Narrowing the Gap Between Serverless and its State with Storage Functions

Tian Zhang
University of Utah

Dong Xie
University of Utah

Feifei Li
University of Utah

Ryan Stutsman
University of Utah

ABSTRACT

Serverless computing has gained attention due to its fine-grained provisioning, large-scale multi-tenancy, and on-demand scaling. However, it also forces applications to externalize state in remote storage, adding substantial overheads. To fix this "data shipping problem" we built *Shredder*, a low-latency multi-tenant cloud store that allows small units of computation to be performed directly within storage nodes. Storage tenants provide Shredder with JavaScript functions (or WebAssembly programs), which can interact directly with data without moving them over the network.

The key challenge in Shredder is safely isolating thousands of tenant storage functions while minimizing data interaction costs. Shredder uses a unique approach where its data store and networking paths are implemented in native code to ensure performance, while isolated tenant functions interact with data using a V8-specific intermediate representation that avoids expensive cross-protection-domain calls and data copying. As a result, Shredder can execute 4 million remotely-invoked tenant functions per second spread over thousands of tenants with median and 99th-percentile response latencies of less than 50 µs and 500 µs, respectively. Our evaluation shows that Shredder achieves a 14% to 78% speedup against conventional remote storage when fetching items with just one to three data dependencies between them. We also demonstrate Shredder's effectiveness in accelerating data-intensive applications, including a *k*-hop query on social graphs that shows orders of magnitude gain.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

ACM Reference Format:
Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and its State with Storage Functions. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3357223.3362723

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6973-2/19/11.
https://doi.org/10.1145/3357223.3362723

1 INTRODUCTION

Recent trends in cloud applications are improving resource utilization at the expense of efficiency; today, there is a growing gap between cloud applications and their data. This has gap has been widened by serverless computing, which cloud providers and their customers are increasingly embracing. Customers provide short functions to a cloud provider, who takes care of invoking and scaling instances in response to customer events. Serverless computing enables fine-grained resource provisioning, high-density multi-tenant resource sharing, and on-demand scalability through new lightweight isolation techniques [13, 15, 25]. This benefits cloud providers by driving up utilization, and it benefits customers by better fitting costs to resource use.

A key challenge, though, is that serverless functions are stateless. Cloud providers count on this to ease provisioning and fault-tolerance. Realistic applications need access to data and state; this limitation forces functions to access all data remotely, and it forces them to externalize all state into remote cloud storage in order to preserve it across calls or to pass state to another function. Today, this is done with traditional cloud storage systems [11, 12, 14, 40], forcing serverless into a scheme that "ships data to code" [26]. This results in inefficiency in moving data over the network to and from low-throughput, high-latency stores. Moreover, traditional cloud storage systems cannot offer the same level of fine-grain resource accounting and on-demand scalability, thus compromise the benefits of the serverless model.

Others have recognized this and have augmented serverless architectures with fast "ephemeral" stores designed to hold state between different functions in chains of serverless functions [32, 47]. Fast ephemeral storage helps, but it doesn't address the fundamental problem of data shipping, and it doesn't help applications that want to efficiently compute on durable data at its location of record. Some databases are automatically scaled and have fine-grained accounting [56], but they often only provide limited data access models; customers cannot run arbitrary functions on their data. For example, SQL works for many applications, but it is a poor fit for many emerging applications. In particular, applications with deeper data dependencies like those with graph data models and applications like inference serving that require short, but computationally-intensive functions aren't practical with SQL.

Instead, we argue serverless represents a new opportunity not just for compute but also for storage. The solution to this inefficiency is to reverse the flow of data to compute by exploiting the natural portability of compute inherit in the serverless design. In this paper, we demonstrate this reversal with Shredder, a new cloud store designed for serverless function chains. In Shredder,

customers embed small *storage functions* within the store, allowing them to operate directly on their data.

Embedding computation within storage creates several challenges, so Shredder has the following key goals:

- **Programmability:** Tenants should be able to embed arbitrary functions within storage with seamless access to their data.
- **Isolation:** Both the code and storage of different tenants should be isolated from each other to ensure security and data integrity.
- **High Density and Granularity:** Shredder should support thousands of tenant functions with fine-grained resource tracking to maximize resource utilization.
- **Performance:** Shredder should make as much of the raw performance of a storage server available to applications as possible.

Shredder's isolation is provided by V8 runtime with a novel design where tenants' data is bound into their runtime avoiding runtime boundary crossing costs. To meet its performance goals, Shredder uses kernel-bypass networking and a user-level TCP stack, and tenants directly route function invocations requests to particular storage server cores in order to avoid centralized request dispatching bottlenecks. Combined with its unique approach to avoiding data-interaction overheads, a single Shredder server can process more than 4 million function invocation requests per second with less than 50 µs response times. We also demonstrate Shredder's effectiveness in accelerating data-intensive applications, including a k-hop query on social graphs that shows orders of magnitude of performance gain.

To summarize, this paper makes the following contributions:

- We identify the gap between serverless computing and state storage; we propose *storage functions* to address this gap; and we demonstrate the promise of the approach through Shredder's design and implementation.
- We show how storage functions eliminate the network gap between applications and data, and we propose a novel technique that relies on an V8 intermediate representation (CSA) to avoid runtime boundary crossing costs, closing the gap further.
- We show how embedding compute within storage fits together with modern, high-performance networking goals; specifically, we show it helps support techniques like kernel-bypass with zero-copy data movement and scalable request dispatching.
- Finally, we conduct a comprehensive evaluation showing Shredder's benefits and cost. In addition, we show how much acceleration Shredder can provide for data-intensive applications.

The rest of the paper organized as the following: §2 describes the overall design of Shredder and its trust model. Then, we describe Shredder's implementation details, including its novel system techniques Shredder adopted in §3. In S4, we dissect the costs and benefits of Shredder. Furthermore, we show how much Shredder is able to accelerate data-intensive applications. Finally, we cover related work in §5 and conclude in §6.

2 SHREDDER DESIGN

Internally, Shredder consists of three layers: a networking layer, a storage layer, and a function layer (Figure 1). Each CPU core runs all three layers, but CPU cores follow a shared-nothing design; the state of these layers is partitioned across CPU cores to avoid contention

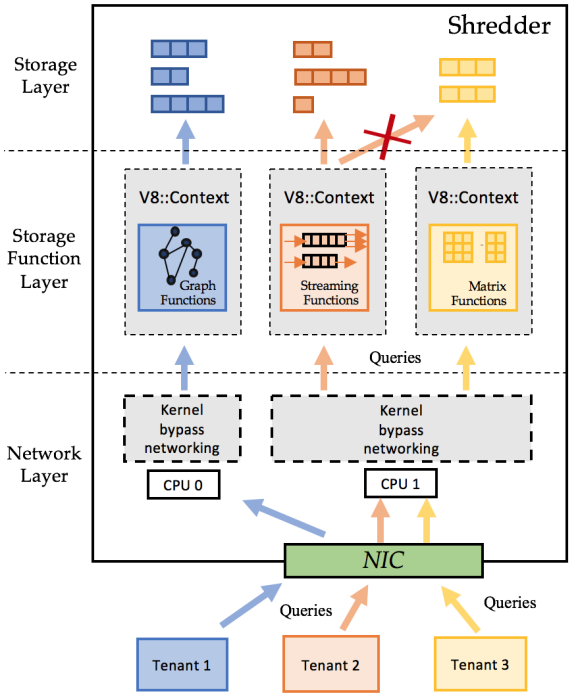


Figure 1: Shredder storage functions add functionality to the storage data model, can safely use low-level hardware, and are isolated with inexpensive language-level guarantees.

and synchronization overheads. The *storage layer* hosts all tenants' data in memory and has a `get()`/`put()` key-value interface. The *network layer* handles network connections, protocol processing, and the requests of all tenants. For each incoming request it calls through to the storage layer if the request is a simple `get()` or `put()` operation. If a request specifies a particular storage function, then the network layer calls through to the function layer. Finally, the *function layer* matches incoming requests to their storage function code and context (that is, the environment and state associated with that storage function), and it executes the operation within a per-core instance of the V8 runtime. Each V8 runtime has a set of embedded trusted access methods to avoid expensive calls between the function runtime and the storage layer.

The reason for this three layer design is rooted in Shredder's goal of efficient but fine-grained resource sharing. First, each of the layers can map any resources (compute, networking, storage) to any tenant at any granularity. For example, the storage layer makes it so that storage resources need not be mapped in and out of V8 runtimes, which would be costly. Similarly, a single V8 instance can only host a single thread at a time, so binding data to a particular runtime also fixes the amount of computation that can be done on that data. By separating storage from functions, Shredder avoids this coupling. Shredder must also be able to fluidly map network resources across tenants. One option for maintaining the performance of kernel-bypass while managing thousands of tenants would be to give each tenant its own hardware-virtualized access to the network card. However, even data center network interface cards typically

only support about 100 virtual functions and cannot scale to thousands of tenants on a single machine [1]. Beyond that, Shredder's networking stack needs a global view across tenant requests to ensure fairness and flow control. Shredder's separated network layer takes care multiplexing these resources, which simplifies its function layer and improves scalability.

2.1 Storage Layer

Shredder's storage layer consists of a single in-memory record heap upon which all tenants' records are allocated. Different tenants' records are intermixed at arbitrary granularities. Having no spatial-segregation eliminates any internal fragmentation between tenants' storage [48]. Storage isolation is enforced by segregating hash tables; the store keeps a hash table for each tenant that maps keys for its records to data. To ensure performance, the storage layer is implemented in native code and exposes `get()` and `put()` functions.

2.2 Storage Function Layer

The function layer keeps an independent instance of the V8 runtime per-core. This lets tenants install and run storage functions while providing lightweight isolation. Specifically, tenants provide their own storage functions in JavaScript (or compiled to WebAssembly [58]), and each function is bound to a per-tenant context in which it can be invoked. The V8 runtime's just-in-time (JIT) compiler translates functions into efficient code, and it also acts as a sandbox, preventing one tenant's code from manipulating stored data belonging to another tenant. V8 is lightweight enough that it can easily support thousands of concurrent tenants. Entry/exit in/out of V8 contexts is less expensive than hardware-based isolation mechanisms, keeping request processing latency low and throughput high. When the network layer receives a request to run a storage function, the function's context is reused and re-entered, which avoids any overhead in creation the function's environment and allows functions to keep volatile state across calls.

Separating storage from functions does come at a cost. Functions are expected to interact intensively with data, so crossing in and out of the runtime can add significant overhead (196 ns per call) and can limit optimizations like inlining of storage access by the JIT compiler. As a result, each tenant context contains logic from the storage layer in an intermediate V8 representation (the CSA IR, §3.3). The storage layer retains ownership of the physical resources that make up the records and indexes, but CSA code within the runtime is given read-only access to these structures from within V8. The CSA version of `get()` implements the same hashing algorithm as the C++ hashing function for hash tables in the storage layer, so that it directly looks up values in the hash tables. From the storage function perspective, the store still supports the same `get()` and `put()` storage interface. Using code generated from the IR, tenants can efficiently traverse the structures without execution ever leaving the V8 runtime. For some data-intensive operations that access or aggregate over many records Shredder's unique trusted CSA functions can accelerate operations by more than 3× (§4.3).

Overall, these storage functions enable "shipping code to data," letting applications run selected functions directly on data and undoing the forced separation of compute and storage. Data-intensive application logic can be implemented in JavaScript to avoid network roundtrips and request processing overheads. With a precise view of function execution time, stores can precisely account for the compute and memory resources each function invocation consumes.

2.3 Network Layer

The network layer handles incoming packets, user-level network protocol processing (TCP), RPC request framing, user-level task scheduling, and request dispatch. It uses kernel-bypass networking and each CPU core in Shredder polls for incoming network packets. This avoids the overheads of the Linux kernel networking stack, system calls, interrupts, and scheduler interactions, and it gives Shredder high-throughput, low-latency, and fine-grained control over request routing and scheduling.

In particular, the network layer works together with tenants to spread load over cores to avoid centralized bottlenecks and spread load. Tenant network flows are routed to specific cores directly by the network card [27]. This also helps Shredder exploit per-tenant locality.

2.4 Trust Model

Store operators and tenants are the key agents that interact with Shredder. Tenants do not share state and cannot communicate through Shredder; its only goal is tenant isolation. Tenants should not be able to access each others' state, and tenants should not be able to access store state without going through its storage-function-facing interface.

Tenants provide functions to the store, and they can invoke those functions. Every tenant's code is run in a separate V8 `Context` that is associated with that tenant, and storage function code interacts with the store through the same `get()` and `put()` interface that remote tenant operations use. So, the interface itself provides no new avenues for attack by tenants. Each `Context` is bound to the tenant that installed it, and the store never allows operations on records associated with a tenant id that differs from the `Context` operating on it. The store only returns views of records to a tenant that were put there by that tenant. Bounds checking on those views ensures tenants cannot see each others' data. Isolated V8 code cannot manipulate these views or otherwise forge and manipulate references.

Shredder's Trusted Computing Base (TCB) includes the OS kernel, libraries (including DPDK [4] for kernel-bypass), and hardware (CPU, network card) on top of which it runs; these are not directly exposed to users, but vulnerabilities in them can still lead to potential exploits. Shredder's safety critically depends on V8's isolation of untrusted code, which is not proven and does see a few reported vulnerabilities per year.

Recent speculative execution attacks [33] that leak data through microarchitectural side channels complicate confidentiality for Shredder. Both hardware manufacturers and runtime developers have worked to mitigate these attacks, but some of them are notoriously difficult to mitigate without significant performance impacts [7]. Furthermore, the discovery of these types of attacks has persisted [38, 41, 54, 55, 59]. V8 mitigates speculative bounds-check-bypass and protects against branch target poisoning [53], though

Mechanism	Context Creation	Two-way Context Switch
Processes/C++	763 µs	2,242 ns
v8::Context/WASM	889 µs	110 ns

Table 1: Isolation context creation and context switch cost for OS, hardware-based isolation versus V8. Context switch time includes the time to transfer control into an isolated no-op procedure and back to trusted code. (See Section 4 for hardware setup.)

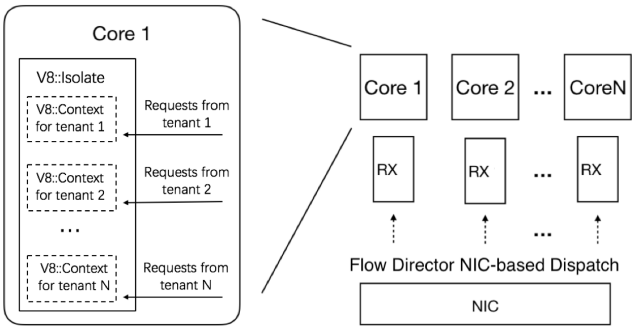


Figure 2: Tenant Isolation in Shredder. Tenant's requests are steered to specific cores by the NIC hardware. Each tenant keeps a `Context` that isolates its state and code from other tenants'. A per core V8 `Isolate` processes each incoming request one-at-a-time within the `Isolate`. Tenant-provided code can manipulate records by making calls from within the executing `Context` into a native-code interface provided by the storage; §3.3 discusses how Shredder optimizes away these cross boundary calls in most cases.

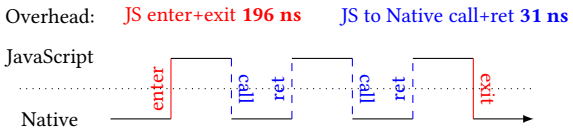


Figure 3: Boundary Crossing Costs.

other attacks like speculative store bypass don't have inexpensive software fixes and one must rely on hardware mitigations [2, 7] to avoid severe performance penalties. Even with mitigations, Shredder's approach has confidentiality risks if tenants are fully and mutually distrusting since future attacks are likely to emerge.

3 IMPLEMENTATION

Shredder is built on Seastar [49], a runtime that combines kernel-bypass through Intel's DPDK [4] with a lightweight event-driven, asynchronous programming framework built around a shared-nothing philosophy, making it easy to scale across cores. Network connections are sharded by the network card, so that each core exclusively handles a subset of tenant connections; we augment this dispatching strategy with a tenant-aware partitioning rather than just random, flow-based load distribution. Communications between cores happen through message passing.

Each core in Shredder runs with a single kernel thread pinned to it to avoid context switch overhead and scheduler interference. Each core executes a loop that is part of the networking layer. This loop polls incoming packet receive queues; performs TCP transport processing; and dispatches requests to handler fibers (fibers in Shredder is stackless). Storage function requests are dispatched to V8 fibers that invoke the local V8 isolate to run the requested function. In the scope of this paper, storage functions are implemented in JavaScript. Requests access records indexed by unordered hash-based tables that are stored in the underlying key value store in an in-memory shared record heap (Figure 1). Functions can emit data to incorporate it in the response message for the request. The result is returned to the event dispatching fiber which returns the result to the tenant over the network.

We describe the details of each of the key aspects of Shredder's design in the remainder of the section.

3.1 Isolation and Context Management

Cloud databases commonly host thousands of tenants per instance [56], so Shredder's isolation costs must be minimal. Shredder must enforce isolation in both for record storage and for tenant-provided code. Tenants can only fetch records through the hash table that indexes the records they stored themself; furthermore, the code they provide is memory-safe, so they cannot access records that they did not create.

Shredder relies on V8's `Contexts` [3] (Figure 2) to isolate tenant-provided code. Each `Context` is an isolated execution environment that allows independent code to run in the same V8 runtime instance. An independent V8 instance (an `Isolate`) is allocated for each core. Each `Isolate` runs code from different tenant `Contexts` one-at-a-time; the contexts enforce inter-tenant isolation of code. After initial creation within a specific `Isolate`, a `Context` can be entered and exited repeatedly. We measured the cost to transfer control into the V8 runtime and then back to the native store code at 196 ns and the cost for JavaScript functions to transfer control to native storage code and then back to JavaScript at 31 ns (Figure 3), making it more than an order of magnitude faster than process context switch (Table 1). Table 1 also shows that the bare cost of a two-way switch into and then out of a V8 `Context` in C++ code without calling a JavaScript function within the `Context` is 110 ns. Overall, these fast enter/exit times keeps Shredder's throughput high, and Section 3.3 shows how the cost of calling native code from a JavaScript function can be avoided altogether during data access, limiting this overhead to the initial procedure invocation and response.

3.2 Zero-copy Data Access

Each storage function invocation runs within a V8 context, which receives data in the request receive buffers as arguments. Each invocation can output values that they compute or values from the store into response transmit buffers.

```
function k_hop(id, k) {
    // Get "id"; cast as uint32 array.
    var friends = new Uint32Array(get(id));
    var sum = friends.length;

    // Argument "k" is the number of hops
    if (k == 0)
        return sum;

    for (var i = 0; i < l; i++) {
        // Recurse
        sum = sum + k_hop(friends[i], k-1);
    }
    return sum;
}
```

Listing 1: k-hop query on the social graph.

All key-value pairs stored in Shredder are stored outside of V8 contexts, and they are intermixed within the storage heap. Storage functions use `get()` and `put()` calls to access these records. In the future, we expect to support range operations over ordered indexes as well, but the current version only supports point lookups via unordered hash indexes.

Local `get()` and `put()` operations are optimized to avoid copying records into and out of storage whenever possible. That is, function logic can operate over records *in-situ* with no copying. This is a powerful optimization exclusive to storage functions, allowing operations on large numbers of records with minimal overhead. Whereas operating over records through remote `get()` `put()` requires moving every record over the network.

Shredder makes this work through in-`Context` `ArrayBuffers`. `ArrayBuffers` allow the Shredder store to safely pass bounded views of memory into V8 `Contexts` with no copying. Since `ArrayBuffer` works as a bounded buffer over generic binary data, code within a `Context` can read arbitrary data from the buffer. V8's `DataView` allows code to give semantics to the underlying data; for example, operating over portions of it as an unsigned integer or as a floating point value.

`ArrayBuffer` works in the other direction as well. When isolated code creates a new `ArrayBuffer`, the underlying memory are allocated through `ArrayBuffer::Allocator`, which can be specified to use `malloc()` to directly allocate memory from the Shredder store's heap. `ArrayBuffer` provides an `Externalize()` function to transfer ownership of memory to the store, which prevents garbage collection by V8 while the store holds a view to the underlying data. Overall, `ArrayBuffer` provides an efficient way to transfer references to chunks of data between isolated V8 `Contexts` and the Shredder store. Keys and values in Shredder are stored as opaque binary blobs, so `ArrayBuffers` are used with them pervasively.

Listing 1 shows an example that implements a k-hop operation that finds the transitive closure of a set of friends rooted at a single user profile in a social graph up to some depth k. The store maps user ids to friend lists, which holds variable length arrays containing user ids (Figure 4). Then, the function fetches the specified user's friend list, and it iterates over the list and recurses. The `get()` operation in the function only binds an `ArrayBuffer` avoiding data movement within the store as friend lists are traversed. The code only uses simple `get()` operations, but workloads like this are common enough that Facebook implements a customized social graph store for similar queries [18]. Here, Shredder saves round trips and data movement without the need for a specialized store.

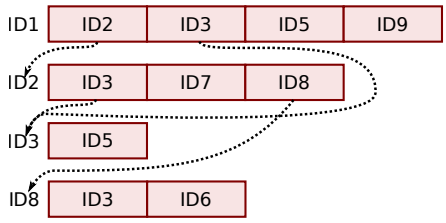


Figure 4: Structure of the social graph data.

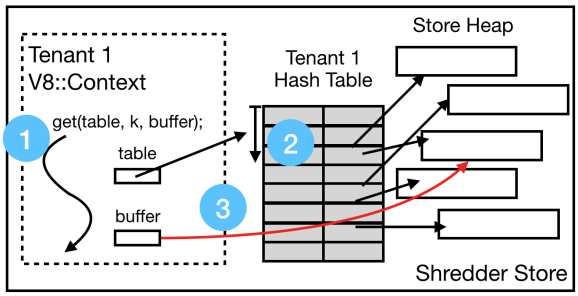


Figure 5: Exitless `get()` Operations via CSA. `get()` is implemented entirely within in tenant `Contexts`, so no cross-domain calls are needed to fetch views of records within tenant-provided functions. 1. Functions take as input a global `ArrayBuffer` "table" which points to the tenant's hashtable, and provides another empty `ArrayBuffer` "buffer" which is used to point to the result value. 2. Internally the `get()` CSA computes a key hash and indexes into the tenant-specific hash table which "table" points to. 3. `get()` rebinds the tenant-provided `ArrayBuffer` "buffer" to point to the requested record.

3.3 Eliminating Boundary Crossings with CSA

For some operations that work with a few larger chunks of data (a few tens of kilobytes), Shredder's simple zero-copy `get()`/`put()` interface is efficient. However, some data-intensive operations suffer when control must be continuously transferred back and forth between a `Context` and trusted store code. The control transfers are implemented via procedure call, and, by our measurements, simple calls from JavaScript to native code only suffer 31 ns (Figure 3) of overhead per call. These costs add up and they prevent optimization and inlining by V8. For functions that work with lots of fine-grained records, the impact is significant. For example, these costs can slow down iteration over large record sets as much as 3× in some simple cases we encountered.

To solve this, Shredder completely eliminates control transfer between tenant-provided code and the store for simple read-access to

records using a unique form of data structure co-design where untrusted tenant code can directly access parts of the store's metadata and indexes. Figure 5 shows how this works. Rather than calling into a native code procedure provided by the store to perform the lookup of a value, `get()` operations are implemented within V8 `Contexts` themselves. The store binds the bucket array of a tenant's hash index into its `Context`. As a result, the tenant can traverse the hash index itself to find values, saving the trust boundary crossing and eliminating the call overhead.

Shredder achieves this by implementing these in-`Context` store operations with V8's CodeStubAssembler (CSA) [9]. CSA is a low-level intermediate representation built on top of V8's TurboFan code generation architecture [10]. CSA is portable across hardware architectures, and it can be translated into highly efficient machine code. Both the V8 Ignition interpreter [5] and TurboFan optimization compiler leverage CSA to achieve portability, efficiency and interoperability. V8 developers also use CSA to improve `Context` builtins – for example, the typical base functions that are part of every JavaScript context. So, many of the functions defined in the ECMAScript specification are written in CSA. Historically some builtins were written in C++, while performance critical builtins were hand written in assembly for each hardware platform to eliminate expensive V8-to-C++ calls. The hand written assembly wasn't portable and was hard to maintain, so many builtins are now ported to CSA to avoid such problems. CSA is low-level but it can perform many simple data manipulation operations efficiently. For example, CSA code can load data from a specified address, and it can modify the internal data of JavaScript objects.

The Shredder CSA-based `get()` operation uses an existing output `ArrayBuffer` provided by the tenant procedure running within its `Context`; it is populated with the output value upon return. Shredder's `get()` takes an input key, computes its hash, indexes into the bucket array, and compares the key at the allocation pointed to by the bucket array. If the key matches, the output `ArrayBuffer` is bound to cover the return value, so the procedure gets a view of the data with no intervening copy. Tenants can't force arbitrary memory accesses through `get()`, because `get()` closes over the hash table `ArrayBuffer`. Tenants can only get access to the hash index through `get()`, which can only happen when the tenant's `Context` is executing the supplied `get()` CSA code. Tenants cannot overwrite or manipulate the CSA code, since V8 doesn't allow code to construct or manipulate pointers.

Finally, object creation and garbage collection is known to be costly in V8. Shredder must avoid these overheads on the fast path, which would otherwise erase the gains from its optimizations. Shredder's interface makes this easy by allowing objects like `ArrayBuffers` to be recycled. For example, when iterating over large sets of records, function code can create a single `ArrayBuffer` and pass it to each `get()`. Shredder's CSA-based `get()` simply retargets the `ArrayBuffer` to the new value, avoiding object allocation, deallocation, creation, or destruction costs.

4 EVALUATION

In our evaluation setup, we use three Emulab D430 machines (see Table 2 for full specifications). One machine runs the Shredder server; two others run a client. Both the clients and the server use a user-level TCP stack with kernel-bypass networking. One client machine simulates load offered by 1,024 tenants, each making requests at the same rate. Each simulated tenant runs in a closed-loop load, and server load is varied by varying the delay between requests and the number of requests that each tenant keeps pipelined to the server at a time. Each of the 1,024 tenants accesses its own set of 250,000 128 byte records, so the server stores about 32 GB of data in total. Tenants access keys according to YCSB-C's default skewed Zipfian distribution with $\theta = 0.99$. The second client machine probes median and 99^{th}-percentile response latency of the server with simple, back-to-back `get()` operations that fetch a single value while the background load runs. This makes the plotted response latencies independent of any response time variance in the workloads that we test with, which can vary widely.

CPU	2×Xeon E5-2630v3 2.40 GHz
RAM	64 GB 2133 MT/s DDR4
NIC	Intel X710 10GbE PCI-Express
OS	Ubuntu 14.04, Linux 4.4.0-116, DPDK 17.02.0, 16×1 GB Hugepages

Table 2: Experimental configuration. The evaluation was carried out on a three node cluster consisting of two clients and one storage server on CloudLab. Each node has two CPU sockets and 16 physical cores.

The Shredder server runs on 16 cores each servicing a disjoint set of tenants. Tenants are mapped randomly (but statically) to different cores directly by steering requests from a particular tenant's TCP connections to a specific DPDK receive queue that is paired with a single storage server core.

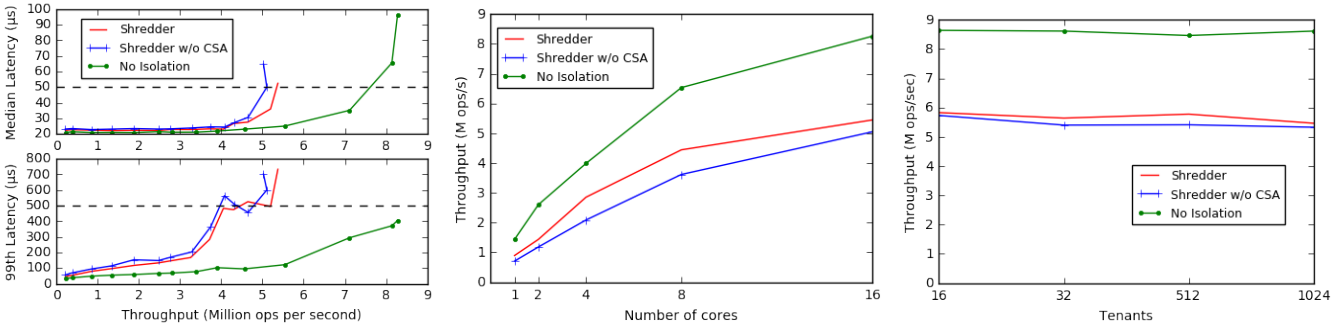
Shredder adds no extra expressivity to what tenants can compute over their data client-side, so its value comes when it can improve server throughput or reduce tenant-visible delays. Our evaluation explores Shredder's overheads compared to simple `get()` and `put()` operations and where those costs stem from. This calibrates how much work Shredder's programmability needs to make up for in order to improve efficiency relative to a typical key-value store. Finally, we explore applications ranging from select-project-join with simple aggregations, to simple graph analytics, and machine learning classifiers that show the end-to-end gains Shredder achieves along with where the approach breaks down.

Our evaluation focuses on four questions:

1. Does Shredder preserve low-latency operation? Our results show a Shredder store responds to 4 million storage-function invocations per second in 50 µs with tail latencies around 500 µs–several times faster than conventional multi-tenant stores.

2. Does Shredder scale across cores to achieve good throughput and server-side efficiency? Our results show Shredder's isolation increases server-side overhead, but Shredder still scales with the number of server cores when tenants can be divided among cores.

3. Can Shredder support thousands of isolated tenants? Our



(a) Latency vs. throughput **(b) CPU Scalability** **(c) Tenant Scalability**

Figure 6: Comparison of `get()` performance with and without using storage functions.

	1 Core	8 Cores	16 Cores
Shredder	479 ns	853 ns	1567 ns
No Isolation	52 ns	58 ns	73 ns

Table 3: Time spent performing a single `get()` request (excluding networking and dispatching time).

results show very small impact on throughput when running with more than 1,000 tenants.

4. Does Shredder's use of CSA improve the performance of data-intensive operations? Our results show that CSA can increase server throughput by about 3.5× with some data-intensive applications. The following subsections detail these results.

4.1 Overheads and Costs

Invoking operations through V8 adds overhead. To understand V8's costs we created a storage function that calls `get()` on the store and returns a 128 byte value. In one configuration (labeled `No Isolation`), the `get()` operation is serviced directly by the storage server, skipping the function layer altogether. In another configuration (labeled `Shredder w/o CSA`), this storage function is invoked, and the function calls `get()` on the storage layer, transferring control flow from V8 to the storage layer and back to fetch the value. In the final configuration (labeled `Shredder`), the storage function is invoked, and it performs the `get()` *without* transferring control flow to the storage layer; instead, the operation is executed entirely within the V8 runtime using trusted code generated using CSA. In practice, this storage function represents an unrealistic case for Shredder; tenants that need a single value from the store can remotely invoke `get()` without using a storage function to avoid overhead; however, it is useful for demonstrating the upper bound on costs of the approach relative to a standard key-value store. In the experiment, each tenant pipelines 8 requests at a time, pacing operations to vary the load.

Cost of Isolation: Figure 6a shows both the median and 99th-percentile response latency for each of the three configurations. In all cases, once the offered load exceeds the limits of the store, response time spikes as expected. Differences in the saturation points show the relative CPU cost of each of the approaches. Shredder's target is around 50 µs access times with a hard bound on 99th-percentile latency within 10× for `get()` operations, so we compare the throughputs of the different approaches at 50 µs median latency and 500 µs the 99th-percentile response latency. (In other experiments we ignore the latency requirement and just show the optimal throughput upper bound.) Native `get()` operations implemented and directly executed by native code (in C++) achieve 7.5 Mop/s; however, this approach is not CPU-bound in this case. Instead, the network is saturated for the `No Isolation` case; Figure 8a shows with smaller return value size native code can achieve 12.5 Mop/s. Calling `get()` operations from V8 slows the store to about 4 Mop/s. As a result, if each storage function invocation can perform just four data-dependent `get()` operations per invocation it can offset the slowdown due to V8's isolation. Note that these numbers are measured with Shredder running on 16 cores, later in CPU Scalablity section we see that when Shredder is running on 16 cores the V8 isolation cost is much higher than when Shredder is running on 1 core due to poor V8 scalability. We believe that with some engineering effort it's possible to improve V8 scalability, then the V8 isolation cost will be much cheaper than that we measured here.

CPU Scalability: To measure how the system scales, we measured the throughput of these same three configurations in performing a `get()` on different number of CPU cores (from 1 to 16 cores). At 16 cores, all of the physical cores of both NUMA sockets of the machine are occupied.

Figure 6b shows the system can scale nearly linearly. The reason that `No Isolation` shows lower than linear scalability at 16 cores is because the 10 Gbps network bandwidth is nearly saturated. Later experiments show that with smaller value sizes `No Isolation` can achieve higher throughput. Shredder and `Shredder w/o CSA` both show lower scalability; although we create independent V8 `Isolates` on each core, this may be in part due to the fact that there is still some resource sharing in the V8 engine (e.g. background code optimization and garbage collection tasks).

To better compare the scalability of Shredder and `No Isolation` without the impact of network saturation, we ran another microbenchmark to measure the CPU time spent performing a single `get()` request for Shredder and `No Isolation`. Table 3 shows the results. For Shredder, the CPU time nearly doubled when scaling

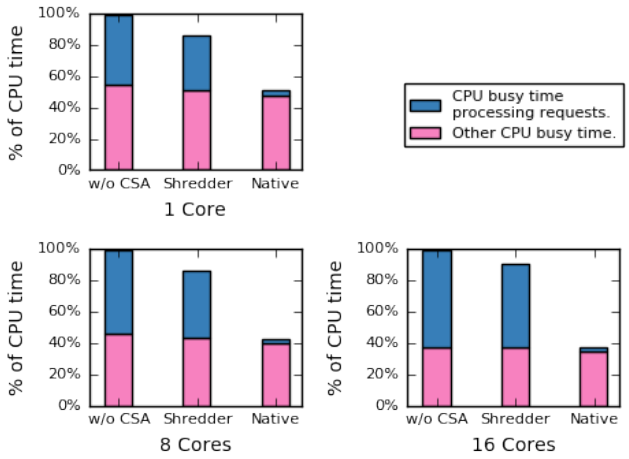


Figure 7: CPU Time Analysis

from 1 core to 8 cores and from 8 cores to 16 cores. Whereas for `No Isolation` the CPU time increase is much less. For Shredder, each request goes through the V8 engine, this confirms that the poor scalability scalability of V8 is the root cause for Shredder's low scalability. It is likely that V8's poor scalability can be improved, so we don't view this as a fundamental issue with Shredder's approach.

Tenant Scalability: In this experiment we run the server on 16 cores while fixing the number of client connections at 1,024 and varying the number of tenants on the server. When less than 1,024 tenants are simulated, multiple client connections will share the same tenant context. We run the same simple `get()` operations and measure the throughput. Figure 6c shows the number of tenants has almost no impact on `No Isolation` approach as expected. For Shredder and `Shredder w/o CSA` tenant contexts are implemented using `V8::Context` structures, and the V8 engine needs to switch between different `Contexts`. The experiment shows that switching between `V8::Contexts` adds a small overhead and the server scales well to more than 1,000 tenants with about 6% performance decline.

CPU time analysis: This experiment analyzes the amount of CPU time spent performing these `get()` requests for each of the three configurations, and we run it on 1, 8 and 16 cores respectively to show the impact of scaling to more cores.

Although server CPU is always running due to DPDK polling, we don't count polling toward the total since it effectively represents CPU idle time. Apart from total CPU busy time, we also measure the time spent processing requests, which is measured starting when a new request is received from the network stack until the response is written to an output buffer. Other CPU busy time, which is computed as the total CPU busy time minus CPU time processing requests, is mainly time spent in the user-level network stack receiving and sending data.

Figure 7 shows the results. The gap between the top of the bars to the top of each subgraph is CPU idle time. The graph shows that under the same load, the time spent on networking is almost the same for the three approaches, and the differences of time spent on processing requests shows the overhead of V8 isolation. There are two kinds of V8 isolation overhead: the first is the overhead of entering/exiting V8 to invoke an operation; the second is the overhead of V8-contained code getting data from the storage layer. Shredder eliminates the second overhead with CSA, so the gap between Shredder and `Shredder w/o CSA` shows CSA's benefit. The graph also shows V8 overhead increases when Shredder scales to more cores. On 1 core, requests processing accounts for 41% of Shredder's CPU busy time, which increases to 59% on 16 cores, due to the increase of V8 overhead.

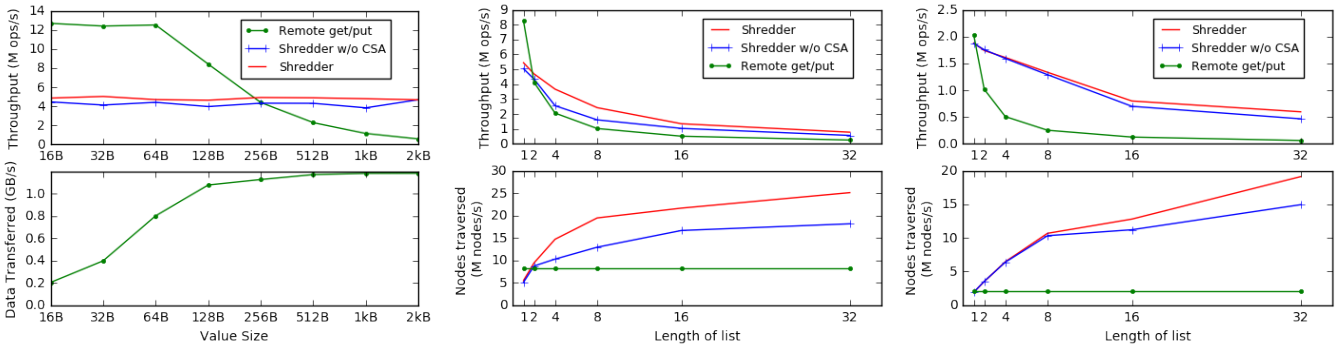
4.2 The Benefit

Minimizing data movement: In many cases pushing code to the storage server can greatly reduce the data transferred over network. To show this benefit we run a simple experiment in which clients issue requests to get a projection of multiple values stored on the server. Each stored value contains an array of 32-bit integers, and the goal of each request is to get the first element in the array. With Shredder the projection can be done on the server with a one-line storage function (Listing 2), which gets the value from the storage layer, projects the first 32-bit integer from the value, and returns the result. Without Shredder clients need to get the whole value from the server and then get the first integer from the value, which results in more data transferred over the network. We vary the size of the value and measure the throughput achieved by the 3 approaches.

The upper half of Figure 8a shows request processing throughput. Shredder performance is independent of value size, since the storage function returns one integer to the client and the amount of data transferred over the network is constant. The performance of `Remote get/put` drops for values of 128 bytes or more. The bottom half of Figure 8a shows the amount of data transferred per second over the network for the `Remote get/put` approach. For value sizes of 128 B or more network bandwidth saturates, and the server becomes bottlenecked. As a result, Shredder improves performance when its storage functions can reduce the amount of data transferred back to the client to less than 128 B.

Minimizing remote requests: Another benefit of pushing code to storage is in reducing the number of remote requests a client has to make to perform an operation, which result in fewer network round trips and client stalls. To show this benefit we demonstrate a simple application that traverses lists in the storage layer and returns the value at the end of each traversal. Each node in each list is stored as a single value in the storage layer; the key for the next node in the list is stored at the head of the value as a 32-bit integer. With Shredder the traversal can be implemented as a function shown as Listing 3. Without Shredder the client has to iteratively read value from the database to find the key of the next node.

The experiment varies the length of the list traversals. The upper half of Figure 8b shows the throughput each approach can achieve for traversals of varied length. The performance of `Remote get/put` drops greatly with longer traversals, because one network round trip is needed for each node traversed. Hence, these extra requests generate extra load, saturating the server and stalling clients waiting for responses. With traversals that access two or more nodes Shredder performs better than `Remote get/put` because



(a) Projection throughput as value size varies. **(b) Throughput for different gets per request.** **(c) List traversal without kernel bypass.**

Figure 8: Performance with and without using storage functions for projection and data dependent accesses.

```
function projection(key) {
    // Get value, cast, return first uint32.
    return new Uint32Array(get(key))[0];
}
```

Listing 2: Projection function.

```
function list_traversal(key, length) {
    // "key" arg indicates where to start.
    // "length" arg indicates traversal depth.
    for (var i = 0; i < length; i++) {
        // Cast as uint32 array.
        // First element indicates next node.
        key = new Uint32Array(get(key))[0];
    }
    ...
}
```

Listing 3: List traversal as a tenant-provided procedure.

only one network round trip (or more than four nodes when ignoring network bottlenecks). The bottom half of Figure 8b shows the number of nodes traversed per second at the server. With longer traversals Shredder can traverse many more nodes per second, since network request processing is amortized over more node accesses. Shredder performs significantly better than `Shredder w/o CSA` in this application, because this application accesses much more data than the previous examples; eliminating data movement within the store itself is key to good performance in this case.

Impact of Kernel Bypass: Figure 8c shows the result of running the list traversal experiment without kernel bypass. Kernel bypass lowers network overhead, so configurations of each of the approaches that depend more heavily on request-response are impacted most by eliminating kernel bypass. For example, of the three approaches shown, using traditional `get()` and `put()` operations suffers the most, since it needs to synchronously fetch every value traversed from the store. This shows that Shredder's storage functions are even more effective if network overheads are high. Without kernel bypass, CSA makes less of a difference than it does with kernel bypass, unless each request interacts with many data items.

4.3 Graph Application

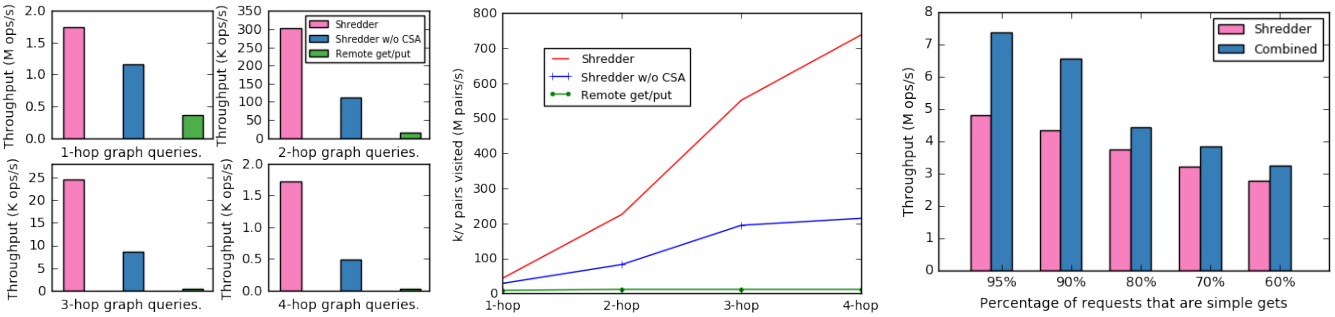
To evaluate Shredder with a more realistic workload, we built a graph application on the ego-Facebook dataset from the Stanford Large Network Dataset Collection [35]. The dataset consists of "circles" (or "friends lists") from Facebook. In Shredder, the dataset is stored as key value pairs where keys are IDs identifying a person in the social graph and the value is an array of IDs representing the friend list of that person (Figure 4). The graph has 4,039 nodes and 88,234 edges in total.

The application performs k-hop queries for randomly generated keys on the social graph. For example, for a one-hop query it gets the friend list of a person, then it gets the friend list of each friend and it sums up the length of the friend lists. Listing 1 shows the implementation of k-hop query. For `Remote get/put` a remote request is needed to get each friend list.

Figure 9a shows throughputs of the three approaches with different values of k. For one-hop queries we implement the `Remote get/put` approach on the client side and measure the throughput. For the other values of k, running `Remote get/put` is slow, so we estimate its best base time by calculating the number of nodes it needs to visit and dividing by best case throughput. The result shows that Shredder is a great improvement over `Shredder w/o CSA` and `Remote get/put`.

As k grows the number of visited social graph nodes (key-value pairs) explodes. Figure 9b shows the number of key value pairs accessed per second. With CSA, Shredder can traverse nearly 740 million key-value pairs per second — orders of magnitude (61×) more than using a plain key-value store and nearly 3.5× as many as without CSA.

Real-world workloads mix of different kinds of queries. One common combination is workloads where most queries can be served by simple primary-key lookup `get()` requests with a small portion of requests being more complicated queries. In fact, simple `get()` requests can just be served without V8 isolation to leverage the low cost of non-isolated `get()`s. Figure 9c compares the two approaches under mixed workloads of simple `get()` and one-hop graph queries. For the Shredder approach all requests including simple `get()`s are



(a) Throughput of k-hop graph query. **(b)** Number of k/v pairs visited per second. **(c)** Simple `get()` and 1-hop graph query mixed.

Figure 9: Graph Application Evaluation.

served through functions. For the other approach (labeled "Combined") the simple `get()` requests are served directly by the database without entering V8, and the one-hop graph queries are served through the function in V8. Throughput of two approaches are measured under varying percentage of simple `get()` requests. The result shows that the combined approach improves the performance significantly. The higher the percentage of simple `get()` requests in the workload, the more benefits from the combined approach.

4.4 Neural Network Application

For AI and machine learning use cases, databases are often used to serve trained models. Once trained, models must be deployed to a prediction serving system to provide low-latency predictions at scale. In many cases many models are needed to reflect different feature representations, modeling assumptions, and machine learning frameworks. In addition, multiple models can be combined in ensembles to boost prediction accuracy and provide more robust predictions with confidence bounds [19]. Usually in an online prediction serving system models are stored in a database. To make a prediction, the corresponding model is fetched from the database. Sometimes the input dataset is also stored in the same database.

Prediction systems can leverage Shredder to push prediction code to the database that stores the models, which can greatly reduce data movement over the network, as models can be very large in size. But there's also a downside of this approach, because prediction code are compute intensive, on Shredder prediction functions must be implemented in JavaScript or WebAssembly, both are slower than C++ native code[29]. It is a question if the benefits of Shredder can outweigh the downside for such applications.

To answer the question, we built an prediction service on Shredder. We trained fully connected neural networks with one hidden layer on the Iris dataset and Wine dataset from the UC Irvine Machine Learning Repository [20] and load the resulting models in Shredder. Neurons are represented as arrays of 32-bit float numbers, and the neural network is stored as one big value of neurons concatenated sequentially. For the Iris dataset, which has four features, the trained neural network has two neurons in its hidden layer to achieve optimal accuracy and three output neurons; the total size of the neural network is 76 bytes. The trained neural network for the Wine dataset, which has 13 features, has five neurons in its

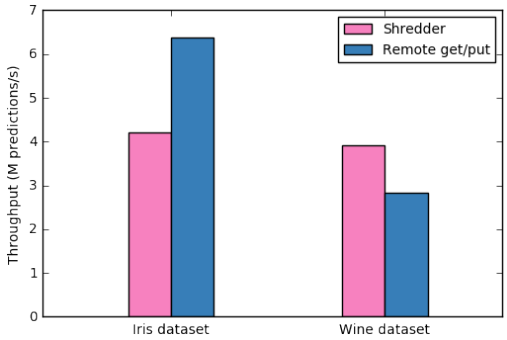


Figure 10: Throughput of neural network predictions

hidden layer and the total size of the network is 352 bytes. The inputs for the predictions are also stored in the database. The storage function `get()`s the model and an input vector from the store and then makes an inference. The `Remote get/put` approach fetches the neural network and the input and computes the inference at the client.

Figure 10 shows the throughput of Shredder and `Remote get/put` approach for the two datasets. With small models of 76 B, `Remote get/put` approach achieves higher throughput than Shredder. This actually shows the case in which the benefits of Shredder cannot overweigh the disadvantage. When requests are CPU-bound and data transfer over the network is small, the `Remote get/put` approach achieves better performance, because inference is computed in C++ code at client, whereas on Shredder inference is implemented as JavaScript storage function which is less efficient than C++ code. For the Wine dataset, which has larger models, Shredder outperforms `Remote get/put`, since `Remote get/put` becomes bottlenecked by the network bandwidth. The result shows that Shredder can still show benefits even in cases of CPU-bound requests if `Remote get/put` entails a large amount of data transfer over the network. In real world applications most models are much larger than these, so Shredder has potential to provide benefit over traditional `Remote get/put` approach without requiring a specialized model-serving service or customization at the storage layer.

5 RELATED WORK

Shredder's core dispatch is similar to many other recent kernel-bypass-based in-memory stores [21, 22, 31, 36, 37, 45] except that its set of operations are runtime extensible. These fast stores perform millions of operations per second per machine with a few microseconds response time, which led to Shredder's different approach to code isolation that eschews conventional hardware-based protection mechanisms. This leads to another place where Shredder differs from past systems: in request heterogenity. Most kernel-bypass-based stores service small, simple requests, but Shredder operations can be general-purpose code. Some recent work has started to tackle the scheduling issues this creates [30, 44, 46], but, in Shredder, tenant-code must be cooperative in order for it to maintain fairness and good response times.

RDMA is another way to make storage more efficient, since it eliminates CPU overhead at the receiver. But, RDMA operations cannot have application-specific semantics, so they are inefficient in practice: tenants would still have to perform the same computation on the data that they receive that a server could have. Shredder still uses efficient kernel bypass networking, but avoids (so called, one-sided) RDMA for this reason.

Using V8 to isolate code in a store or database isn't a new idea. NodeJS [43] is a general process container for running arbitrary JavaScript or WebAssembly code. MongoDB [42] popularized JavaScript within databases, and multi-tenant versions inspired by it are operated by major cloud infrastructure providers [16]. Shredder really differs primarily in its focus on performance, which forces some aggressive design decisions. In particular, Shredder's focus on kernel-bypass networking with dense multi-tenancy emphasizes all overheads; this leads to its unique approach centered on storage structure co-design, which allows operation over structures fully within tenant-provided code.

Comet [24] is an decentralized hash table that allows applications to extend its functionality using Lua sandboxed extensions. Similarly, Splinter [34] is a Rust-extensible in-memory key-value store. Like Shredder, extensions in these systems also interact with the store through its get/put interface. Shredder storage functions are language-agnostic since they can be supplied as WebAssembly, and Comet is missing Shredder's emphasis on performance. Neither Comet nor Splinter eliminates the boundary between functions the data they operate on.

Finally, software-fault isolation (SFI) [28, 50, 52] has long sought to create low cost control transfer schemes [6, 17, 23, 39, 60] to contain untrusted computation within a process. Many schemes have explored approaches to avoid copying data across these trust boundaries, but this generally involves increasing trust in the isolated code (for example, only restricting write access to some state). Postgres supports database extensions [8], and hardware isolation [51] and SFI [57] have both been applied to it, though at very different time scales and performance.

6 CONCLUSION

Today's cloud storage systems avoid coupling compute and storage to ease scaling and fault-tolerance and to drive high-utilization. However, as compute grows more granular, applications are increasingly impaired by this gap between computation and data.

Shredder takes a step toward resolving that gap: while applications can use it as a normal key-value store, they can also embed compute directly next to their data. Serverless applications already logically decompose applications into fine-grained units in order to ease scaling; we view Shredder as the first step toward exploiting that by recognizing that logically decoupling compute and storage need not mandate physical decoupling.

Shredder aggressively eliminates data movement by bringing functions as near as possible to the data they operate on. It eliminates movement across the network, between the network card and storage functions, and between storage functions and data. By attacking all layers of the storage server, Shredder achieves millions of tenant-provided functions per second over stored data with remote access latencies of 50 µs.

ACKNOWLEDGMENTS

Thanks to our anonymous reviewers and to our shepherd, Tim Harris, whose feedback helped improve this work. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1750558, CNS-1566175, III-1816149, and III-1619287. 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] http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-x710-brief.pdf.
[2] https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html.
[3] https://github.com/v8/v8/wiki/Embedder%27s-Guide, Title = Embedder's Guide.
[4] dpdk. http://dpdk.org.
[5] Firing up the Ignition interpreter. https://v8.dev/blog/ignition-interpreter.
[6] NaCl and PNaCl. http://developer.chrome.com/native-client/nacl-and-pnacl. Accessed: 6/19/2017.
[7] NVD - CVE-2018-3639. https://nvd.nist.gov/vuln/detail/CVE-2018-3639.
[8] PostgreSQL: Documentation: 10: H.4. Extensions. http://www.postgresql.org/docs/10/static/external-extensions.html. Accessed: 6/19/2017.
[9] Taming architecture complexity in V8 - the CodeStubAssembler. https://v8project.blogspot.com/2017/11/csa.html.
[10] V8: Behind the Scenes (February Edition feat. A tale of TurboFan). https://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition.
[11] Amazon Web Services, Inc. Amazon DynamoDB - Overview. https://aws.amazon.com/dynamodb/.
[12] Amazon Web Services, Inc. Amazon Simple Queue Service (SQS) | Message Queuing for Messaging Applications | AWS. https://aws.amazon.com/sqs/.
[13] Amazon Web Services, Inc. AWS Lambda. https://aws.amazon.com/cn/lambda/.
[14] Amazon Web Services, Inc. Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service. https://aws.amazon.com/s3/.
[15] Amazon Web Services, Inc. Firecracker. https://firecracker-microvm.github.io.
[16] Azure DocumentDB. Running JavaScript in Azure DocumentDB with Chakra. https://azure.microsoft.com/en-us/blog/the-road-ahead-for-azure-documentdb-with-chakracore/.
[17] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.
[18] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.
[19] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 613–627, 2017.
[20] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017.

[21] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[22] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70, 2015.

[23] B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, USENIX ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[24] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 323–336, 2010.

[25] Google, Inc. Google Cloud Functions. https://cloud.google.com/functions/.

[26] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[27] Intel Corporation. Flow Director. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[28] C. Jacobsen, M. Khole, S. Spall, S. Bauer, and A. Burtsev. Lightweight Capability Domains: Towards Decomposing the Linux Kernel. *SIGOPS Operating Systems Review*, 49(2):44–50, Jan. 2016.

[29] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: analyzing the performance of webassembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[30] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for Microsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, pages 345–360, 2019.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[32] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[33] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[34] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *Proceedings of the Thirteenth USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, 2018.

[35] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[36] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. 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*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[39] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[40] Microsoft, Inc. Blob Storage | Microsoft Azure. https://azure.microsoft.com/en-us/services/storage/blobs/.

[41] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom. Fallout: Reading Kernel Writes From User Space. *CoRR*, abs/1905.12701, 2019.

[42] MongoDB, Inc. MongoDB for GIANT Ideas | MongoDB. http://www.mongodb.com/, 2017.

[43] Node.js Foundation. Node.js. http://nodejs.org/, 2017.

[44] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, pages 361–378, 2019.

[45] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, Aug. 2015.

[46] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[47] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association.

[48] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, 2014. USENIX.

[49] Scylla DB, Inc. Seastar. http://www.seastar-project.org.

[50] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI 96)*, pages 213–227. ACM, 1996.

[51] M. Sullivan and M. Stonebraker. Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems. In *VLDB*, pages 171–180, 1991.

[52] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 207–222. ACM, 2003.

[53] B. L. Titzer and J. Sevcik. A year with Spectre: a V8 perspective. https://v8.dev/blog/spectre, 2019.

[54] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[55] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[56] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[57] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[58] WebAssembly Community Group. WebAssembly Specification. https://webassembly.github.io/spec/core/index.html, 2018.

[59] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[60] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.