Safe, Fast Sharing of memcached as a Protected Library

Chris Kjellqvist
ckjellqv@u.rochester.edu
Department of Computer Science
University of Rochester

Mohammad Hedayati
hedayati@cs.rochester.edu
Department of Computer Science
University of Rochester

Michael L. Scott
scott@cs.rochester.edu
Department of Computer Science
University of Rochester

ABSTRACT

Memcached is a widely used key-value store. It is structured as a multithreaded user-level server, accessed over socket connections by a potentially distributed collection of clients. Because socket communication is so much more expensive than a single operation on a K-V store, much of the client library is devoted to batching of requests. Batching is not always feasible, however, and the cost of communication seems particularly unfortunate when—as is often the case—clients are co-located on a single machine with the server, and have access to the same physical memory.

Fortunately, recent work on *protected libraries* has shown that it is possible, on current Intel processors, to amplify access rights quickly when calling into a specially configured user-level library. Library instances in separate processes can then share data safely, even in the face of independent process failures. We have used protected libraries to implement a new version of memcached in which client threads execute the code of the server themselves, without the need to send messages. Compared to the original, our new version is both significantly simpler, containing 24% less code, and dramatically faster, with a 11–56× reduction in latency and a roughly 2× increase in throughput.

CCS CONCEPTS

• **Information systems** → *Key-value stores*; • **Software and its engineering** → **Access protection**; Client-server architectures.

KEYWORDS

key-value store, protected library, cross-application sharing, memory protection keys

ACM Reference Format:
Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Safe, Fast Sharing of memcached as a Protected Library. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3404397.3404443

1 INTRODUCTION

The ubiquitous client-server model distinguishes strongly between client processes, which request a service, and server processes, which provide it. Any resource on which the server depends for its correct operation is generally mapped into the server's address space only; to access the resource, clients must make structured requests to the servers, typically via socket communication. Server threads listen on those sockets; receive, unpack, and validate requests; perform requested operations; and return results, again via socket communication. Because the server and client are isolated from one another, a correctly functioning server can enforce access rights, check the consistency of arguments, and ensure that every operation preserves resource invariants and executes atomically, even in the face of client crashes.

Unfortunately, socket communication can add substantial overhead to simple services, even when no physical network is traversed. On a recent Intel box in our lab, datagram messaging over Unix domain sockets incurs a minimum round-trip latency of 3.3—9.6 μs, depending on which cores or hyperthreads are running the communicating threads.

Key-value stores are a case in point. The widely used memcached [21] is organized as a process containing an adjustable number of server threads that communicate with clients over sockets. Socket communication allows the server to be used in a data center, where it runs on a different machine from most of its clients. Memcached is also widely used, however, in more local environments, where it shares a single multicore machine with its clients. In such an environment, the latency of socket communication overwhelms that of the actual service, which is essentially a hash table lookup. Moreover much of the complexity of the memcached code base—roughly a fifth—is devoted to communication management. This, too, seems like a waste in the local case.

Concerns over the cost of accessing services are not new. They contributed heavily to the debate over microkernels almost 30 years ago [31]. Researchers sought to increase modularity, reliability, security, and maintainability by moving significant functionality out of the operating system kernel and into user-space servers. Communication with those servers was still mediated by the kernel, however, making the baseline overhead of calls reflected to a server roughly double that of a call that was handled in the kernel. Whether this overhead was significant, and whether it was overshadowed by other issues was hotly debated, but developers largely voted with their feet, and monolithic kernels still dominate today.

Several recent systems have sought to achieve the modularity benefits of microkernels at lower cost. Dune [1] uses hardware virtualization to run each user process in a separate virtual machine, giving it direct access to protected hardware features under control of the hypervisor. Arrakis [27], inspired in part by Dune, separates "control plane" and "data plane" operations in the I/O system, and leverages single-root I/O virtualization (SR-IOV) [15] hardware to allow applications to interact directly with memory-mapped devices. Similar functionality is provided by a variety of other recent

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404443

systems [13, 14, 18, 23]. IX [2] and ZygOS [29] build upon Dune to provide this same direct-to-device functionality while maintaining a protection boundary between the application and the I/O library. Snap [24] achieves similar protection without virtualization by dedicating one or more cores to actively spinning server threads, which scan shared in-memory queues for client requests.

Most of these recent systems have focused on performance for individual applications, with limited attention to cross-application sharing or system-wide resource management (e.g., to enforce quality of service guarantees). IX and Zygos could potentially accommodate sharing, and Snap already addresses QoS, but each has limitations (its dependence on virtualization or on dedicated cores), and the other systems would be hampered by their lack of a protection boundary between the library and the application.

More recently, our work on the Hodor project [12] has shown how to leverage protection key hardware on recent Intel processors to implement low-overhead *protected libraries* without the need for virtualization. We noted that such libraries can be used for safe, cross-process sharing; we used this capability to implement shared access to the Silo library database [32] and to Intel's DPDK networking library [13]. Similar functionality is provided (in a similar way) in the concurrently developed ERIM project [34], though the authors focus on applications to intra-process sandboxing for security. Protection between the library and the application might also be provided by a trusted compiler [20, 37] or through source or binary rewriting [33, 36], but these impose significant instrumentation costs throughout a program's execution.

We observe that any mechanism that allows a library database or network stack to share data safely across applications can be used to convert a server like memcached to provide safe, direct, function-call access to clients rather than requiring them to communicate over sockets. We have performed this conversion on memcached. The conversion eliminates not only the cost of message packing, transmission, and unpacking, but also the cost of context switches: application threads perform operations on the K-V store themselves.

After a review of Hodor in Section 2, we describe our variant of memcached in Section 3, including its approach to memory management and position independence, the interface provided to applications, the integration with Hodor, and fault tolerance. Section 4 then presents experimental results. For the Yahoo! Cloud Service Benchmark (YCSB) [7], we measure a reduction in latency of 11–56× and a roughly 2× increase in throughput. When use is confined to a single multicore machine, we were also able to eliminate about 26% of the code base, while adding about 2% new code. After returning briefly to related work in Section 5, we summarize conclusions in Section 6, and consider future work, including hybrid (sharing + communication) models and the use of persistent memory.

2 HODOR

Full details on the Hodor system can be found in a previous paper [12]; code is available at http://github.com/hedayati/hodor. We review essential concepts here (see Figure 1).

The key idea in Hodor is to control access to a memory-mapped resource by making it accessible only while executing trusted library code. An application gains access to the resource when it calls into a Hodor library; it loses access when it returns. Threads that are executing outside the library are unable to access the resource even when other threads are executing inside the library. Moreover the operating system arranges for each library call to complete (with certain limits on execution time) even if the process to which its thread belongs crashes due to activity in another thread; this allows the library to ensure integrity in the face of independent client failures. (A crash that occurs *inside* library code is considered unrecoverable.)

The Hodor paper explored three ways to build protected libraries. Our experiments here use the preferred implementation, which is based on the memory protection keys (Protection Keys for Userspace—PKU) of recent Intel processors. PKU harvests four previously unused bits in each page table entry to associate one of 16 "key" values with the page. A new 32-bit `pkru` register, writable in user space, associates two bits with each key. These bits allow the running thread to reduce permissions for pages marked with that key: (0,0) allows whatever access is otherwise permitted to the page; (0,1) eliminates write access; (1,∗) eliminates all access.

PKU appears to have been designed as a safety feature: it allows an application to minimize the impact of stray pointer or array subscripting bugs by turning off access to critical data structures when they are not being actively used. Hodor arranges for real protection by controlling the circumstances under which the `pkru` register can be written. Specifically, a modified version of the OS's (trusted) loader scans the binary of an about-to-be-executed program. It dynamically links the code of any specified Hodor libraries for which the application has access rights. For each library entry point, it installs a *trampoline* that changes stacks and uses the `wrpkru` instruction to change access rights—dropping restrictions on the protected resource on the way in and re-enabling them on the way out. It also installs an initialization routine, called before `main`, that enables restrictions at startup.

If the `wrpkru` opcode appears anywhere in the binary *other* than a trampoline, the loader places a hardware breakpoint at that address, to prevent its execution. Extensive scans of existing binaries confirm that stray instances are extremely rare. They can be avoided entirely with minor compiler changes. If more than four (the number of breakpoint registers) appear in any one program, they can also be accommodated (at some cost) by changing page permissions.

In addition to changing stacks and the value of the `pkru`, trampolines can optionally copy arguments and return values into and out of the library, rather than leaving them in the application's main protection domain (where they could potentially be modified by non-library threads while the library is using them). We do not enable this option by default in our experiments; rather we copy, manually, only those arguments that are security-sensitive; more on this in Sections 3.3 and 3.4.

An empty call into a Hodor library takes about 40 ns on the machine in our lab, round trip—about two orders of magnitude faster than an empty messaging round trip on Unix domain sockets.

3 IMPLEMENTATION

In converting memcached from a socket-based application to a Hodor protected library, there is actually much more code removal

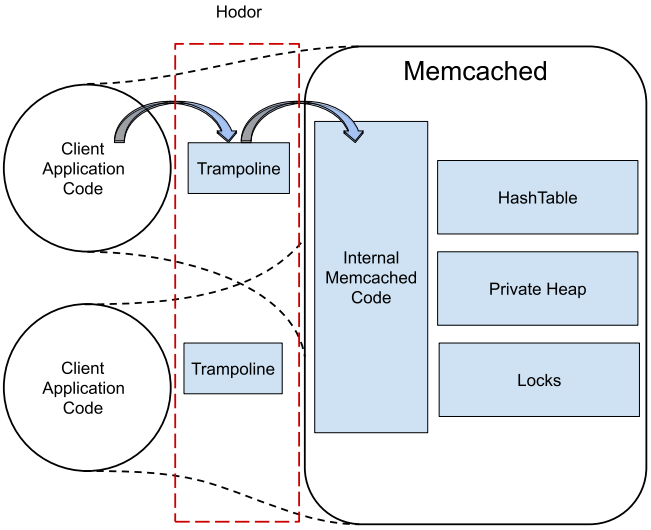

Figure 1: Protected libraries using Hodor

than there is code addition. Standard memcached is a large program (approx. 26 KLoC) in part because of its customizability and flexibility. Clients can issue requests, for example, in either an ASCII based, readable format or a compact, binary format: one offers superior debugability, the other better performance. Without a network interface, the ASCII format loses its attraction. Call parameters will never need to be viewed in a text editor (as they might when debugging the distributed version): they will be viewed in a symbolic debugger, where binary format is perfectly acceptable.

3.1 Interface

We implemented a modified version of the `libmemcached` API. Each call takes as an argument a `memcached_st`, which includes server information, protocol details, and the state of the current operation, none of which are required for direct-through-Hodor calls. We therefore provide two separate APIs—one that is identical to memcached's and one that omits the `memcached_st` argument. Preservation of the original interface allows us to use our modified memcached as a drop-in replacement in existing applications; provision of the newer API allows a modified application to avoid unnecessary overhead. Calls designed, in the original interface, to change the network protocol configuration are now treated as no-ops by default; alternatively, they can be flagged as errors in order to facilitate migration to the newer interface.

Memcached also provides an asynchronous API designed to hide the latency of socket-based communication. A programmer can issue a query with a callback function that will be invoked when data is returned. While callbacks could, in principle, be added to a system like Hodor, they are not supported at present, and it would not be trivial to add them. Fortunately, they are not needed for memcached: since all calls complete immediately, our version of `libmemcached` can call into Hodor for service and invoke the callback immediately after the trampoline returns.

3.2 Memory Management

Hodor manages the protection boundary between an application and its shared libraries, but it does not automatically arrange to share space among library instances in separate applications. For that we need to address a variety of challenges, including set-up and clean-up; cross-process synchronization; static and dynamic memory allocation; and background, bookkeeping tasks. For several of these challenges we rely on the Ralloc memory allocator of Cai et al. [3] (code available at http://github.com/qtcwt/ralloc).

Ralloc provides a *shared heap* abstraction on top of a shared, memory-mapped file. To create a key-value store (an instance of memcached), we launch a *bookkeeping* process that uses Ralloc to create a shared heap, or to map it into memory if its file already exists. (Ralloc supports the ability to have multiple shared heaps, but we only need one for our experiments.) The bookkeeping process remains alive as long as its K-V store is in use. During operation, it is responsible for intermittent "cleaning" of the store—eviction of less-needed items when space runs low. On shutdown, it flushes all updates back to the underlying file. This convention allows us to restart a store with its contents already intact. (Enhancements to allow the store to survive full-system crashes are a subject of future work; see Section 6.)

Each memcached client, on startup, uses Ralloc to map the shared heap into its own address space. While we might hope to map the heap to the same address in every client process, it is difficult to do so in practice, given the possibility that any given address range might be needed by some client for other purposes. Fortunately, Ralloc provides a *persistent pointer* (`pptr`) abstraction that enables the creation of *position-independent data*. In C++, the `pptr` type is implemented as a templated smart pointer that holds the signed distance between its own location and that of its target [5, 6]. So long as the target resides in the same shared heap, a `pptr<T>` can be loaded into or stored from a `T*`, quickly and correctly, in any address space.

We use Ralloc for all dynamic allocation of buckets, keys, and values. Internally, Ralloc uses `pptr`s for all its metadata; we converted memcached to use them for all pointers in the K-V store as well.

Because threads in multiple address spaces now access the K-V store directly, synchronization must work across process boundaries. We therefore updated the initialization options on all locks in the memcached code base to specify the `PTHREAD_PROCESS_SHARED` attribute. Locks used to protect metadata in a conventional allocator would also need to be shared, but Ralloc, it turns out, is entirely nonblocking. It also scales extremely well, due in large part to the extensive use of per-thread caches, and it partitions blocks of different sizes into separate *superblocks*, leading to low internal fragmentation and no external fragmentation for the block sizes used in memcached. This efficient space management obviates the need for memcached's own "slab-based" allocator, which we deleted.

As a starting point for data structure access (in our case, for access to the K-V store), Ralloc supports the notion of *persistent roots*. They are statically allocated near the beginning of the shared heap, and contain `pptr`s to internal structures. Each is identified by a symbolic ID.

```
// in global scope
// doesn't need extra indirection
// pthread_mutex_t lru_locks[POWER_LARGEST];
pthread_mutex_t* lru_locks;
...
// on initialization of file
lru_locks =
    (pthread_mutex_t*)pm_calloc(POWER_LARGEST,
    sizeof(pthread_mutex_t));
pm_set_root(lru_locks, RPMRoot::LRULocks);
...
// on restart
lru_locks = pm_get_root<pthread_mutex_t>
    (RPMRoot::LRULocks);
```

Figure 2: Sharing objects with a fixed location

Figure 2 illustrates the use of persistent roots for data structures that are statically allocated in the Ralloc heap. Memcached's `lru_locks`, originally a static array of `pthread_mutex_t` objects, is now, in our version of the code, a pointer to an array of such objects. When the memcached bookkeeping process starts up, it allocates this array in the Ralloc heap using `pm_calloc` and sets a persistent root to point at it. When a memcached client process starts, it uses `pm_get_root` to obtain (an address-space-appropriate version of) the pointer stored in the persistent root.

The `lru_locks` are used to protect least-recently-used lists that allow the background process to choose victims to evict from the hash table when space runs low. The original version of memcached places items into an LRU list based on the size class to which they belong in the server's custom allocator. Because we now use a separate allocator, we chose to decouple the LRU functionality from the allocator internals. We tried putting all items into a single list, but this caused unacceptable lock contention at high thread counts. We currently use a set of lists, and choose among them (and their locks) based on the hash of an entry's key.

We encountered similar trouble with contention on the lock used to protect statistics on client requests. We therefore chose to scatter these statistics across the slots of a shared array. Most updates are now made to a slot that is not being used concurrently. Statistics-retrieving calls must scan the whole array. Neither the `lru_locks` nor the statistics lock is a bottleneck in the original memcached code, where requests are serviced less frequently and where the maximum number of active threads is smaller.

To access data structures that may be reallocated during execution, persistent roots can be used with an extra level of indirection. Figure 3 illustrates this idiom for the root of the primary hashtable, whose location may change due to occasional table resizing.

In practice, not all data need to be shared between processes. The threads of the bookkeeping process, for example, maintain information that is not needed by clients during library calls. Memcached also makes use of some temporary buffers during individual calls. These can be local to a single client, so long as they reside in the `libmemcached` protection domain in Hodor, to prevent concurrent access by threads executing in the main code of the client application.

```
// item **hashtable = 0;
pptr<pptr<item>> *hashtable_storage = 0;
...
// on initialization, allocate storage,
// and store hashtable there
hashtable_storage =
    pm_malloc(sizeof(pptr<pptr<item>>));
*hashtable_storage = (pptr<item>*)
    pm_calloc(hashsize(hashpower),
    sizeof(pptr<item>));
pm_set_root(hashtable_storage,
    RPMRoot::PrimaryHT);
...
// on restart, the table already exists;
// fetch it
hashtable_storage = (pptr<pptr<item>>*)
    pm_get_root<pptr<pptr<item>>>
    (RPMRoot::PrimaryHT);
```

Figure 3: Sharing objects whose location may change

3.3 Integration with Hodor

To integrate our API with Hodor, we identified the internal functions associated with each API call and wrapped them with Hodor trampolines; these in turn are tagged with special macros in the source code.

Figure 4 shows our implementation of `memcached_get`. Memcached incorporates significant machinery to parse and execute commands. These two tasks are deeply interwoven—a fact that made it difficult for us to understand, "unpackage," and replicate the API. The task would presumably have been easier if writing a Hodor application from scratch, because commands would never have been marshalled into network packets.

At the same time, the use of Hodor introduces certain security concerns. Specifically: no privileged information in the library should be leaked into client code, and no data created by the client should be trusted by the library. If arguments to library calls have internal consistency requirements, they should be copied to space that is not writable by the client before performing consistency checks, to prevent concurrent corruption by other client threads. This idiom can be seen in Figure 4, where we copy the client's key into a buffer (`key_prot`) that we have created *inside* the library. By contrast, the return value `buffer` is allocated using the standard `malloc`, so it will be visible after returning to the client.

Hodor relies on file system permissions to control the mapping of shared libraries. In our case, it executes the `libmemcached` initialization routine with the effective user ID of the memcached bookkeeping process, allowing it to open and map the file containing the K-V store. Once initialization is complete, it reverts the effective ID to that of the client process. These conventions allow us to share data across protected library instances without leaking the contents of the actual K-V store file.

3.4 Fault Tolerance

Our code must take care to ensure that faults experienced in one process do not affect the integrity of the library or of other processes.

```
HODOR_FUNC_ATTR
char *
memcached_get_internal
  (const char * key, size_t key_length, size_t
      *value_length, uint32_t *flags,
   memcached_return_t *error){
  assert(run_once && "You must run memcached_init
      before calling memcached_functions");
  char* buff = NULL;
// Can't use user pointers inside
// protected function. Copy to private
// buffer before resources are acquired
  char * key_prot = (char*)RP_malloc(nkey);
  memcpy(key_prot, key, nkey);
// Get the item using the protected key
  item* it = item_get(key_prot, nkey, 1);
// We don't need the key anymore
  RP_free(key_prot);
  if (it == NULL){
    *error = MEMCACHED_NOTFOUND;
  } else {
// We found the item. Need place to
// copy its data to protected buffers
// while important resources are held
// (ie ref counts)
    char * dat_prot = (char*)RP_malloc(it->nbytes);
    memcpy(dat_prot, ITEM_data(it), it->nbytes);
    size_t flag_prot = it->it_flags;
    size_t buffLen_prot = it->nbytes;
// release our resources
    item_remove(it);
// allocate output buffer
    buffer = (char*)malloc(buffLen_prot);
// copy protected data to
// user-accessible locations
    *buffLen = buffLen_prot;
    *flags = flag_prot;
    memcpy(buffer, dat_prot, *buffLen);
// free data buffer
    RP_free(dat_prot);
    *error = MEMCACHED_SUCCESS;
  }
  return buff;
}
HODOR_FUNC_EXPORT(memcached_get_internal, 5);
```

Figure 4: `memcached_get` using Hodor

The original memcached uses locks to ensure that server threads perform their operations atomically. In our version the same code is executed by client threads. Each operation—assuming it completes—completes atomically. The only new issue is the potential for failure in the middle of an operation.

If the process of a Hodor thread is terminated by action outside the library—a `SIGKILL` signal, for example, or a segmentation fault in a concurrent thread in the client's regular code—Hodor allows the thread in the library to continue running until it has completed its call or a generous timeout has expired.

If a Hodor thread encounters an error of its own—a segfault for example—it will indeed terminate abruptly. Protected libraries, like system call handlers, must be carefully written to avoid this possibility, or to ensure that termination happens only when no locks are held and all invariants still hold. Pointers received from the client are the most common source of potential problems. They cannot safely be dereferenced while locks are held. (Even if they could be verified before use, it would always be possible for a concurrent thread to unmap the target memory while the library was active.) As a standard practice, we therefore copy the targets of all pointers into memory located in a Hodor protected region before acquiring locks or performing other changes to shared state. In Figure 4, `key_prot` is used for precisely this purpose. For our final allocation, `buffer`, which we return to the client program, we use ordinary `malloc`, but only *after* releasing all resources that we acquired.

4 EXPERIMENTAL RESULTS

We used the Yahoo! Cloud Service Benchmark (YCSB) [7], a workload generator for databases, to test the performance of our protected library implementation against the original version of memcached. YCSB allows users to select from default workloads or to generate their own with specific chosen properties. We created a set of workloads that test the performance of our key-value store on value sizes of 128 bytes and 5 kilobytes, with read/write distributions of 50/50 and 95/5. Our results were collected on a single-socket Dell machine with a 10-core (20-hyperthread) Intel Xeon Gold 5215 processor equipped with 192 GB of DRAM and running at 2.5 GHz.

Since writes are uncommon compared to reads in production environments, we consider a 50/50 split between reads and writes to be a "write heavy" workload; the 95/5 case is considered "read heavy." For workloads with values of size 128 B, we store 4×10^7 key-value pairs and perform 10^6 operations on those pairs. For workloads with values of size 5 KB, we store 10^6 key-value pairs so that the total memory consumption of the application remains about the same. Operations were performed with a Zipfian distribution over the keys. Latency is reported in μs for operations in a single thread. Throughput is reported in thousands of transactions per second (KTPS), so higher numbers are better.

For the original memcached, we set the maximum data size to 60 GB; we provide the same limit to Ralloc in our modified version. In the original version, the hash table starts with 2^{16} buckets and resizes several times. In our modified version, we report results for a fixed size of 2^{25} buckets (our resizing code in the background process is not yet working correctly). If anything, this decision penalizes our code: in the small-key experiments, our table ends up with a load factor of about 1.2.

4.1 Latency and Bandwidth

Once a request is received, the original memcached and our modified version perform the same internal operations. The main difference is that in the absence of socket communication, the overhead of initiating a call in our version is much lower. As shown in Figure 5, calls in the original memcached vary from 13–54 μs; in our version they never take more than 1.5 μs. These numbers represent speedups of 11–56×.

	Memcached	Plib, w/Hodor	Plib, No Hodor
Get 128 B	13 μs	0.67 μs (19×)	0.64 μs (20×)
Get 5 KB	13 μs	0.67 μs (20×)	0.64 μs (21×)
Set 128 B	13 μs	1.2 μs (11×)	1.2 μs (11×)
Set 5 KB	17 μs	1.5 μs (11×)	1.5 μs (11×)
Delete	10 μs	0.21 μs (48×)	0.18 μs (56×)
Increment	54 μs	1.6 μs (34×)	1.5 μs (36×)

Figure 5: Operation latency and speedup

When measuring throughput scalability, there is a fundamental mismatch between the original and shared-library versions of memcached: since server and client threads are distinct in the original, their numbers can vary independently; in our version they are always the same. In an attempt at a fair comparison, we vary the number of client threads on the X axis in Figures 6–9, and show two curves for the original memcached: one with 4 server threads and one with 8.

With 4 server threads, the original memcached scales linearly to 10 clients—the number of cores on the machine. The addition of hyperthreads has no significant impact, suggesting that the server threads may be a bottleneck. To test this hypothesis, we ran the experiment again with 8 server threads. Performance for client counts up to 10 is virtually identical to the 4-server-thread case, but it continues to scale, at a slower rate, all the way out to the tested limit of 40 clients—2 for every hyperthread on the machine.

The explanation, we believe, lies in an understanding of the critical path of the microbenchmark. When a server thread completes a request in the original code, it calls into the operating system kernel to perform a `write` on a socket. It then immediately performs a `select` syscall to obtain another request. Whether that call returns immediately or waits (incurring a context switch to another process) depends on whether another client has already performed its matching `write`. As the number of client threads increases, the odds that one of them has performed a `write` on a socket in the `select` set gradually increases, increasing the probability that the kernel can simply return into the server, rather than switching to a client context.

In our new, protected-library version of memcached, by contrast, client threads perform their own requests. There are no system calls on the critical path, and the overall system bottleneck becomes the synchronization employed in hash table critical sections. In the read-heavy workload, throughput peaks at 6–8 threads. In the write-heavy workload, where the average operation takes a little longer, throughput peaks at slightly fewer transactions per second, and takes a few more active threads to get there. In all cases (large and small values, read- and write-heavy workload), bandwidth degrades slightly as contention increases, out to the number of hyperthreads on the machine (i.e., 20), but remains essentially flat thereafter, at roughly 2× the throughput of the original memcached. At peak throughput, the protected library reaches 3× the throughput of the original memcached. To assess the marginal overhead of the protected library mechanism, we have shown results both with and without Hodor protection. The version without improves throughput by roughly 5%, but of course it is not safe.

4.2 Code Complexity

As noted in Section 3, our updates to memcached deleted more code than they added. Specifically, on an original base of ~26 K lines of code, we removed ~6800 lines and added ~600, for a net reduction of ~24%. Of the deleted lines, ~5200 were devoted to socket communication and to packing and unpacking of message buffers; ~1600 were devoted to slab-based memory management (Section 3.2).

With regard to conceptual complexity, our subjective impression is that the removed code was more complex than the added code. With Hodor, a thread performs its own request, and the flow of control is very clear. In a separate server, threads must keep track of multiple client connections, `select` from among their sockets, unpack request buffers, and pack reply buffers for return.

On the other hand, protected libraries in Hodor are significantly more subtle than "ordinary" libraries—particularly when they share data with instances in other processes. As noted in Section 3.4, a Hodor library routine is more akin to a kernel-level syscall handler than it is to an ordinary function: it must check its arguments for consistency, manage its space separately from that of the caller, and ensure that faults (e.g., due to incorrect pointers to client data) never occur while holding locks on shared data. After modifying the first few library routines, we settled on an idiom that copies client data into Hodor-allocated buffers before acquiring any locks.

5 RELATED WORK

Over the past decade, extensive research has aimed to limit the operating system's involvement in resource management to what is required for set-up and access control (i.e., the *control plane*), and to avoid OS intervention on individual operations (i.e., the *data plane*) once that access has been granted. Specialized I/O stacks (e.g., DPDK [13], SPDK [14], and mTCP [18]) have been designed to bypass the OS and avoid the overhead of the general-purpose system-call path on data transfers. Similarly, our work relies on the OS to set up page table permissions for protected libraries, while avoiding per-op OS interventions when querying memcached.

Unlike our work, most previous efforts at avoiding OS intervention have not considered the possibility of sharing resources across distrusting domains. Our work allows for safe sharing of memcached among independently developed local processes with independent failure modes.

While intra-address space isolation [33, 34, 36] has been used to protect secrets (e.g., encryption keys) or security critical regions (e.g., shadow stacks) in a single application, researchers have only recently begun to consider isolation for libraries. Hodor [12] protects user-space libraries for kernel-bypass I/O and in-memory databases. Treasury [8] uses Intel PKU to protect a user-space non-volatile memory file system called ZoFS. Our work transforms memcached from a client/server model to a protected library and uses Hodor to provide both protection and sharing.

Exokernel [10] was one of the first projects to compile OS components as libraries and link them to applications as a *Library OS*. Other such systems include Drawbridge [28] and EbbRT [30]. Separation of components from the kernel offers the prospect of better system security and more rapid evolution of the code base. Unfortunately, library OSes provide no easy way to share the same set

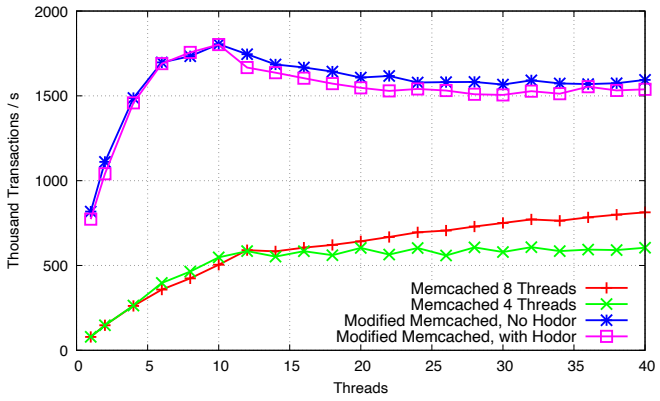

Figure 6: Field length 128 B – Write Heavy

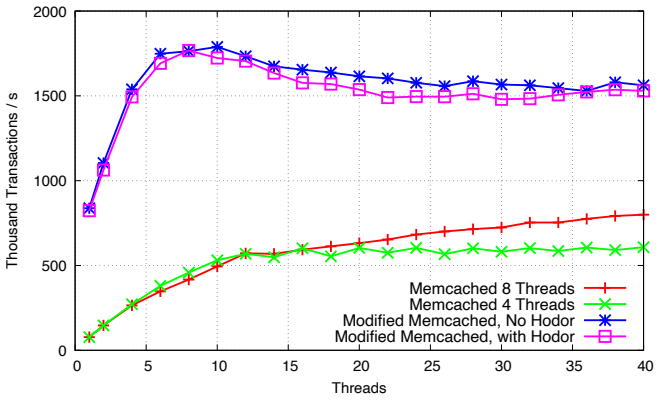

Figure 7: Field Length 5 KB – Write Heavy

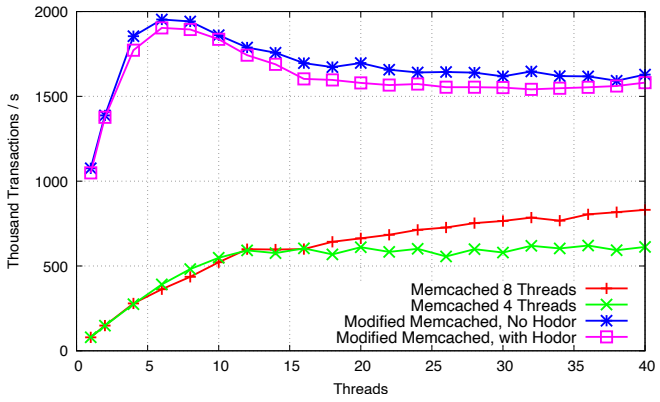

Figure 8: Field length 128 B – Read Heavy

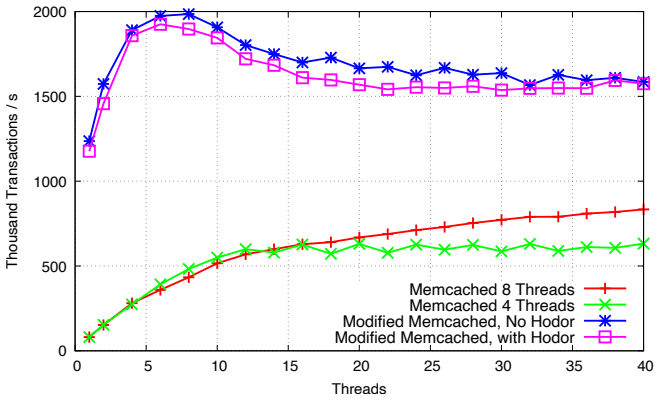

Figure 9: Field length 5 KB – Read Heavy

of resources across multiple instances of the exported components. Microkernels [9, 19], several of which pre-date Exokernel, avoid this problem by separating address spaces and following a client/server model in which each resource is handled by a user-level server. The main drawback of microkernel design has long been the overhead of communication [25]. While at a higher abstraction level than a traditional OS service, current memcached deployments follow the microkernel pattern. We see recent work on protected libraries, including ours, as a means of combining the isolation and sharing benefits of microkernels with the performance gains of exokernels.

6 CONCLUSIONS AND FUTURE WORK

We have presented a new, shared-memory version of the ubiquitous memcached key-value store. Our code is based on the Hodor protected library system [12] and the Ralloc memory allocator[3]. For clients running on the same machine as the server, we achieve an 11–56× improvement in latency and a roughly 2× improvement in throughput compared to communicating with the standard version of the server over Unix domain sockets. In addition to offering better performance, our version is significantly simpler, suggesting that similar servers, written from scratch, would be easier to build using Hodor as a base.

Because our modified memcached only can interact with processes on the same node, it is no longer a *distributed* key-value store. There is no reason, however (other than code complexity), not to allow the memcached background process to provide a socket-based interface for remote clients while still permitting local clients to use the Hodor interface. We intend to support this option in a future release of the code.

More ambitiously, we are experimenting with the possibility of making memcached resilient to system crashes. As noted in Section 3.2, our Hodor memcached flushes its K-V store contents to the backing file when the bookkeeping process shuts down. Because the data in this file is position independent, it can be loaded back into memory and reused when the bookkeeping process is restarted. Significantly, this reload and reuse adds *no extra code* to the system. It may be particularly appealing when memcached is used not as a cache for off-line data but as a stand-alone in-memory store.

When memcached *is* used as a cache, and when the backing file is located on a magnetic or flash device, reloading may be only slightly faster than rebuilding—it saves the hash table update time, but not the I/O time. On the other hand, if the file is located in nonvolatile,

direct-access (DAX) memory—e.g., on a machine with Intel Optane DIMMs [17]—then a reload can be almost instantaneous: it will consist only of page table updates.

The challenge, of course, is to support stand-alone use in the face of possible crashes, when rebuilding is not an option. In such an environment, operations must be not only isolated and consistent (in traditional database terminology [11]), but also failure-atomic and durable. Various groups are currently exploring mechanisms to provide failure atomicity for lock-based critical sections or transactions [4, 6, 16, 22, 26, 35]; we look forward to leveraging this work.

REFERENCES

[1] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, 335–348.

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 49–65.

[3] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *19th Intl. Symp. on Memory Management*. Earlier version published as arXiv:2003.06718 [cs.DC] and TR 1008, Computer Science Dept, Univ. of Rochester, March 2020. Extended abstract presented at *PPoPP* 2020.

[4] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *ACM Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. Portland, OR, 433–452. https://doi.org/10.1145/2660193.2660224

[5] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-volatile Memory. In *50th IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*. Cambridge, MA, 191–203. https://doi.org/10.1145/3123939.3124543

[6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, 105–118. https://doi.org/10.1145/1950365.1950380

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symp. on Cloud Computing (SoCC)*. Indianapolis, IN, 143–154. https://doi.org/10.1145/1807128.1807152

[8] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *27th ACM Symp. on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, 478–493. https://doi.org/10.1145/3341301.3359637

[9] Kevin Elphinstone, Amirreza Zarrabi, Kent Mcleod, and Gernot Heiser. 2017. A Performance Evaluation of Rump Kernels as a Multi-Server OS Building Block on SeL4. In *8th Asia-Pacific Workshop on Systems (APSys '17)*. Mumbai, India, Article 11, 8 pages. https://doi.org/10.1145/3124680.3124727

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 251–266. https://doi.org/10.1145/224057.224076

[11] Jim Gray. 1981. The Transaction Concept: Virtues and Limitations (Invited Paper). In *7th Intl. Conf. Very Large Data Bases (VLDB)*. Cannes, France, 144–154.

[12] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conf. (ATC)*. Renton, WA, 489–504.

[13] Intel Corp. 2018. Intel DPDK: Data Plane Development Kit. http://www.dpdk.org.

[14] Intel Corp. 2018. Intel SPDK: Storage Performance Development Kit. http://www.spdk.io.

[15] Intel Corp. 2018. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

[16] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, 427–442. https://doi.org/10.1145/2872362.2872410

[17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). http://arxiv.org/abs/1903.05714

[18] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Conf. on Networked Systems Design and Implementation (NSDI)*. Seattle, WA, 489–502.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and et al. 2009. SeL4: Formal Verification of an OS Kernel. In *22nd ACM Symp. on Operating Systems Principles (SOSP)*. Big Sky, MT, 207–220. https://doi.org/10.1145/1629575.1629596

[20] James Larus and Galen Hunt. 2010. The Singularity System. *Commun. ACM* 53, 8 (Aug. 2010), 72–79. https://doi.org/10.1145/1787234.1787253

[21] libMemcached.org. 2011. libMemcached. libmemcached.org/libMemcached.html.

[22] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *51st IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*. Fukuoka, Japan, 258–270. https://doi.org/10.1109/MICRO.2018.00029

[23] I. Marinos, R. N. M. Watson, and M. Handley. 2014. Network Stack Specialization for Performance. In *ACM SIGCOMM Conf.* Chicago, IL, 175–186. https://doi.org/10.1145/2619239.2626311

[24] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. 2019. Snap: A Microkernel Approach to Host Networking. In *27th ACM Symp. on Operating Systems Principles (SOSP)*. Huntsville, ON, Canada, 399–413. https://doi.org/10.1145/3341301.3359657

[25] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-process Communication for Microkernels. In *14th ACM SIGOPS European Conf. on Computer Systems (EuroSys)*. Dresden, Germany, Article 9, 15 pages. https://doi.org/10.1145/3302424.3303946

[26] Faisal Nawab, Joseph Izraelevitz, Terrence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st Intl. Symp. on Distributed Computing (DISC)*. Vienna, Austria, Article 37, 16 pages.

[27] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 1–16.

[28] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. *SIGPLAN Not.* 46, 3 (March 2011), 291–304. https://doi.org/10.1145/1961296.1950399

[29] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *26th Symp. on Operating Systems Principles (SOSP)*. Shanghai, China, 325–341. https://doi.org/10.1145/3132747.3132780

[30] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building per-Application Library Operating Systems. In *12th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, 671–688.

[31] Andrew S. Tanenbaum and Linus Torvalds. 1999. The Tanenbaum-Torvalds Debate. In *Open Sources: Voices from the Open Source Revolution*, Chris DiBona, Sam Ockman, and Mark Stone (Eds.). O'Reilly & Associates. Appendix A. https://www.oreilly.com/openbook/opensources/book/appa.html.

[32] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *24th ACM Symp. on Operating Systems Principles (SOSP)*. Farmington, PA, 18–32. https://doi.org/10.1145/2517349.2522713

[33] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, 75–88.

[34] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *28th USENIX Security Symp. (SEC)*. Santa Clara, CA, 1221–1238.

[35] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, 91–104. https://doi.org/10.1145/1950365.1950379

[36] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *14th ACM Symp. on Operating Systems Principles (SOSP)*. Asheville, NC, 203–216. https://doi.org/10.1145/168619.168635

[37] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. Toronto, ON, Canada, 99–110. https://doi.org/10.1145/1806596.1806610