THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Adaptive Placement for In-memory Storage Functions

Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman, *University of Utah*

https://www.usenix.org/conference/atc20/presentation/bhardwaj

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Adaptive Placement for In-memory Storage Functions

Ankit Bhardwaj Chinmay Kulkarni Ryan Stutsman
University of Utah

Abstract

Fast networks and the desire for high resource utilization in data centers and the cloud have driven disaggregation. Application compute is separated from storage, but this leads to high overheads when data must move over the network for simple operations on it. Alternatively, systems could allow applications to run application logic within storage via user-defined functions. Unfortunately, this ties provisioning and utilization of storage and compute resources together again.

We present a new approach to executing storage-level functions in an in-memory key-value store that avoids this problem by dynamically deciding where to execute functions over data. Users write storage functions that are *logically* decoupled from storage, but storage servers choose where to run invocations of these functions *physically*. By using a server-internal cost model and observing function execution, servers choose to directly run inexpensive functions, while preferring to execute functions with high CPU-cost at client machines.

We show that with this approach storage servers can reduce network request processing costs, avoid server compute bottlenecks, *and* improve aggregate storage system throughput. We realize our approach on an in-memory key-value store that executes 3.2 million strict serializable user-defined storage functions per second with 100 µs response times. When running a mix of logic from different applications, it provides throughput better than running that logic purely at storage servers (85% more) or purely at clients (10% more). For our workloads, it also reduces latency (up to 2×) and transactional aborts (up to 33%) over pure client-side execution.

1 Introduction

Today, in data centers and the cloud, compute is disaggregated from storage. Separating compute and storage eases provisioning and keeps utilization high by decoupling their allocation. Fast networks have made this practical, but moving all data to compute comes at a cost.

Beyond conventional, higher-level approaches like SQL, many systems have evolved to embed more functionality within storage servers to make storage operations more expressive and to reduce inefficient data movement. For example, some databases allow compile-time extensions [38, 47], user-defined functions [34], and stored-procedures [19, 22, 33, 38, 48]. Among key-value and object stores, some stores offer a fixed set of extra operators [2, 43], while others allow runtime extension with just-in-time [14, 26, 45] or ahead-of-time compiled user-supplied operations [26]. All of these approaches move user operations closer to the data that they operate on.

The downside is that these approaches fix the ratio of compute to storage, so compute at storage servers can quickly become a bottleneck. The result is that the state-of-practice is to prefer easy provisioning and high utilization while keeping a hard network boundary between compute and storage.

However, the steady decrease in the granularity of compute allocation and scheduling in the cloud (from virtual machines, to containers, to serverless functions) has raised a possibility: application compute need not be statically embedded within storage; nor must it be the case that it is always run separately. Storage servers that support running granular user-supplied functions at low cost create the opportunity to dynamically adapt where functions on stored data are executed. By shifting processing of storage functions back to storage client machines, a storage server can avoid CPU-intensive operations when under load to avoid becoming bottlenecked, choosing instead to send data back to clients for processing. By shifting processing onto itself, a server can eliminate data movement, lend its spare CPU capacity to clients, and reduce its own request processing load. Since moving user-logic into the server reduces the number of requests clients make for data, counter-intuitively, a server can improve its own throughput by taking on more of client applications' compute work.

To show the benefits of such an approach, we developed a new scheme for executing storage functions on top of Splinter [26], which is an extensible in-memory key-value store. Beyond fast `get()`/`put()` key-value operations, applications can push compiled, binary-code extensions containing storage functions to Splinter. These functions can be invoked over the network by clients with low overhead such that even operations that only perform a few microseconds of compute are practical and efficient. Our new approach builds on Splinter to imbue it with a profiler that tracks storage function execution. Clients attempt to invoke their functions at servers. Servers use an internal cost model that weighs the CPU cost to the server if the function were to continue to run at the server against the CPU cost to the server if the function were to run at the client (which would result in extra remote requests to the server for data). Functions invocations that compute over large amounts of data are deemed beneficial and are run at the server, since running them at the client would require transferring large amounts of data. Functions invocations that are compute-intensive but access little data are *pushed back* to the client, where the client must perform the computation.

Beyond the server side, the framework includes a smart storage client library that makes "pushback" cases transparent to applications. The server and the storage client library both provide a binary compatible runtime, so functions are unaware of whether they are run at a storage server (where data access is local) or at a client (where data access is remote). Applications attempt to invoke their storage functions, and the client library transparently executes any client function invocation requests that are pushed back by the storage server before returning the result to the application.

In our model, invocations may execute on the server, at the client, or partially on both, so ensuring consistency is a challenge. Our approach adapts techniques from distributed optimistic concurrency control protocols (OCC) [3, 27, 51] to solve this. All storage functions run within strict serializable transactions, which ensure that clients observe the same strong consistency regardless of where functions execute. These transactions play a key role in the function execution model itself; when a function's execution is transferred from a server to a client, its transaction's read/write set is shipped along with it, avoiding extra requests back to the server for data.

We demonstrate adaptive storage function placement (or *ASFP*) with functions drawn from different domains including aggregation, graph traversal, machine learning classifiers, and authentication. We show these workloads have heterogeneous compute demands, often with compute-to-storage-access ratios varying within one application's functions. Even so, ASFP provides throughput better than running functions purely at storage servers (85% more) or purely at clients (10% more), and it automatically adjusts, optimizing throughput as workloads and server network costs vary and change.

2 Background and Motivation

Today, cloud and data center applications keep data in one set of servers and compute over it on another. This "client-side" function execution model serves as a baseline. Our question is, can a system consistently beat the performance of this client-side approach without creating server bottlenecks?

To do this, one needs a way to embed application logic within storage to compute on data. Our approach relies on the Splinter multi-tenant in-memory key-value store (KVS) [26]. Similar to other low-latency in-memory stores like RAMCloud [40] and FaRM [12], remote clients issue `get()`, `put()`, `multiget()`, and `multiput()` operations to a Splinter server. Unlike most other systems, clients also send compiled *extensions* with custom *storage functions* to it at runtime, which they *invoke* remotely to perform operations over their data. Invoking a storage function only incurs 1,400 cycles of overhead and adds no other no runtime overheads. Splinter achieves low-latency and high-throughput via kernel-bypass networking; one server handles 6.5 million `get()` or 13.5 million no-op `invoke()` requests per second over the network with tens of microseconds of delay. It supports thousands of inter-isolated tenants per server; each application and its storage functions can only access and modify data that it owns.

Extensions reduce requests to storage. With them, a single request could fetch a "user profile" object along with the

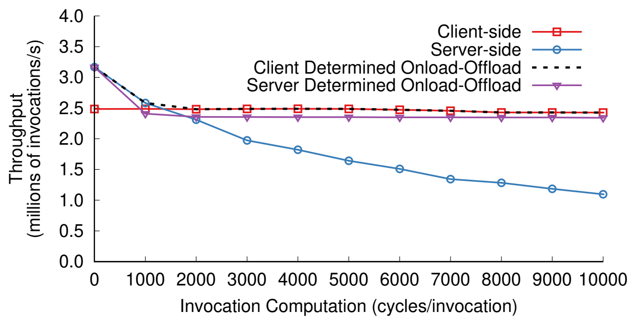

Figure 1: Server-side vs client-side throughput when CPU cost varies. Throughput is inversely related to the compute applied to two accessed values per invocation when run server-side. When the logic is run client-side, the server must process more requests (2 `get()` vs 1 `invoke()` request), but computation is offloaded to clients.

profiles of friends listed within that profile. Another application could make recursive *k*-hop queries by traversing edge lists stored in values or make classification requests to stored models. Storage functions access multiple values, reducing server request processing costs, but they are most effective for inter-dependent data accesses since these accesses would otherwise require multiple requests separated by a round-trip.

2.1 Understanding the Impact of Placement

Storage functions can lower server overhead, but if functions perform too much computation, then the benefits of eliminating requests are offset as the server CPU becomes a bottleneck. Figure 1 shows this effect. When functions are run server-side (circles), a server performs 3.2 million invocations/second if the functions perform no computation, but its throughput is inversely proportional to the CPU cycles spent computing on the values (x-axis). When run client-side (squares), the server only processes the two `get()` operations for each function; the extra computation is run at clients, which have sufficient idle CPU to perform the work. (Later, we show that if clients do not have idle CPU capacity, our approach shifts work to the server still so long as it is not overloaded. In either case, a global bottleneck is avoided.) When functions perform no computation on values, the two `get()` requests incur higher server-side overhead than sending one `invoke()` request, so server CPU becomes a bottleneck when servicing the equivalent of 2.5 million invocations/second.

Sometimes pure server-side execution provides better system throughput and other times pure client-side execution does. The key insight of this paper is that with lightweight performance tracking, the server can determine this cross-over point, and it can separate invocations into those that should be kept at the server from those that are better run at clients.

2.2 Challenges in Execution Placement

Ideally, a server could get the best of both worlds if it could perfectly determine where to execute an invocation. We simulate this by manually controlling where invocations run based on how much computation they do on data

(`Client Determined Onload-Offload`, dashed line). Here, clients never issue compute-heavy invocations to the server – so, performance matches pure server-side execution for data-intensive invocations and pure client-side execution for compute-intensive invocations.

However, real clients (and real servers) do not know how much computation an invocation will use a priori; different functions vary, and even invocations of the same function could access values and use the CPU in different ratios. Statically determining how much data or how much computation an invocation will use is undecidable in general. Static analysis or modeling might help make good guesses, but the analysis could be fragile and have pathologies.

Our approach is to measure rather than guess; rather than using history, another option is to optimistically assume invocations should run at the server and then try to minimize the cost of correcting mistakes. Figure 1 shows the cost of this conservative, "black box" approach (`Server Determined Onload-Offload`, triangles). Here, clients always invoke functions at the server, but the server quickly sheds invocations that consume CPU without accessing many values. This adds overhead for compute-intensive functions, since the server wastes a small amount of compute before realizing the mistake, but these results show this only hurts throughput 3% for compute-intensive invocations (all other invocations benefit).

Simple enhancements to this scheme are likely to work in practice. Tracking the history of the costs of a particular function's last few invocations can help. If a function's invocations are determined better to be run client-side a few times in a row, then running the next several invocations client-side makes sense. This would work for many applications, but we intentionally avoided such tweaks in this paper. Our approach never relies on the history of invocations (neither across nor within a function); optimizations that make better predictions are likely limited to only recovering that 3% of performance.

In summary, ASFP based on optimistic onloading of application compute to storage with pushback to clients achieves the best of both worlds. For storage functions that access a great deal of data, ASFP avoids data movement costs; for functions that are compute-costly it avoids server bottlenecks.

3 ASFP Design

Applications vary in how they work with data they hold in remote storage. Compute-bound applications may access little data, so moving data to computation is efficient; for data-intensive applications moving computation is more efficient. Multi-tenant stores take this to an extreme: they see a diverse set of applications with a wide variety of compute and data needs. The key idea of ASFP is to exploit this diversity by optimistically colocating functions with the data they access and then profiling storage function execution costs to dynamically relocate invocations that would create a bottleneck.

ASFP relies on *mechanisms* for running storage functions at servers, at clients, or split between both and *policies* to control the mechanisms and decide placement. The four main mechanisms are needed for ASFP are:

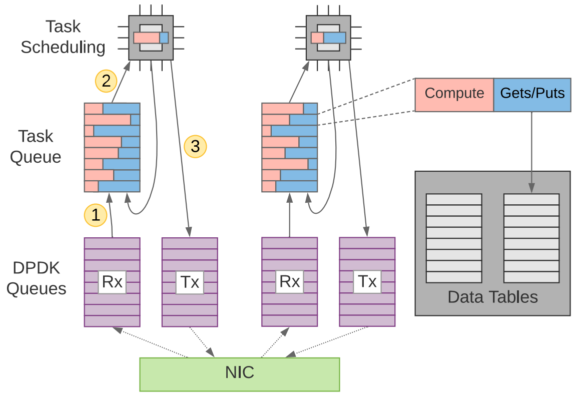

Figure 2: Splinter Request Execution. Each server core has a dedicated network receive queue where clients steer requests. Each core polls this queue and creates a task for each incoming request. Tasks are run round-robin; storage functions can access key-value pairs and perform custom computation on them within the server.

Server-side Storage Functions (§3.1.1). Tenant-provided storage functions reduce data movement. They are key for improving server performance for data-intensive functions. This functionality already pre-exists in Splinter.

Server-to-client Pushback (§3.1.2). ASFP uses a *pushback* scheme that relocates costly function invocations back to clients to avoid server-side bottlenecks.

Concurrency Control (§3.1.3). Since a single function invocation could run partly server-side and partly client-side, consistency becomes an issue. ASFP ensures that this does not cause repeated effects or inconsistencies. It uses OCC to ensure strict serializability of invocation operations, and it integrates with OCC read/write set tracking to preserve work for invocations that are pushed back to clients.

Client-side Runtime (§3.1.4). Clients locally execute invocations that are pushed back from the server, and the ASFP client library makes this transparent. Applications wait for invocations to complete; the client library runs pushed back invocations, fetching data from the server as needed.

The server's primary objective in ASFP is to minimize the CPU usage per function invocation and to optimize its own throughput, which, indirectly optimizes the throughput of the entire system. The ASFP policy relies on three key components to do this:

Invocation Profiling. Each server tracks each function invocation as it runs to account for its CPU time.

Request/Response Cost Modeling (§3.2.1). Similarly, each server dynamically profiles networking CPU costs to determine a CPU cost model for data movement. This projects how much server CPU is being saved by running each invocation at the server. If an invocation has consumed substantially more CPU cycles at the server than the request/response cost model projects have been saved by running it at the server, then it is pushed back to the client.

Overload Trigger (§3.2.2). Even compute-bound functions run more efficiently at the server than they do at clients since they can avoid data movement. All invocations run at the server if there is spare CPU capacity available, so long as they don't create a bottleneck at the server. Hence, pushback is only triggered when our server deems itself overloaded.

First, we describe ASFP's mechanisms to show how storage functions, pushback, and concurrency control work; then, we explain how its measurements and policies drive its mechanisms. Overall, ASFP constitutes about 7,500 lines of code split across additions to the Splinter server and a the new client library, which shares much of its code with the server (available at `https://github.com/utah-scs/splinter/`).

3.1 ASFP Mechanisms

3.1.1 Server-side Storage Functions

ASFP is built on top of the Splinter in-memory KVS. Splinter supports typical KVS remote `get()` and `put()` operations. It is a good starting point because it also supports installation of client-supplied native-code extensions. These extensions add storage functions to the server that can be called remotely via `invoke()` requests. ASFP uses `invoke()` requests to move computation to data, and it extends Splinter with new profiling, policy, and function invocation relocation functionality.

Internally, Splinter multitasks between `get()`, `put()`, and (possibly longer-running) `invoke()` requests, so each incoming request is converted into a *task*. The server runs these cooperative tasks round-robin until they complete or yield; this prevents head-of-line blocking when functions take awhile to execute. Tasks handling `invoke()` operations maintain state for the running storage function as a coroutine stored in the task. Figure 2 illustrates request processing. Each server core polls a CPU-core-specific network receive queue and creates a task for each incoming request (①), each of which is added to a per-core task queue. Each queued task is run once (②), then the core polls its receive queue again. The core transmits a response (③) when a task completes and then destroys it.

Invocations run interleaved due to cooperative scheduling, but they can also run in parallel too. Clients steer requests to specific CPU cores to reduce overhead, but server cores steal work from each others' receive queues to keep throughput high under load imbalance. Hence, pipelined invocations from a client can run in parallel at the server.

3.1.2 Pushing `invoke()`s Back to Clients

ASFP lets Splinter servers selectively shed load. When a storage server's cores are overloaded, it *may* perform a *pushback* on tasks. These tasks are terminated at the server and restarted client-side. Figure 4 shows the state transition diagram of the lifecycle of an `invoke()` request at a server. ASFP adds a new `Offload` state to server-side tasks to support pushback.

For each incoming `invoke()` request, a server creates a task and tries to run it to completion, sending a `Result` response (top of Figure 3). However, if a server is past an *overload trigger point* (§3.2.2), then it chooses some `Ready` `invoke()` tasks that are good candidates for client-side execution based on a *threshold function* (§3.2.1), and it moves them to `Offload`.

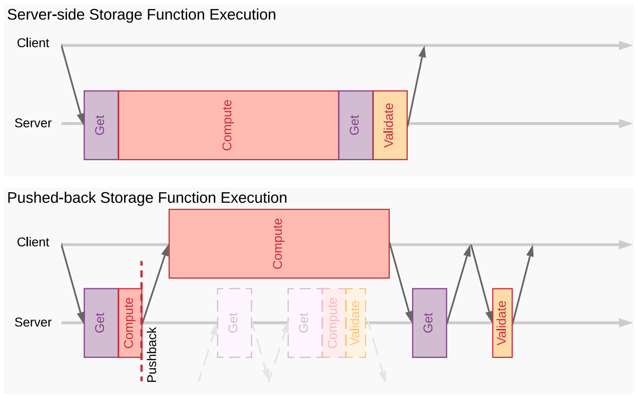

Figure 3: Timeline of a function invocation request when run server-side (top) and when pushed back to the client side (bottom). In this case, offloading this relatively long-running function to the client gives the server extra CPU resources to service other requests.

When tasks in the `Offload` state are scheduled, a `Pushback` response is generated that informs the client that it should run the function client-side. The client runs the function, falling back to making `get()` requests to the server to fetch needed values (`put()`s are cached locally and installed atomically when the invocation completes, §3.1.3). Figure 5 shows this. If the client receives a `Result` response, the work of the requested invocation has been done, and there is nothing left to do. If the client receives a `Pushback` response, the client begins to execute the function logic itself in a fashion similar to the server. The bottom of Figure 3 shows the interactions between the server and the client when an invocation is pushed back to the client; as shown, this avoids a bottleneck in this case, freeing the server to process other requests at the server.

3.1.3 Consistency and Concurrency Control

Storage functions and pushback create interrelated challenges, especially for consistency. First, `invoke()` tasks run concurrently at the server; this can happen because tasks run interleaved at the server and because server cores perform work stealing. Pushed back requests also create concurrency, since those functions run at the client in parallel with server tasks. Second, when tasks are pushed back, the client restarts execution of that function from the beginning – pushback has no means to preserve the running state of a function to resume it at the client. This means that without care, clients might repeat operations, which would affect the concurrent behavior of functions and make it hard to reason about consistency.

To solve these consistency issues, invocations are run as strict serializable transactions. This makes it easy to reason about consistency regardless of where an invocation is run. ASFP adds OCC transactions to Splinter. When a server receives an `invoke()`, it creates an empty read/write set. The server tracks the version of each value that a task sees and the values that the tasks wishes to install in storage. If the task

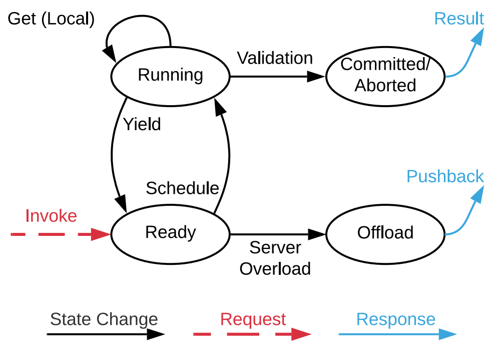

Figure 4: Server-side task states for an `invoke()`.

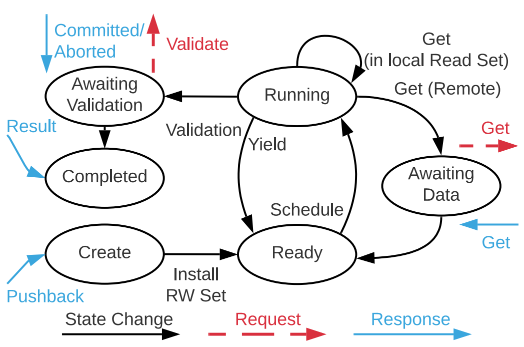

Figure 5: Client-side task states for a pushed-back `invoke()`.

completes on the server, validation is performed by latching the invocation's read/write set. For each key in the read set, if the associated value versions remain unchanged, then its write set is installed and the client is informed of commitment; otherwise, the server indicates abort.

The server's read/write set has a second purpose: by tracking what values an invocation has read, the server can save work by returning those values immediately on pushback. The client installs this read/write set locally before restarting the function. This way, the server will never have to repeat any work for a pushed back task: all of the values the task needs have already been delivered to it up to the point that it was terminated at the server. This is key: pushed back requests never generate extra work for the server. This bears similarities to reconnaissance queries in deterministic databases [50].

On completion of a pushed back task, the client sends the write set and version metadata for the values it read to the server where validation is performed the same as if the task has completed server-side. This is another advantage of OCC: the server need not keep any state about an invocation once it is pushed back. For example, the server retains no metadata or locks on behalf of a pushed back task. This makes any recovery or state reclamation unnecessary on client failures.

3.1.4 Client Runtime for `invoke()`s

Splinter client requests consist of basic `get()/put()` requests and `invoke()` requests that attempt to invoke a storage function within the store. Clients register *extensions* at the server before invoking the functions they contain. ASFP requires that the same extensions are registered at the client library as well, so that they can handle pushed back `invoke()` requests.

On each `invoke()` response from the server, the client checks a `Pushback` response flag. If it is set, the client performs logic similar to request dispatching on the server: it creates a task and coroutine similar to the ones used on the server, and it places the task in a client-local task queue (Figure 5). The main difference is that the read/write set returned from the server is used to pre-populate the read/write set of the invocation before it starts at the client side.

Clients put `Ready` tasks in their task queue and run tasks round-robin, just like the server. This lets clients make progress on invocations while continuing to issue new operations to the server.

The ASFP client library provides a binary-compatible interface with the server, so identical versions of storage functions work whether they run at the client or at the server. Splinter extensions have a restricted `get()/put()` interface for interacting with storage, and they have a restricted set of whitelisted library functions they can run beyond that. Extensions on the client-side have the same restrictions, so that pushed back invocations will run the same way in both places.

Client-side execution does run different from server-side in one important regard: the client must access key-value pairs remotely. This is solved by passing in handles into storage functions through which they request access to data. On the server-side the handles call `get()/put()` functions to access data directly; on the client-side the handles issue remote `get()/put()` requests. Requests to the store take about 10 µs to service, and task context switch time is just 24 cycles, so tasks waiting for responses from the server enter an `Awaiting Data` state. Each invocation has a unique client-side id; whenever the client library receives a response to a particular extension invocation, it adds the record to the local read/write set for that invocation. Clients read through their read set; after a "hit" in their read set or upon the completion of a remote access, the task is returned to the `Ready` state.

3.2 ASFP Policies

3.2.1 `invoke()`s Profiling and Classification

Splinter is both multi-tenant and extensible. Together, these mean that it must deal with different access patterns and functions with varying compute-to-storage-access ratios. This also means a server will be able to find many suitable functions to run that can reduce its load. When overloaded, it must determine which `Ready` tasks should be pushed back; however, pushing back the wrong tasks can *hurt* its throughput.

Whether a task is beneficial to server throughput when run server-side is determined by two things: the amount of CPU time it uses computing on the values it accesses (which hurts throughput) and the number of values it interacts with (which benefits throughput, since each access run at the server eliminates a network request that it otherwise would have had to process). Effectively, each time a task accesses a stored value it should be credited for the amount of server CPU it saved by having run that operation locally. Likewise, whenever it performs other computation that does not save server CPU work it should be debited, since this slows request processing.

This results in a natural threshold for when tasks would be pushed back to clients, which we call the *pushback threshold*. It is defined by

$$c < nD - (D + I)$$

where c is the amount of computation done by an invocation so far, n is the number of values accessed by the invocation so far, D is the request processing CPU cost, I represents the cost to perform an invocation (beyond request processing cost). Effectively, nD is the work the server would have done if the client issued n `get()` requests. $(D + I)$ represents CPU cost at the server of an `invoke()` request. Hence, so long as $c < nD - (D + I)$, server-side work is saved by letting the invocation remain at the server.

This inequality divides all tasks into two classes, $\mathcal{S}$ and $\mathcal{C}$. Tasks in $\mathcal{S}$ are beneficial to run at the server, and tasks in $\mathcal{C}$ improve server throughput when run at clients. The inequality does not hold when an invocation accesses zero or one values; these invocations save the server less work than the cost of an `invoke()` request. It never makes sense to run them at the server, and it would also be unusual for a client to try to do so. The model is simple and linear, so the server can calibrate it inexpensively at runtime. Just by profiling the cost of an `invoke()` operation and a `get()` operation, it can accurately assess which invocations should be pushed back.

As discussed in Section 2.2, it is undecidable in general to determine the class of an invocation. The input parameters to an `invoke()`, the data its accesses, the server hardware, and its cache policies/pollution all influence performance. Our approach simply assumes all invocations should be initially attempted server-side. Exploiting history or domain knowledge would improve performance in cases where functions are pushed back. However, we explicitly avoid relying on such information since its effectiveness is workload dependent, and it can only provide a few percent performance improvement for invocations in $\mathcal{C}$ (and would only hurt functions in $\mathcal{S}$).

3.2.2 Server Overload

The final piece of ASFP is overload detection. Functions should always run at the server when it has idle CPU; this still eliminates data movement costs, and it frees client CPUs to do other work. However, when overloaded, the server must shed load to improve throughput and control response times.

Algorithm 1 shows how the server detects overload (others use similar approaches [39]). The server is under high load when it receives new tasks and the requests from previous scheduling passes have not completed. At the beginning of a server's round-robin pass through its set of tasks, it polls its receive queues and creates up to B tasks, one for each

Algorithm 1: Server Overload Detection

```
Function Scheduler()
    totalTime ← 0;
    while true do
        t ← taskQueue.Dequeue();
        if t = DispatchTask then
            newTasks, dispatchTime ← PollRecvQueue();
            if totalTime ≫ dispatchTime then
                if taskQueue.length ≥ B/k and
                  newTasks.length ≥ B/k then
                    taskQueue.ClassifyAndPushback();
                    taskQueue.Enqueue(newTasks);
                end
                taskQueue.Enqueue(t);
                totalTime ← 0;
            end
        else if t = RequestTask then
            t.getPutTime, t.computeTime, t.state = t.Run();
            totalTime += (t.getPutTime + t.computeTime);
            if t.state ∉ {Committed, Aborted} then
                taskQueue.Enqueue(t);
            end
        end
    end
```

incoming request (where B is the maximum receive batch size, which we fix at 32). Then, it compares the amount of time spent dispatching requests in that round of scheduling (time spent polling network queues and creating tasks) with the time spent executing invocation tasks in that scheduling pass. If invocation task execution time is the dominating factor, the scheduler checks the task queue length. If it contains at least B/k tasks and processing incoming requests would create at least another B/k tasks, then the scheduler sets a flag indicating the server is overloaded. Higher values of k trigger overload more easily; $k = 2$ works well, and we keep it fixed in our experiments. This guarantees that the scheduler:

1. only pushes back work if load is mainly from `invoke()`s;
2. keeps at least B/k tasks in the queue after pushback; and
3. only pushes back when $\geq 2B/k$ requests await service.

On overload, the server tests the threshold inequality (§3.2.1) on all old `Ready` tasks, triggering pushback on some of them.

4 Evaluation

We compare three models for storage functions. *Client-side* runs them on clients and issues `get()` requests to the server. This is state-of-practice and the baseline. *Server-side* runs functions on the server and represents Splinter's approach. *Pushback* is our approach, which runs functions on the server, pushing some back to clients. We focused on these questions:

Does ASFP improve storage server throughput? For an application mix consisting of machine learning classification and graph-based storage functions, ASFP can improve throughput by 10% (§4.5.3). For functions with dependent data accesses, ASFP improves throughput by 42% (§4.2).

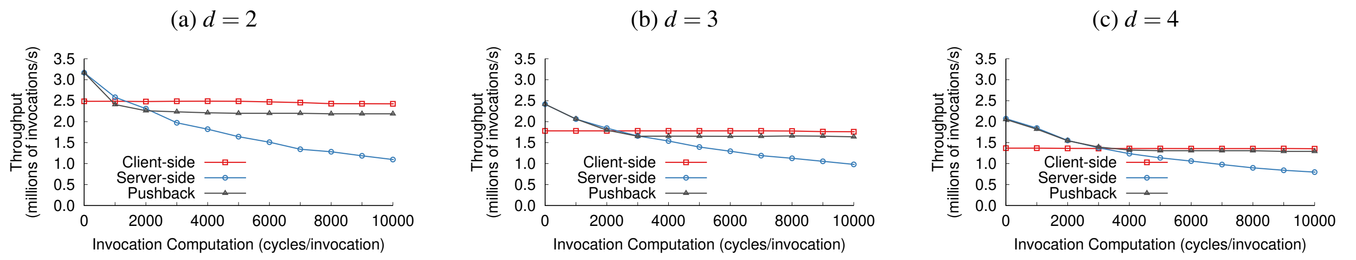

Figure 6: A function with 2, 3, or 4 dependent `get()`s followed by varied computation lengths. ASFP improves throughput by 42% over client-side execution. For compute-intensive invocations, ASFP throughput is within 15% of pure client-side in the worst case.

What is the cost of using ASFP? For invocations that improve server throughput (those in class $\mathcal{S}$), ASFP gives the full performance benefit of server-side execution with no measurable overhead. For compute-intensive invocations that access little data, classification and client-side re-execution delivers performance within 15% (§4.2) of pure client-side execution in the worst case.

How effective is the ASFP classifier? For a mix of predominantly $\mathcal{C}$-class invocations with significant compute variance, 87% of all $\mathcal{C}$-class invocations are offloaded to clients (§4.3). The remaining are accurately classified but retained by the server because it has idle compute to execute them, improving overall system throughput.

How does ASFP impact latency? For invocations in class $\mathcal{S}$, ASFP saves on round trips to the server, which reduces latency by as much as $2\times$ (§4.4). For invocations in class $\mathcal{C}$, ASFP's read/write set and server overload-based optimizations can help reduce latency by 15% compared to executing client-side. For extremely compute-intensive invocations, ASFP matches client-side execution.

How do ASFP and OCC interact? Beyond providing consistency, OCC lets the server send back read/write sets on pushback, improving throughput by 33% (§4.6). ASFP also exploits idle compute at both servers and clients speeding up transactions and reducing abort rates (§4.6.1).

4.1 Experimental Setup

Evaluation is on five machines; four as clients and one server (unless otherwise noted) on CloudLab [13] (Table 1). All use DPDK [11] over Ethernet. Eight of ten server cores process requests; Splinter uses two cores for task management. Clients also use eight cores; each core pipelines `invoke()` requests up to a depth of 32 and receives responses in a closed-loop.

Using a closed-loop is helpful. ASFP demands complex, heterogeneous workloads; an open-loop load requires careful manual pacing of the request rate for each storage function type. To ensure we always measure the server at saturation (unless otherwise noted), we control client thread count instead of manually tuning per-storage-function request rates.

The server held 15 GB as 120 M records (30 B keys, 100 B values) unless otherwise noted. On pushback, clients transparently ran functions locally, issuing remote record requests.

CPU	Ten-core Intel E5-2640v4 at 2.4 GHz
RAM	64GB Memory (4x 16 GB DDR4-2400 DIMMs)
NIC	Mellanox CX-4, 25 Gbps Ethernet
Switch	Mellanox SN2410 48-port, 25 Gbps per port
OS	Ubuntu 16.04, Linux 4.4.0-138, DPDK 17.08 Rust 1.29.0-nightly, 16×1 GB Hugepages

Table 1: Experimental setup. Evaluation used one machine as a server and four as clients. All experiments were run on CloudLab.

4.2 ASFP Throughput Benefits & Costs

Benefits. ASFP combines the benefits of server- and client-side execution; invocations with low compute-to-data access ratios run on overloaded servers, otherwise they are offloaded to clients. To show this, we run a microbenchmark that varies the number of records accessed and the amount of compute performed within an invocation.

Clients issue `invoke()`s that do d data-dependent `get()`s followed by x cycles of compute. Figure 6 shows server throughput when the function is run purely client-side, purely server-side, and with adaptive pushback for d from 2 to 4. With a small x, server-side execution prevents clients from stalling on remote `get()`s. With a large x, client-side execution with remote value access avoids a server CPU bottleneck.

Here, ASFP's simple model works well. In Figure 6 (a), invocations that perform little compute over values stay at the server, improving throughput over client-side execution by 27%. Invocations using more CPU are pushed back, and throughput tracks the client-side approach. For increasingly CPU-intensive invocations ($x >$ 6,000), the throughput of pure server-side execution tends toward zero, so the benefits of pushback over server-side execution grow (until all client CPUs saturate, but realistic servers will service many clients).

These results show that the more data an invocation accesses, the more savings pushback provides; increasing d to 3 and 4 gives savings of up to 33% and 42%, respectively (Figure 6 (b), (c)). The area between pushback and client-side can be large for CPU-inexpensive functions (left side of graphs), but the area between pushback and server-side for CPU-expensive functions (right side) is also large since real functions will vary even more in how much CPU they use.

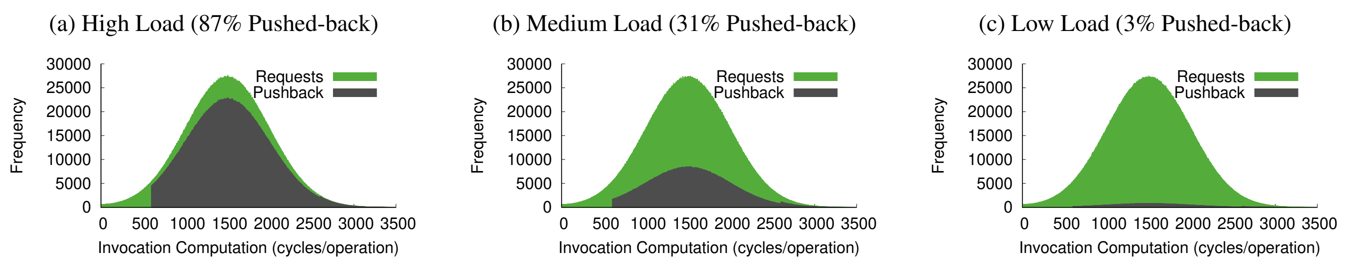

Figure 7: Distribution of invocations generated overlaid with those pushed back to one client under different loads. ASFP can use idle server compute to run some of the invocations classified as $\mathcal{C}$. This is why the distributions do not completely overlap.

Costs. These graphs also show ASFP's costs. On a pushback, there are two costs: the first is the cost of having the server process an extra request for `invoke()` and validation; the second is the computation the invocation did before it was terminated. Most of the first cost is eliminated by shipping all accessed values back to the client on pushback. An `invoke()` only costs 9% more than a `get()`, and all practical functions receive some values when they are pushed back. Hence, the cost of the `invoke()` is offset by the fact that it eliminates the need for the client to issue at least one `get()`.

The second cost explains the gap between the client-side and the pushback approach. This experiment is a pessimistic case: the function first accesses all of its values, then it performs compute over those values. For compute-intensive functions, this means the server runs them longer before it pushes them back; for functions where data access and computation are intermingled, pushback would achieve performance closer to the client-side case. Even so, in all cases where client-side execution would outperform server-side, pushback is only 13 to 15% slower than running everything at clients.

Cost Breakdown. This 15% overhead for ASFP for $\mathcal{C}$-class functions in Figure 6 (a) has two components. The first is the cost of suspending an invocation and sending its read/write set back to the client, but this only accounts for 3% of the 15%. Figure 1 shows this; in it, the performance difference between (omniscient) client-determined placement and server-determined placement that observes each invocation is only 3% even when invocations are in $\mathcal{C}$. The second component of the overhead (12%) comes from an interplay between overload detection and $\mathcal{C}$-class invocations. In Figure 1, the server never executes an invocation in $\mathcal{C}$ to completion, even if the server is idle, but ASFP completes invocations server-side, regardless of class, if the server is underloaded. However, a server's load can shift rapidly at fine timescales. Leaving $\mathcal{C}$-class invocations onloaded is a form of speculation about whether invocations will arrive in the near-term that will overload the server. This 12% is due to cases where the server performed some $\mathcal{C}$ work, and it became overloaded during that work. This effect can be seen in §4.3 Figure 7 (a) as well; even at high load some $\mathcal{C}$-class functions are run at the server. This can be controlled; making overload detection more aggressive reduces this overhead (down to 3%, if desired); the trade-off is that the server may sit idle in more cases to ensure it has capacity when $\mathcal{S}$-class tasks arrive.

4.3 Invocation Heterogeneity

Real invocations are likely to be heterogeneous in two ways: first, the total compute performed might vary across invocations (*inter-invocation heterogeneity*), and second, compute might be clustered at points of execution instead of being evenly distributed across data accesses (*intra-invocation heterogeneity*). To be effective, ASFP must be able to accurately classify invocations (as $\mathcal{S}$ or $\mathcal{C}$), as well as efficiently use both server and client CPU under such forms of heterogeneity.

Inter-Invocation Heterogeneity. To demonstrate ASFP's efficiency under inter-invocation heterogeneity, we configured one client to generate `invoke()`s where the number of cycles of extra compute performed (after two dependent `get()`s) is drawn from a normal distribution, $\mathcal{N}(1500, 500)$. Figure 7 (a) plots the distribution of the generated requests overlaid with the distribution of those that were pushed back and completed on the client. This figure shows two things; first, no requests that perform less than 600 cycles of work are pushed back, so inexpensive functions are executed at the server; second, the two distributions do not completely overlap, so many compute-intensive invocations still complete at the server. With just one client, the server has some idle CPU capacity; as a result, many of invocations in $\mathcal{C}$ run server-side. As the load on the server decreases, this spare capacity increases, allowing more $\mathcal{C}$ invocations to run server-side (Figure 7 (b), (c)). This shows that ASFP can be efficiently split work between the server and client(s); any idle compute at storage can accelerate clients and improve throughput.

Intra-invocation Heterogeneity. Figure 6 presented the benefits and costs of ASFP when compute is performed after all records are accessed by an invocation. Under this scenario, pushed back invocations benefit from the shipped back read/write set. However, real function invocations are likely to perform compute at different points of execution (between record accesses for example). Figure 8 explores such scenarios. 'Pushback-y' represents a run where invocations perform compute after issuing y `get()`s (out of a total of 4 per

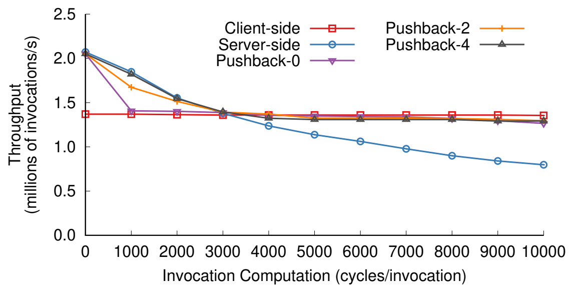

Figure 8: Throughput when the position of compute within an invocation varies. 'Pushback-*y*' is when invocations perform compute after issuing *y* `get()`s. ASFP is never worse than pure client-side.

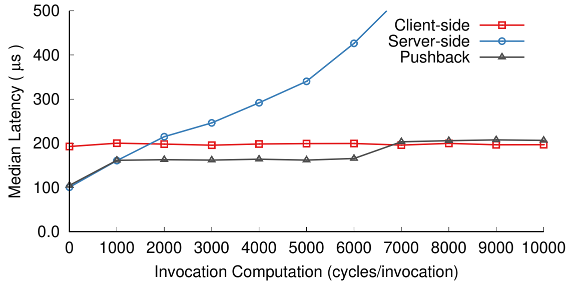

Figure 9: Effect of ASFP on median latency. ASFP improves latency between 15% to 2× compared to pure client-side.

`invoke()`). ASFP throughput is always better than or equal to pure client-side execution; for cases where compute is performed early on (Pushback-0), compute inexpensive invocations (left side of the graph) get pushed back earlier, resulting in lower gains over pure client-side execution.

4.4 ASFP Impact on Latency

Figure 9 shows median latency for an experimental setup similar to Figure 6 (a). When compute is less than 600 cycles, ASFP reduces round trips by running invocations on the server, improving latency over pure client-side execution by as much as 2×. As compute increases, invocations get pushed back; ASFP's latency is still better (15%) because these invocations receive their read/write set, resulting in one less RPC compared to client-side execution. The pure server-side approach bottlenecks, causing its response times to spike. As compute increases beyond 6,000 cycles, pushed-back invocations cause clients to saturate, reducing server load. This makes ASFP's overload detection loop retain more invocations on the server, increasing median latency to track that of pure client-side execution. Since, ASFP restarts pushed-back invocations at clients, it can increase the latency by up to 2× in the worst case. However, the only invocations that could experience this worst case are ones that never access values, since invocations that access values always execute more quickly on the client-side after pushback due to read/write set shipping. These functions should be rare and should not be attempted at storage servers; clients have little reason to send them to the server since they never access data.

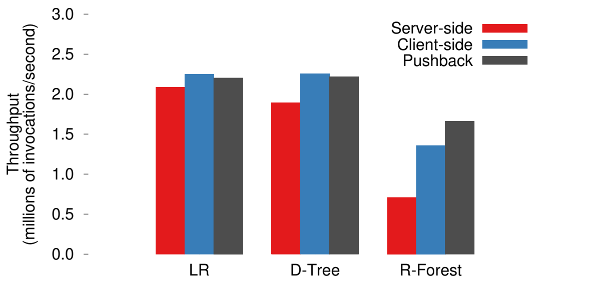

Figure 10: Machine learning classifiers. LR and D-Tree are within 2% of pure client-side. For R-Forest, ASFP leverages idle server compute, improving performance by 22% over pure client-side.

4.5 Realistic Applications

Beyond microbenchmarks, we applied ASFP to more realistic functions. We use three types of functions from different domains that we believe would be a possible fit for low-latency in-memory storage services. The first is an application barely in class $\mathcal{S}$ that accesses little data and performs little compute per invocation: Facebook's TAO social graph database [5]. The second is an application barely in class $\mathcal{C}$ that accesses little data with compute requirements slightly higher than the $\mathcal{S}/\mathcal{C}$ threshold: a machine-learning based disk failure prediction. The last is an application well in class $\mathcal{C}$ that accesses little data and uses significant CPU: authentication [42]. To be effective, ASFP must classify (as $\mathcal{S}$ or $\mathcal{C}$) and place invocations (on the server or the client) and improve overall throughput. We show ASFP can do so for these functions and for mixes of both $\mathcal{S}$ and $\mathcal{C}$ invocations.

4.5.1 Machine Learning

We use disk failure prediction [17, 28, 41, 44] for our first application. This application consults a classifier to predict whether a disk in a data center is about to fail. We chose classifiers because they benefit from Splinter's model that supports complex but native functions; they are a realistic and expected use; and their compute requirements vary.

We evaluated three classifiers: logistic regression (LR), a decision tree (D-Tree) and a random forest (R-Forest) (an ensemble of decision trees). Classifiers are trained offline from a data set with 25 features [37]. The server holds data points to be classified (loaded/streamed in a priori). Two clients generate `invoke()`s that classify two data points each.

Figure 10 shows how the classifiers perform. All three are in $\mathcal{C}$, so client-side execution outperforms server-side. R-Forest is the most CPU-intensive. ASFP outperforms both pure client-side (22%) and pure server-side (2.3×) execution because invocations are placed on both the server and the client. LR and D-Tree are harder cases; they are nearer to the $\mathcal{S}/\mathcal{C}$ split; the extra overhead of initially running them server-side before pushing them to the client cuts into ASFP's benefits. As a result, for these two classifiers, pure client-side execution marginally outperforms ASFP (by $< 2\%$) even considering the extra compute capacity that the server provides.

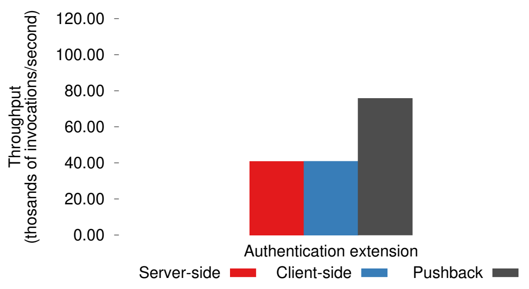

Figure 11: Authentication application. Compared to running server-side or client-side, ASFP can exploit idle cycles anywhere among the machines, improving throughput by nearly 2×.

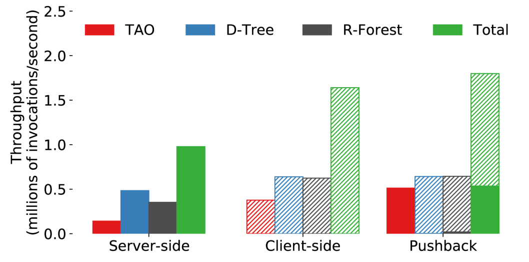

Figure 12: TAO/D-Tree/R-Forest Mix. ASFP correctly places invocations, improving overall throughput by 10%. Solid-colored regions of the bars show throughput due to invocations that ran server-side; hashed regions show throughput due to those that ran client-side.

4.5.2 Authentication

Authentication is another application; it uses bcrypt [42] to verify user identity. It uses few values, and it is computationally costly (well in class $\mathcal{C}$). Even so, it can still benefit from ASFP. We ran an experiment over 128,000 records, each containing a 30 B username and a 40 B salted hash (16 B salt, 24 B hash). One client issues `invoke()`s with a username and a 72 B AES-encrypted password. The salted hash is applied to the password. If the result matches the stored salted hash, then the invocation returns success, otherwise it returns failure.

Figure 11 shows throughput. Purely server- and client-side execution perform about 40,000 authentications/s. Both are CPU bottlenecked; bcrypt takes about 450,000 cycles per request. With pushback, throughput is nearly doubled over both approaches, as expected; with ASFP, CPUs on both the server and the client can be used to perform authentication.

4.5.3 Application Mix

Splinter is expected to run multi-tenant workloads, and pushback is primarily beneficial in a setting where there are a wide and heterogeneous set of `invoke()` requests. To create such a scenario, we ran a mixed workload comprised of an R-Forest classifier (class $\mathcal{C}$), a D-Tree classifier (class $\mathcal{C}$ by a small margin), and an implementation of Facebook's TAO [5, 26] data model which consists of dependent data accesses (class $\mathcal{S}$). Three client machines generated requests to the server.

Figure 12 shows how ASFP improves throughput for this mix. The solid-colored regions of the bars show throughput achieved from running invocations server-side. The hashed regions of bars show throughput achieved from running invocations client-side. The final bar in each group shows the aggregate throughput of the three applications (both at clients and the server). R-Forest requests are CPU-intensive; so, they are bottlenecked by server CPU in pure server-side execution, *and* they hurt the throughput of the other applications sharing the server. Running functions client-side avoids this bottleneck and interference, raising the throughput of the other applications. However, this runs TAO at clients as well, which creates extra server load since it is in class $\mathcal{S}$. Hence, ASFP provides the best results. R-Forest and D-Tree are classified as $\mathcal{C}$ and run (almost completely) at clients. TAO is classified as $\mathcal{S}$ and runs at the server improving server throughput. Hence, ASFP provides 10% better throughput than a conventional, disaggregated approach. Interestingly, onloading TAO creates CPU headroom at the server that R-Forest is able to exploit.

This workload is a challenging one for ASFP; a majority of the TAO requests only access one data item per invocation (60%); hence, all of the applications perform fairly well when executed client-side. As a result, the system only experiences modest gains when TAO is run at the server.

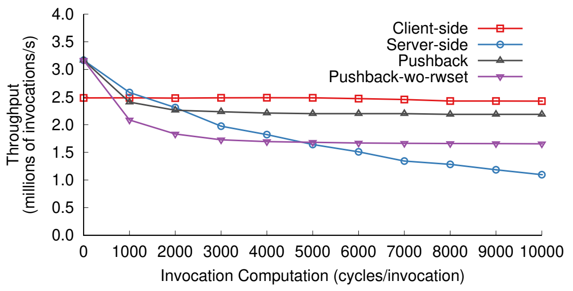

Figure 13: Impact of read/write set shipping on ASFP. When turned off, throughput suffers by 33% since pushed-back invocations load the server by reissuing remote `get()` requests.

4.6 Concurrency Control and ASFP

Beyond providing consistency, OCC improves ASFP's throughput; instead of reissuing `get()` requests to the server and increasing its request processing load, pushed-back requests reuse their read/write set. We explore this optimization with a setup similar to Figure 6 (a). Figure 13 shows that disabling read/write set shipping for pushed back invocations hurts throughput by 33% (Pushback-wo-rwset). Note, this workload only accesses two records per invocation; invocations that access more records would benefit more.

4.6.1 ASFP Impact on Abort Rate

Moving execution between servers and clients affects transaction commit latency and abort rates. To study this, we used YCSB+T's *Closed Economy Workload* [9] with four clients generating a request distribution where 50% of the requests are read-only and the remaining are read-modify-writes. We added a parameter to the read-modify-write transactions to control how much compute each one performs.

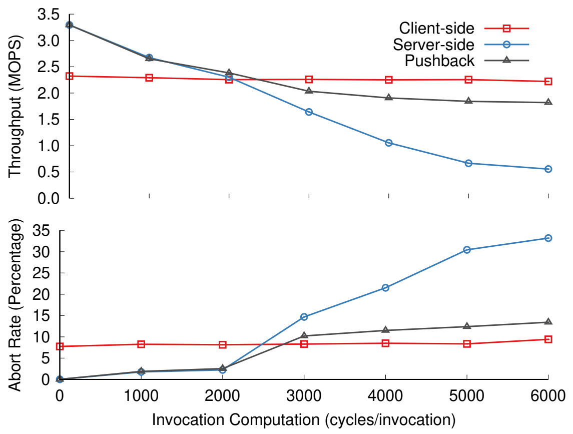

Figure 14: Impact on abort rate. ASFP leverages idle compute to speed up transactions, reducing read/write conflicts and abort rates.

Figure 14 shows trends similar to §4.2. ASFP speeds transactions/invocations in $\mathcal{S}$ by placing them on the server, reducing read/write conflicts and aborts. Server-side execution would bottleneck and slow transactions/invocations in $\mathcal{C}$, increasing conflicts. Here, ASFP uses clients to speed invocations, reducing aborts. The conflict window of each transaction is mainly determined by its latency. This relationship can be clearly seen when comparing these results to those in Figure 9. ASFP avoids bottlenecks, controls latency, and reduces aborts regardless of how much compute invocations use. We omitted results for a 90-10 read/write ratio; aborts are always negligible (0.02%), even for invocations in $\mathcal{C}$.

5 Discussion

Security. ASFP builds on Splinter's unique function isolation model that uses Rust's type system. This software-based scheme has a broad attack surface including Splinter's code; Rust (its type system, compiler, and standard library comprising millions of lines of code); and underlying libraries including libc and DPDK. This model is also complicated by micro-architectural side channels and speculative execution attacks, which continue to surface. For example, Splinter does not include the micro-architectural state flushes needed on protection domain switches to protect against information leaking via Spectre v2 and other similar vulnerabilities [6, 23].

ASFP is independent of Splinter's isolation and trust model, but there two ways that its isolation costs affect ASFP's applicability to other systems. First, its software-based isolation has extremely low protection domain/context switch costs. Tenant function invocations cause neither page table nor stack switches; hence, `invoke()`s are only 9% more expensive than `get()`s. With stronger isolation schemes, like conventional page table switching, each `invoke()` would need to make up for these costs, which can add up to several microseconds of CPU. With Splinter, some functions that only access two records improve efficiency when run server-side; if a page table switch were needed per invocation, invocations that accessed less than tens of records would be inefficient server-side. The second impact of Splinter's model is that it supports thousands of tenants per machine with low overhead, increasing the heterogeneity of operations it would be offered by tenants. Overall, this means using stronger isolation primitives would result in providers dedicating at least one server core to each tenant to avoid protection domain switch costs; this would limit the diversity of functions each server handles.

Larger-than-DRAM data & distribution. ASFP targets low-latency in-memory storage where only small, hot records are economical to store, which simplifies its cost model. Records are so small that the CPU cost of copying them (in/out of network buffers) is negligible, and neither I/O CPU cost nor storage throughput limits need to be considered. Addressing more complex systems is an interesting problem.

ASFP is focused on one server and its clients. As-is it can work in a sharded store where data is partitioned (e.g. by key). In the future, we plan to extend its OCC model for distributed transactions while factoring in data movement costs and abort rates in deciding placement of operations.

Idle client-side CPU assumptions. ASFP assumes clients have sufficient idle CPU to run pushed back invocations. This relies on provisioning client capacity according to state-of-practice: as if all invocations run client-side. When invocations are shifted server-side, this can only produce extra idle capacity at clients and servers.

State migration. A full system would need sharding, load balancing (similar to Slicer [4]), state migration to consolidate load [25], and a means to deprovision idle CPUs. ASFP is complementary; load and state must be rebalanced in any cluster with or without ASFP. For the heterogeneous invocations offered to servers, ASFP optimizes server CPU regardless of how state is sharded across the cluster.

Restart vs. resume. Process/function migration [10, 32, 35] is costly and complex. Resumed functions would need to send intermediate state to clients; that additional state capture, transmission, and restoration would need to be incorporated into ASFP's cost model. Further, restarted, pushed back functions take no more client-side CPU than they would in today's client-side approaches. Worst case, a function could be (nearly) computed at the server and repeated at a client. In the cases we have looked at redundant work is small, so it would be hard to offset function shipping/resuming costs.

Predicting placement. Speculatively onloading a function only adds 3% server load even when it is always pushed back (Figure 1). Pathological cases could access many records after pushback. These would perform nearly the same as today's pure client-side approach, but history/prediction could help. Simple approaches that track recent invocation misclassifications could be used to bias a function's future invocations to stay server-side. Pushback only happens on overload, so some misclassification has little impact; the server need only classify enough tasks correctly to mitigate overload.

Other key-value stores. ASFP can work in any extensible store, like Redis [43]. Splinter's kernel-bypass networking simplifies cost modeling; modeling kernel TCP costs would add complexity but would increase potential savings over our implementation. ASFP is also targeted toward diverse, multi-tenant workloads with heterogeneous operations that it can place. Single-application functions added to a single-tenant store could likely be statically classified as server- or client-side, eliminating some benefits of dynamic profiling.

Network congestion. Congestion isn't a problem in our high-bandwidth setup. However, exposing transport layer information (window sizes) to ASFP could let it choose placement to minimize network traffic under congestion. If an invocation accesses little data and enqueues many bytes for transmission while the network is congested, the server could return the data instead, forcing the client to compute the result.

6 Related Work

Adaptive pushback for Splinter builds on many ideas.

Storage Procedures, UDFs, and Database Extensions. There are several common approaches for pushing computation to databases and data stores. SQL is ubiquitous, though it is a poor fit for specialized computation, especially for microsecond timescales. SQL stored procedures [34] and UDFs [19, 22, 33, 38, 48] allow more specialized, procedural logic to be added to stores, and they can often be compiled for performance. Some databases also allow dynamic libraries to be loaded as well for specialized operations [49]. Some key-value stores and object stores support similar user-provided functions or extensions provided either at server-start time [43] or at runtime [14, 26, 45, 53], some relying on just-in-time compilation and some ahead-of-time compiled.

All of these approaches can ship computation to storage, but they do not address the question of whether doing so is beneficial for storage servers or its clients. Our approach could be applied to stored procedures and UDFs.

Thread and Process Migration. In the 1990s, both process and thread migration were pursued as ways to move computation at a fine-grain, often to place computation near data [10, 32, 35]. These approaches are often complex and highly runtime-specific, since moving in-progress computation requires precise reasoning about the state it closes over. We take a much simpler approach; rather than moving running functions, we preserve some of the work they have done through their read/write sets and restart functions from the beginning at clients. This assumes that invocations tend to be short, which is true for the small timescales that we target.

Fast, Disaggregated Storage. Fast networks have led to disaggregated storage and even disaggregated memory [16, 30, 31]. Many works focus on building scalable in-memory stores that move data efficiently [29], and many more have used techniques like kernel-bypass and RDMA (both one-sided and two-sided) to minimize the CPU cost of request processing for fast storage [20, 21, 29, 52]. These approaches reduce server-side CPU consumption and improve throughput for the simple operations that these stores provide, but they provide no way to move computation into storage when it would improve server efficiency. FaRM [12] is an exception. Clients can do this manually since each node in FaRM is both a client and a server; functions can be compiled into the storage server for custom request handlers. However, FaRM lacks an adaptive mechanism to move invocations of these functions between clients and remote servers.

Cell [36] is a distributed in-memory B-tree that uses RDMA. Cell uses a similar idea to pushback. In Cell, when clients lookup keys in the B-tree they can use one-sided RDMA reads to fetch nodes from the B-tree and perform the tree traversal client-side, or clients can send a request to a server to have it do the traversal. Clients track round-trip times to estimate queuing delay to determine whether the server network card or server CPU is under pressure. This lets them intelligently choose between the two approaches to improve server throughput. Our approach is similar, but ASFP is black box; it assumes no visibility into the functions that clients want to run. As a result, it must track and predict the relative client-side versus server-side cost of operations.

Offloading and migrating code has also been pursued in other contexts like edge computing and mobile devices where there is a large imbalance between the capabilities of devices and where moving data over edge links incurs high cost [8,15, 18,24,46]. Our approach and Splinter are also similar to Active Disks [1, 7] that allow application code to be downloaded to and executed on disk- and flash-based storage devices.

7 Conclusion

Today, data center and cloud storage systems disaggregate compute; clients must fetch data to compute on it, resulting in wasted work. When clients can send computation to storage, both clients and storage servers can benefit; however, to be practical, storage servers need a means to avoid becoming a bottleneck. ASFP does this by keeping client functions logically decoupled from storage and deciding physical placement of their invocations at runtime. By profiling invocations and observing both the CPU costs and savings they create at the server, storage servers can dynamically determine when invocations should be forced back for client-side execution.

We show ASFP's promise; servers and smart clients adapt function placement at microsecond timescales, improving throughput even when storage function CPU cost varies. We show it works when running a mix of different applications' logic, providing better throughput than running that logic purely at storage servers (85% more) or clients (10% more).

Acknowledgments. Thanks to Mazhar Naqvi and Jacob Barzee for contributing to ASFP, to the reviewers for their comments, and to our shepherd, Changwoo Min. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors 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] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast Key-value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 113–119. ACM, 2019.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 23–34, 1995.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat 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.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 249–266, USA, 2019. USENIX Association.

[7] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[8] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[9] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+ T: Benchmarking Web-scale Transactional Databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[10] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(8):757–785, 1991.

[11] DPDK Project. Data Plane Development Kit. `http://dpdk.org/`. Accessed: 2020-01-15.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[14] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry 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.

[15] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code Offload by Migrating Execution Transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, Hollywood, CA, 2012. USENIX.

[16] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[17] Greg Hamerly, Charles Elkan, et al. Bayesian Approaches to Failure Prediction for Disk Drives. In *ICML*, volume 1, pages 202–209, 2001.

[18] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, August 1997.

[19] Guy Harrison and Steven Feuerstein. *MySQL stored procedure programming*. " O'Reilly Media, Inc.", 2006.

[20] Anuj Kalia, Michael Kaminsky, and David 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.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, 2016. USENIX Association.

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, August 2008.

[23] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[24] Michael Kozuch and Mahadev Satyanarayanan. Internet suspend/resume. In *WMCSA*, volume 2, page 40, 2002.

[25] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 390–405. ACM, 2017.

[26] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, 2018. USENIX Association.

[27] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[28] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Hard Drive Failure Prediction Using Classification and Regression Trees. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 383–394. IEEE, 2014.

[29] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael 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, 2014. USENIX Association.

[30] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ACM SIGARCH computer architecture news*, volume 37, pages 267–278. ACM, 2009.

[31] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level Implications of Disaggregated Memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[32] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1997.

[33] Microsoft, Inc. Stored Procedures (Database Engine) - SQL Server. `https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-2017`. Accessed: 2020-01-15.

[34] Microsoft, Inc. User-Defined Functions - SQL Server. `https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions?view=sql-server-2017`. Accessed: 2020-01-15.

[35] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *ACM Comput. Surv.*, 32(3):241–299, September 2000.

[36] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 451–464, Denver, CO, 2016. USENIX Association.

[37] JF Murray, GF Hughes, and K Kreutz-Delgado. Comparison of Machine Learning Methods for Predicting Failures in Hard Drives. *Journal of Machine Learning Research*, 6, 2005.

[38] Oracle, Inc. Oracle PL/SQL. http://www.oracle.com/technetwork/database/features/plsql/index.html. Accessed: 2020-01-15.

[39] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[40] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Operating Systems Review*, 43(4):92–105, December 2009.

[41] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[42] Niels Provos and David Mazières. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[43] Redis. http://redis.io/. Accessed: 2020-01-15.

[44] Felix Salfner, Maren Lenk, and Miroslaw Malek. A Survey of Online Failure Prediction Methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[45] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *Proceedings of the 12th European Conference on Computer Systems*, Eurosys '17, pages 175–190. ACM, 2017.

[46] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[47] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.
.

[48] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[49] The PostgreSQL Global Development Group. PostgreSQL: Documentation: 10: H.4. Extensions. http://www.postgresql.org/docs/10/static/external-extensions.html. Accessed: 2020-01-15.

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[51] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[52] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[53] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.