interact with the cache coherence system, we argue that the easiest and most efficient approach to thread synchronization at large scale is to provide hardware-specific features such as those previously mentioned.

6. Conclusion

The paper presents two optimizations for delegation over cache-coherent shared memory: (i) backoff in local-spin loops to minimize collision with hardware prefetchers and (ii) weakly-ordered streaming stores to avoid memory-model limitations. Although simple, these two optimizations subtly interact with the cache-coherency protocol and the hardware prefetchers of modern x86 processors to achieve unprecedented throughput for the execution of critical sections. Hence, concurrent counters, stacks, and queues implemented with our optimized delegation solution outperform the most efficient NUMA-oblivious and NUMA-aware, both blocking and nonblocking alternatives in most cases, especially under heavy contention.

Acknowledgments

We would like to thank Martin Biely, Tudor David, Žarko Milošević, Adam Morrison, Omid Shahmirzadi and Vasileios Trigonakis for their comments on an earlier version of this paper. We are also very thankful to the DIAS laboratory at EPFL, especially Danica Porobić and Pinar Tözün, for arranging access to the Intel Xeon machine.

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 scalable multicore systems. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[2] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97, 2013.

[3] J. Cleary, O. Callanan, M. Purcell, and D. Gregg. Fast asymmetric thread synchronization. *ACM Transactions on Architecture and Code Optimization*, 9(4):27:1–27:22, Jan. 2013.

[4] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, 2013.

[5] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[6] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010.

[7] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions Programming Languages and Systems*, 15(5):745–770, Nov. 1993.

[8] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*, February 2014.

[9] D. Klaftenegger, K. Sagonas, and K. Winblad. Brief announcement: Queue delegation locking. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 70–72, 2014.

[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 *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[11] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions Computer Systems*, 9(1):21–65, Feb. 1991.

[12] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996.

[13] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[14] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, 1999.

[15] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim. Location-aware Cache Management for Many-core Processors with Deep Cache Hierarchy. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 20:1–20:12, 2013.

[16] D. Petrović, T. Ropars, and A. Schiper. Leveraging Hardware Message Passing for Efficient Thread Synchronization. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014.

[17] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the 7th annual ACM symposium on Parallel algorithms and architectures*, 1995.

[18] D. Sorin, M. Hill, and D. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[19] M. A. Suleman, O. Mutlu, M. Qureshi, and Y. Patt. Accelerating Critical Section Execution with Asymmetric Multicore Architectures. *IEEE Micro*, 30(1):60–70, Jan. 2010.

[20] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[21] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.