ffwd: delegation is (much) faster than you think

Sepideh Roghanchi
University of Illinois at Chicago
srogha2@uic.edu

Jakob Eriksson
University of Illinois at Chicago
jakob@uic.edu

Nilanjana Basu
University of Illinois at Chicago
nbasu4@uic.edu

Abstract

We revisit the question of delegation vs. synchronized access to shared memory, and show through analysis and demonstration that delegation can be much faster than locking under a range of common circumstances. Starting from first principles, we propose fast, fly-weight delegation (*ffwd*). The highly optimized design of *ffwd* allows it to significantly outperform prior work on delegation, while retaining the scalability advantage.

In experiments with 6 benchmark applications, and 6 shared data structures, running on four different multi-socket systems with up to 128 hardware threads, we compare *ffwd* to a selection of lock, combining, lock-free, software transactional memory and delegation designs. Overall, we find that *ffwd* often offers a simple and highly competitive alternative to existing work. By definition, the performance of a fully delegated data structure is limited by the single-thread throughput of said data structure. However, due to cache effects, many data structures offer their best performance when confined to a single thread. With an efficient delegation mechanism, we approach this single-threaded performance in a multi-threaded setting. In application-level benchmarks, we see improvements up to 100% over the next best solution tested (RCL), and multiple micro-benchmarks show improvements in the 5–10× range.

1. Introduction

As processor manufacturers increasingly depend on multicore designs to improve system performance and energy efficiency, the importance of safe and efficient access to shared variables and data structures continues to grow. The most common solution is mutual exclusion, or locking, where atomic instructions are used to acquire a lock before entering a critical section where a data structure in shared memory is accessed. The design of efficient and scalable locks has a long history, and remains an active area of research today [9, 18, 19, 23, 27, 48, 54, 55, 70, 78, 84].

By locking the entire shared data structure, a technique also known as coarse-grained locking, performance is upper-bounded by the single-thread throughput of the data structure. Several approaches aim to provide concurrent access to the data structure by multiple threads, including fine-grained locking approaches [43], lock-free data structures [11, 37, 38, 41, 42, 44, 50, 62, 63, 69, 82, 83, 85, 86, 89–91, 95] and software transactional memory. Here, the fine-grained locking and lock-free approaches require (sometimes extensive) modifications to the specific data structure, while software transactional memory [28, 30, 33, 35, 45, 56, 64, 72, 75, 81, 92, 93] provides transactional semantics that automatically permit concurrent but independent operations on any shared data structure.

With *delegation*, one thread (the server) acts on behalf of multiple client threads. Combining [26, 31, 32, 40, 66, 79, 94] is a form of delegation, where threads temporarily take on the role of the server and *combine* their own critical section with those of one or more other threads waiting for the lock currently held, increasing efficiency. Conventional delegation methods [17, 25, 49, 53, 68, 84] use one or more dedicated server threads. Here, the server has exclusive access to the data structure, and interacts with clients via a messaging interface. Dedicating a hardware thread to a delegation server is a significant sacrifice. However, this choice also enables more efficient implementation than combining.

An efficient delegation system aims to provide single-threaded data structure performance, in a multi-threaded setting. Thus, the most attractive data structures to delegate are those that perform best on a single thread. Conversely, highly parallel data structures generally do not benefit from delegation. Our system: fast, fly-weight delegation (*ffwd*, pronounced “fast-forward”), is a stripped-down implementation of delegation that is highly optimized for low latency and high throughput. *ffwd* offers up to 10× the throughput of the state-of-the-art in delegation, RCL [53], in micro-benchmarks, or up to 100% in application-level benchmarks. With respect to locking and other methods, *ffwd* is often able to improve performance by 10× or more, for data structures suitable to delegation.

ffwd achieves this performance by effectively hiding the high latency of the interconnect link between the server and

SOSP '17, October 28, 2017, Shanghai, China

ISBN 978-1-4503-5085-3/17/10. . . 15.00

DOI: https://doi.org/10.1145/3132747.3132771

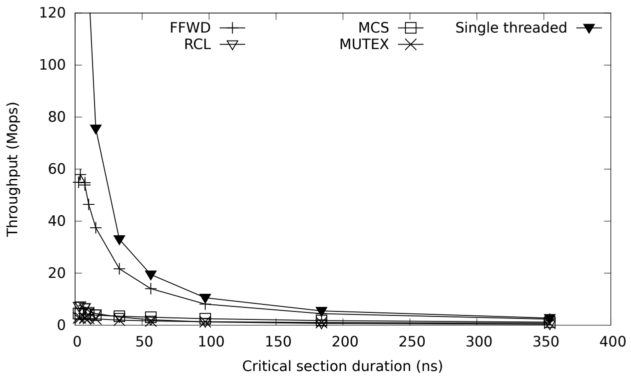


Figure 1: Throughput as a function of critical section duration, for single-threaded execution, delegation, and locking (higher is better).

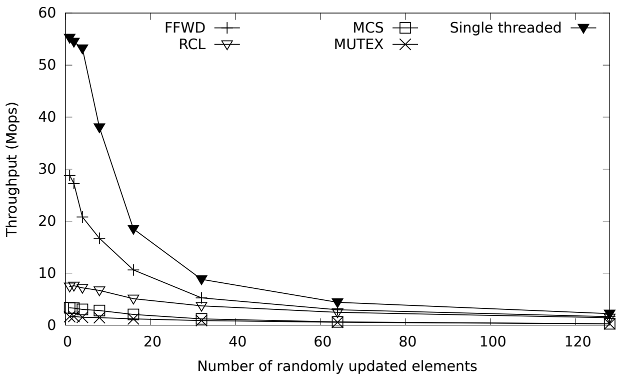


Figure 2: Throughput as a function of memory accesses, for single-threaded execution, delegation, and locking (higher is better).

client threads. This is done through a combination of instruction level parallelism and carefully managing memory accesses. Among the techniques used, we eschew the use of atomic instructions in the server to allow instruction reordering, pack requests and responses into cache line pairs based on client proximity, buffer responses to minimize cache coherence traffic, and more. The primary contributions of this paper are as follows:

- The design and implementation of *ffwd*, to our knowledge the fastest delegation system to date. At approximately 40 cycles of server-side overhead per request, *ffwd*-delegated data structures closely approximate single-thread performance, in a multi-threaded setting.
- A low-level analysis of the performance bounds on locking vs. delegation, from an architectural perspective, setting the stage for further improvements.
- A publicly available implementation of *ffwd* [2], including a general purpose API, and a set of benchmark programs ported to delegation, for the community to reproduce and build upon our results.

Below, we analyze the factors that limit delegation performance in §2, and contrast this to a similar analysis of coarse-grained locking. We then describe the design of *ffwd* in detail in §3, before a comparative performance evaluation in §4. We discuss how to port existing programs to *ffwd* in §5, followed by a literature review in §6 and conclusions §7.

2. Lock and Delegation Performance Bounds

Below, we discuss the performance limits of locking and delegation. Succinctly, single-lock performance is constrained by the interconnect latency. Delegation, meanwhile, is primarily limited by server processing capacity.

Figure 1, shows the number of critical sections executed per second (Mops) with varying critical section lengths (in this case, iterations of an empty for-loop), for several competing methods. Here, the *single-threaded* benchmark program consists of an outer loop that repeatedly calls a function containing the empty for-loop. This represents an upper bound on delegation performance, which is as high as 320 million operations per second (Mops) for a one-iteration critical section. The MUTEX and MCS programs use 128 hardware threads, contending for one shared (pthread mutex/MCS) lock and executing one critical section before releasing the lock. The FFWD and RCL programs are delegation-based. FFWD uses 120 client threads, and 1 server thread[1]. Clients delegate the inner loop to the server.

Notably, delegation (*ffwd*) dramatically outperforms locking for short critical sections, but falls short of the single-threaded program's performance. While single-threaded throughput upper-bounds delegation throughput, several factors restrict the performance of a delegation system, including interconnect bandwidth, interconnect latency, store buffer capacity, and message demarshalling overhead, each discussed separately below.

Consider an idealized system. Assuming no back-to-back acquisitions, the maximum single-lock throughput is $t_{lock} = \frac{1}{l+c_{lock}}$, where l is the mean one-way interconnect latency, and c_{lock} is the mean duration of the critical section. When ($c_{lock} \rightarrow 0$), lock throughput is dominated by bus latency which, on our systems, is approx. 200 ns between sockets, or approx. 80 ns between cores on a single socket. This corresponds to a single-lock throughput of 5 Mops and 12.5 Mops respectively. Only by increasing the number of locks can a locking program go beyond this fundamental limit.

While locking explicitly serializes both the coordination and the execution of the critical section, delegation serializes the execution of delegated functions, but leaves the exchange of requests and responses up to the implementation. The interconnect bandwidth on Intel/AMD architectures tends to be very high relative to the bus latency, as high as 25

[1] These numbers are explained in §3.

Gbyte/s, or 390 million cache lines per second on a single link. Thus, an implementation that can harness part of this bandwidth could achieve significant performance gains. We now discuss several constraints on delegation performance in more detail.

Interconnect Bandwidth In our evaluation systems, the interconnect bandwidth on a single link is 150–390 million cache lines per second. For a lower bound, consider a single link of our slowest interconnect, and one cache line per request, in each direction. Conservatively (not counting in-socket bandwidth), the bandwidth bound is then 75 Mops per link. Our systems have two links per socket, for a total of 150 Mops. More efficient design, faster links, and using multiple servers, can together increase this number considerably.

Interconnect Latency Delegation requires a round-trip on the interconnect bus: one way for the request, and one for the response. Thus, the maximum delegation *per-client* throughput is $\frac{1}{2l}$, or 2.5 Mops for inter-socket communication.

Interconnect Parallelism: Store Buffers To go beyond the single-client throughput, multiple request/response pairs need to traverse the high-latency interconnect in parallel. Clients naturally issue requests in parallel, but the single server is more constrained. Here, a number of store buffers exist that hold (and order) cache-coherent stores until the relevant cache line is locally available for writing. The store buffers on our Broadwell CPUs support 42 concurrent in-flight stores. Thus, assuming one store per response, and if all in-flight stores are used for responses, the throughput limit is $42 \times 2.5 = 105$ Mops.

Demarshalling Overhead Finally, the server processing throughput is a function of the critical section length when run on the delegation server, c_{del}, and the demarshalling overhead, o_{del}. Each request requires at minimum: loading the request, reading parameters into registers, calling the delegated function, and writing a response. Thus, the maximum processing throughput is $\frac{1}{o_{del}+c_{del}}$. It is unknown how efficient a delegation server can be. However, our current implementation achieves 55 Mops on a 2.2 GHz CPU, or 40 cycles per request, whereas locking requires more than 450 cycles per request.

In summary, with locking, throughput is limited to 5 Mops per lock, or 12.5 Mops when running on a single socket. With delegation, performance is limited primarily by server processing capacity, and the number of processor cycles spent on each delegated function.

2.1 Performance Bounds with Larger Critical Sections

Naturally, longer critical sections are also common. For longer critical sections, the delegation advantage in terms of parallel communication fades, as seen in Figure 1. However, not captured in this figure is the potential memory locality advantage of the delegation approach.

Figure 2 shows the results of a similar experiment investigating the effect of memory accesses. Here, the critical section updates a variable number of randomly selected elements within a 1 Mb statically allocated array. Here, the lack of contention for the elements, and the effective use of the LLC cache, allows *ffwd* to substantially outperform the other approaches throughout the range.

2.2 When to use *ffwd*

Based on the above analysis, we expect delegation to consistently outperform coarse-grained locking on multi-core systems, as long as the number of hardware threads is sufficiently large that setting one thread aside for the server is not unreasonable.

Fine-grained locking can outperform delegation under the right circumstances. In particular, long critical sections are better suited to fine grained locking, due to the added parallelism provided. For short critical sections, delegation is likely the better choice due to the per-lock throughput bound (≈5 Mops on our systems). Unless, that is, the data structure can be partitioned enough to support a large number of independent locks (e.g. a hash table).

Considerably higher performance can sometimes be obtained with specialized data structure designs. In our evaluation, we highlight several such data structures. It is important to note, however, that these data structures require sophisticated engineering effort while delegation can often provide a high-performance alternative with minimal code changes.

The current state of the art in delegation, RCL [53], places a heavy emphasis on supporting the re-engineering of legacy applications to delegation. Beyond the automatic profiling and code rewriting tools described, the RCL delegation protocol itself was designed with re-engineering as first priority, and performance second. This is evident in several details of the design: RCL supports locking and blocking system calls in delegated functions. The name, Remote Core Locking, is embodied in the fact that after delegating a critical section, the server still acquires the lock, to preserve correctness should other threads acquire the lock in another part of the code. Also, the use of a request *context* is convenient for automatic rewriting, but results in lower performance, as the server must first read the request, then read the context based on the pointer passed in.

ffwd's design focuses on performance rather than re-engineering of legacy applications. As a result, we are able to achieve ≈10× speedup over RCL, widening the range of programs where delegation outperforms other methods. Below, we describe the design of *ffwd* in more detail.

3. fast, fly-weight delegation (*ffwd*)

ffwd provides an API for delegating the execution of a normal C function to a remote server: the client sends a request to the server, specifying a function and a set of parame-

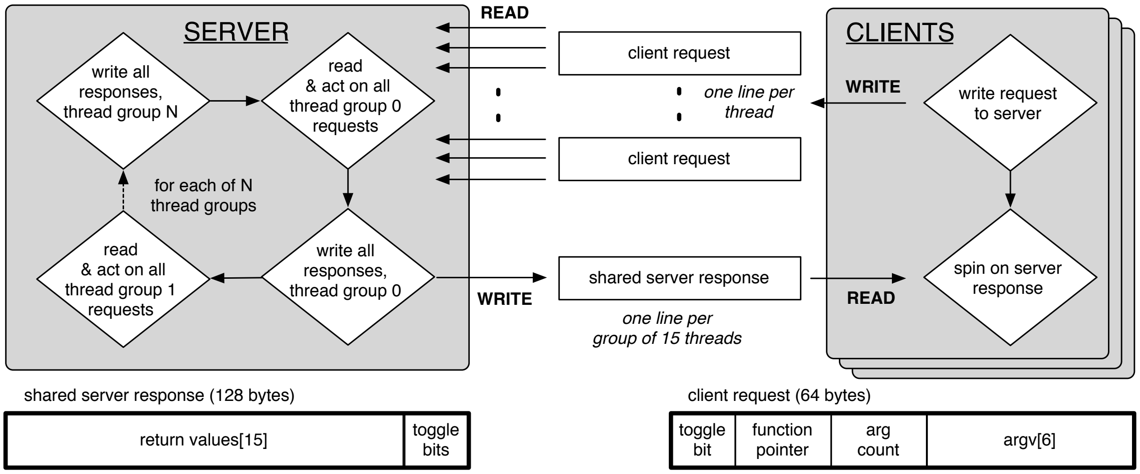

SERVER
write all responses, thread group N
read & act on all thread group 0 requests
for each of N thread groups
read & act on all thread group 1 requests
write all responses, thread group 0
READ
WRITE
client request
one line per thread
client request
shared server response
one line per group of 15 threads
CLIENTS
WRITE
write request to server
READ
spin on server response
shared server response (128 bytes)
return values[15]
toggle bits
client request (64 bytes)
toggle bit
function pointer
arg count
argv[6]

Figure 3: High-level design of fast,fly-weight delegation (*ffwd*). Request and response data structures are designed to minimize cache coherence traffic and latency.

ter values. It then awaits the function's return value in the server's response.

Figure 3 illustrates the overall operation of *ffwd* at the message-passing level. For illustration purposes, we describe the design in terms of system constants taken from our evaluation systems (64-byte cache lines, up to 32 threads per socket). However, the design principles behind *ffwd* generalize well to other constants.

Each client core maintains a dedicated 128-byte request cache line pair. This is exclusively written to by the hardware threads of that core, and read only by the server. After writing to its request line, each client thread spins on its dedicated response slot within its 128-byte response line pair. The response line pair is exclusively written to by the server, and shared (for reading) by multiple cores of a given socket.

The server processes requests in a round-robin, socket-batched fashion, polling all request lines and handling any new requests from one socket before proceeding to the next. It buffers individual return values locally until processing for the current response group has finished, then writes all responses to that group's response line pair.

Each request contains a toggle bit, a function pointer, an argument count, and up to 6 arguments to the function. The response line pair is shared between up to 15 clients on a single socket, containing per-client toggle bits, and 8-byte return values. Together, the requests and the shared response form up to 15 individual channels between the clients on a single socket, and one server. The toggle bits indicate the state of each individual request/response channel. If the request and response toggle bits that correspond to a given client differ, a new request is pending. If they are equal, the response is ready. To process a request, the server loads the arguments provided into the appropriate registers, and calls the specified function.

3.1 Design Motivation

Several key aspects of *ffwd* are listed below. This is followed by a discussion of and motivation for each design choice.

- Allocations are multiples of 128-byte aligned line pairs.
- Though multiple cores may read a given line pair, only one core ever writes to each one.
- Several clients on a socket share a response line pair.
- The server buffers responses locally, then copies all responses to the appropriate response line pair in one uninterrupted series of writes.
- Overhead for serving a request is minimal: load parameters into registers, call the supplied function pointer, then copy the return value to the local response buffer.
- The server does not acquire any locks prior to executing the delegated function.

Without loss of generality, consider a slice of the system consisting of a single server, and up to 15 client hardware threads on a single remote socket. At a high level, the key to high messaging performance is minimizing the amount of cache coherence communication on the interconnect.

Eliminating false sharing A first step to this is to eliminate any source of false sharing. On Intel's Xeon family of architectures, cache lines are 64 bytes, but the L2 cache of each core includes a spatial (pair) prefetcher, which treats memory as consisting of 128-byte line-pairs, and automatically prefetches the "other" 64-byte line when one of the lines is otherwise retrieved. Thus, independent, false-sharing-free memory access (aside from cache associativity effects, which do not play a role here) is only available in 128-byte granularity on Xeon.

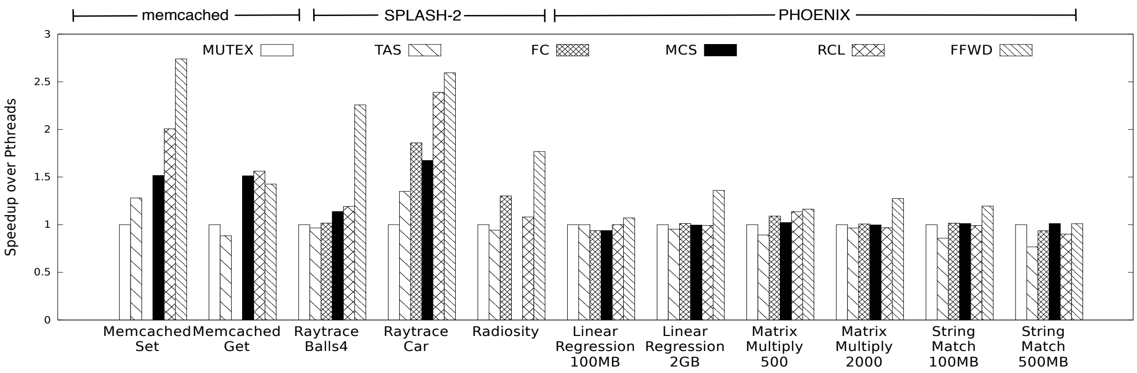


Figure 4: Application benchmark speedup, for best performing number of threads, relative to the best performance of POSIX on Broadwell. *We were not able to run the RCL version of memcached with all thread counts, due to persistent failures.

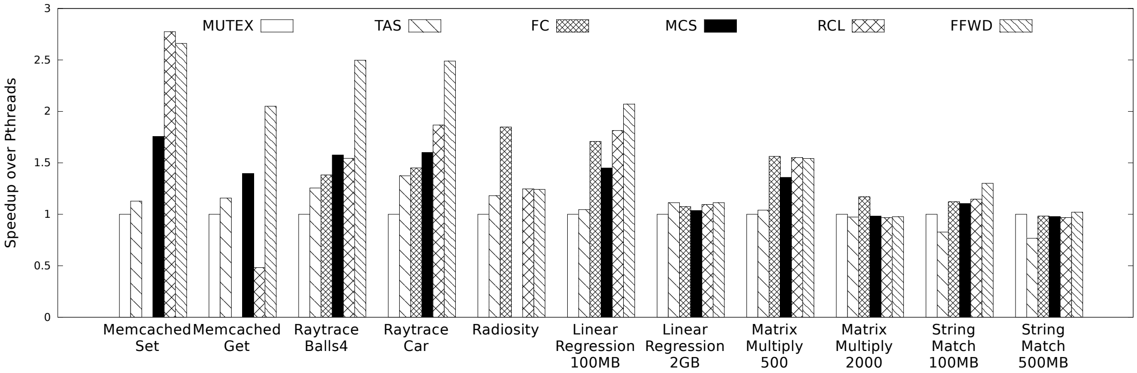


Figure 5: Application benchmark speedup, for best performing number of threads, relative to the best performance of POSIX for Abu Dhabi.

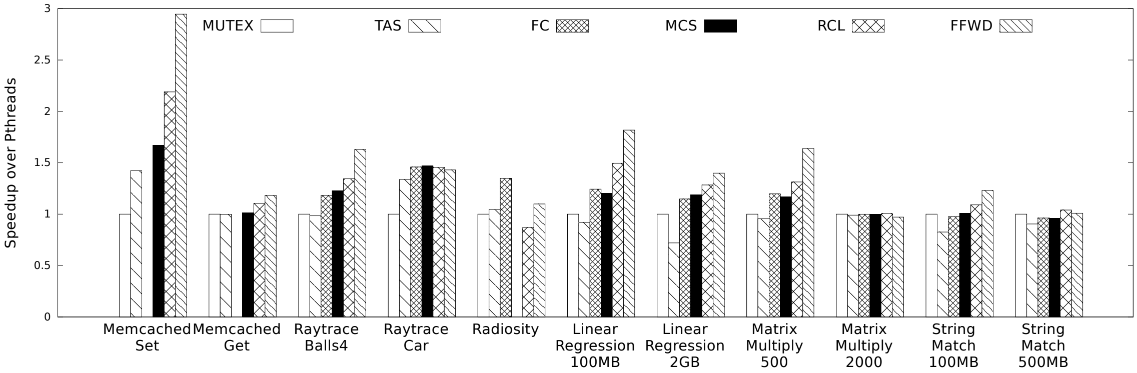


Figure 6: Application benchmark speedup, for best performing number of threads, relative to the best performance of POSIX for Sandy Bridge-EP.

Independent, per-core request lines We allocate one 128-byte line pair per core, which is split equally between the two hardware threads supported by our three Xeon machines. Allocating a full 128-byte pair eliminates any write contention on the requests: the only contention is between the writing client, and the reading server. The server's read request transitions the cache line to the shared (S) state and copies the contents over the interconnect, while the client's subsequent write invalidates the server's copy (without data transfer), and transitions the line to the modified (M) state.

Buffered, shared response lines By contrast, we allocate 128-byte response line pairs to be shared by a group of up to 15 hardware threads of a common socket, containing toggle bits and 8-byte return values. Here, the server's write invalidates all copies of the response line pair at the corresponding socket. While this invalidation is in process, the server's buffered responses are written to the local core's store buffer, virtually guaranteeing that the entire 128-byte pair can be written to without incurring further coherence traffic. The toggle bits are copied last. The first subsequent read by a client transitions the cache line pair to the S state and copies the data. Subsequent reads by other clients on the same socket are served by their local last-level cache. In short, every round of service, serving up to 15 clients on one socket, incurs at most 17 cache line data transfers and 17 corresponding cache line invalidations over the interconnect.

NUMA-aware allocation of request and response lines Request lines are allocated on the NUMA node of the clients, and response lines on the NUMA node of the server, which we found to provide a substantial performance advantage. While measuring cache coherence traffic (beyond cache misses) directly is outside the scope of this paper, we hypothesize that this NUMA allocation strategy avoids extra steps in the invalidation process, as the relevant cache home agent always shares a socket with the writer.

Minimal demarshalling overhead Beyond message passing, the processing of requests on the server must incur minimal overhead vs. the client performing the work, or server processing would quickly become a performance bottleneck. To this end, our server starts processing a request by reading the specified number of parameters from the request into the parameter passing registers. It then calls the specified function, and finally buffers the eventual return value for subsequent combined transmission.

No atomic instructions Finally, the server does not acquire any lock during its operation. In x86 processors, atomic operations are not reordered with respect to other loads and stores, degrading performance. For example, in our fetch-and-add micro benchmark holding a local, uncontended lock around each delegated function reduced throughput from 55 Mops to 26 Mops. Not acquiring locks does impact applicability, as in most cases the server can only operate on data structures local to the server thread. However, due to the server processing bottleneck present in all delegation systems, performance considerations anyway suggest that servers be denied access to any data structures shared with clients.

3.2 *ffwd* Delegation API

Having the server call a passed-in pointer combines our two goals of high performance and ease of use: the function specified in a request can be any non-blocking C function, with up to 6 8-byte parameters. To facilitate ease of use for application developers, *ffwd* provides a small API:

FFWD_Server_Init() Starts a server thread, allocates and initializes the request and response lines.

FFWD_Delegate(s, f, retvar, argc, args...) This macro delegates function f to server s, with specified arguments. Stores return value in retvar.

Using the *ffwd* API, clients may delegate the execution of a function to the server using a single line of code, specifying the function, desired parameters and return value variable.

4. Experimental Evaluation

Below, we compare *ffwd* to several competing approaches, across 6 different data structures, 6 application benchmarks, 4 systems (Table 1) using up to 128 hardware threads.

Overall, the experimental results reflect the conclusions of the analysis in §2: delegation excels when critical sections are brief, and contention is high. Under these circumstances, contention and communication dominate and degrade the performance of every type of lock-based synchronization, whereas *ffwd* is at peak performance.

As we stray from these ideal conditions, exploring smaller thread counts, greater inherent parallelism in the program, and either long critical sections or long intervals between them, *ffwd* remains highly competitive up to a point, after which the advantages of *ffwd* diminish, and its server-side processing bottleneck becomes more apparent.

4.1 Experimental Design

We report evaluation results from experiments run on the four machines described in Table 1. However, due to space constraints, most plots are from the 64-core Broadwell system—except where noted, the trend was similar across all four systems. For full evaluation results on all four systems, please refer to our technical report [73].

To accurately characterize the performance of *ffwd*, we perform both application level and micro-benchmark experiments comparing *ffwd* to a wide variety of competing approaches: conventional locks including test-and-set spinlock (TAS), test-and-test-and-set spinlock (TTAS), posix mutex lock (MUTEX), the MCS lock (MCS) [61], the CLH lock (CLH) [23], the conventional ticket lock (TICKET) [61] and its hierarchical counterpart (HTICKET) [27], combining approaches including flat combining (FC) [40], and delegation (RCL) [53]. For some specific benchmarks, we included several implementations specific to that benchmark, including lock-free, software transactional memory, and combining approaches.

Except where noted, we use a number of threads equal to the number of supported hardware threads on the machine in question. One significant exception to this is the FFWDx2 method, which over-subscribes each hardware thread with two user/green threads. These user threads yield the CPU immediately after sending a request. FFWDx2 is only evaluated in our micro-benchmarks. In the case of *ffwd*, we dedicate one core per socket as a delegation server in our exper-

Machine	RAM Latency (ns) local	RAM Latency (ns) remote	LLC latency (ns) local	LLC latency (ns) remote	Interconnect Bandwidth (GB/s)
4×16-core Xeon E5-4660, Broadwell, 2.2 GHz	81	238	63	135	77
4×8-core Xeon E5-4620, Sandy Bridge-EP, 2.2 GHz	71	249	70	177	57
4×8-core Xeon E7-4820, Westmere-EX, 2.0 GHz	168	235	98	185	47
4×8-core AMD Opteron 6378, Abu Dhabi, 2.4 GHz	×	×	92	163	51

Table 1: Specifications and preliminary performance measurements using MLC [1] for Intel, and ccbench [25] for AMD on our experimental platforms. ccbench did not report RAM latency. In the table, **local** denotes communication between cores within a socket, and **remote** represents communication with cores on other sockets.

iments even when it is unused, leaving up to 30 threads per socket to run client/application code on Broadwell machine. Though dedicating one server core per socket is not a design requirement of *ffwd*, it keeps the experimental design simple and flexible. The incremental performance gain from using these last few cores is negligible for *ffwd*.

In all experiments, benchmark threads are pinned to hardware threads in a pre-determined order. On Xeons, we are first filling one socket with one thread per available core, then filling subsequent sockets in the same fashion, and finally revisiting each socket in the same order to populate the second thread supported by each core. For FFWDx2, each hardware thread populated immediately results in two user threads. We follow a similar order on the AMD system, as pairs of Opteron cores share some resources. The first pass populates one core in each pair: second pass populates the second core. Finally, the data points in the subsequent plots represents the mean of at least 10 independent runs, except for memcached, which is a long-running benchmark, and linked-list/hash-table, which were configured to run for 15 seconds per data point.

All programs were compiled with gcc 5.4.0, using optimization level 3, running under 64-bit Ubuntu 16.04. These experiments were run with the glibc standard allocator (ptmalloc). We also experimented with Hoard [3], slab[7], and jemalloc [4], and saw no significant improvements for these workloads (figures omitted for brevity). The concurrent operations in these workloads are not allocation-intensive, and generally make small, fixed-sized allocations. Also, we found that ptmalloc creates approximately arena per thread, largely avoiding contention on area locks.

4.2 Application level benchmarks

Our application level benchmarks were taken from the SPLASH-2 [8], Phoenix v2.0.0 [71] benchmarks, and Memcached v1.4.6 [5] / Memslap v1.0.2 [6] using the same subset of benchmarks as the RCL paper [53]. In turn, [53] selected these benchmarks based on the fraction of time spent in critical sections. Several concerns have been raised about the Phoenix benchmark suite, including: (a) a false-sharing problem was discovered in the linear regression program [52, 65], though [65] indicates that gcc with optimization level -O3 eliminates this problem, (b) the matrix multiply program may not be competitive with the state of the art in matrix multiplication, and (c) the string-match program may not be a representative workload. Thus, a comparison between delegation and locking based on the Phoenix suite should be taken with a grain of salt. Nevertheless, using the same set of benchmarks enables a direct comparison to RCL. We include instances of each of the three benchmarks with two input sizes. We did not include the BerkeleyDB benchmark due to time constraints. Figure 4 summarizes our application-level results on the Broadwell machine, showing speedup relative to standard Posix mutex locks. We were unable to run the radiosity benchmark with MCS successfully, and were unable to implement Memcached with flat combining case due to time constraints. Overall, we find that *ffwd* further improves upon the delegation gains already demonstrated by RCL.

While Figure 4 shows the speedup vs. pthreads at the ideal thread count for the respective method, Figures 7–8 show how performance varies with the number of threads, telling a very different story.

For both Raytrace-Car and Memcached-Set, *ffwd* performance continues to improve slightly after 16 threads, while all other approaches scale negatively. The spinlock (TTAS) shows a characteristic congestion collapse, whereas the other lock types, and RCL, exhibit a more graceful decline. Several lines in both benchmarks stop abruptly, due to persistent “server disconnection” errors at high thread counts that we were unable to resolve. We traced this back to a connection loss between the memslap [6] client and the memcached server, but could not isolate the fault further.

We find that for all benchmark applications evaluated, on all platforms tested, *ffwd* matches or outperforms all lock types (including combining), as well as RCL. Overall, the performance gains can be attributed to two main sources: eliminating contention for highly contended data structures, and improved memory locality, for all shared data. For the comparison with RCL, the cause of the performance difference is less obvious, as RCL and *ffwd* follow the same design principle. However, the micro-benchmark results below suggest a simple explanation: while *ffwd* and RCL do the same job, *ffwd* does it more efficiently, as described in §3.

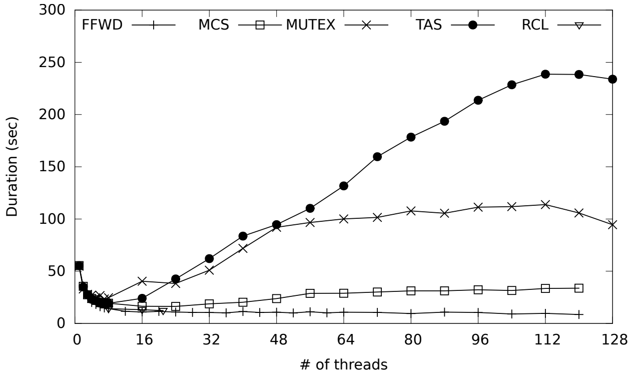
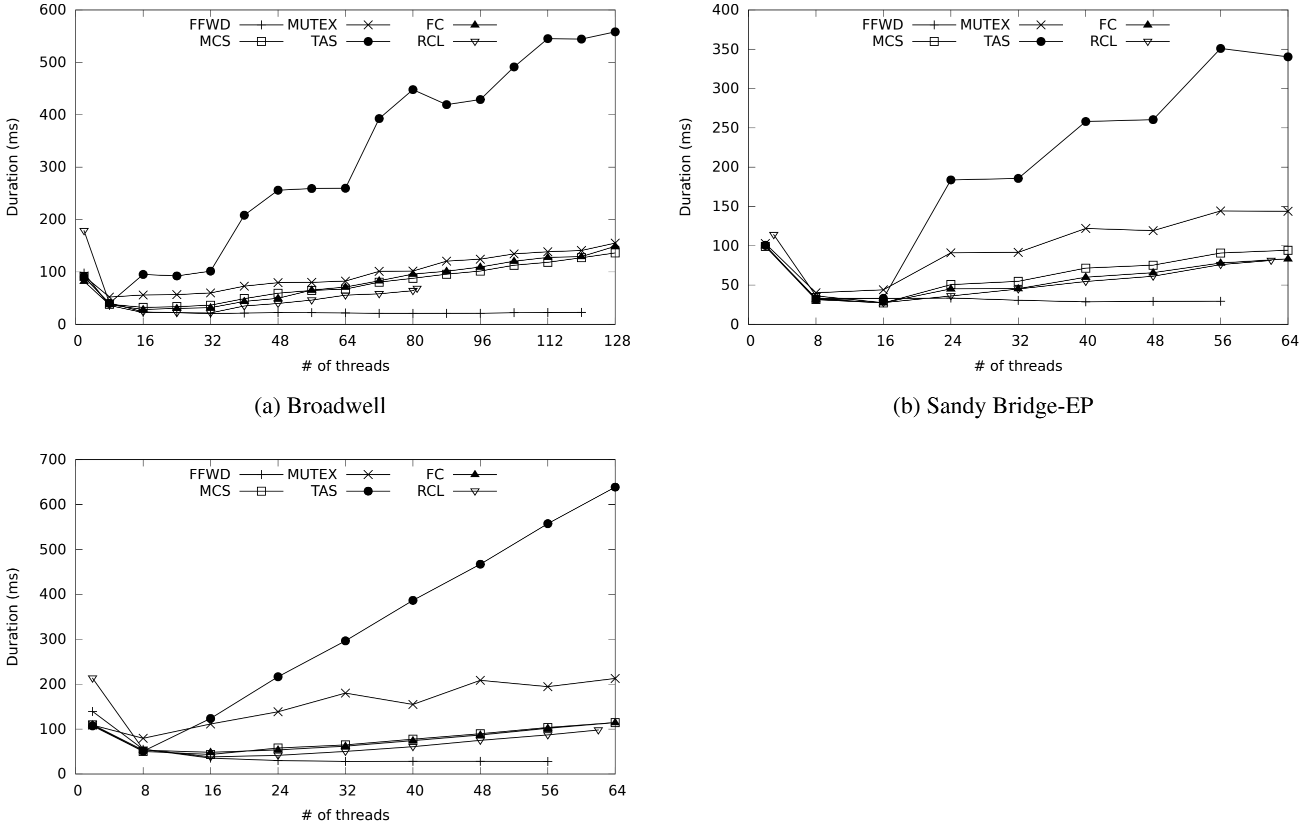


Figure 7: Memcached-Set benchmark runtime for varying number of threads on Broadwell (lower is better).



(a) Broadwell

(b) Sandy Bridge-EP

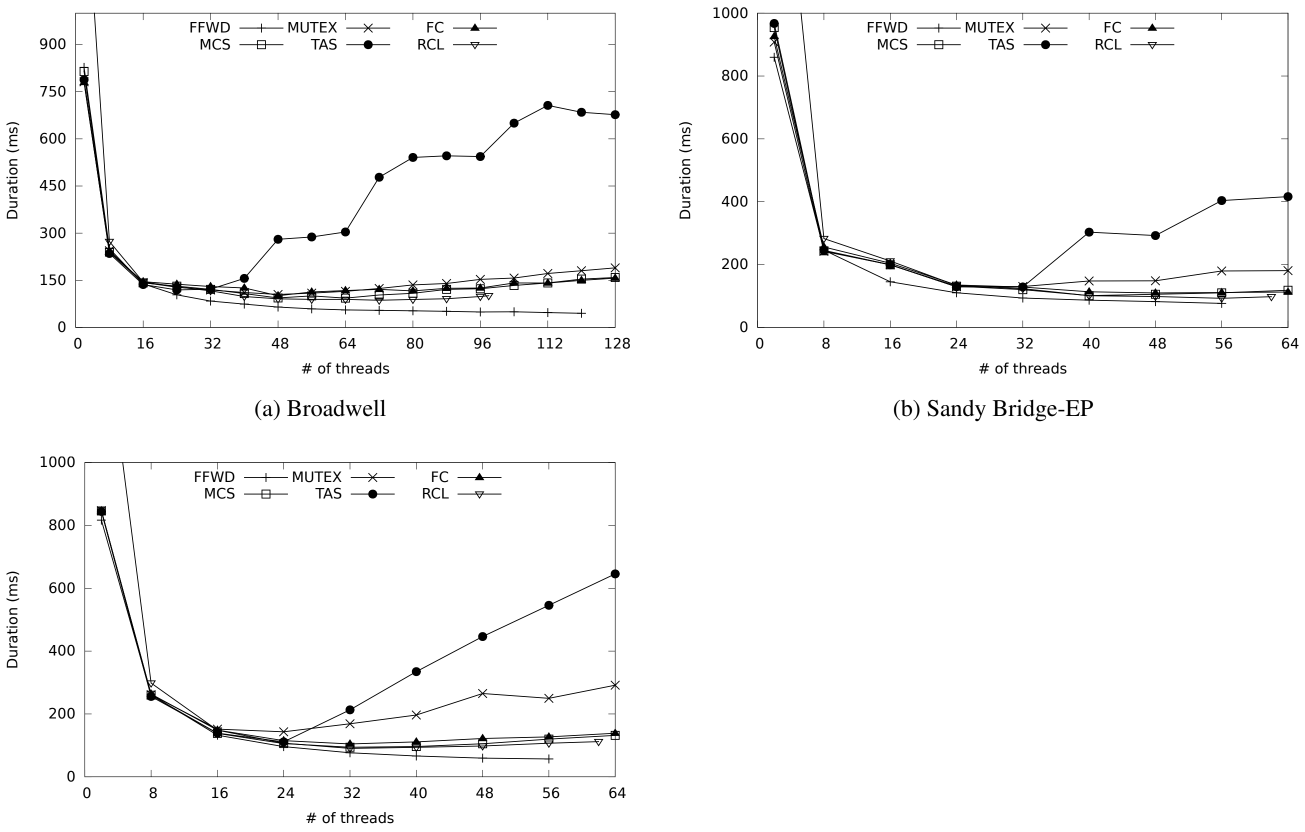
(c) Abu Dhabi

Figure 8: Splash2 raytrace-car application run time for varying number of threads (lower is better).

4.3 Micro-benchmarks

To gain a better understanding of the expected performance of *ffwd*, we include several micro-benchmark programs in our evaluation, described in more detail below: fetch-and-add (§4.3.1), stack and queue (§4.3.2), linked list (§4.3.3), search tree (§4.3.4), and hash table (§4.3.5).

To avoid one thread acquiring a lock multiple times consecutively, we introduce a small delay between increments, outside any critical section or delegated function. Figure 17 shows the relationship between this delay, the percentage of back-to-back lock acquisitions (labeled MUTEX % B2B ACQ) for MUTEX, and lock throughput for several lock types, using 128 threads and a single lock. Without a delay, back-to-back acquisition is very common, which greatly distorts the performance of some of the simpler locks. This distortion is even more severe for atomic increment.



(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 9: Splash2 raytrace-balls application run time for varying number of threads (lower is better).

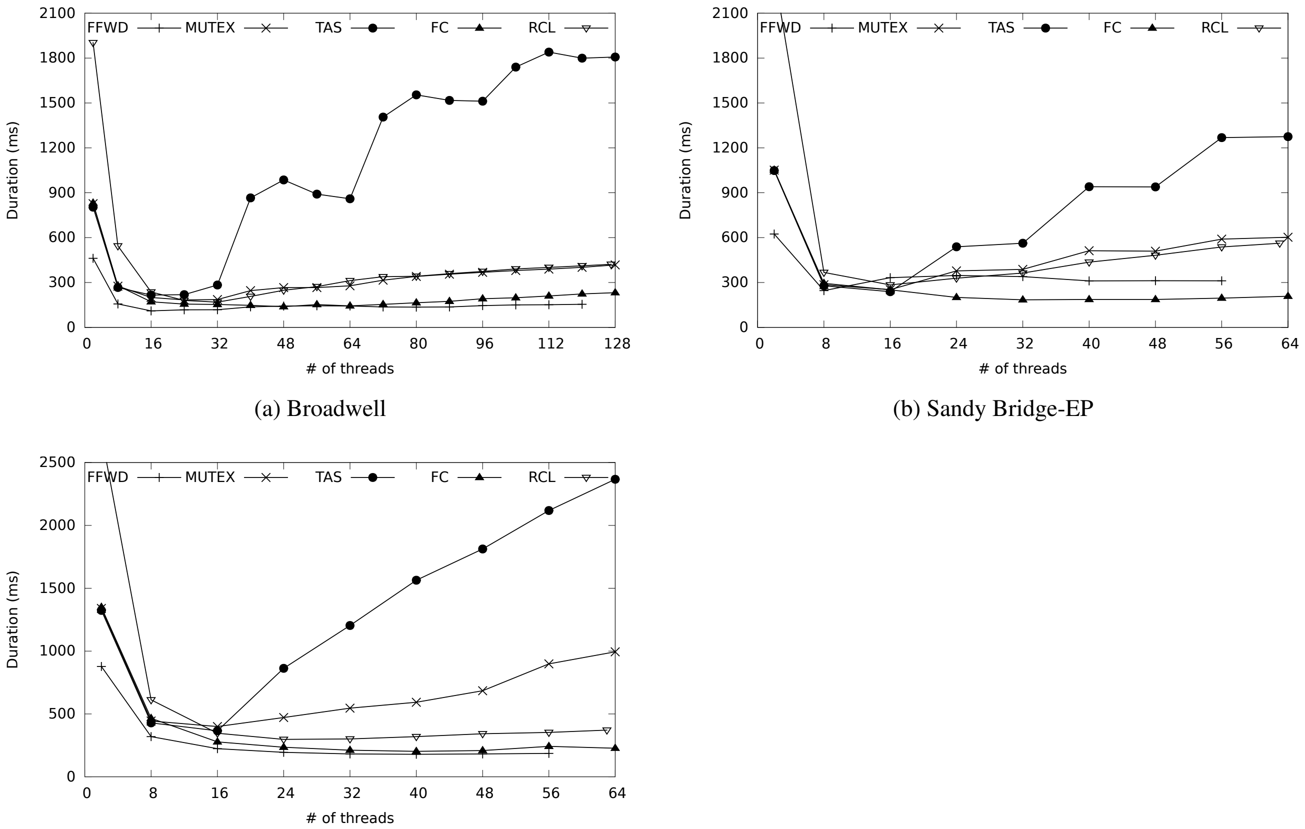
Thus, to ensure representative results across all methods, we use a delay loop of 25 `PAUSE` instructions between critical sections to prevent such back-to-back acquisition, translating to ≈500 cycles of delay on our Xeon machines, or about one round-trip on the interconnect bus. In this test, threads on average wait for the critical sections of 127 other threads to complete before successfully acquiring the lock. Thus, this small added delay has no significant impact on performance in the absence of back-to-back acquisitions. Nevertheless, for fairness, the same 25 `PAUSE` delay is imposed on all methods evaluated.

4.3.1 fetch-and-add

This benchmark demonstrates *ffwd* performance and scalability for very short critical sections, for the single-variable case, and investigates the performance trade-off between memory locality and parallelism, in the multi-variable case.

In the fetch-and-add micro-benchmark, there is a variable number of integer counters. In the case of locking and combining, each counter occupies its own cache line pair of 128-bytes, which it shares with an associated lock to minimize cache misses. In the case of *ffwd*, the counters reside in a standard C array, without any particular cache alignment. The work consists of selecting a global variable at random, and incrementing it, for 10 seconds. Before starting, all threads wait at a barrier, to ensure the threads run concurrently. For locking and combining, each thread selects a variable, acquires the associated lock, increments the variable, then releases the lock. For *ffwd*, each client thread instead selects a variable, then delegates an increment function (with the variable index as a parameter) to the variable's pre-assigned server and awaits the server's response. Since fetch-and-add also has dedicated hardware support on the x86 architecture in the form the `LOCK INC` atomic instruction we also report the performance using this (ATOMIC) instead of a lock and a non-atomic increment.

Figure 18 compares the performance of *ffwd* and its alternatives on this benchmark program on our Broadwell machine using all 128 hardware threads, as we vary the number of global variables (and in the case of locking and combining, associated locks). *ffwd* and its over-subscribed variant show a dramatic advantage over the alternatives until the number of variables approaches the number of hardware threads in use. The over-subscribed FFWDx2 outperforms standard *ffwd* as the number of variables grows from 1–4. With four variables, *ffwd* is able to take advantage of the



(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 10: Splash2 radiosity application run time for varying number of threads (lower is better).

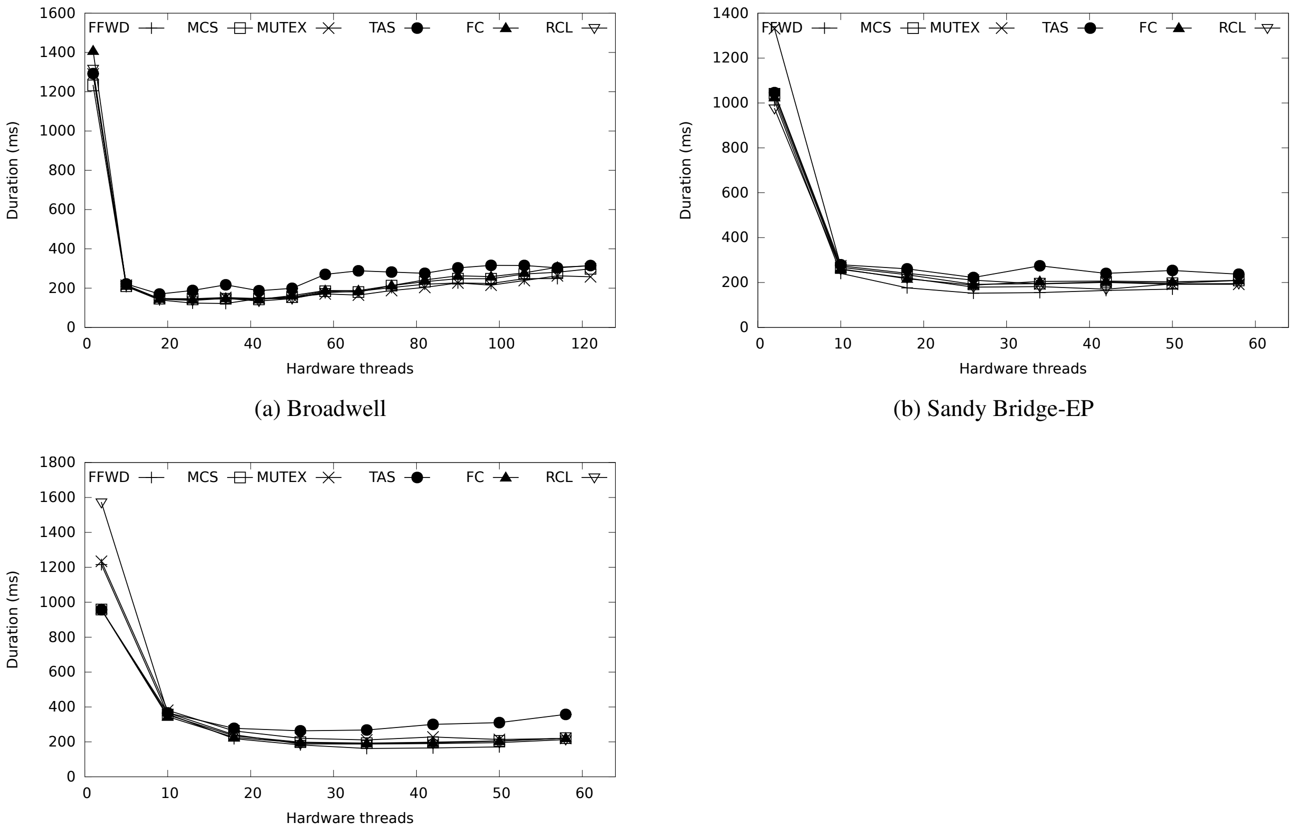
four available servers, eliminating the server processing bottleneck. FFWDx2, meanwhile, is also able to circumvent the interconnect latency limit by increasing parallelism, further increasing throughput.

With more than 128 variables, some locking methods start to outperform *ffwd*. This is unavoidable: for a sufficiently parallel program, the centralized model of delegation cannot compete with locking. Notably, however, neither flat combining nor RCL perform well on this micro-benchmark. Figure 19 offers a hint at the problem, showing total system throughput for a single variable, while varying the number of threads, on all four systems. While RCL is relatively competitive on the Abu Dhabi system, it does not scale well beyond a single socket on the Intel systems. It is possible that either design decisions or parameter settings were chosen to better reflect the AMD architecture.

Though the final performance is a function of a large number of factors, cache misses can often have a large impact. In this benchmark, *ffwd* incurred an average of 0.72 cache misses per operation, while RCL saw 3.07 cache misses per operation. Based on the RCL design, there should be one miss per request, one dependent miss to transfer the context, and one miss for the response. Meanwhile *ffwd* appears to benefit from cache pre-fetching to get below the expected $1 + \frac{1}{15} = 1.06$ misses per operation.

Figure 19 illustrates the similarity in trend between the systems tested, though the absolute performance varies. Locking does relatively well on a single lock when the number of threads is small, but quickly drops off with thread count. Particularly severe is the step between single-socket and multi-socket performance, at 16–32 threads in Figure 19(a), and 8–16 threads in the remaining sub-figures, as the added latency of the interconnect kicks in.

For *ffwd*, we note a steady increase in performance as the thread count grows. Since the delegated function is very brief, *ffwd* is latency-bound in most of this experiment. Adding threads effectively increases the amount of pipelining possible on the long-latency but high-bandwidth CPU interconnect. It is interesting to note that except when operating on a single socket, *ffwd* significantly outperforms even the atomic increment instruction, which was built to do exactly what this benchmark does. This is a true testament to the high cost of sequential communication, which is unavoidable for both atomic instructions and lock-based methods. The FFWDx2 program showcases two benefits of the added threads: first, on the Broadwell system, being



(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 11: Phoenix string match benchmark runtime for varying number of threads (lower is better).

able to run 32 concurrent requests on a single socket rather than 16 almost doubles throughput, as the system is latency constrained. Second, the Abu Dhabi system appears to be latency constrained for all thread counts, which FFWDx2 effectively circumvents. While this latency difference is not apparent in Table 1, delegation is considerably more complex than the measurements performed for that table.

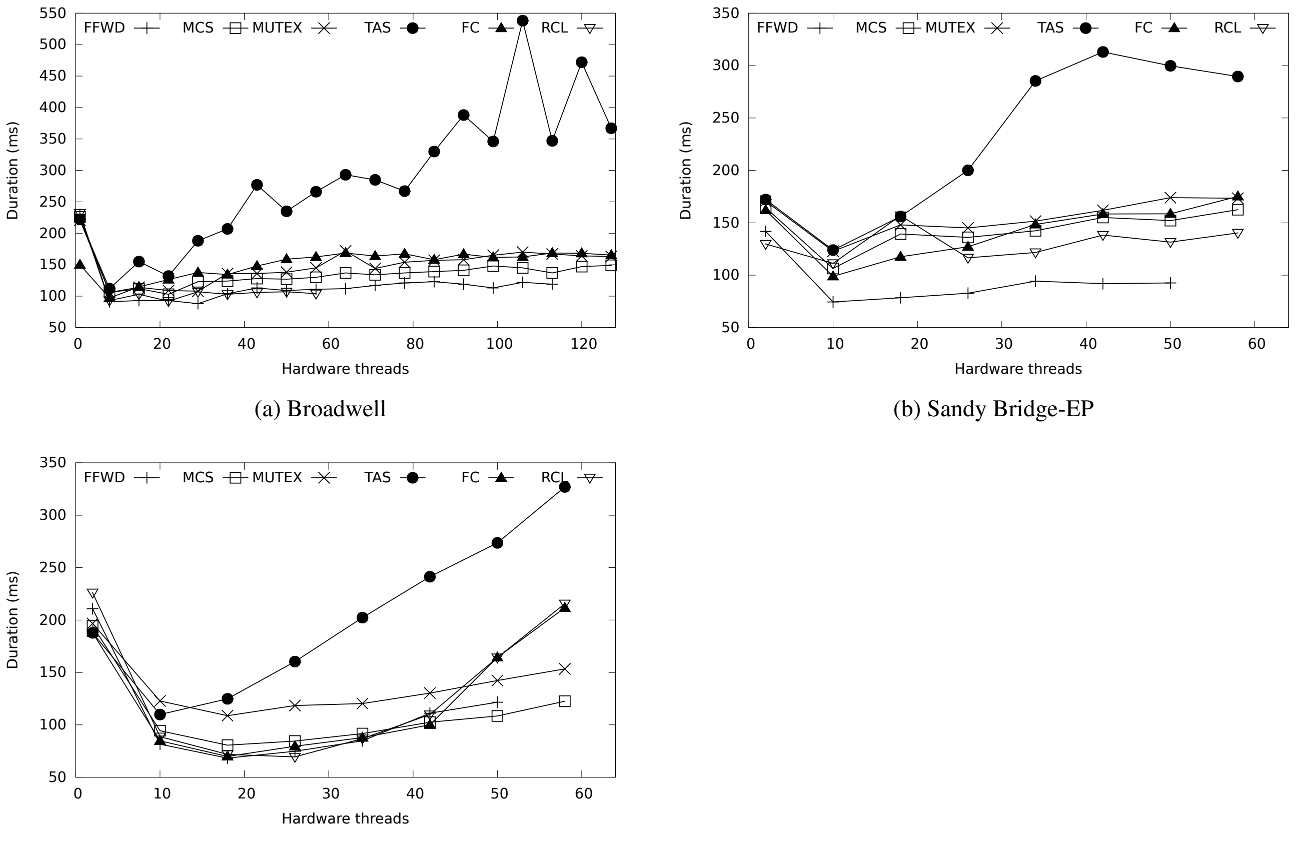
4.3.2 queue, and stack

We now compare the performance of different access methods for queue and stack data structures, adopted from [32]. The queue implementation is based on the two-lock queue algorithm proposed in [63]. In this algorithm, the head and the tail of the queue are protected by two distinct locks of the same type, allowing parallel enqueues and dequeues.

Figures 20–21 compare the performance of state-of-the-art locking and combining-based synchronization methods with that of *ffwd* for a varying number of threads. The experiments performed in this section are similar to those in [32]. Every thread executes 10^6 pairs of enqueues/pushes and dequeues/pops. There is a random number of increment loops (between 0 and 64) after each operation to simulate the work done in between.

In these figures, FC, CC, DSM, and H are combining-based implementations of the stack and queue benchmarks respectively. FC is based on the Flat Combining algorithm proposed in [40], in which a thread becomes the combiner by acquiring a global lock. It then serves any active requests from other threads in addition to its own critical section. The other three combining methods, proposed in [32], though similar to flat combining in principle, differ from it in the following way: a FIFO queue is used to both implement the lock and store the threads' requests, and the combiner is always the thread at the head of such a queue. CC-Synch (CC) is designed to have good performance for cache coherent (CC) shared memory systems, while DSM-Sync (DSM) is more suited for distributed shared memory (DSM) systems. H-Sync (H) is a hierarchical version of CC-Synch. SIM is a wait-free implementation based on the Sim algorithm [31]. MS is a lock-free queue implementation using the algorithm presented by Michael and Scott in [63]. LF is another lock-free implementation by Fatourou and Kallimanis [32]. BLF is the lock-free queue in the Boost library.

For the *ffwd* implementation, we simply remove the lock acquisition code, and delegate the entire enqueue/dequeue, or push/pop functions as appropriate using the *ffwd* API.



(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 12: Phoenix matrix multiply benchmark runtime for varying number of threads (lower is better).

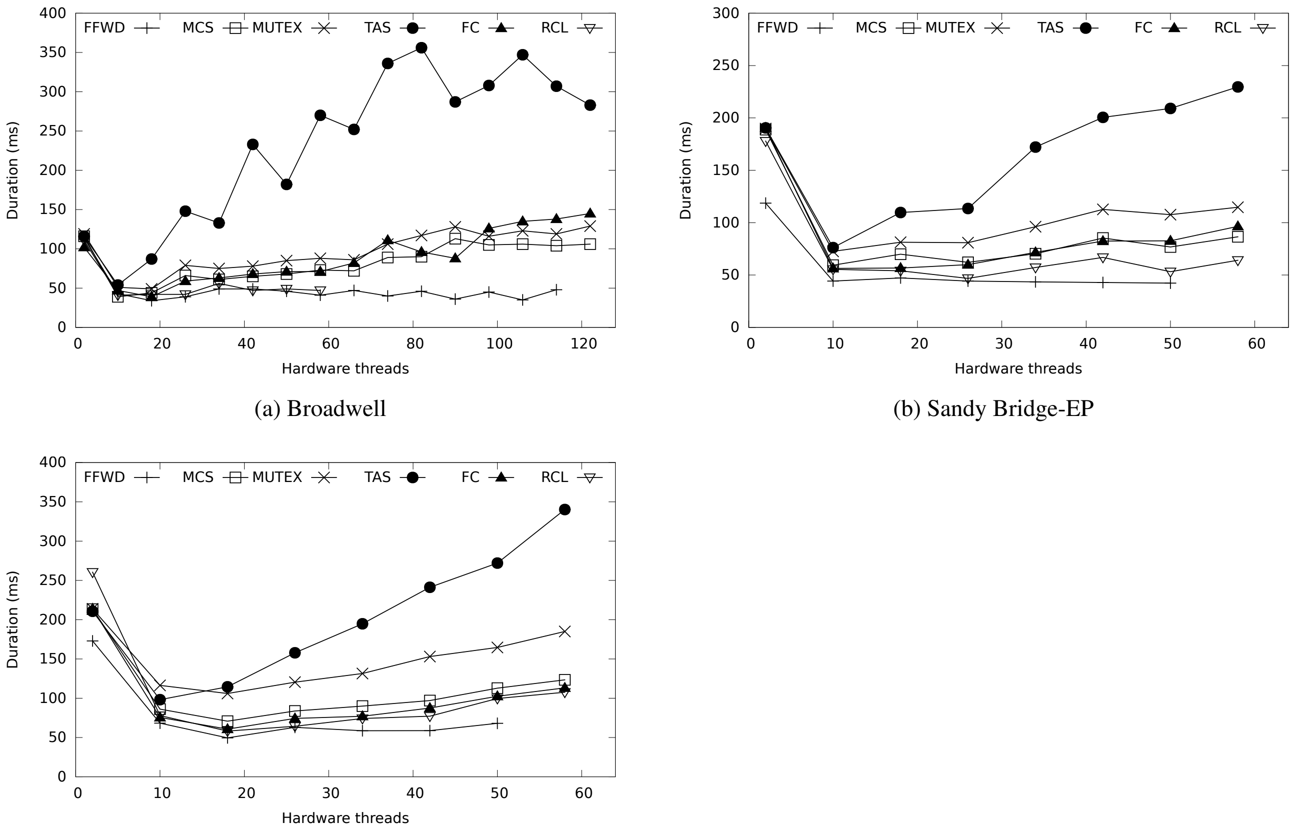
Figures 20–21 show the measured throughput for the queue and stack benchmarks respectively. Here, *ffwd* matches and often significantly outperforms the best lock-based and combining-based schemes. The lock-based and combining methods are competitive up to the socket boundary, then drop off significantly. For *ffwd*, a minor drop-off is observed between 15 and 16 threads (one thread on each socket being used for the server), but because communication is concurrent in *ffwd*, the added latency incurred when crossing sockets is less impactful. The over-subscribed version, FFWDx2 has an significant advantage for small thread counts (where *ffwd* performance is heavily latency constrained), but maintains only a small lead over *ffwd* for larger thread counts. We also see that the combining approaches significantly outperform the lock-based ones, with the more recent approaches leading. Nevertheless, delegated stacks and queues with *ffwd* are on average twice as fast as the best combining-based stack and queue implementations.

Another key observation is that for all approaches, except *ffwd*, the queue performance is generally higher than the stack performance, due to the two locks used in the queue. In *ffwd*, however, a single server handles all delegations, implicitly serializing them. Thus, *ffwd* performance is essentially identical for both data structures.

4.3.3 linked list

In this benchmark, a single linked list contains integers in sorted order, representing a set of integers. Threads query the list for membership (70%), and alternate inserting members into, and removing members from the list (30%). For synchronized access, a single global lock protects the list from concurrent access. For delegation, both query and update methods are delegated. FFWDx2 results are elided: they are essentially identical to FFWD due to the long critical sections in this benchmark.

Figure 26 shows the performance of this basic linked list implementation, with an initial list length of 1024, as we vary the number of threads. Given the use of a single lock for the list, the performance of the locking methods drops quickly with the number of contending threads. Note too that *ffwd* throughput does not increase with the number of threads, as was the case with the fetch-and-add benchmark §4.3.1. This is because the time taken to traverse the relatively long linked list bounds server performance. Nevertheless, *ffwd* provides a significant performance boost over

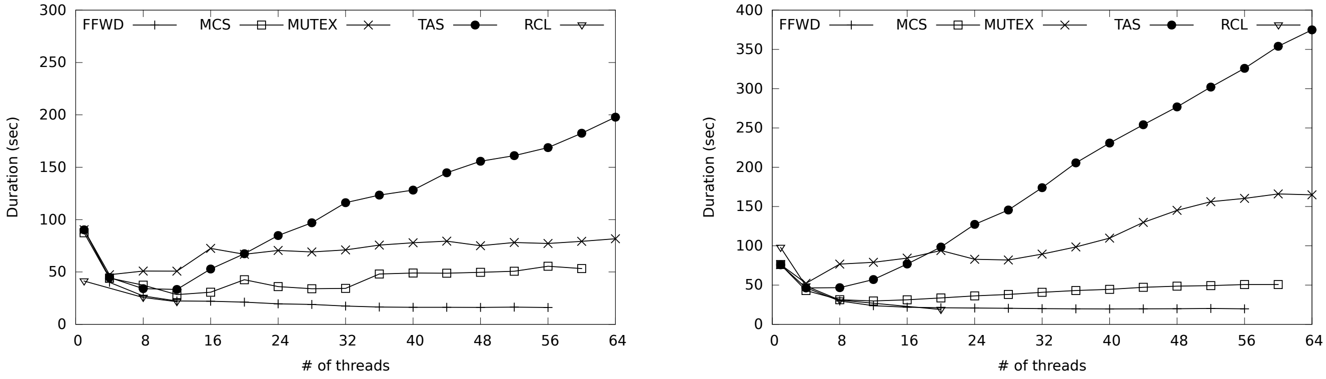


(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 13: Phoenix linear regression benchmark runtime for varying number of threads (lower is better).



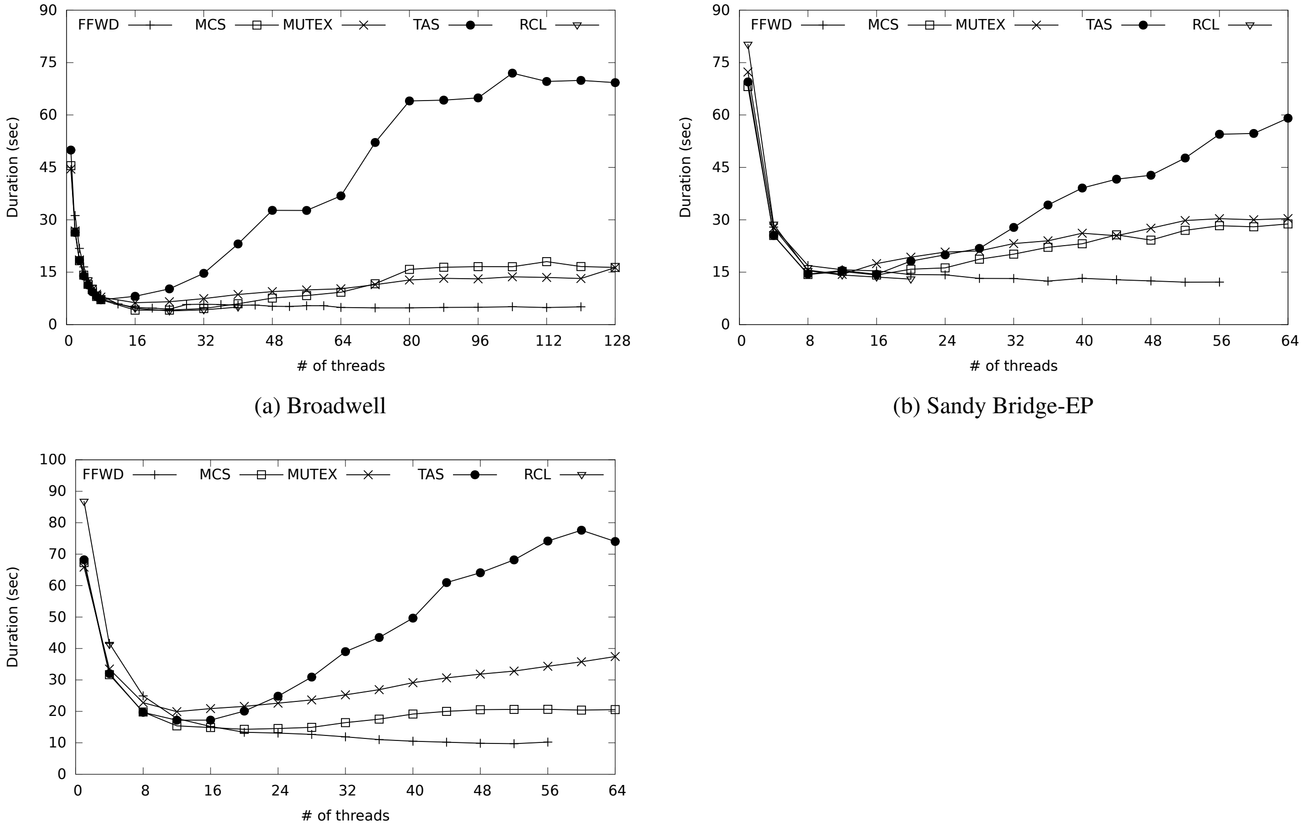
(a) Sandy Bridge-EP

(b) Abu Dhabi

Figure 14: Memcached set operation run time for varying number of threads (lower is better).

locking on this naïve version of a linked list. Interestingly, the added concurrency provided by the software transactional memory approach (STM) [28] results in a substantial performance boost for modest thread counts, and relatively graceful performance degradation as the thread count grows.

A faster "lazy" linked list for concurrent use is presented in [39]. Here, threads traverse the list in parallel, and fine-grained locking is used to update the list. For the *ffwd* version the updates operations are delegated to a single server: as in the locking version, clients traverse the list in parallel.

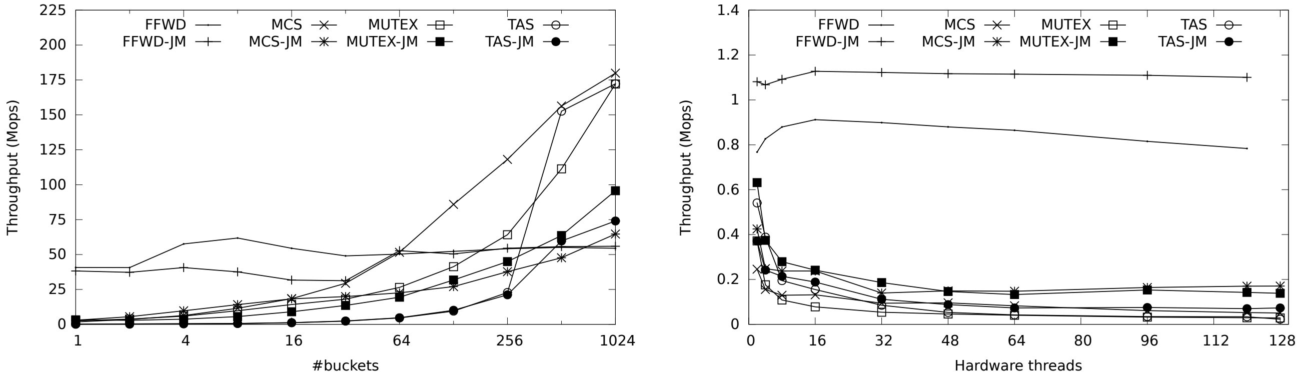


(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 15: Memcached get operation run time for varying number of threads (lower is better).



(a) Comparison between use of malloc and jemalloc allocators for average throughput of hash table over 120 threads for *ffwd*(128 threads for all other methods) on Broadwell

(b) Comparison between use of malloc and jemalloc allocators for average throughput of naïve linked list with 1024 initial list elements, 30% update ratio on Broadwell

Figure 27 shows the performance of the lazy linked list, in an experiment that uses the same settings as Figure 26. Here, the *ffwd* implementation of the lazy linked list is labeled FFWD-LZ. Besides the performance improvement across the board, the lazy list exhibits a very different trend. Due to the concurrent traversal, and the fine-grained locking for updates, adding threads increases concurrency, and therefore throughput. In absolute performance terms, *ffwd* also benefits significantly from the lazy list version. However, its single server cannot keep up with many threads and 1024 locks.

As an alternative to delegating this highly concurrent data structure, we also evaluate using a data structure better suited

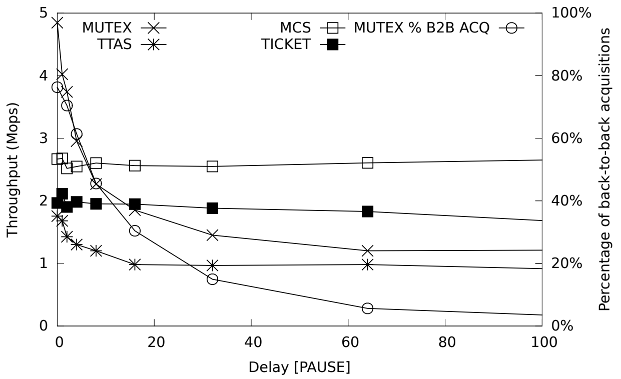


Figure 17: Percentage of back-to-back acquisitions, and lock throughput, for varying delay between critical sections (higher is better).

to single-threaded execution. The lines labeled FFWD-SK and MCS-SK show the performance of a skip list [51], under the same conditions. While FFWD-LZ falls far behind other methods, FFWD-SK is highly competitive with the lazy list. Meanwhile, the coarse-grained MCS lock version of the skip list (MCS-SK) is unable to scale beyond a handful of threads, as it offers no concurrency, and poor memory locality. For reference, we also include the Harris list [37].

To better understand the dynamics of these linked lists, Figure 28 shows their performance as we vary the length of the list. While FFWD-LZ falls behind the lock-based lazy linked lists as the initial size grows, the origin of this drop-off is not strictly server processing: in the delegated lazy list implementation, the server is only responsible for updates, which does not include list traversal. Rather, the design of the lazy linked list, in combination with delegation, results in a serialization of communication delays. Specifically, every write by the server results in a cache miss, causing the server (and with it, all pending requests) to stall when its store buffers are depleted. Figure 29 illustrates this effect: the most severe performance degradation in *ffwd* coincides with the rise in store buffer stalls, which at its peak consumes 80% of the server's cycles. As the list grows further, clients slow down, taking some of the load off the server.

Meanwhile, FFWD-SK illustrates the benefit of using a high-performance single-threaded data structure: as the list grows beyond 2048 elements, even the massive parallelism of the lazy list cannot make up the $O(N)$ vs. $O(logN)$ difference in computational complexity. We did not evaluate the performance of concurrent skip lists [34, 46, 80, 85].

In summary, one cannot choose a data structure in isolation. While both the lazy list and the skip-list are better than the naive list under all circumstances, the skip-list is very well suited to delegation, while the lazy list is better suited for multi-threaded execution. When used with a suitable data structure, we find that delegation is very competitive with alternative approaches.

4.3.4 binary search tree

Figure 30 shows the performance of several binary tree implementations. The initial size of the tree is 1,024 elements, and the workload consists of 50 percent reads, 50 percent updates (equal proportions of insert and delete).

FFWD delegates all tree operations to a single server, which uses a barebones binary tree implementation. The inserts and deletes are random, which results in an approximately balanced tree, and the tree operations in this benchmark do not rebalance the tree. The same design is used by the RCU [20], RLU [57], SWISSTM [28] and VTree [95] programs, while the VRBTree [95] implements a balanced Red-Black tree. For this relatively small tree, FFWD significantly outperforms the other alternatives, as the critical section is short, allowing a single-threaded solution to outperform more sophisticated, parallel trees.

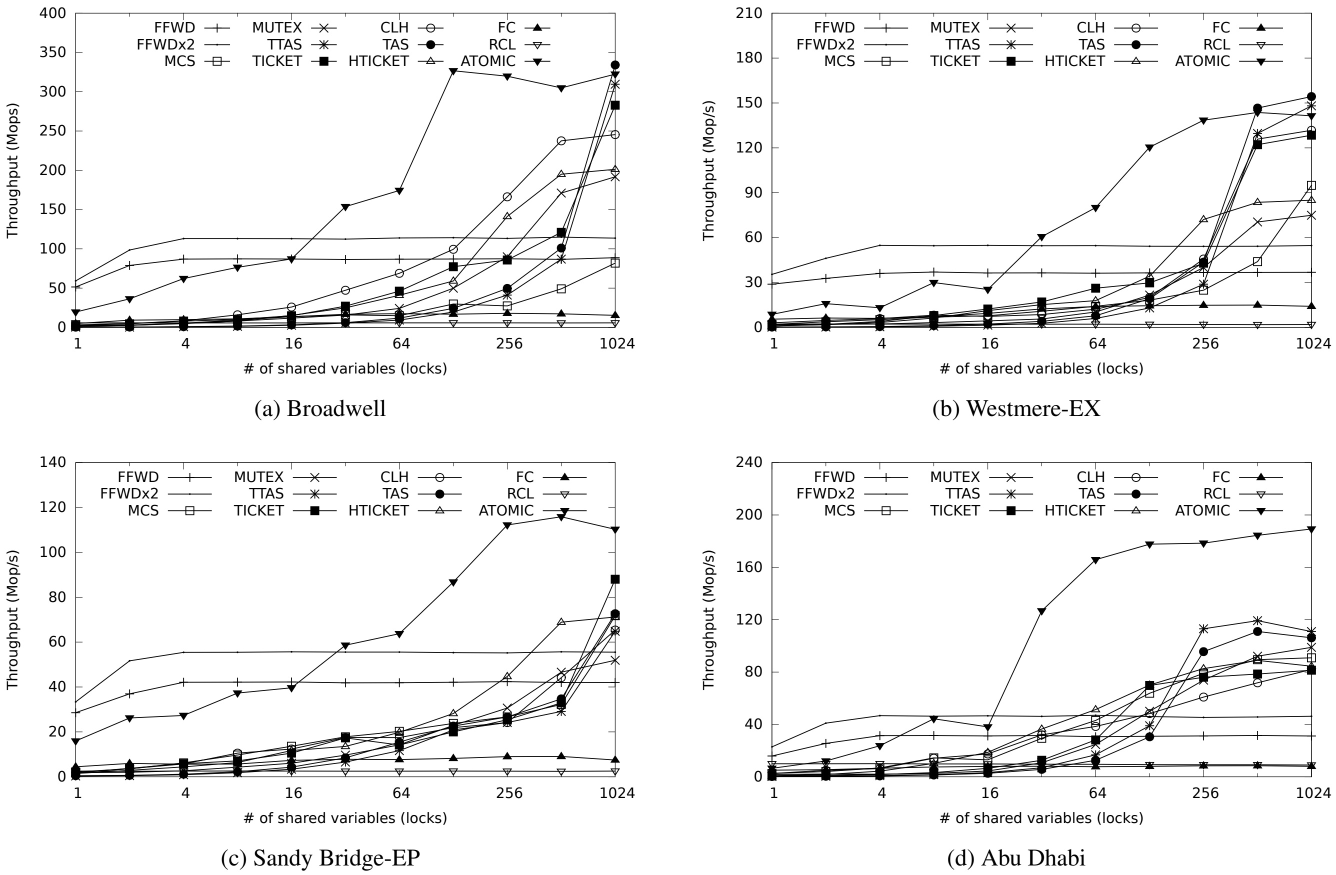
However, Figure 31 tells a more complete story. Here, the tree size is varied between 128–128k elements, with the larger sizes resulting in much longer critical sections due both to the number of levels of the tree, and the decidedly poorer memory locality. The *ffwd* results reflect this reality: as the tree grows, *ffwd* throughput drops steadily, while RLU and SWISSTM take advantage of the larger tree to improve concurrency. The line labeled "single threaded" shows the throughput of the data structure when running the same workload in a single thread. The narrow gap between *ffwd* and single-threaded performance shows how well *ffwd* is making single-threaded performance available to this multi-threaded application. For very large trees, RCL also approximates the single-threaded performance.

In order to achieve performance on par with RLU and SWISSTM, it is necessary to go beyond a single server. FFWD-S4 shows *ffwd* performance after partitioning (sharding) the key space into four separate trees, and delegating each of these trees to a different server, yielding a 4× increase in throughput. Note, however, that this basic design relies on a static partitioning, which is not ideal for all workloads.

4.3.5 hash table

In our final benchmark program, the hash table, a set of integers is stored in a hash table. As above, 70% of operations are membership queries, while 30% add elements to, or remove from the set. Each bucket in the hash table contains a simple linked list, which typically holds only a small number of items. For locking methods, each bucket has its own lock, which is acquired prior to accessing the list. For delegation, both membership queries and updates are delegated to the server in charge of the hash bucket in question.

A hash table is perhaps a counterintuitive evaluation choice for a delegation system. Hash tables are an ideal target for fine-grained synchronization, and thus, we would expect locking to outperform delegation. However, the hash table benchmark presents an opportunity to directly manip-



(a) Broadwell

(b) Westmere-EX

(c) Sandy Bridge-EP

(d) Abu Dhabi

Figure 18: Average throughput of fetch-and-add, for a varying number of locks/shared variables (log scale) (higher is better).

ulate the amount of concurrency in the system, similar to fetch-and-add, making it a useful evaluation tool, even if a large hash table is a highly unlikely target for delegation.

Figure 32 shows the throughput of the hash table, as we vary the number of buckets in the hash table. The number of entries is also adjusted, so that buckets on average have one entry each. The plot bears a close resemblance to the fetch-and-add results in Figure 18.

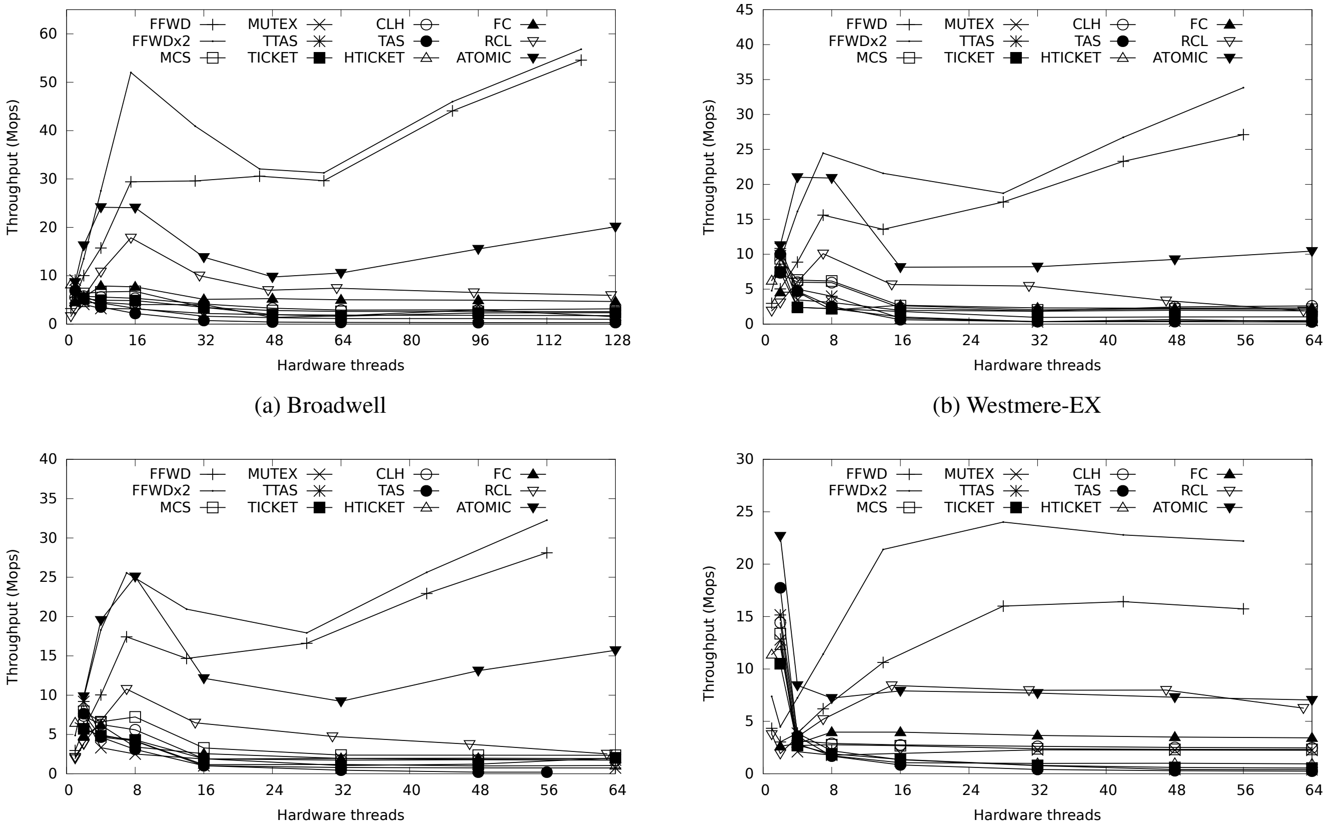
Notably, *ffwd* is only competitive up to 64 buckets, whereas in the fetch-and-add experiment, it kept pace with the other methods up to 128 global variables. In hash table case, the greater complexity of managing linked lists vs. incrementing integers reduces the throughput of the 4 server threads, whereas locking remains communication and contention-bound, and is largely unaffected by an increase in the processing burden.

5. Porting Code to Delegation

Overall, delegation is best suited to serving a fast, single-threaded data structure to multiple client threads. This requires little effort, and usually results in high performance. Transitioning this type of code to delegation with *ffwd* is simple: delegation requires both shorter and less complex code than locking does, and runs faster.

However, another common use case would be to port existing locking code to delegation. In [53], an automated method for doing this is described for the RCL delegation system. This is elegant, but requires functionality that is currently unsupported by *ffwd*, including lock acquisition and blocking system calls inside delegated functions. Potentially, the automatic rewriting approach could be modified to accommodate *ffwd*, but we leave this to future work.

In terms of manually porting code to delegation, locking code can be converted to single-server delegation by removing the locking code, and delegating all critical sections that access the shared variables or data structures in question. However, this approach is similar to using a single lock in a concurrent program; it often reduces program concurrency, an can sometimes result in a performance degradation. Significantly, *ffwd* does not provide any means of synchronization between servers. Thus, to take advantage of multiple servers, each server must serve independent data structures, or independent partitions of a single data structure, as in the tree benchmark (§4.3.4).



(a) Broadwell

(b) Westmere-EX

(c) Sandy Bridge-EP

(d) Abu Dhabi

Figure 19: Average throughput of fetch-and-add, for a single lock/shared variable, and varying number of threads. Delegation performance grows rapidly with the number of threads, outpacing even atomic instructions once threads spread over more than one socket (higher is better).

For all benchmarks but memcached, porting the application to *ffwd* was a matter of a handful of changed lines of code. Memcached, on the other hand, required significant effort: 1460 lines of code were either moved, added or deleted, with the vast majority being lines moved from critical sections into functions to be delegated. Because we don't acquire any locks before entering the critical section, *every* access to a delegated data structure must be delegated. In large or complex software, especially involving function pointers and/or lock pointers, even identifying every such access can be hard. In addition, acquiring locks in delegated functions can degrade server performance, since it may block the server, or even result in deadlock if the lock is not released before returning. Therefore, it is often preferable to eliminate any such nested locking, and instead delegate all accesses that use these locks as well, in effect causing a cascade of porting effort. Here, the RCL approach, which focuses on re-engineering multi-threaded programs, has a significant advantage in terms of effort required.

5.1 Combining Delegation and Locking

ffwd provides tremendous performance improvements over locking and combining in some settings, but a spinlock is often the best-performing solution for highly parallel programs. This suggests a hybrid solution. Nothing prevents the coexistence of *ffwd* and locking in a program, as long as the respective data structures they manage are independent. Thus, for maximum performance, one may use *ffwd* for a central shared work queue, but spinlocks to protect the million-bucket hash table using fine-grained locking.

6. Related Work

Access to shared data structures is a common performance bottleneck in concurrent programs, which has spawned a range of approaches aimed at improving performance and scalability. Synchronization through mutual-exclusion (locking) remains the most popular scheme for access to shared variables, and the design of efficient and scalable locks has been a rich topic of research for decades [9, 12, 15, 18, 19, 23, 25–27, 36, 48, 54, 55, 59, 61, 70, 77, 78, 84]. We review the locking literature and other approaches below.

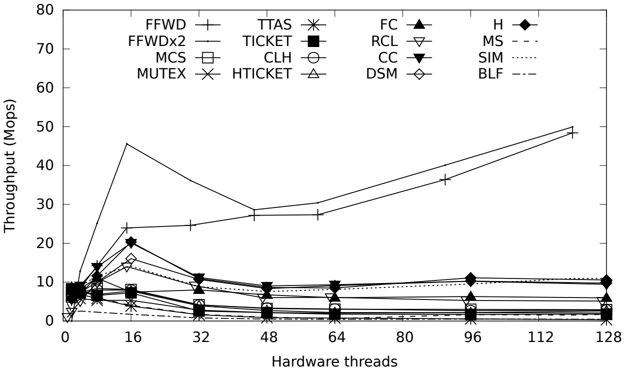


Figure 20: Average throughput of *two-lock* [63] queue with different synchronization and lock-free methods on Broadwell (higher is
better).

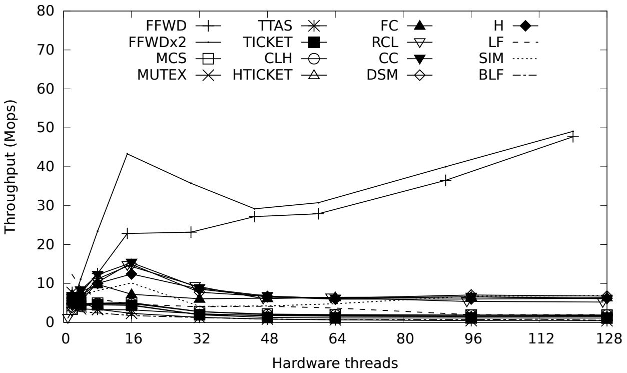


Figure 21: Average throughput of stack with different synchronization and lock-free methods on Broadwell (higher is better).

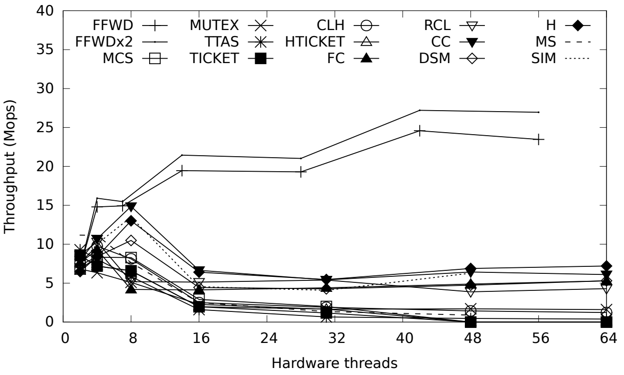


Figure 22: Average throughput of *two-lock* [63] queue with different synchronization methods on Sandy Bridge-EP (higher is better).

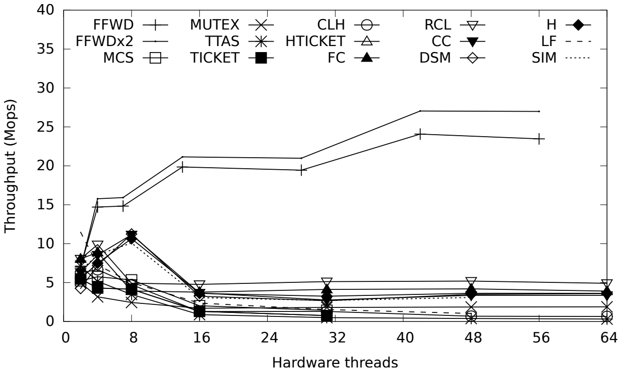


Figure 23: Average throughput of stack with different synchronization methods on Sandy Bridge-EP.

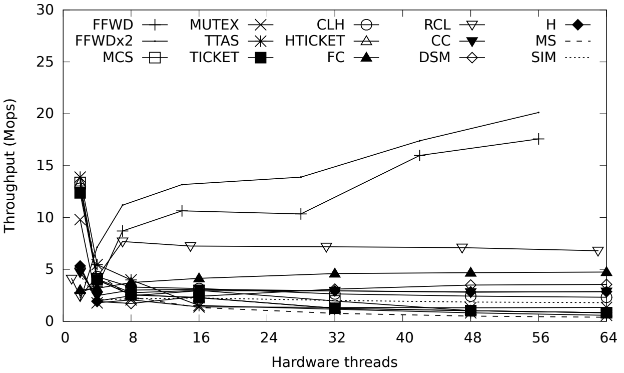


Figure 24: Average throughput of *two-lock* [63] queue with different synchronization methods on Abu Dhabi.

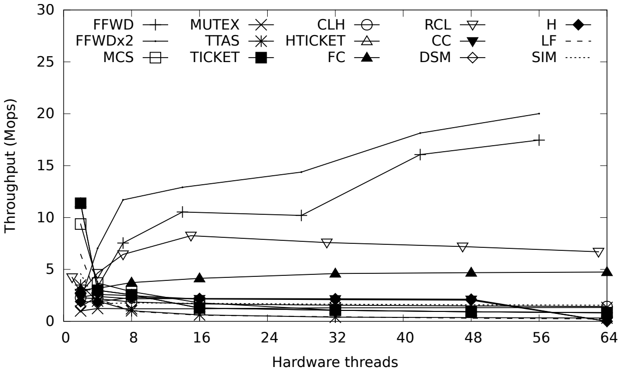
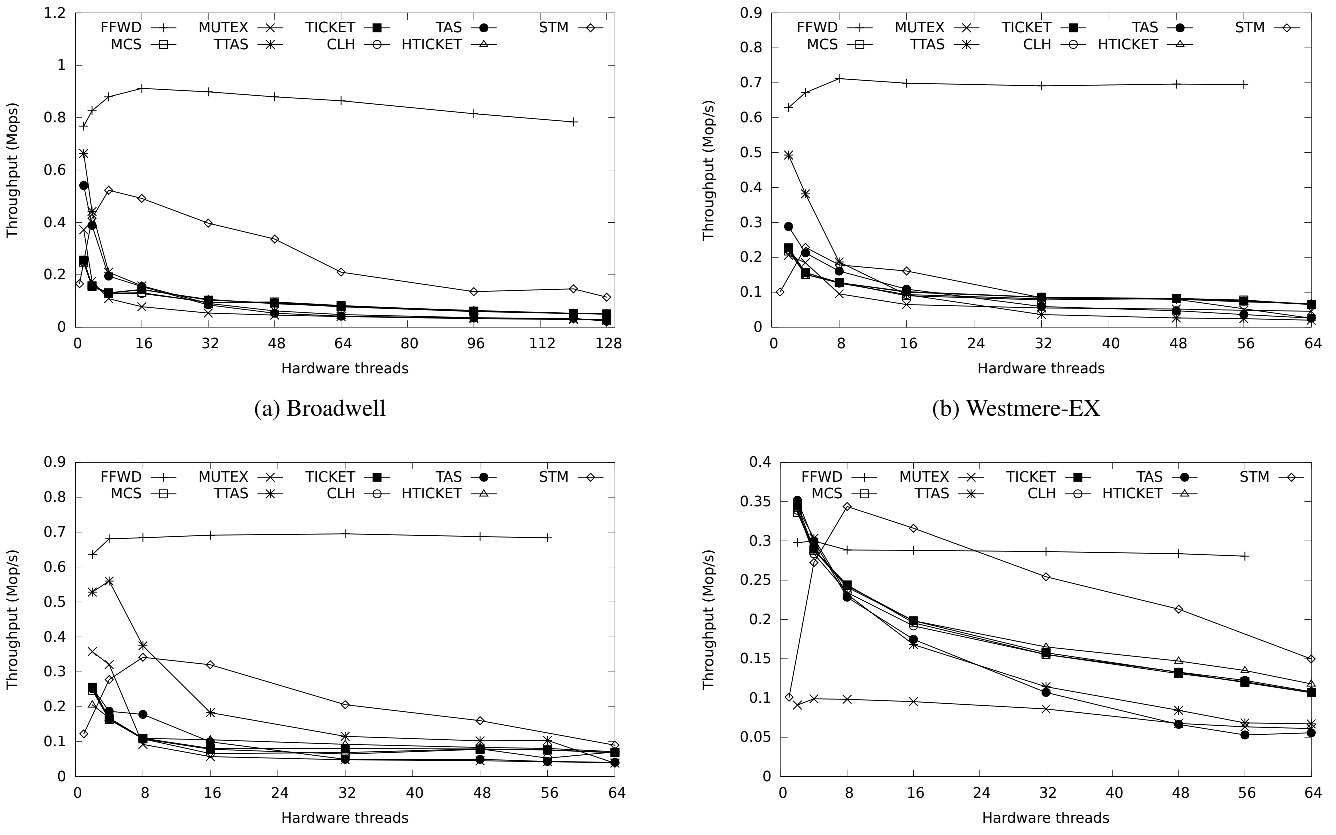


Figure 25: Average throughput of stack with different synchronization methods on Abu Dhabi (higher is better).



(a) Broadwell

(b) Westmere-EX

(c) Sandy Bridge-EP

(d) Abu Dhabi

Figure 26: Average throughput of naïve linked list with 1024 initial list elements, 30% update ratio (higher is better).

6.1 Spinlocks

Spinlock approaches [12, 43, 61, 74] generally spin on a shared memory location until the lock is acquired. Closely related, ticket locks [61] trade-off some performance in low-contention regimes for improved fairness and scalability. Due to their simplicity, spin locks outperform other lock types in low contention settings. In high contention settings, however, spin locks suffer from scalability problems, as several threads concurrently attempt to access a single shared lock variable using relatively long-latency atomic operations. Hierarchical versions [27, 54, 70], order acquisitions to reduce cache coherence traffic.

6.2 Queue-based Locks

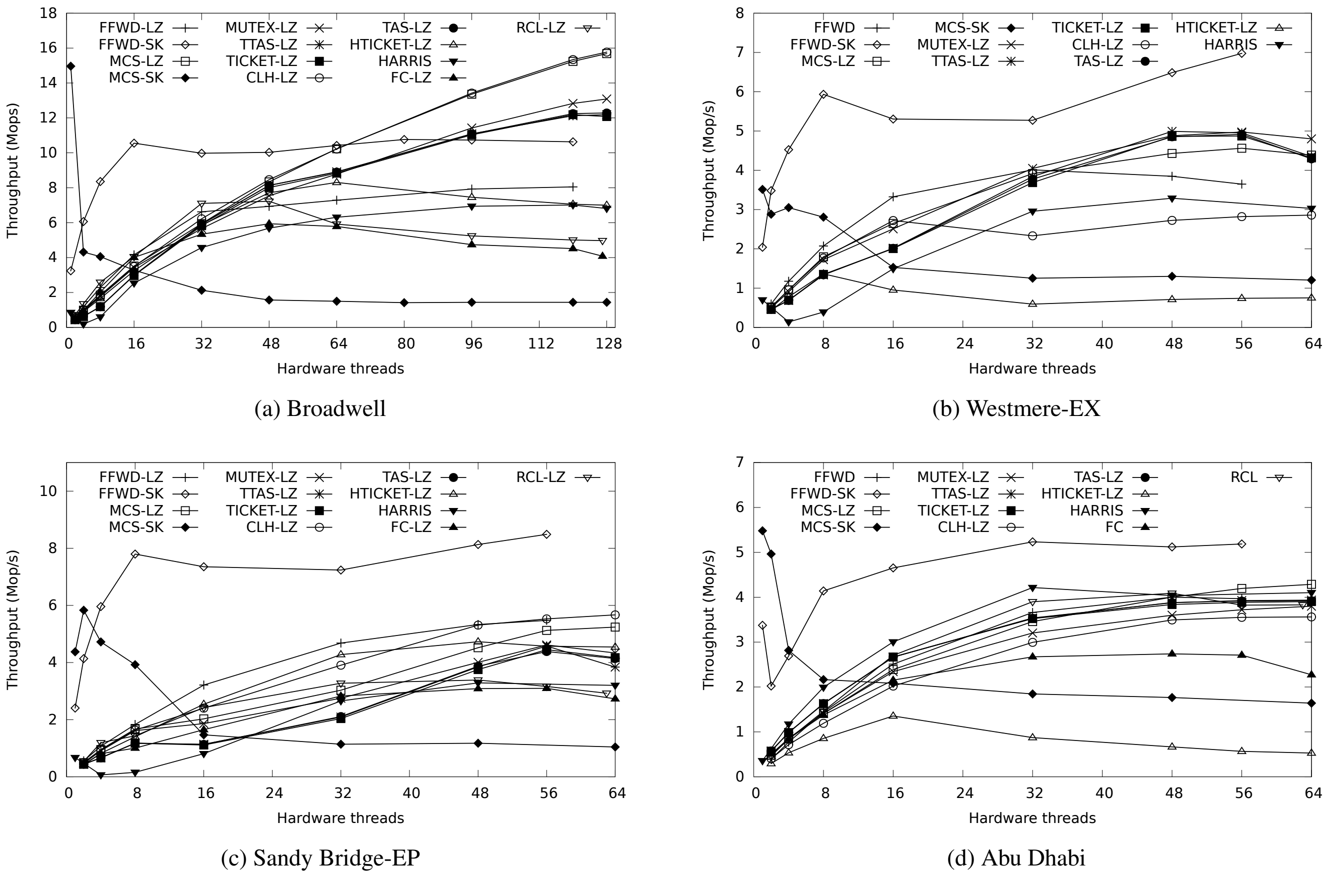
Queue-based locks [23, 55, 61, 77, 78], such as CLH [23] and MCS [61], address scalability issues by spinning on a local memory location rather than the global lock, polling only the previous lock holder's state. Queue-based approaches are designed based on waiting in a queue, which results in improved fairness and reduced cache coherence communication, but memory management can be complex in these schemes, and the added overhead makes these locks slower than spinlocks in a low contention regime. Like hierarchical spin locks, hierarchical queue locks [18, 19, 27, 54, 70] can improve scalability, at the cost of reduced fairness and added complexity.

6.3 Lock-Free Data Structures

Some popular data structures have lock-free implementations [11, 37, 38, 41, 42, 44, 50, 62, 63, 69, 82, 83, 85, 86, 89–91, 95]. Here, modifications to a shared data structure are first made speculatively, which are then committed through a single atomic operation, typically compare-and-swap (CAS), that makes the change visible to other threads. If the commit fails (i.e. a concurrent change made the CAS fail), the operation is restarted. Lock-free data structures can be more efficient than locking, as they replace serialized critical sections with only serialized commit operations. However, for highly contended data structures, frequent retries are common, often leading to poor performance.

6.4 Read-Copy-Update

Read-Copy-Update (RCU) implementations of data structures [13, 21, 47, 87, 88], and the related Read-Log-Update (RLU) [57] typically provide lock-free access to readers, while ensuring mutual exclusion between updaters. This is similar to reader-writer locking [22, 58, 60], except that mul-



(a) Broadwell

(b) Westmere-EX

(c) Sandy Bridge-EP

(d) Abu Dhabi

Figure 27: Average throughput of lazy linked list, skip list and harris's lock-free list with 1024 initial list elements, 30% update ratio (higher is better).

tiple readers operate on the data structure concurrently with the writer. The overall idea is to maintain multiple versions of the same data structure, and only reclaiming an old version after all readers have finished using it. While RCU can be very advantageous for the reader side, updates usually carry a higher cost, in addition to relying on mutual exclusion. This makes the RCU approach best suited to data structures that are relatively infrequently updated.

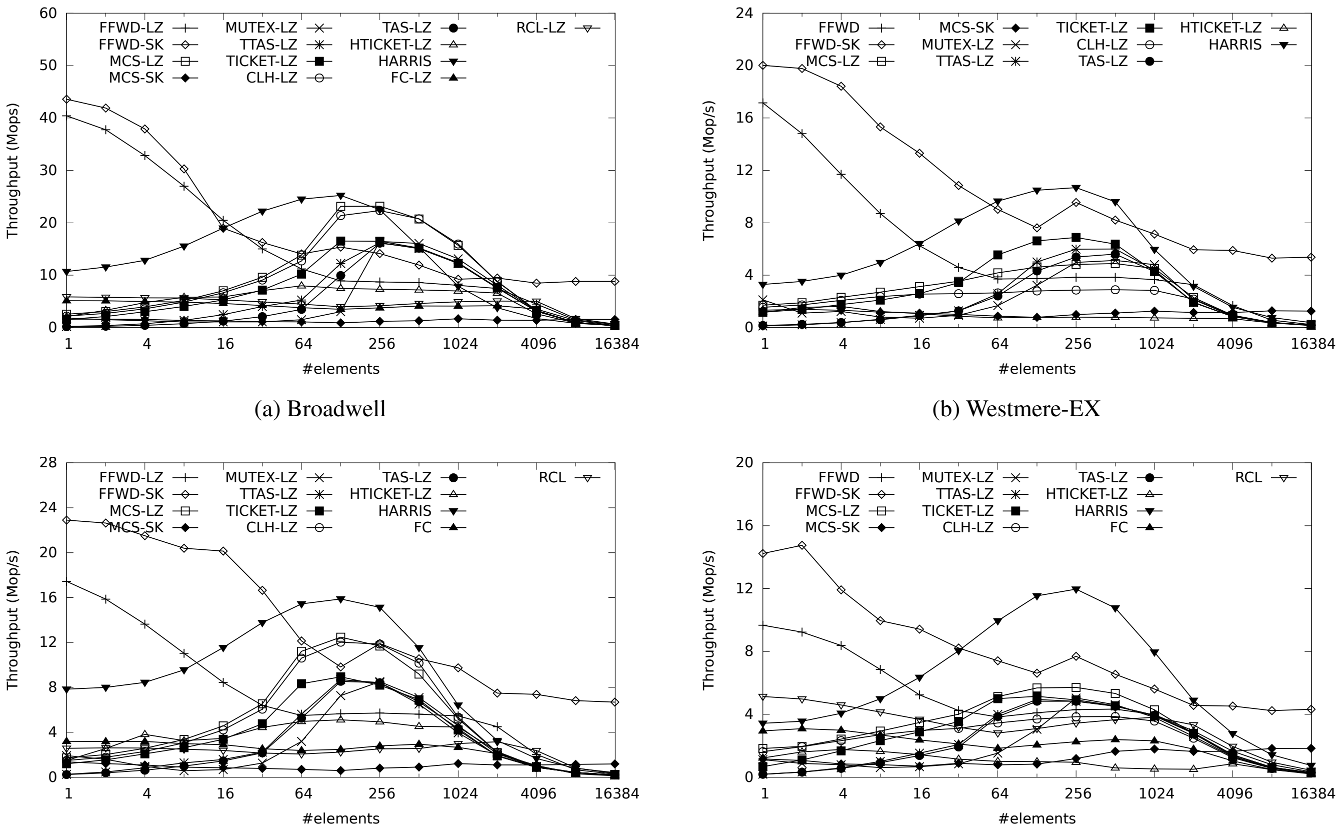
6.5 Software Transactional Memory

Software Transactional Memory (STM) [28, 30, 33, 35, 45, 56, 64, 72, 75, 81, 92, 93] generalizes the main concept behind lock-free data structures (speculative execution) to arbitrary programs. Here, during a *transaction* initiated by the programmer, every store to memory is buffered, and every read is logged, establishing the transaction's *read set*. The STM runtime verifies that no concurrent modifications were made to the values in the read set, before committing the writes to memory. Otherwise, the transaction is aborted and restarted. STM provides concurrent, yet atomic execution of transactions, but incurs significant serial overhead. Similar to lock-free data structures, STM is not efficient for highly contended data structures due to transaction restarts.

6.6 Delegation and Combining

Several delegation techniques [17, 25, 26, 31, 40, 66, 68, 79, 94] have been proposed to address the generally poor performance of lock-based methods on highly contended locks. In delegation, a server thread operates on a shared data structure on behalf of client threads. Here, at most one thread plays the role of server for a single data structure at any given time, thus reducing the number of synchronization operations needed. In the extreme case of a dedicated server (as in *ffwd*), no synchronization is required at all. The other benefit of delegation methods over lock-based schemes is that the shared data structure remains in the server's cache.

A popular category of delegation methods is combining [26, 31, 40, 66, 79, 94]: in Flat Combining [40], threads add the work to be done to a list. One of the threads becomes the server/combiner by acquiring a global lock and executes the critical sections of other threads in addition to its, before releasing the lock. Over time, different threads may play the role of server/combiner. CC-Synch, DSM-Synch, and H-



(a) Broadwell

(b) Westmere-EX

(c) Sandy Bridge-EP

(d) Abu Dhabi

Figure 28: Average throughput of lazy linked list (log scale) over 120 threads for *ffwd* (128 threads for all other methods), 30% update ratio (higher is better).

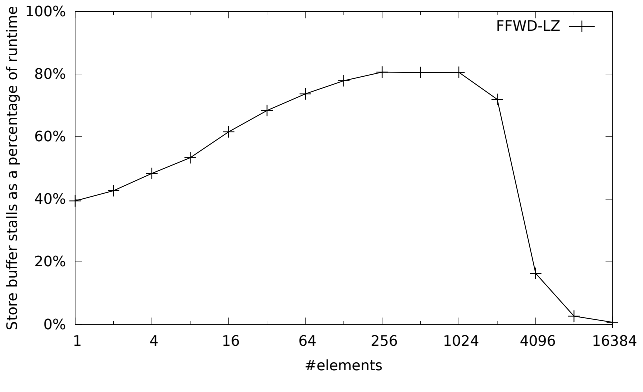


Figure 29: Server store buffer stalls for lazy linked list execution with *ffwd*, for varying list size (log scale) (lower is better).

Synch [32], improve on the flat combining method by using a FIFO queue both to implement the global lock (as in the MCS lock) and to store the threads' requested work. In high contention, combining approaches outperform locking, but come with extra memory management and overhead.

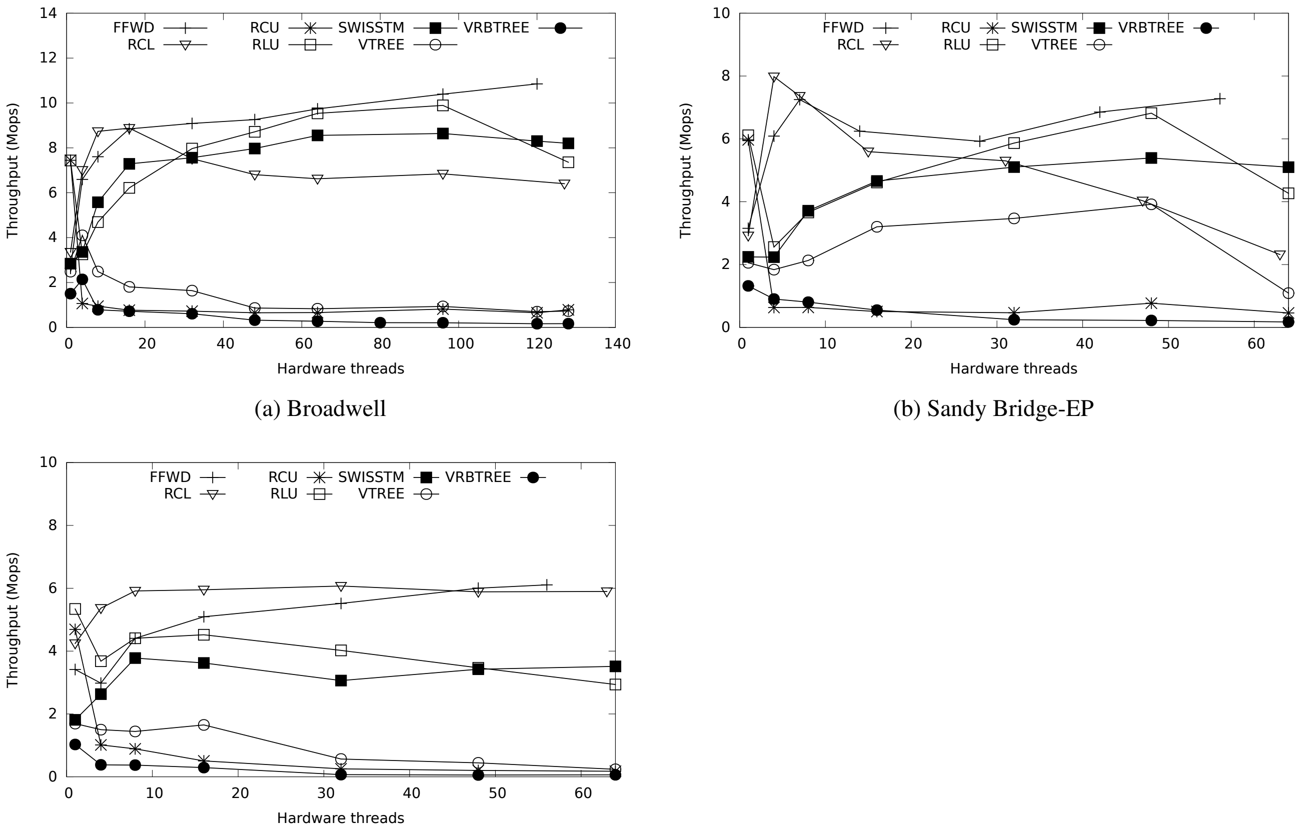
Contrary to the combining approaches, Remote Core Locking (RCL) [53] and *ffwd* employ one or more dedicated server threads. Here, RCL aims to provide a drop-in, delegation-based replacement for locks, resulting in an elegant and scalable, but complex and high overhead solution. The more barebones, but highly efficient *ffwd* generally offers higher throughput, but can be more challenging to use due to its somewhat restricted functionality. In [14], a delegation-based OS design was proposed, arguing for delegation as the path to scalability.

6.7 Batched Data Structures

In batched data-structures [10, 16, 24, 29, 67, 76] multiple operations are collected into a batch before being applied, in parallel, to the data structure. A delegation server or combiner could serve a batched data structure, potentially combining the benefits of both approaches.

7. Conclusion

To summarize, *ffwd* offers an efficient, high-performance design and implementation of delegation for multi-core computers. Given a data structure that runs faster on a single



(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

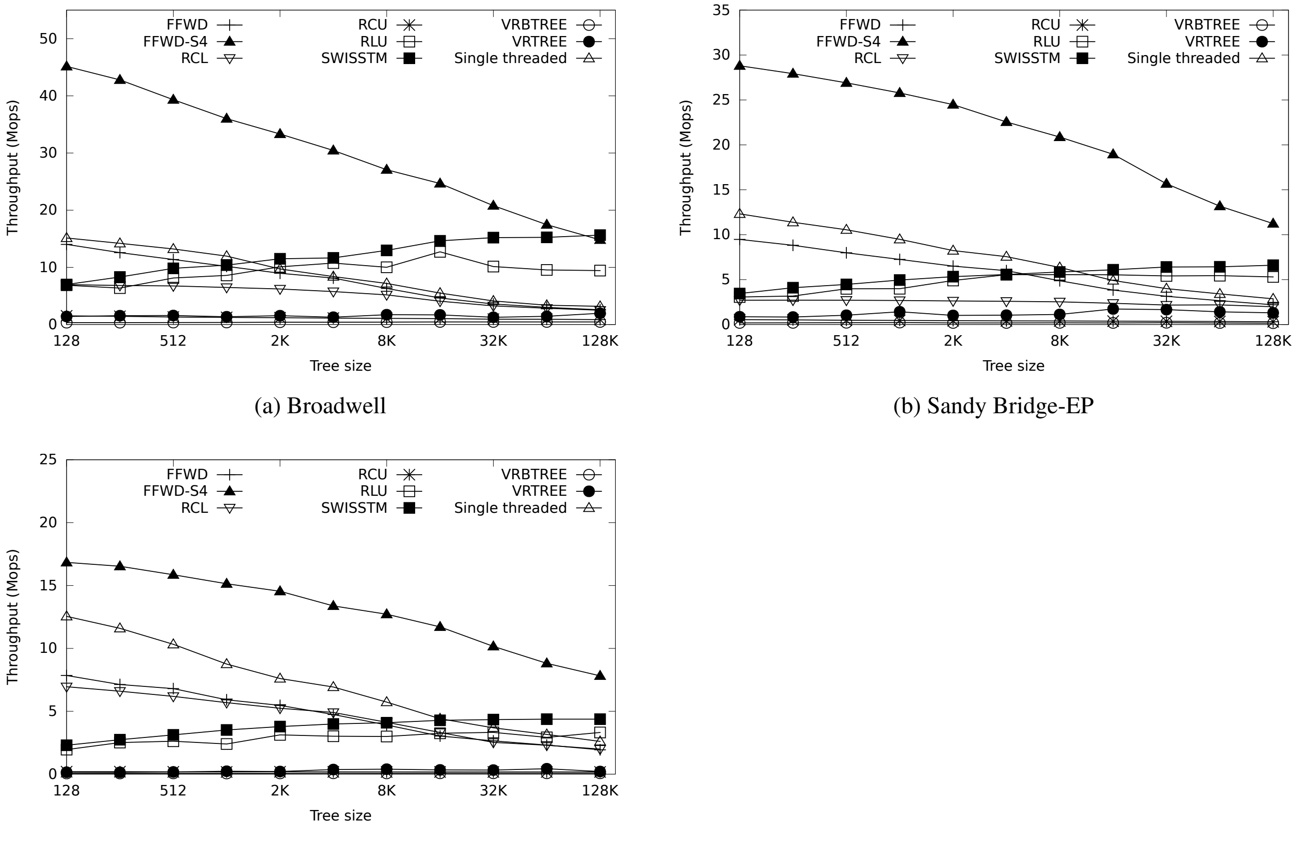
Figure 30: Average throughput of a 1024 node binary tree for varying number of threads (higher is better).

thread than on multiple threads, *ffwd* provides unmatched performance in making this data structure available within a multi-threaded program. For other uses, the case is less clear cut. Given enough parallelism, and long critical sections that saturate the delegation server, fine grained locking and parallel data structures are sometimes the better design choice.

In describing *ffwd*, and making a strong case for delegation as an attractive, easy to use, and high-performance means of accessing a shared data structure, we hope to inspire the community to further explore its potential.

8. Acknowledgments

This work was made possible through National Science Foundation grants CNS-1320235, CNS-1617992 and CNS-1703425. We would also like to acknowledge the priceless insights of our shepherd Don Porter, as well as Joseph Devietti, Timothy Merrifield and Andreas Willig for many helpful exchanges leading up to this work.

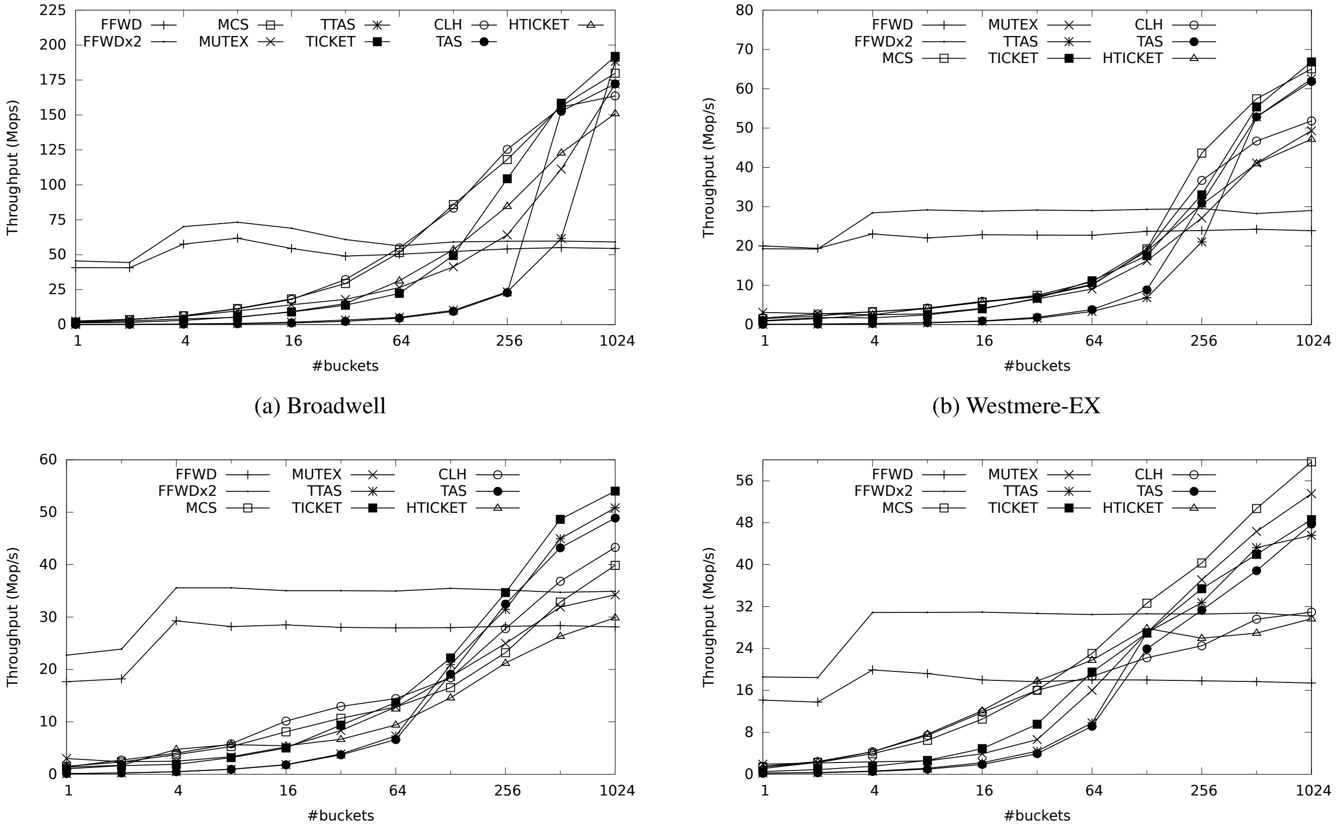


(a) Broadwell

(b) Sandy Bridge-EP

(c) Abu Dhabi

Figure 31: Average throughput for binary tree, for varying tree size (log scale) (higher is better).



(a) Broadwell

(b) Westmere-EX

(c) Sandy Bridge-EP

(d) Abu Dhabi

Figure 32: Average throughput of hash table (log scale) over 120 threads for *ffwd* (128 threads for all other methods), initial load factor 1, 30% update ratio (higher is better).

References

[1] Intel® memory latency chekcer. URL `https://software.intel.com/en-us/articles/intelr-memory-latency-checker`.

[2] Fastforward @ github. URL `https://github.com/bitslab/ffwd`.

[3] Hoard. URL `https://www.hoard.org`.

[4] Jemalloc. URL `http://jemalloc.net`.

[5] Memcached, . URL `https://memcached.org`.

[6] Memslap, . URL `http://docs.libmemcached.org/bin/memaslap.html`.

[7] Slab. URL `https://github.com/bbu/userland-slab-allocator`.

[8] Splash-2. URL `http://www.capsl.udel.edu/splash`.

[9] J. L. Abell, J. Fern, M. E. Acacio, et al. Glocks: Efficient support for highly-contended locks in many-core cmps. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 893–905. IEEE, 2011.

[10] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 84–95. ACM, 2014.

[11] S. Al Bahra. Nonblocking algorithms and scalable multicore programming. *Queue*, 11:40, 2013.

[12] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:6–16, 1990.

[13] M. Arbel and H. Attiya. Concurrent updates with rcu: search tree as an example. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 196–205. ACM, 2014.

[14] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[15] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21:246–254, 1994.

[16] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49:4–21, 1998.

[17] 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. Springer, 2013.

[18] M. Chabbi and J. Mellor-Crummey. Contention-conscious, locality-preserving locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 22. ACM, 2016.

[19] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High performance locks for multi-level numa systems. In *ACM SIGPLAN Notices*, volume 50, pages 215–226. ACM, 2015.

[20] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47:199–210, 2012.

[21] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47:199–210, 2012.

[22] P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14:667–668, 1971.

[23] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993.

[24] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. *Mathematical Foundations of Computer Science 1998*, pages 722–731, 1998.

[25] 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*, pages 33–48. ACM, 2013.

[26] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining numa locks. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 65–74. ACM, 2011.

[27] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. In *ACM SIGPLAN Notices*, volume 47, pages 247–256. ACM, 2012.

[28] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM sigplan notices*, volume 44, pages 155–165. ACM, 2009.

[29] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31:1343–1354, 1988.

[30] R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2006.

[31] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 325–334. ACM, 2011.

[32] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.

[33] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[34] K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[35] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25: 5, 2007.

[36] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, volume 99, pages 87–100, 1999.

[37] T. Harris. A pragmatic implementation of non-blocking linked-lists. *Distributed Computing*, pages 300–314, 2001.

[38] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67: 1270–1285, 2007.

[39] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems*, pages 3–16. Springer, 2005.

[40] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

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

[42] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:745–770, 1993.

[43] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[44] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 522–529. IEEE, 2003.

[45] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[46] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[47] P. W. Howard and J. Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 26: 2684–2712, 2014.

[48] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. *ACM SIGARCH Computer Architecture News*, 38:117–128, 2010.

[49] D. Klaftenegger, K. Sagonas, and K. Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *European Conference on Parallel Processing*, pages 572–583. Springer, 2014.

[50] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5:190–222, 1983.

[51] S. Lists. A probabilistic alternative to balanced trees, william pugh. *Communications. of the ACM*, 33:668–676, 1990.

[52] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. *ACM Sigplan Notices*, 46: 3–18, 2011.

[53] J.-P. Lozi, F. David, G. Thomas, J. L. Lawall, G. Muller, et al. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX Annual Technical Conference*, pages 65–76, 2012.

[54] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical clh queue lock. *Euro-Par 2006 Parallel Processing*, pages 801–810, 2006.

[55] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[56] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing*, pages 354–368. Springer, 2005.

[57] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183. ACM, 2015.

[58] P. E. McKenney and B. Kingsbury. Reader-writer lock for multiprocessor systems, Nov. 23 2004. US Patent 6,823,511.

[59] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *ACM SIGPLAN Notices*, 26:269–278, 1991.

[60] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *ACM SIGPLAN Notices*, volume 26, pages 106–113. ACM, 1991.

[61] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9:21–65, 1991.

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

[63] 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*, pages 267–275. ACM, 1996.

[64] M. S. Moir. Hybrid software/hardware transactional memory, July 1 2008. US Patent 7,395,382.

[65] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield. Whose cache line is it anyway?: operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 141–154. ACM, 2013.

[66] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Dis-*

tributed Computing for Symbolic and Irregular Applications, volume 16. Citeseer, 1999.

[67] P. Paige. Parallel algorithms for shortest path problems. In *Proc. the 1985 International Conference on Parallel Processing*, 1985.

[68] D. Petrović, T. Ropars, and A. Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, page 17. ACM, 2015.

[69] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. In *International Symposium on Distributed Computing*, pages 108–121. Springer, 2005.

[70] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 241–252. IEEE, 2003.

[71] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.

[72] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, pages 1–10. Association for Computing Machinery (ACM), 2006.

[73] S. Roghanchi, J. Eriksson, and N. Basu. ffwd: delegation is (much) faster than you think. Technical report, University of Illinois At Chicago, 2017.

[74] L. Rudolph and Z. Segall. *Dynamic decentralized cache schemes for MIMD parallel processors*, volume 12. ACM, 1984.

[75] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.

[76] P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49:86–97, 1998.

[77] M. L. Scott. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8:1–221, 2013.

[78] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *ACM SIGPLAN Notices*, volume 36, pages 44–52. ACM, 2001.

[79] O. Shalev and N. Shavit. Predictive log-synchronization. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 305–315. ACM, 2006.

[80] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.

[81] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997.

[82] J. M. Stone. A simple and correct shared-queue algorithm using compare-and-swap. In *Supercomputing'90., Proceedings of*, pages 495–504. IEEE, 1990.

[83] J. M. Stone. A non-blocking compare-and-swap algorithm for a shared circular queue. *S. Tzafestas et al., editors, Parallel and Distributed Computing in Engineering Systems*, pages 147–152, 1992.

[84] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multicore architectures. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 253–264. ACM, 2009.

[85] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 11–pp. IEEE, 2003.

[86] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *OPODIS*, pages 240–255. Springer, 2004.

[87] J. Triplett, P. E. McKenney, and J. Walpole. Scalable concurrent hash tables via relativistic programming. *ACM SIGOPS Operating Systems Review*, 44:102–109, 2010.

[88] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference*, page 11, 2011.

[89] J. D. Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[90] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[91] J. D. Valois. Lock-free data structures. 1996.

[92] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48. IEEE Computer Society, 2007.

[93] A. Welc, S. Jagannathan, and A. Hosking. Transactional monitors for concurrent objects. *ECOOP 2004–Object-Oriented Programming*, pages 494–514, 2004.

[94] P.-C. Yew, N.-F. Tzeng, et al. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100:388–395, 1987.

[95] Y. Zhan and D. E. Porter. Versioned programming: A simple technique for implementing efficient, lock-free, and composable data structures. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, page 11. ACM, 2016.