Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage

Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman, *University of Utah*

https://www.usenix.org/conference/osdi18/presentation/kulkarni

This paper is included in the Proceedings of the
13th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '18).
October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage

Chinmay Kulkarni Sara Moore Mazhar Naqvi Tian Zhang Robert Ricci Ryan Stutsman
University of Utah

Abstract

In-memory key-value stores that use kernel-bypass networking serve millions of operations per second per machine with microseconds of latency. They are fast in part because they are simple, but their simple interfaces force applications to move data across the network. This is inefficient for operations that aggregate over large amounts of data, and it causes delays when traversing complex data structures. Ideally, applications could push small functions to storage to avoid round trips and data movement; however, pushing code to these fast systems is challenging. Any extra complexity for interpreting or isolating code cuts into their latency and throughput benefits.

We present *Splinter*, a low-latency key-value store that clients extend by pushing code to it. Splinter is designed for modern multi-tenant data centers; it allows mutually distrusting tenants to write their own fine-grained extensions and push them to the store at runtime. The core of Splinter's design relies on type- and memory-safe extension code to avoid conventional hardware isolation costs. This still allows for bare-metal execution, avoids data copying across trust boundaries, and makes granular storage functions that perform less than a microsecond of compute practical. Our measurements show that Splinter can process 3.5 million remote extension invocations per second with a median round-trip latency of less than 9 µs at densities of more than 1,000 tenants per server. We provide an implementation of Facebook's TAO as an 800 line extension that, when pushed to a Splinter server, improves performance by 400 Kop/s to perform 3.2 Mop/s over online graph data with 30 µs remote access times.

1 Introduction

Today's model of separated compute and storage is reaching its limits. Fast, kernel-bypass networking has yielded key-value stores that perform millions of requests per second per machine with microseconds of latency [22, 37, 45, 55, 71]. These systems gain much of their speed by being simple, allowing only lookups and updates. However, this simplicity results in inefficient data movement between storage and compute and costly client-side stalls [6, 51]. To efficiently exploit these new stores, applications will be under increasing pressure to push compute to them, but the granularity at which they can do so is a concern. At microsecond timescales, even small costs for isolation, containerization, or request dispatching dominate, placing practical limits on the granularity of functions that applications can offload to storage.

We resolve this tension in *Splinter*, a multi-tenant in-memory key-value store with a new approach to pushing compute to storage servers. Splinter preserves the low remote access latency (9 µs) and high throughput (3.5 Mops/s) of in-memory storage while adding native-code runtime *extensions* and the dense *multi-tenancy* (thousands of tenants) needed in modern data centers. Tenants send arbitrary type- and memory-safe extension code to stores at runtime, adding new operations, data types, or storage personalities. These extensions are exposed so tenants can remotely invoke them to perform operations on their data. Splinter's lightweight isolation lets thousands of untrusted tenants safely share storage and compute, giving them access to as much or as little storage as they need.

Splinter's design springs from the intersection of three trends: *in-memory storage with low-latency networking*, which is driving down the practical limits of request granularity; *massive multi-tenancy* driven by the cloud and the efficiency gains of consolidation; and *serverless computing*, which is already training developers to write stateless, decomposed application logic that can run anywhere in order to gain agility, scalability, and ease of provisioning.

Together, these trends drive Splinter's key design goals:

No-cost Isolation. Since extensions come from untrusted tenants, they must be isolated from one another. Hardware-based isolation is too expensive at microsecond time scales; even a simple page table switch would significantly impact response time and throughput.

Zero-copy Storage Interface. Extensions interact with stored data through a well-defined interface that serves as a trust boundary. For fine-grained requests, it must be lightweight in terms of transfer of control and in terms of data movement. This effectively requires extensions to be able to directly operate on tenant data *in situ* in the store, while maintaining protection and preventing data races with each other and the storage engine.

Lightweight Scheduling for Heterogeneous Tasks. Extensions are likely to be heterogeneous. Some extensions might involve simple point lookups of data or constructing small indexes; others might involve expensive computation or more data. Preemptive scheduling involves costly context switches, so Splinter must avoid preemption in the normal case, yet maintain it as an option to contain poorly-behaving extensions. It must also be able to support high quality of service under heavy skew, both in terms of the tenants issuing requests at different rates and extensions that take different amounts of time to complete.

Adaptive Multi-core Request Routing. With multiple tenants sharing a single machine, synchronization over tenant state can become a bottleneck. To minimize contention, tenants maintain locality by routing requests to preferred cores on Splinter servers. We can't, however, use a hard partitioning, as we don't want high skew to create hotspots and underused cores [58]. Routing decisions can't get in the way of fast dispatch of requests [7].

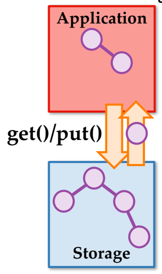
These goals give rise to Splinter's design. Developers write type-safe, memory-safe extensions in Rust [2] that they push to Splinter servers. Exploiting type-safety for lightweight isolation isn't new; SPIN [8] allowed applications to safely and dynamically load extensions into its kernel by relying on language-enforced isolation. Similarly, NetBricks [56] applied Rust's safety properties to dataplane packet processing to provide memory safety between sets of compile-time-known domains comprising network function chains. Splinter combines these approaches and applies them in a new and challenging domain. Language-enforced isolation with native performance and without garbage collection overheads is well-suited to low-latency data-intensive services like in-memory stores — particularly, when functionality must be added and removed at runtime by large numbers of fine-grained protection domains.

Splinter's approach allows it to scale to support thousands of tenants per machine, while processing more than 3.5 million tenant-provided extension invocations per second with a median response time of less than 9 μs. We describe our prototype of the Splinter key-value store and its extension and isolation model. We evaluate it on commodity hardware and show that a simple 800 line extension imbues Splinter with the functionality of Facebook's TAO [10]. On a single store, the extension can perform 3.2 million social graph operations per second with 30 μs average response times, making it competitive with the fastest known implementation [22].

2 Motivation

Splinter's key motivation is the desire to support complex data models and operations over large structures in a fast kernel-bypass stores. Existing in-memory stores trade data model for performance by providing a simple key-value interface that only supports `get` and `put`. Many real applications organize their data as trees, graphs, matrices, or vectors. Performing operations like aggregation or tree traversal with a key-value interface often requires multiple `get`s. Applications are usually *disaggregated* into a storage and compute tier, so these extra `get`s move data over the network and induce stalls for each request.

Figure 1 illustrates this problem with a storage client that traverses data logically organized as a tree. The client must first issue a `get` to retrieve the tree's root node. Next, it must perform a comparison and move down the



```
fn find_in_tree(n: &Node, key: u64)
  -> Option<Value>
{
  if n.key == key { // Found correct value
    Some(n.value)
  } else {
    // Traverse left or right
    let next = if key < n.key { n.left }
               else { n.right };
    if let Some(next) = next {
        // Fetch each node from storage
        find_in_tree(get(next), key)
    } else {
        None // Break if dead end
    }
  }
}
```

Figure 1: Tree traversal using `get()` operations over a key-value store. Each step requires a lookup at the storage layer, which is latency-bound and expensive for deep traversals. If multi-tenant stores could be safely extended this function could avoid remote access stalls and request processing costs.

tree by issuing another `get`. It must repeat this for every step of the traversal. Each `get` incurs a round trip that fetches a single node from storage; since the control flow is dependent on the data fetched, the client can only issue one request at a time. The number of round trips needed is proportional to the tree's depth, and a significant portion of the tree gets moved over the network. Even with modern low-latency networking, latency still dominates the client's performance: network transmission and processing takes tens of microseconds while the actual comparisons take less than a microsecond [55].

One solution is to customize the storage tier of each application to support specialized data types. However, to improve efficiency and utilization, storage tiers are usually deployed as multi-tenant services [14, 19], so they cannot be customized for every possible data structure. SQL could be used at the storage tier, but SQL is known to be a poor fit for data types like graphs and matrices, does not support abstract data types, and is too expensive at microsecond timescales. Instead, Splinter takes a different approach; it allows applications to push small pieces of native compute (extensions) to stores at runtime. These extensions can implement richer data types and operators, avoiding extra round trips and reducing data movement.

2.1 The Need for Lightweight Isolation

Multi-tenancy at the storage layer makes running extensions challenging; a tenant cannot be allowed to access memory it does not own, starve others for resources, or crash the system. The major challenge is that, at microsecond timescales, context switches and data copying across isolation boundaries significantly hurt performance.

To quantify the overhead of hardware isolation, we simulated an 8-core multi-tenant store that isolates extensions using processes while varying the numbers of tenants making requests to it. Simulated requests con-

Xeon Architecture	Context switch delay (µs)	
	Pre KPTI	KPTI
D-1548, Broadwell	1.60	2.40
E5 2450, Sandy bridge	1.50	2.48
Gold 6142, Skylake	1.40	2.16

Table 1: Context switch overhead for different Intel Xeon architectures as measured on CloudLab. Each number represents the median of a million samples. Based on these measurements, we chose 2.16 µs and 1.40 µs for the context switch overhead with and without KPTI in our simulations.

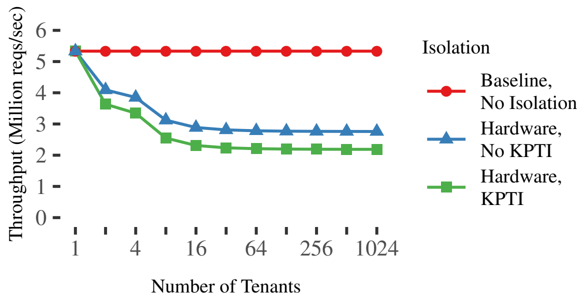


Figure 2: Simulated throughput versus the number of tenants. With hardware isolation, even modestly increasing the number of tenants to 16 (just twice the number of cores) leads to a significant drop in throughput. "No isolation" represents an upper bound where isolation costs are zero.

sume 1.5 µs of compute at the store; this is based on our benchmarks of simple unisolated operations on Splinter (§5.2); our numbers are similar to those reported by others' kernel-bypass stores [55]. Different context switch costs are simulated to show the overheads of hardware-based isolation of tenant code. The simulation only accounts for context switch costs; copying data across hardware isolation boundaries has also been shown to have significant performance costs [56]. Nearly all extensions will access data, which will force data copying when using hardware isolation and hurt throughput further. Based on measurements we made on different processor microarchitectures (Table 1), we simulate 1.40 µs of overhead for a basic context switch and 2.16 µs for a KPTI [16] protected kernel (which mitigates attacks that can leak the contents of protected memory [46]). The request pattern is uniform; all tenants make the same number of requests. The results are similar with skew. The simulator is also optimistic; whenever a request is made and an idle core is available at the store that last processed a request from the same tenant, the isolation cost is assumed to be zero.

Figure 2 presents simulated throughput at different tenant densities. The baseline represents an upper bound where extensions are run un-isolated at the storage system. The simulations show that throughput with hardware isolation (irrespective of KPTI) is significantly lower than the baseline. Even at just 16 tenants, context switch costs alone cut server throughput by a factor of 1.8.

Overall, for these types of fast stores, hardware isolation limits performance and tenant density. The challenges that we face in Splinter, and our design goals, stem from the need to (nearly) eliminate trust boundary crossing costs, to keep data movement across trust boundaries low, and to perform efficient fine-grained task scheduling.

3 Splinter Design

Each Splinter server works as an in-memory key-value store (Figure 3). Like most key-value stores, tenants can directly `get` and `put` values, but they can also customize the store at runtime by installing safe Rust-based extensions (shared libraries mapped into the store's address space) (Figure 3 ①). These extensions can define new operations on the tenant's data, including extensions that stitch together new data models in terms of the store's low-level get/put interface. Each tenant-provided extension is exported over the network, so a tenant can remotely invoke the procedures it has installed into the store.

Tenants send requests to a Splinter store over the network using kernel bypass (②). Splinter currently only supports a simple, custom UDP-based RPC protocol, though other optimized transports may provide similar performance [38]. Each tenant's requests are steered to a specific receive queue by the network card, improving locality (③). Each receive queue is paired with a single kernel thread (or *worker*) that is pinned to a specific core. Each worker pulls requests from its receive queue and creates a user-level task for the requested operation. Tasks provide an accounting context for resources consumed while executing the operation, the storage needed to suspend/resume the operation, and a unit of scheduling. Each worker has a task queue of new and suspended tasks, and it schedules across them to make progress in processing the operations (④). Scheduling is cooperative; as tasks yield and are resumed, they store/restore their state, so when a worker schedules a task no stack switch is performed. As tasks execute user-provided logic, they interact with the store through a `get/put` interface similar to the one exposed remotely (⑤); the key difference is that the functions exposed to extensions take and return references rather than forcing copies (Table 2).

Beyond fast kernel-bypass network request processing, Splinter's speed depends on exploiting the Rust compiler in two key ways: first, to enable low-cost isolation and, second, to enable low-cost task switching. The two are intertwined. Splinter uses stackless generators to suspend and resume running extensions, which require compiler support. That is, the Rust compiler analyzes extension code, determines the state that needs to be held across extension cooperative-yield/resume boundaries, and generates the code to suspend and resume extension operations. No separate stack is needed, and the code needed to yield/resume is transparent to the extension.

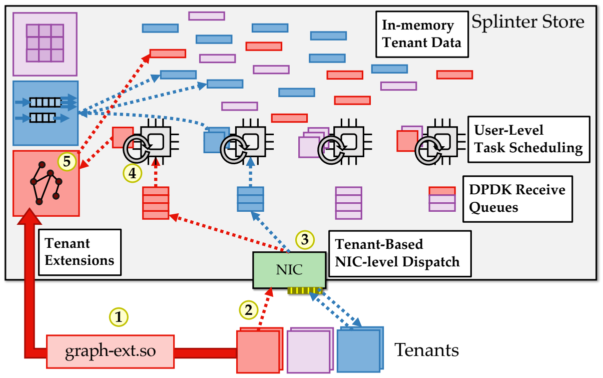


Figure 3: Overview of Splinter. Tenant data is stored in memory, and tenants can invoke extensions they have installed in the store (①). Extensions are type safe, but compile to native code. The NIC uses kernel bypass for low latency (②) and assists in dispatch by routing tenant requests to cores (③). Each core runs a single *worker* kernel thread that uses a user-level task scheduler to interleave the execution of tenant requests (④).

These lightweight tasks are key, but Splinter's careful attention to object lifetimes, ownership, and memory safety make them effective, since otherwise full context switch would be needed between tasks for isolation. A key challenge in Splinter is ensuring its fine-grained tasks from different trust domains—compiled to native code, and mapped directly into the store's memory—remain low-overhead while still operating within Rust's static safety checks. Low-overhead trust boundary crossings are essential to Splinter's design; they enable easy and inexpensive task switching, dispatch (§3.3), and work stealing (§3.4), which keep response latency low and CPU utilization high across all the cores of the store.

Another key challenge is that extension invocations introduce more irregularity into request processing than a simple get/put interface. By avoiding hardware context switches, Splinter keeps task switch costs down to about 11 nanoseconds, but the difficult tradeoff is that this forces it to handle these variable workloads without traditional preemptive scheduling. At the same time, it cannot use fully cooperative scheduling, since the store does not trust tenants to supply well-behaved extensions. Splinter's per-worker task scheduler resolves this tension by multiplexing long-running and short-running tasks to build mostly-cooperative scheduling. This is backed up by having an extra thread that acts as a watchdog for the others to support preemption when needed.

3.1 Compiling and Restricting Extensions

The Splinter store cannot directly load native code provided by tenants. Code must be compiled and type checked to ensure its safety before it can be loaded into a store, and extensions face some extra restrictions that must be enforced at compile time. The compiler is trusted and must be run by the storage provider. Tenants must not be able to tamper with the emitted extension, so it must be loaded directly into the store by the provider or the provider must ensure its integrity in transit between the trusted compiler and the store. Aside from Rust's standard type and lifetime checks (§3.1.2), Splinter extensions have the following static restrictions:

No Unsafe Code. Unsafe code could skip compiler checks resulting in memory unsafety. So, our wrapper over `rustc` disallows unsafe code in extensions (§3.1.3).

Module Whitelist. Code from external dependencies could include unsafe code, and that unsafe code shouldn't be incorporated into untrusted extensions unless it is trusted. Even beyond memory safety, such unsafe blocks could, for example, make syscalls. So, our wrapper restricts external dependencies to modules that are re-exported by a Splinter library that includes many standard functions and types. This restriction applies to the standard library (`std`) as well: the wrapper only exposes whitelisted `std` functionality to extensions.

These checks combine with three other runtime guarantees to ensure isolation: the store only accepts or provides references to insert/fetch a value under a key if the same tenant owns both the extension and the key (§3.2); it prevents uncooperative extensions from dominating CPU time and stack, heap, or record memory (§3.3); and it catches panics (runtime exceptions) and stack overflows that occur while executing an extension operation (§3.3). Next, we describe what guarantees this gives the storage provider and its tenants; the runtime checks are described later along with details about the execution model.

3.1.1 Trust Model

There are two stakeholders for a Splinter store: the storage provider and storage tenants. Splinter should protect tenants from each other and the provider from the tenants. Tenant misbehavior could be unintentional, in the form of bugs or unexpectedly high application load, or it could be malicious, in the form of tenants attempting to read others' data, deny service, or use an unfair fraction of resources. We consider threats from "within" the store; threats from "without" such as an attacker gaining root access to the machine by exploiting other services running on it should be dealt with using standard security best practices.

Aside from providing good quality of service to tenants, service providers have one key concern: protecting the secrecy and integrity of tenants' data. Extensions don't share state with one another, and Splinter provides no means for inter-extension communication. So, no complex sharing policies are needed; Splinter's only goal is extension isolation. Rust references act as capabilities; they ensure that extensions cannot fabricate arbitrary references to storage state or to other tenants' state (§3.1.2).

Like any database, Splinter's Trusted Computing Base (TCB) includes the libraries, compilers, hardware, etc. on which it is built; while this code is not directly exposed to tenants, vulnerabilities in it can still lead to exploits. Dependencies include LLVM [42], the CPU, the network card (NIC) and its kernel-bypass libraries (DPDK [20]).

Splinter's design provides a larger attack surface relative to other databases in some ways, but decreases the attack surface in others. Because it allows execution of tenant code, Splinter's safety depends on the soundness of Rust's type system, which is not proven. While some soundness issues in the compiler have been found [34], progress is being made in proof efforts [35], and Splinter automatically benefits from such progress. If extensions cannot violate Rust's safe types, the remaining avenue for attack is unsafe code in the system; extensions cannot supply unsafe code, but they can indirectly call it in the interfaces and libraries that Splinter explicitly exposes to extensions. On the plus side, extensions *must* break one of these layers of protection before they can attack other code: they do not have direct access to system libraries, system calls, etc. and can only gain it by breaking out of Rust's safe environment.

Splinter decreases the attack surface with respect to the virtual memory system – both hardware and kernel components. Because it doesn't rely on virtual address translation for isolation, recent Meltdown speculation attacks don't affect its design [46]; however, Spectre-based speculation attacks do affect Splinter [40, 41]. Like any system that runs untrusted code or operates on untrusted inputs, Splinter would require special steps to mitigate these side channels. It already limits them in part because it doesn't provide explicit timing functions to extensions. Full protection will require compiler support [13], hardened storage interfaces (like the Linux kernel [17]), and hardened libraries for extensions. The measurements in this paper do not include these mitigations.

3.1.2 Memory Safety

Rust's memory safety (and data race freedom) is guaranteed through a strong notion of *ownership* that lets the `rustc` compiler reason statically about the lifetime of each object and any references to it. The compiler's *borrow checker* statically tracks where objects and references are created and destroyed. It ensures that the lifetime of a reference (initially determined by its binding's scope) is subsumed by the lifetime of its referent. Rust separates immutable and mutable references; an immutable reference is a reference that when held restricts access to the underlying object to be read-only. The compiler disallows multiple references (of either type) to an object while a mutable reference exists, which prevents data races.

Often, the lifetime of an object cannot be restricted to a single, static scope. This is especially true in a server that processes requests across threads, where the lifetime of many objects (RPC buffers, extension runtime state) is defined by request/response. Rust provides various accommodations for this, such as moving ownership between bindings and runtime reference counting that is safe but implemented in unsafe Rust. Splinter efficiently handles these issues while working within `rustc`'s static safety checks (§3.2.2). Unlike C/C++ pointers, Rust references cannot be fabricated or manipulated with arithmetic; they always refer to a valid, live object. Rust supports pointers but their use is restricted for safety.

Store Operations for Extensions
get(table: u64, key: &[u8]) → Option⟨ReadBuf⟩ Return view of current value stored under ⟨`table`, `key`⟩.
alloc(table: u64, key: &[u8], len: u64) → Option⟨WriteBuf⟩ Get buffer to be filled and then `put` under ⟨`table`, `key`⟩.
put(buf: WriteBuf) → bool Insert filled buffer allocated with `alloc`.
args() → &[u8] Return a slice to procedure args in request receive buffer.
resp(data: &[u8]) Append data to response packet buffer.

Table 2: Extensions interact with the store locally through an interface designed to avoid data copying.

3.1.3 Restricting Unsafe Rust

An important extra restriction that Splinter imposes beyond Rust is that extension code must be free from *unsafe* Rust, a superset of the language that allows operations that could violate its safety properties. For example, unsafe code can dereference pointers, perform unsafe casts, omit bounds checks, and implement low-level synchronization primitives. All unsafe code in Rust requires an `unsafe` block, which Splinter disallows in extension code.

Extensions cannot implement unsafe code, but they can invoke it indirectly. This is often desired. For example, extensions execute some unsafe code when they ask the store to populate a response packet buffer. In some cases it is not desired. For example, file I/O can be induced through the Rust standard library. To prevent this, Splinter restricts extensions to use a subset of the standard library that doesn't include I/O or OS functionality.

Our experience has been that safe Rust combined with basic data structures from its standard library are sufficient to write even complex imperative extensions like Facebook's TAO [10]. In cases where unsafe code could provide a performance benefit, the store can provide that functionality if it is deemed safe to do so, since it is trusted and can include `unsafe` code (§3.2.3).

3.2 Store Extension Interface

The interface that extensions use on the server to interact with stored records is similar to the external, remote interface that clients use in any conventional key-value store (Table 2). The main differences are in careful organization to eliminate the need to copy data between buffers.

All persisted records are stored in a *table heap*. Keeping records in a identifiable region will be essential to support replication, recovery, and garbage collection as Splinter's implementation evolves.

3.2.1 Storing Values

Extensions can `put()` data they receive over the network or new values that they produce into the store. When an extension invocation request is received from a tenant, the store invokes the indicated operation. Incoming data is in a packet buffer that is registered with the NIC. Those buffers cannot be used for long-term storage because the NIC must use them to receive new requests; data that must be preserved needs to be copied into the store.

Splinter tries to ensure that data can be moved from NIC buffers into the store with a single copy. This requires `put()` to be split into two steps. First, an extension calls `alloc(table, key, length)` to allocate a region in the table heap for a record. The extension receives a bounded slice (a view) to the underlying allocated memory. Then, it copies data from the request's receive buffer, unmarshalling as it does so, if needed. Extensions use `args()` to directly access data (by reference) in the receive buffer to perform this copy. An extension may produce its own data values as part of this process either from input arguments or together with values read from the store. Once the allocated region is properly populated, it is inserted into the table with `put()`, which takes ownership of the buffer and inserts it into a hash table.

Problems like use-after-free are prevented by Rust's borrow checker; extensions cannot hold references to a buffer once ownership is transferred to the store, eliminating the need for copying data into the store for safety. The receive packet buffer has the same guarantee. Rust's borrow checker ensures references to it cannot outlast the life of the RPC, eliminating the need to copy received arguments or data into the extension for safety.

Values stored by `put()` must be allocated from the table heap; extensions should not be able to pass arbitrary (heap or stack allocated) memory to `put()`. Splinter enforces this so that it can optimize record layout; keys and values can be forced into a single table heap allocation, which eases heap management and eliminates cache misses for hash table lookups. As a result, Splinter wraps allocations with a type (`WriteBuf`) that extensions cannot construct, ensuring they can only pass buffers acquired from `alloc()`. `WriteBuf` has a method to get a reference to the underlying buffer, so extensions can fill it.

```rust
fn aggregate(db: Rc<DB>) {
  let mut sum = 0u64;
  let mut status = SUCCESS;
  let key = &db.args()[..size_of::<u64>()];

  if let Some(key_lst) = db.get(TBL, key) {
    // Iterate KLEN sub-slices from key_lst
    for k in key_lst.read().chunks(KLEN) {
      if let Some(v) = db.get(TBL, k) {
        sum += v.read()[0] as u64;
      } else {
        status = INVALIDKEY;
        break;
      }
    }
  } else {
    status = INVALIDARG;
  }
  db.resp(pack(&status));
  db.resp(pack(&sum));
}
```

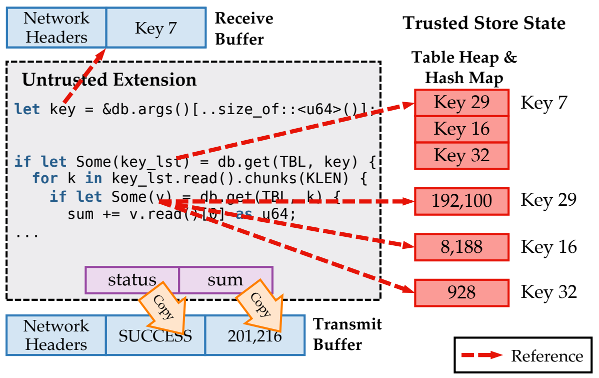
Listing 1: Example aggregate extension code. The extension takes a key as input (directly from a request receive buffer), looks it up in the store, and gets a reference to a value that contains a list of keys. It looks up each of those keys, it sums their values, and directly appends the result to a response buffer.

3.2.2 Accessing Values

Extensions can interact with stored data in a similar way, requiring only one copy into a response buffer to return values from the store. When an extension procedure is invoked, it is also provided with a response buffer that can be incrementally filled via `resp()`. On each extension procedure invocation, the store pre-populates the response buffer's packet headers; extensions can only append their data after these headers. All response buffers are pre-registered with the NIC for transmission.

Extensions call `get(table, key)`, and they receive back a reference to the underlying portion of the table heap that contains the value associated with `key`. No copying is needed at this step; the store tracks this reference and prevents the table heap garbage collector from freeing the buffer while an extension has a live reference to the data. Since values are never updated in place, extensions see stable views of values. Extensions can compute over the value or many values concurrently (by calling `get()` multiple times), and they can copy portions of the data they observe or any results they compute directly into the response buffer. Once the extension procedure has populated the response buffer, Intel's DDIO [32] transmits the data directly from the L1 cache, which avoids the cost of memory access for DMA of stored data.

Listing 1 and Figure 4 show an example of how this works for a simple extension that sums up a set of values stored under keys that are listed as part of another stored



```
let key = &db.args()[..size_of::<u64>()];


if let Some(key_lst) = db.get(TBL, key) {
  for k in key_lst.read().chunks(KLEN) {
    if let Some(v) = db.get(TBL, k) {
      sum += v.read()[0] as u64;
...
```

Figure 4: References during aggregation. All data accessed by the extension in Listing 1 is by reference whether that data is part of the arguments in the receive buffer or part of a record in the store. References work in reverse for the response; the extension passes references to data to the store, and the store copies that data into the response buffer.

value without any extra data copying. In Line 4, the extension obtains a reference to its transmit buffer to find which key it should look up in order to find a list of keys that will be aggregated over. Line 6 passes a reference to that same location to the store in order to obtain a reference to the value that contains the key list. In Line 8, still without copying, the extension iterates over that value in chunks equal to the length of the keys stored in the value. Each step of the iteration produces a reference that the extension uses to `get()` references to values for each of the stored keys, one at a time (Line 9). Using each of those references, it extracts a field that it adds to `sum`, a local variable. Finally, the extension passes references to `status` and `sum` to append them to the response buffer. In all, data copying is only forced where it is needed, so the compiler has flexibility in optimizing extension code.

The store's `get()` call returns a `ReadBuf` rather than a plain slice (`&[u8]`) in order to satisfy Rust's borrow checker. Calling `get()` cannot return an immutable reference or slice to a stored value, because the borrow checker wouldn't be able to statically verify that the reference would always refer to a valid location. For example, the compiler couldn't be sure that the store wouldn't garbage collect the value while the reference still exists. Furthermore, extension invocations are generators, and they must yield regularly (§3.3). Yielding marks the end and start of a new static scope, so each time the generator is resumed, the calling scope could vary. Any obtained references to a stored value couldn't be held across yields, because the borrow checker wouldn't be able to verify that those references would still be valid on reentry.

The `ReadBuf` returned by `get()` solves this. It is a smart pointer that maintains a reference count to ensure the underlying stored object isn't disposed, and it allows the extension code to (re-)obtain a reference to the underlying object data. Once a `ReadBuf` is returned to a generator, it is stored within the generator's local state, so the generator owns this `ReadBuf`. Extensions cannot hold references between yields, but by working with the `ReadBuf` it can (transparently) re-obtain a reference to the data without performing another `get()`. Rust's `Arc` smart pointer does the same; `ReadBuf` hides its constructor from extensions and disallows duplication. This prevents extension code from creating `ReadBuf`s that persist beyond the life of a single request/response, which could otherwise hold back table heap garbage collection.

3.2.3 Avoiding Serialization and De-serialization

Allowing extensions to interact directly with receive buffers, transmit buffers, and table heap buffers eliminates copying for opaque data, but Rust's safety makes avoiding some copies harder. Extensions cannot perform unsafe operations, otherwise they could thwart Rust's memory safety guarantees. Unfortunately, this means safe Rust code cannot cast an opaque byte array to/from different types to avoid the need to serialize/de-serialize data. For example, if `args()` returned an 8-byte slice an extension may desire to treat that slice data as a 64-bit unsigned value. Safe Rust disallows this.

For small arguments, extensions can convert between formats with arithmetic, but for richer data models, arguments, stored values, and responses will have more complex, structured formats. To accommodate this, Splinter's interface provides a mechanism for extension code to convert between byte slices and references to a small set of types. If a slice (`&[u8]`) is naturally aligned to the desired type, Splinter allows conversion to a reference of that type (`&T`), where `T` is limited to signed/unsigned integers and compound types built from them.

These casts are safe, but they are meaningless across architectures. As a result, they can only be used between a client and the store when they have the same underlying platform (e.g. x86-64). Similarly, they can only be used with extensions' `get/alloc/put` interface if all stores in the system (e.g. before/after recovery, source/destination for migration) have matching hardware platforms.

3.3 Cooperatively Scheduled Extensions

Splinter is designed to work well regardless of whether tenant-provided extensions are short and latency-sensitive or long-running and compute- or data-intensive. In fact, the best mix of tenants will mix these operations, keeping CPU, network, and in-memory storage better utilized than would be possible with a single, homogeneous workload. Even so, latency-sensitive operations can easily suffer under interference from heavier operations.

This means Splinter must multiplex execution of tenant extension invocations not only across cores but also within a core. Long-running procedures cannot be allowed to dominate CPUs, but preemptive multitasking is too costly even when page table switching can be avoided.

Rust's lightweight isolation is part of the solution, since calls across trust domains have little overhead. Splinter already relies on `rustc` for safety, but it can also rely on it to help minimize task switching costs. When a new request comes into the store, Splinter calls into the responsible extension to allocate a stackless coroutine (a generator) that closes over the state needed to process the request. Generators support a `yield` statement that suspends execution and enables cooperative scheduling; extension code is expected to periodically call `yield` to allow other tasks to run. `rustc` produces generators specific to the extension, so the cost to create them and switch between them is low. Splinter invokes the created generator. Whenever it yields, Splinter's per-core task scheduler runs another generator task. Since yielding requires no costly hardware boundary crossing and no stack switch, it is fast and inexpensive to yield frequently.

Like other similar systems, to avoid jitter due to kernel thread context switches and migrations, Splinter runs the same number of worker threads as cores in the system (Figure 3), and each is pinned to a specific core. Generators are invoked on the worker's stack, avoiding a stack switch. Note that the compiler generates the structure to hold a suspended task's state across yields. Consequently, a worker's stack never concurrently contains state for different tenants (or even tasks); furthermore, whenever a task yields or completes, the worker's stack contains no extension state. This makes it easier to handle uncooperative extensions (§3.3.1) and load imbalance (§3.4).

3.3.1 Uncooperative and Misbehaving Extensions

All calls through the store interface include an implicit yield, so extensions can only dominate CPU time with infinite or compute-intensive loops. Nonetheless, such behavior can disrupt latency-sensitive tasks and constitute a denial-of-service attack in the limit.

To solve this, Splinter uses ideas from user-level threading for latency-sensitive services [59] and adapts them for untrusted code. An extra (mostly idle) thread acts as a watchdog. If a task on a core fails to yield for a few milliseconds, the watchdog remedies the situation. First, the worker thread on the core with the uncooperative task is re-pinned to a specific core that is shared among all misbehaving threads and low-priority background work that the store performs. Second, a new worker kernel thread is started and pinned to the idle core left behind after the misbehaving thread was re-pinned. Finally, the new worker steals the tasks remaining in the scheduler queue for the re-pinned worker and resumes execution for these tasks. Note, this is safe in part because all of the state of a suspended task is encapsulated. Tasks only have state on a worker's stack if they are running, so the misbehaving task is the only one the new worker cannot steal. Whenever a misbehaving task finally yields, the scheduler on that worker realizes that it has been displaced, and the worker thread terminates along with the task.

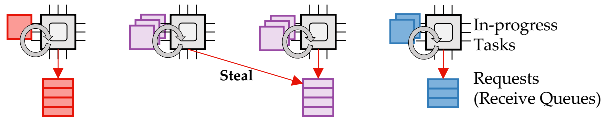


Figure 5: Dispatch tasks on each core steal requests from the receive queue of the core to their right whenever they have no requests in their own receive queue. As a result, work from overloaded cores get redistributed without generating high contention. Here, core 1's in-progress tasks were induced by requests stolen from core 2's queue.

Hence, misbehaving tasks don't block other requests, but they can still cause disruption. Creating and migrating kernel threads is expensive, so there must be a disincentive against forcing watchdog action. Tenants that run uncooperative tasks will experience poor quality of service, since they must share a core with other disruptive work. Furthermore, when a worker is re-pinned the watchdog also takes away access to its receive and transmit queues, so tenants cannot get responses from bad requests and, thus, benefit from their misbehavior. Even so, billing policies should ensure such behavior is unprofitable.

Aside from infinite loops, the store must also protect against other things that cannot be prevented with compile-time checks. For example, Rust doesn't have general exceptions, but extensions can raise exceptions with operations like division by zero that raise a `panic`. Splinter must "catch" these panics or they would terminate the worker, since panics unwind the call stack and worker threads call extension code on their own stack. Fortunately, Rust provides a mechanism to do this, and Splinter catches panics and converts them to an error response to the appropriate client. Stack overflows and violation of heap quotas are handled similarly.

3.4 Tenant Locality and Work Stealing

The Splinter store avoids any kind of centralized dispatch core to route requests to cores, since this can easily become a bottleneck [55]. At the same time, it needs to balance requests across cores, while still trying to exploit locality to avoid cross-core coordination overheads. To do this, clients route each tenant's requests to a particular core. This provides cache locality, it reduces contention, and it improves performance isolation. Splinter configures Flow Director [31] so that the NIC directly stores packets with a specific destination port number in a specific receive queue. Each receive queue is paired to a single task dispatcher owned by a worker thread (pinned to a core). As a result, tenants can steer requests to specific cores by placing their tenant id in the UDP destination port field.

CPU	2×Xeon E5-2640v4 2.40 GHz 10 cores (20 hardware threads) per socket
RAM	1 TB 2400 MHz DDR4
NIC	Mellanox CX5, 40 Gbps Ethernet
OS	Ubuntu 16.04, Linux 4.4.0-116, DPDK 17.08, 16×1 GB Hugepages, Rust 1.28.0-nightly

Table 3: Experimental configuration. Evaluation used one machine as server and one as client. Only the NIC-local CPU socket was used on the server.

However, this approach alone can leave cores idle under imbalance, and, as a multi-tenant store, it is important for the system to deliver good resource utilization. Whenever the scheduler on a core has no incoming requests in its local receive queue, it attempts to steal requests from a neighbor's receive queue (Figure 5). Transmit queues aren't bound to specific (server-side) source ports, so the response can be sent directly from the core that stole the request. This simple form of soft affinity works well, and, since tasks are lightweight, it is also relatively easy for Splinter to take advantage of idle compute in the system without costly thread migration.

4 Implementation

The Splinter store is implemented in 7,500 lines of Rust. It uses the NetBricks network function virtualization framework [56] as a wrapper over the DPDK [20] packet processing framework. Splinter also includes 1,100 lines of Rust that provide the store interface to extensions. Extensions import it and compile against it. The store also imports the interface, since it defines how the store interacts with extensions to create a new generator for an invocation. Splinter is open and freely available on github[1].

The store needn't be written in Rust, but doing so has advantages. It prevents data races and segmentation faults within the store, but it also lets the store use Rust's type system and lifetimes to ensure that mistakes aren't made with lifetimes of objects and references handed across trust boundaries, which an adversary could exploit.

5 Evaluation

We evaluated Splinter on five key questions:

1. What is Splinter's isolation overhead?
2. Does Splinter support high tenant densities?
3. How does Splinter perform under operations with heterogeneous runtimes?
4. Do representative extensions see latency and throughput benefits?
5. When does performing operations client-side outperform extension-based operations?

[1]`https://github.com/utah-scs/Sandstorm/`

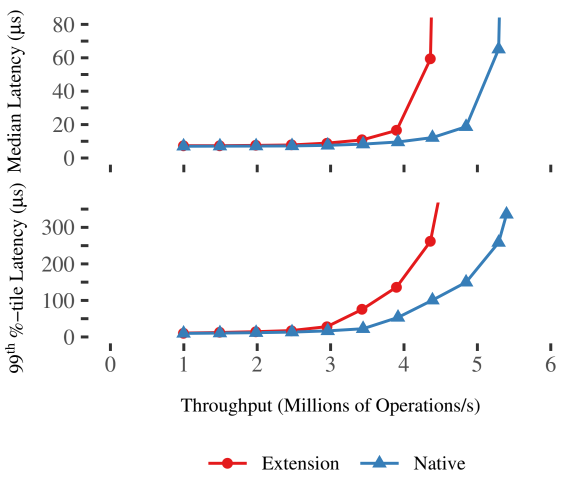


Figure 6: Comparison of YCSB-B performance using native and extension-based `get()` and `put()` operations at a tenant density of 1,024. When using extensions, the server saturates at 4.3 million operations per second. In comparison, native operations are about 23% more efficient, saturating at 5.3 million operations per second.

5.1 Experimental Setup

All evaluation was done on two machines consisting of one client and one storage server on the CloudLab testbed [60] (Table 3). Both used DPDK [20] over Ethernet using Mellanox NICs for kernel-bypass support. The server was configured to use only one processor socket; out of the ten hardware cores, eight were used for request processing, one was used for management and to detect misbehaving extensions, and the last one was used to hold all misbehaving extensions once detected.

To evaluate Splinter and its isolation costs under high load and density, the client ran a YCSB-B workload [15] (95% gets, 5% puts; keys were chosen from a Zipfian distribution with $\theta = 0.99$) that accessed tenant data on the storage server. Unless stated otherwise, the client simulates 1,024 total tenants. Tenant ids for each request were chosen from a Zipfian distribution with $\theta = 0.1$ (unless stated otherwise) to simulate some tenant skew. Each simulated tenant owns one data table consisting of 1 million 100 B record payloads with 30 B primary keys (totaling about 120 GB of stored data). The client always offered an open-loop load to the server.

5.2 Isolation Overhead

Figure 6 compares the performance of YCSB-B under two different cases. In one case ("Native"), the Splinter store executes get and put operations like any other key-value store would; none of Splinter's extension functionality is used. This case sets an upper-bound for Splinter's performance. In the other case ("Extension"), that same get or put is executed as part of a tenant-provided and untrusted

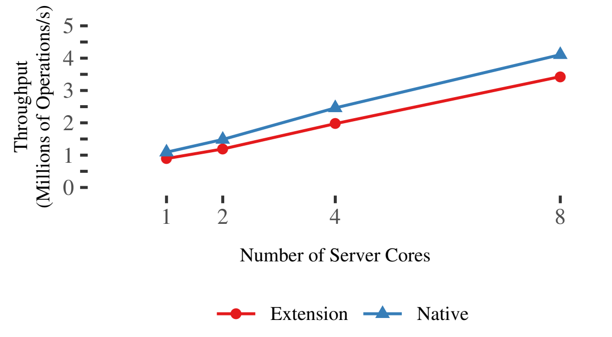


Figure 7: Storage server scalability at a tenant density of 1,024. Points represent throughput when YCSB-B latency crosses 10 μs. Isolation overhead is consistently lower than 20%.

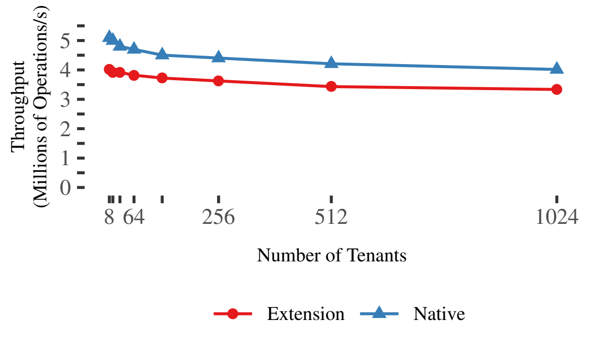


Figure 8: Scaling tenants. Points represent server throughput when YCSB-B latency crosses 10 μs. With isolation, increasing the number of tenants only impacts performance modestly; moving from 8 to 1,024 tenants reduces throughput by 700 Kops/s.

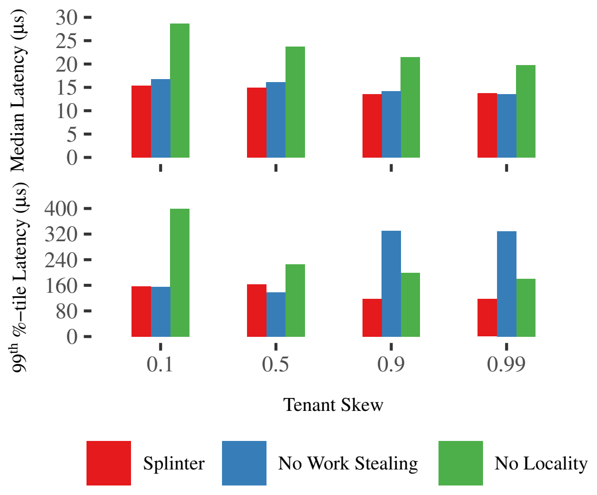


Figure 9: Latency with tenant skew. The server runs near saturation at 4 Mops/s in each case. Without work stealing, tail latency under high skew increases from 138 μs to 330 μs. Without tenant locality, median and tail latencies are affected.

Splinter extension. This teases apart the isolation and dispatch costs for Splinter to run arbitrary tenant-provided logic. For offered loads of less than 3.5 million operations per second (Mops/s), median latency with and without isolation are nearly identical (about 9 μs).

Splinter extensions have some overhead, so the store saturates earlier when gets/puts are executed through extensions. With isolation, the median latency spikes above 4 Mops/s, reaching 59 μs at 4.3 Mops/s. Without isolation, this spike comes at 5.3 Mops/s. Tail latency (99th-percentile) begins to show a difference at 3 Mops/s. On the whole, in this pessimal workload with extremely fine-grained operations all invoked as extensions, Splinter's isolation costs still only impact throughput of the store by about 19%. Compared to the 1.8× (simulated) penalty for hardware-based isolation in Figure 2, this is a significant improvement (a 1.2× penalty over native get/put).

Figure 7 compares YCSB-B scalability when the server is approaching saturation (median latency $>$ 10 μs) under the native and extension-based cases. Invoking get and put operations from extensions instead of directly has no impact on scalability; scalability is near linear in both scenarios. However, as pointed out above, it does affect throughput. At one core, throughput is reduced by 200 Kops/s (18%), while at eight cores, the reduction is 700 Kops/s (17%). This shows that, though extensions do increase the number of cycles each core spends processing requests, it doesn't come at the cost of significant increased coordination between the cores.

5.3 Tenant Density

Figure 8 shows how varying the number of tenants sharing the store impacts its throughput. As in the prior experiments, tenants run YCSB-B under two cases: without isolation ("Native") and with isolation ("Extension"), so the experiment captures extension isolation overheads. The results show that Splinter can efficiently support high tenant densities with minimal overhead. With isolation, the throughput at 1,024 tenants is 3.3 Mops/s, only 700 Kops/s less than the throughput at 8 tenants. Additionally, the throughput with isolation is consistently within 22% of the throughput without isolation.

In practice, offered tenant load will be skewed, since some tenants are likely to have heavier workloads than others. This results in a few heavy workloads that must share the store with a long tail of many more passive ones. We ran an experiment to show that Splinter can handle this imbalance and that its work stealing and tenant locality help maintain Splinter's response times under high load.

Recall that Splinter routes requests for a tenant to a specific core, but cores steal work from each other to combat imbalance. To gauge the benefits of this approach, we compare it against a tenant-partitioned approach with no work stealing and an unpartitioned approach that sprays requests over all cores in a tenant-oblivious fashion. We vary tenant skew, which affects all three approaches.

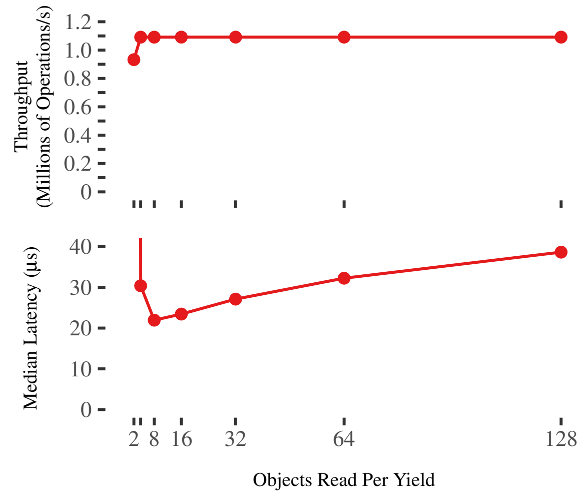


Figure 10: Performance with a small fraction (15%) of cooperative long running procedures that perform 128 `gets`. Yielding frequently can help improve median latency from 38 µs to 22 µs. However, yielding too frequently hurts median latency. The storage server was offered a constant load of 1.1 Mops/s.

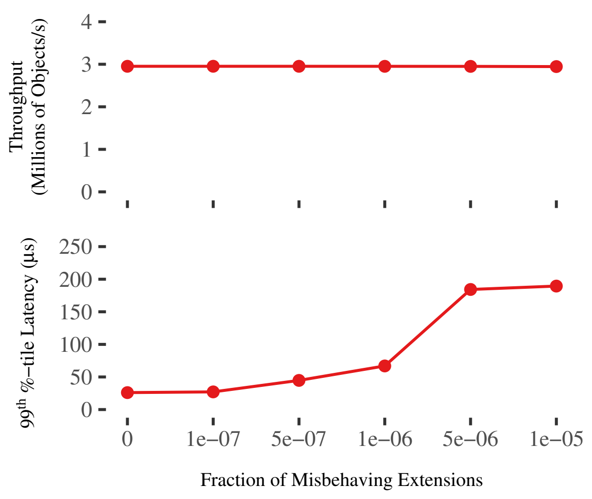


Figure 11: Impact of uncooperative requests on performance. System throughput stays constant at 3 Mops/s throughout. For fractions of uncooperative requests greater than 1 every million, tail latency is significantly affected (> 100 µs).

Figure 9 shows the results. These measurements are with an offered load of 4 Mops/s, keeping the store close to saturation. In each case, the store meets the offered load by running at 4 Mop/s. Without work stealing, Splinter's tail latency suffers by a factor of 2 under high tenant skew (0.9 and 0.99). In this case, partitioning helps throughput due to locality and reduced contention (as evidenced by its relatively consistent median response time), but queues become imbalanced hurting tail latency. The unpartitioned approach doesn't respond as significantly to tenant skew though it is slower overall, as expected. Unpartitioned execution results in 42% to 86% worse median latency with 38% to 155% worse tail latency.

5.4 Request Heterogeneity

Figure 10 investigates the impact of mixing short operations with cooperative longer-running operations. We configured our client so that 15% of extension operations performed 128 `gets` on the storage server. The rest of the requests invoked an extension that performed one `get`. We varied the number of `gets` made by the longer extension per yield (frequency). These measurements were made at an offered load of 1.1 Mops/s. Increasing the frequency of yields improves median latency of the smaller operations by 42% until a frequency of 8 `gets` per yield. Yields add some overhead, and yielding more frequently pushes the store to saturation in this case. As a result, all requests see increased response times. Extensions should yield frequently, but yielding too often is wasteful. Splinter may be able to help with this in the future; Splinter could provide extensions with a yield that is ignored if called too quickly in succession, avoiding the full yield cost.

Figure 11 shows how uncooperative extensions impact system performance. Here, the client invoked a small fraction of extension operations that executed an infinite loop. The remaining fraction of requests invoked a small extension that performed a single `get`. Splinter performs well in the presence of misbehaving extensions. Throughput is steady at 3 Mops/s irrespective of the fraction of misbehaving requests. Median latency isn't shown, but it is steady as well. Tail latency suffers as more requests misbehave, though it is within 100 µs for fractions as high as one in a million requests.

Note that one in a million requests (1e-6) is harsh. The store can execute more than 4 Mop/s, so this represents a misbehaving invocation starting every quarter second; at 1e-5 misbehavior starts about once every 25 ms.

5.5 Aggregation Extension

Online data aggregation is a common task for applications. For example, a user might send a query demanding a movie studio's total earnings in the year 2017. With a key-value data model, this would require two round-trips to storage: one to fetch the list of movies made by the studio and one to fetch the box-office earnings of each of the movies. Splinter improves the user-facing and server-side performance of these types of queries by allowing applications to inexpensively embed their data model (studios and movies) and operations (total earnings aggregation) within storage.

Figure 12 compares a completely client-based and a Splinter extension-based implementation of such an aggregation over 4 records. Each of the store's 1,024 tenants owned a table with 300 K indirection lists pointing to 1.2 million records, totalling about 100 GB of stored data.

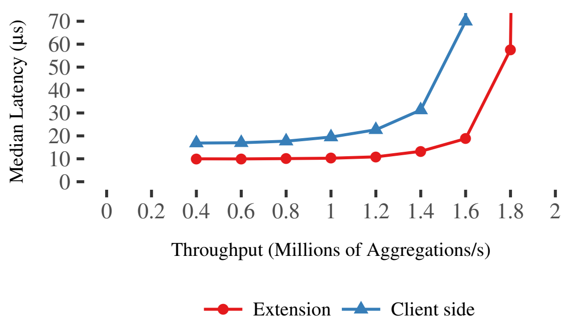


Figure 12: Aggregation throughput versus latency. Aggregations combine 4 records. Under low load, the median latency of a client-side implementation is 1.6× that of an extension-based implementation. Using an extension also improves saturating throughput from 1.2 M to 1.6 M aggregations per second.

The client-based implementation first performed a `get()` to retrieve an indirection list followed by a `multiget()` (a single RPC requesting values for multiple keys) to fetch all of the records indicated in the indirection list. The first field from each of the returned objects is summed up into a single 64-bit result. The extension-based implementation invoked a Splinter extension called `aggregate()` with the same functionality as the client-based approach.

Pushing the aggregation from the client to the server has two key benefits. First, it improves performance from the client's perspective: the extension-based implementation reduces median latency by 38% (from 16 µs to 10 µs) under low load with larger gains under higher loads. This improvement is mainly due to a reduction in the number of round-trips; unlike the client-based extension, the `aggregate()` extension doesn't need to wait for the store to return an indirection list before it can start aggregation. Second, it improves performance from the server's perspective as well. Splinter's extension invocations are more expensive than plain `get()` operations (§5.2), but they eliminate some of the costly network and RPC processing. Hence, saturating throughput improves from 1.2 M to 1.6 M aggregations per second.

Note, this improvement comes in a challenging case for Splinter; at 40 Gbps, Splinter is never network limited. These results show that even if a store is CPU-limited, pushing compute to the store can still provide a throughput benefit, since it can mitigate request processing overheads. On slower networks, Splinter would provide more of a benefit since extensions can reduce network load.

Figure 13 shows the impact of the number of records aggregated on the saturating throughput of the extension-based and client-based implementation. In both approaches, increasing the number of records aggregated increases the work the store has to do per request (`aggregate()/multiget()`), and, hence, decreases the overall throughput of the system. However, if that work

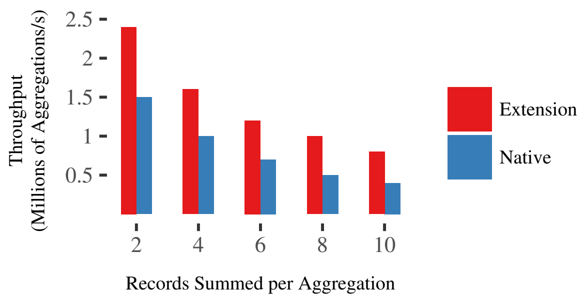


Figure 13: Saturating throughput of aggregation versus the number of aggregated records. The extension-based implementation outperforms the client-side implementation irrespective of the number of records aggregated. The gains are highest when aggregations are over two records (2.4 M versus 1.5 M aggregations per second).

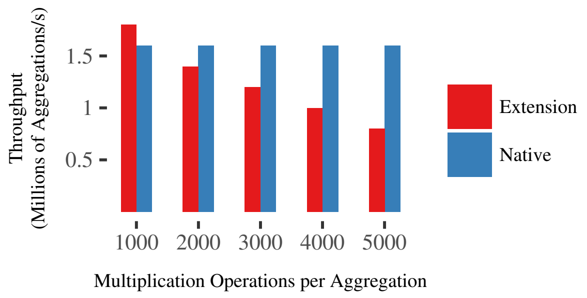


Figure 14: Saturating throughput of the aggregation extension versus the amount of compute per aggregation. After aggregating 2 records, each operation raised the result to the power `n`, implemented as `n` 64-bit multiplications (hence the x-axis). Increasing the order (`n`) increases server-side compute in the extension-based implementation, hurting throughput. At an order of 5000, the client-side approach is 2× faster.

is simple (like summation) it is always better to aggregate at the store. The gain in saturating throughput of the extension-based aggregation is always more than 50%.

For compute-intensive operations, the extra CPU cost of running extensions at the store can outweigh the gains of fewer RPCs. Figure 14 explores this effect. After adding the first field of two records, each operation raises the result to the power `n` (with `n` 64-bit multiplications). Using an extension, increasing `n` above 2,000 slows the store and decreases saturating throughput from 1.8 M to 800 K aggregations per second. The client-side approach can hold throughput constant at 1.6 M aggregations per second; the client has enough idle CPU capacity to compute the result. This shows that extensions are ideal for operations with modest amounts of compute. For compute-intensive operations over data stored on high-load servers, clients should fetch data and perform operations locally.

5.6 TAO Extension

TAO [10] is a graph-oriented in-memory cache used at Facebook to hold objects from the social graph and as-

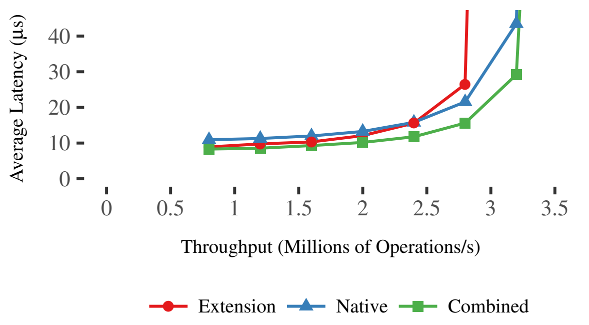


Figure 15: TAO extension throughput versus latency. With 60% `object_get` and 40% `assoc_range` operations, the TAO extension can reach 2.8 Mop/s before saturating with an average latency of 30 µs. By using native `get()` operations for `object_get`, the extension-based approach can outperform a purely client-side implementation by 400 Kop/s.

sociations between those objects. TAO is well-suited to Splinter. It is designed for interactive data, but it embeds knowledge about Facebook's workload to decrease round-trips to the store, which eliminates client-side stalls and improves server-side efficiency. We have implemented its simple operations as an 800-line Splinter extension.

Full details of TAO are beyond the scope of this paper, but the basics are simple. Aside from object put/get, TAO's *association lists* (e.g. `user1`'s "likes") allow one object to be associated to another via a typed, directed edges. For example, `user1`'s "likes" may be represented as an association list `(user1, likes)` $\rightarrow$ [`post1`, `post32`]. Association lists provide simple operations for adding, removing, and counting associations. Entries in association lists are timestamped, and range operations over association lists to fetch subsets of them are common ("get the first 10 entries in the `(user1, likes)` association list").

Figure 15 shows Splinter's performance under three different configurations: an extension-based approach (Extension), a client-based approach (Native), and a combined approach (Combined) that implemented `object_get` using native `get()` operations, and `assoc_range` using an extension. The workload was configured to issue a mix of 60% `object_get` and 40% `assoc_range` operations. We picked this ratio based on Facebook's reported TAO workload [10], which is dominated by reads (99.8%) mostly from these two operations. Each of the 1,024 tenants on the storage node owned a graph with half a million objects and two million edges (associations), totalling about 100 GB of stored data.

Since a significant fraction of requests are single round-trip `object_gets`, the client-based approach has a better saturating throughput than the extension-based approach. However, combining the two improves saturating throughput from 2.8 Mop/s to 3.2 Mop/s at a latency of 31 µs; the native `get()` helps eliminate the isolation overhead while executing an `object_get`, and the extension helps reduce the number of round-trips required by an `assoc_range`.

This makes Splinter competitive with FaRM's TAO implementation which is the fastest known implementation. Interestingly FaRM, takes the opposite approach of Splinter. On FaRM, TAO operations use multiple RDMA reads and careful object layout. FaRM reported 6.3 Mops/s (about 200 Kop/s/core) with a 41 µs average latency; Splinter performs about 400 Kops/s/core with lower latency. Differences in hardware and experimental setup likely account for some of the differences, but it shows Splinter's CPU-active server approach is competitive against FaRM's CPU-passive server approach. Furthermore, Splinter maintains a simple, remote procedure call interface, and the TAO extension enforces strong abstract data types. Splinter TAO clients have no knowledge of the internal layout of the stored data objects.

6 Related Work

Shipping computation to data and isolating untrusted code are well-studied, and Splinter builds on prior work. However, prior work does not address multi-tenancy at Splinter's granularity and number of tenants; further, no work addresses these issues with its throughput and latency goals, which are far beyond most cloud storage systems.

Low-latency RDMA-based Storage Systems. Low-latency, high-throughput key-value stores are now thousands of times faster than conventional cloud storage by exploiting RDMA, kernel-bypass, and DRAM [22, 23, 36, 44, 45, 55]. These systems are well-understood for small, regular workloads, but their simple (get/put, read/write) interfaces make them easy to optimize internally at the expense of application efficiency, since they force clients to make many round trips to storage and to compute locally [21]. RDMA lowers CPU overhead for transmit, but it cannot make up for the fundamental inefficiency of moving large amounts of data over the wire; receivers must still perform the same computation on the data that a server could have. Splinter eliminates this waste, while still using efficient kernel-bypass networking. At 40 Gbps a Splinter store is never network bound, so combining Splinter's approach with (one- or two-sided) RDMA verbs could provide a benefit by freeing up additional compute on store servers.

6.1 Pushing Computation to Storage

MapReduce [18] and Spark [73] ship code to data sets, though latency is not a concern. Even when compute is shipped to a storage (HDFS [63]) node, data is still copied via interprocess communication. Untrusted extensions, like those in Splinter, could eliminate these overheads.

Some distributed systems and frameworks support composing internal storage abstractions to synthesize new services [3, 4, 11, 28, 48, 62]. Malacology [62] claims storage extensions have been popular in the Ceph distributed file system, showing that extensions are useful to developers. In these systems, extensions are trusted, so they don't work for cloud storage; Splinter is also focused on tight integration of fine-grained computation and storage rather than on coarse composition of software services. Comet [26] embedded sandboxed Lua extensions into a decentralized hash table to allow application-specific extensions to get/put behavior. Lua's entry/exit costs are low; it is unclear how the performance of its just-in-time (JIT) compiled runtime would compare to Splinter.

SQL. SQL may be the most widely used approach to ship computation to data, and it also supports use as a stored procedure language [50, 54]. In-memory databases have placed pressure on performance, resulting in JIT compilation for SQL [25, 53]. With JIT, queries run fast, and calls back-and-forth between the database and user logic are inexpensive. SQL is type safe, so it is also easy to isolate. SQL's main drawback is that it is declarative. Often, this is a benefit, since it can use runtime information for optimization, but this also limits its generality. Implementing new functionality, new operators, or complex algorithms in SQL is difficult and inefficient. Some have extended SQL for specific domains, like graph processing [52], scientific computing [47, 57] and simulation [12], showing that SQL by itself is insufficient for many domains.

Native-code Extensions. The popular Redis [1] in-memory store supports native extensions. In FaRM [22, 23], an RDMA-based in-memory store, applications are written as native, storage-embedded functions that are statically compiled into the server. These systems don't allow extensions to be loaded at runtime, and application code is trusted so it does not work for multi-tenant cloud storage. Similarly, H-Store [39], VoltDB [65], and Hazelcast [29] are in-memory stores that support Java-based procedures, though none of them provide multi-tenancy.

6.2 Fault Isolation

Software-fault isolation (SFI) sandboxes untrusted code within a process (or OS kernel [33, 61, 67]) with low control transfer costs [9, 24, 27, 49, 72]. Both hardware isolation [66] and SFI [69] were applied to Postgres [64], which pioneered database extensions [68]. SFI still requires protected data to be copied in/out of extensions, since it relies on hardware paging or address masking that can only restrict access to contiguous memory regions.

Language-level approaches to kernel extension [8, 30] closely match Splinter's design and goals. SPIN let language-isolated extensions run as part of the kernel. It eliminated runtime overheads (aside from garbage collection), since extensions were compiled; it eliminated control transfer overheads, since it didn't require page table switching; and it eliminated copying between protection domains, since type-safe pointers worked as capabilities. Like Splinter, where tenants must write Rust code, a key downside of SPIN was that extensions had to be written in Modula-3, not C, so legacy code couldn't be used. Java also "sandboxed" applets using type-safety and specialized class loaders, which supported inexpensive control transfer and data access between domains [70].

Using Rust for low-cost, zero-copy isolation has been used for inexpensive software fault isolation both generally [5] and for network packet processing pipelines [56]. Splinter builds on these ideas, bringing them to storage and moving beyond static domains to a runtime extensible service. Tock [43] is an embedded OS that decomposes its kernel into untrusted *capsules* by exploiting Rust's safety. Tock's capsules are similar to Splinter's extensions, but they don't protect against denial of service (infinite loops) and capsules are static – they can't be added to a running kernel. These also differ from Splinter in that they assume a small number of trust domains; they are targeted at software decomposition. Splinter targets dense multi-tenancy with no static bound on the number of trust domains.

7 Conclusion

In-memory storage can significantly accelerate data-intensive applications, including those that need fine-grained and real-time access to data. However, as Dennard scaling ends, future cloud storage must not only be faster but also more efficient. Splinter shows that soon legacy hardware isolation techniques will limit resource provisioning granularity in the cloud, but it also provides a way forward. Systems must evolve to support granular, low-overhead shipping of compute to storage, and lightweight isolation between small compute tasks. Splinter works toward that evolution by discarding hardware isolation in favor of static safety checks. As a result, it supports thousands of tenants that can all access data in tens of microseconds while customizing storage operations to their needs and while performing millions of remote operations on modern multicore machines.

Acknowledgments

Thanks to Ankit Bhardwaj and Ethan Ransom for contributing to Splinter; to Abhiram Balasubramanian, Anton Burtsev, and Amit Levy for the conversations that helped lead us to this work; to the reviewers for their comments; and to our shepherd, Jon Howell. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1750558 and CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by Facebook and VMware.

References

[1] Redis. http://redis.io/. Accessed: 2018-09-27.

[2] The Rust Programming Language. http://www.rust-lang.org/en-US/. Accessed: 2018-09-27.

[3] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobbler, T., Wei, M., and Davis, J. D. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, CA, 2012), NSDI '12, USENIX Association, pp. 1–14.

[4] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed Data Structures Over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, PA, 2013), SOSP '13, ACM, pp. 325–340.

[5] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, 2017), HotOS '17, ACM, pp. 156–161.

[6] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the Killer Microseconds. *Communications of the ACM 60*, 4 (Mar. 2017), 48–54.

[7] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO, 2014), OSDI '14, USENIX Association, pp. 49–65.

[8] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, ACM, pp. 267–283.

[9] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, 2008), NSDI '08, USENIX Association, pp. 309–322.

[10] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, 2013), USENIX ATC '13, USENIX Association, pp. 49–60.

[11] Brown, A., Oppenheimer, D., Keeton, K., Thomas, R., Kubiatowicz, J., and Patterson, D. A. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the The 7th Workshop on Hot Topics in Operating Systems* (Washington, DC, 1999), HotOS '99, IEEE Computer Society, pp. 32–37.

[12] Cai, Z., Vagena, Z., Perez, L., Arumugam, S., Haas, P. J., and Jermaine, C. Simulation of database-valued Markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 637–648.

[13] Carruth, C. [SLH] Introduce a new pass to do Speculative Load Hardening to mitigate. http://reviews.llvm.org/rL336990, 2018. Accessed: 2018-09-27.

[14] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (June 2008), 4:1–4:26.

[15] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, 2010), SoCC '10, ACM, pp. 143–154.

[16] Corbet, J. KAISER: hiding the kernel from user space. http://lwn.net/Articles/738975/. Accessed: 2018-09-27.

[17] Corbet, J. Meltdown/Spectre mitigation for 4.15 and beyond. http://lwn.net/Articles/744287/, 2018. Accessed: 2018-09-27.

[18] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation* (Berkeley, CA, 2004), OSDI '04, USENIX Association, pp. 10–10.

[19] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, 2007), SOSP '07, ACM, pp. 205–220.

[20] DPDK Project. Data Plane Development Kit. http://dpdk.org/. Accessed: 2018-09-27.

[21] Dragojević, A., Narayanan, D., and Castro, M. RDMA Reads: To Use or Not to Use? *IEEE Data Engineering Bulletin 40*, 1 (2017), 3–14.

[22] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, 2014), NSDI '14, USENIX Association, pp. 401–414.

[23] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, 2015), SOSP '15, ACM, pp. 54–70.

[24] Ford, B., and Cox, R. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Berkeley, CA, 2008), USENIX ATC '08, USENIX Association, pp. 293–306.

[25] Freedman, C., Ismert, E., and Larson, P. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin 37*, 1 (2014), 22–30.

[26] Geambasu, R., Levy, A. A., Kohno, T., Krishnamurthy, A., and Levy, H. M. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, BC, 2010), OSDI '10, USENIX Association, pp. 323–336.

[27] Google LLC. NaCl and PNaCl. http://developer.chrome.com/native-client/nacl-and-pnacl. Accessed: 2018-09-27.

[28] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation* (Berkeley, CA, 2000), OSDI '00, USENIX Association.

[29] Hazelcast. Hazelcast the Leading In-Memory Data Grid - Hazelcast.com. http://hazelcast.com/. Accessed: 2018-09-27.

[30] HUNT, G., AND LARUS, J. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review 41/2* (April 2007), 37–49.

[31] INTEL CORPORATION. Flow Director. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf`. Accessed: 2018-09-27.

[32] INTEL CORPORATION. Intel®Data Direct I/O technology. `http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`. Accessed: 2018-09-27.

[33] JACOBSEN, C., KHOLE, M., SPALL, S., BAUER, S., AND BURTSEV, A. Lightweight Capability Domains: Towards Decomposing the Linux Kernel. *SIGOPS Operating Systems Review 49*, 2 (Jan. 2016), 44–50.

[34] JUNG, R. LLVM loop optimization can make safe programs crash #28728. `http://github.com/rust-lang/rust/issues/28728`, 2018. Accessed: 2018-09-27.

[35] JUNG, R., JOURDAN, J., KREBBERS, R., AND DREYER, D. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages 2* (2018), 66:1–66:34.

[36] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, 2014), SIGCOMM '14, ACM, pp. 295–306.

[37] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI '16, USENIX Association, pp. 185–201.

[38] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter RPCs can be General and Fast. *CoRR abs/1806.00680* (2018).

[39] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008), 1496–1499.

[40] KIRIANSKY, V., AND WALDSPURGER, C. Speculative Buffer Overflows: Attacks and Defenses. *CoRR abs/1807.03757* (2018).

[41] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy* (2019), S&P '19.

[42] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), CGO '04, IEEE, pp. 75–86.

[43] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, 2017), SOSP '17, ACM, pp. 234–251.

[44] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (New York, NY, 2015), ISCA '15, ACM, pp. 476–488.

[45] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, 2014), NSDI '14, USENIX Association, pp. 429–444.

[46] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 973–990.

[47] MAAS, R., HYRKAS, J., TELFORD, O., BALAZINSKA, M., CONNOLLY, A., AND HOWE, B. Gaussian Mixture Models Use-Case: In-Memory Analysis with Myria. *Third International Workshop on In-Memory Data Management and Analytics (IMDM'15)* (2015).

[48] MACCORMICK, J., MURPHY, N., NAJORK, M., THETH, C. A., AND ZHOU, L. Boxwood: Abstractions As the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Opearting Systems Design and Implementation* (Berkeley, CA, 2004), vol. 6 of *OSDI '04*, USENIX Association, pp. 8–8.

[49] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, 2006), USENIX-SS '06, USENIX Association.

[50] MICROSOFT, INC. Transact-SQL Reference (Database Engine). `http://docs.microsoft.com/en-us/sql/t-sql/language-reference`. Accessed: 2018-09-27.

[51] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference* (Santa Clara, CA, July 2015), USENIX ATC '15, USENIX Association, pp. 291–305.

[52] NEO4J, INC. Neo4j, the World's Leading Graph Database. `http://neo4j.com/`. Accessed: 2018-09-27.

[53] NEUMANN, T. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment 4*, 9 (2011), 539–550.

[54] ORACLE, INC. Oracle Database 12c PL/SQL. `http://www.oracle.com/technetwork/database/features/plsql/index.html`. Accessed: 2018-09-27.

[55] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (Aug. 2015), 7:1–7:55.

[56] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI '16, USENIX Association, pp. 203–216.

[57] PARADIGM4, INC. Paradigm4: Cretators of SciDB a computational Database. `http://www.paradigm4.com/`.

[58] PREKAS, G., KOGIAS, M., AND BUGNION, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, 2017), SOSP '17, ACM, pp. 325–341.

[59] QIN, H., LI, Q., SPEISER, J., KRAFT, P., AND OUSTERHOUT, J. Arachne: Core-Aware Thread Management. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, 2018), OSDI '2018, USENIX Association.

[60] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[61] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (1996), OSDI '96, USENIX Association, pp. 213–227.

[62] SEVILLA, M. A., WATKINS, N., JIMENEZ, I., ALVARO, P., FINKELSTEIN, S., LEFEVRE, J., AND MALTZAHN, C. Malacology: A Programmable Storage System. In *Proceedings of the 12th European Conference on Computer Systems* (2017), Eurosys '17, ACM, pp. 175–190.

[63] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), IEEE, pp. 1–10.

[64] STONEBRAKER, M., AND KEMNITZ, G. The POSTGRES Next Generation Database Management System. *Communications of the ACM 34*, 10 (Oct. 1991), 78–92.

[65] STONEBRAKER, M., AND WEISBERG, A. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin 36*, 2 (2013), 21–27.

[66] SULLIVAN, M., AND STONEBRAKER, M. Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of the VLDB Endowment* (1991), VLDB '91, VLDB Endowment, pp. 171–180.

[67] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 207–222.

[68] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL: Documentation: 10: H.4. Extensions. `http://www.postgresql.org/docs/10/static/external-extensions.html`. Accessed: 2018-09-27.

[69] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles* (New York, NY, 1993), SOSP '93, ACM, pp. 203–216.

[70] WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. Extensible Security Architectures for Java. *SIGOPS Operating Systems Review 31*, 5 (Oct. 1997), 116–128.

[71] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, 2015), SOSP '15, ACM, pp. 87–104.

[72] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), S&P '09, IEEE, pp. 79–93.

[73] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, CA, 2012), NSDI '12, USENIX Association, pp. 15–28.