ffwd: delegation is (much) faster than you think

Sepideh Roghanchi, Jakob Eriksson, Nilanjana Basu
int get_seqno() {
    return ++seqno;
}

// ~1 Billion ops/s
// single-threaded
int threadsafe_get_seqno() {
    acquire(lock);
    int ret=++seqno;
    release(lock);
    return ret;
}

// < 10 Million ops/s
why so slow?
~70 ns intra-socket latency

~14 Mops
Quad-Socket Intel System

150-300 M cachelines/s
10-20 gigabyte/s
~200ns O/W latency
= 5 million ops per second
dedicated server thread
client
client
client
client
client
client
client
CLIENT1

wait for response

request

400x

wait for request

critical section

response

wait for response

critical section
400x still!
CLIENT1

wait for response

400x still!

Dedicated Server Thread

wait for request

critical section
critical section
critical section
critical section
critical section
critical section
critical section
critical section
critical section
critical section

wait for response

CLIENTn
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.

Server acts upon pending requests in batches 15 clients.

Each group of 15 clients shares one 128-byte response line pair.

One dedicated 64-byte request line, per client-server pair

Requests are sent synchronously
A request in more detail

- request is new if: request toggle bit != response toggle bit
- server calls function with (64-bit) arguments provided
- client polls response line until toggle bit == response bit
delegation server thread

local response buffer

..111

group 0 requests

0
0
1
..
delegation server thread

local response buffer

..100

0
0
1
. 
. 

group 0 requests
delegation server thread

local response buffer

..100

global response buffer

modified ←- response cache lines → shared

group 0 requests

0
0
1
.
.

modified ←- response cache lines → shared
shared ←-requests--→ modified

local response buffer

modified ←- - - - - - - - - - response cache lines - - - - - - - - - - shared
performance evaluation
evaluation systems

4×16-core Xeon E5-4660, Broadwell, 2.2 GHz
4×8-core Xeon E5-4620, Sandy Bridge-EP, 2.2 GHz
4×8-core Xeon E7-4820, Westmere-EX, 2.0 GHz
4×8-core AMD Opteron 6378, Abu Dhabi, 2.4 GHz
application benchmarks

- Same benchmarks as in Lozi et al. (RCL) [USENIX ATC’12]
  - programs that spend large % of time in critical sections
- Except BerkeleyDB - ran out of time
RCL experienced correctness issues above 82 threads.
application benchmarks

- comparing best performance (any thread count) for all methods
- up to 2.5x improvement over pthreads, any thread count
- 10+ times speedup at max thread count
telling a very different story. Show how performance varies with the number of threads, ideal thread count for the respective method, Figures 5–6.

Furthermore, this improvement is further improved upon the delegation gains already demonstrated by RCL.

This technique does not pose on all methods evaluated.

Nevertheless, for fairness, the same 25-container load is used for all benchmark applications evaluated, on all platforms tested, including delegating to memory intensiveness.

We find that for all benchmark applications evaluated, on all platforms tested, including delegating to memory intensiveness.

Thus, this small added delay has no significant impact on performance in the absence of back-to-back acquisitions. Thus, back-to-back acquisition is very common, which greatly distorts the performance of some of the simpler locks. This back-to-back acquisition is very common, which greatly distorts the performance of some of the simpler locks.

We use a delay loop of 25 cycles of delay on our Xeon machines, or 500 cycles of delay on our Xeon machines, or 500 cycles of delay on our Xeon machines, or 500 cycles of delay on our Xeon machines, or 500 cycles of delay on our Xeon machines.

We see that the performance loss between the memslap [6] client and the memcached server, but could not isolate the fault further.

RCL experienced correctness issues above 24 threads. We did not get Flat Combining to work.
microbenchmarks
• ffwd is much faster on largely sequential data structures
  • linked list (coarse locking), stack, queue
  • fetch and add, for few shared variables

• for highly concurrent data structures, ffwd falls behind when the lock contention is low
  • fetch and add, with many shared variables
  • hashtable

• for concurrent data structures with long query times, ffwd keeps up, but is not a clear leader
  • lazy linked list
  • binary search tree
naïve 1024-node linked-list, coarse-grained locking

Figure 10: Average throughput of two-lock queue with different synchronization and lock-free methods (higher is better).

Figure 11: Average throughput of stack with different synchronization and lock-free methods (higher is better).

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.

Figures 10–11 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 a 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

![Graph showing throughput vs hardware threads for different methods](image)
two-lock queue
fetch-and-add, 1 variable

Throughput (Mops) vs. Hardware threads

- FFWD
- MCS
- MUTEX
- TTAS
- TICKET
- CLH
- TAS
- HTICKET
- FC
- RCL
- ATOMIC

Hardware-provided atomic increment instruction!
• ffwd is much faster on largely sequential data structures
  • naïve linked list, stack, queue
  • fetch and add, for few shared variables

• for highly concurrent data structures, fwd falls behind when there are many locks
  • fetch and add, with many shared variables
  • hashtable

• for concurrent data structures with long query times, ffwd keeps up, but is not a clear leader
  • lazy linked list
  • binary search tree
128-thread hash table

Throughput (Mops)

#buckets

FFWD
MCS
MUTEX
TTAS
TICKET
CLH
TAS
HTICKET

locking takes the lead when #locks ~ #threads
fetch-and-add, 128 threads
fetch-and-add, 128 threads

Throughput (Mops) vs. 

- FFWD
- MCS
- MUTEX
- TTAS
- TICKET
- CLH
- TAS
- HTICKET
- ATOMIC
- RCL

atomic increment —>
ffwd is much faster on largely sequential data structures

- naïve linked list, stack, queue
- fetch and add, for few shared variables

for highly concurrent data structures, once lock# is similar to thread#, fwd falls behind

- fetch and add, with many shared variables
- hashtable

for concurrent data structures with long query times, ffwd keeps up, but is not a clear leader

- lazy linked list
- binary search tree
Throughput (Mops)

#elements

Throughput (Mops)

128-thread lazy concurrent lists
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.

Figure 13 shows the performance of the lazy linked list, in an experiment that uses the same settings as Figure 12. 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 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 14 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 15 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. $O(\log N)$ difference in computational complexity. We did not evaluate the performance of concurrent skip lists [34, 46, 81, 86].

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.
binary search tree

- simple, unbalanced tree
- 50% queries, 50% updates
- all tree operations delegated for ffwd/RCL
128-thread binary search tree

Throughput (Mops/s)

Initial Size

ffwd is bounded by single-threaded performance
4.3.4 binary search tree

Figure 16 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], SWISSTTM [28] and VTree [69] programs, while the VRBTree [69] 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 17 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 SWISSTTM 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 SWISSTTM, 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 manipulate 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.
what makes ffwd so fast?

- requests are virtually un-contended, contiguous in memory
  - = happy server hardware pre-fetcher
- buffered responses on server
  - 15 responses in one contiguous copy
- 2 modified cache-lines instead of 15
- responses are read-only on the client
  - response line never leaves the server L1
- very light-weight processing on the server
  - plenty of hand-tuning
Why isn’t it even faster?

- Link bandwidth is 300 cache lines per link $\rightarrow$ **300+ Mops**

- Latency suggests 2.5 Mops/client. 120 clients $\rightarrow$ **300 Mops**

- Why are we only seeing 55 Mops?

- Processing limit? 55 Mops = 40 cycles per operation

- Insufficient concurrency: round-trip bandwidth-delay product is 120 cache lines
  - server store / load buffers, reorder window size?
using ffwd

- Free C library available now (Rust is on the way)

- Some current limitations:
  - delegated functions cannot, in turn, delegate functions
  - delegated functions typically should not block (nor acquire locks)
  - up to 6, 64-bit parameters
  - currently assume one client per hardware thread
related work

- Remote Core Locking [Lozi, USENIX ATC’12]
- Barrelfish - delegation-based OS [Baumann, SOSP’09]
- Flat Combining [Hendler, SPAA’10]
- Log-based node replication [Calciu, ASPLOS’17]
in conclusion

• delegation is (much) faster than you thought

• it is easy to use, and has many attractive applications

• similar results on
  Intel Broadwell
  Intel Sandy Bridge
  Intel Westmere-EX, and
  AMD Abu Dhabi
questions?

- UIC has many open CS faculty positions this year, all areas

- libffwd, extended paper and more
  http://github.com/bitslab/ffwd
## comparing parallelism

<table>
<thead>
<tr>
<th>critical section</th>
<th>locking</th>
<th>delegation</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication latency</td>
<td>#locks</td>
<td>#servers</td>
</tr>
<tr>
<td></td>
<td>#locks</td>
<td>#clients</td>
</tr>
</tbody>
</table>