Leave Nothing Idle: Filling Datacenter Resource Utilization Gaps with Quicksand

Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Seo Jin Park, Malte Schwarzkopf, Adam Belay

Abstract. Data centers have poor utilization of CPU and memory, as applications have a varying consumption of these resources that do not always match their varying availability at each server. The problem is that current abstractions for execution—such as tasks, containers, or virtual machines—have a coarse time granularity and inherently couple computation and state at each server, making it impossible to schedule resources efficiently across servers.

We propose new execution abstractions that allow applications to use resources at a finer time granularity without coupling them. Our key idea is to break applications into resource proclets: granular units of execution that dominantly consume a single type of resource and can split, merge, and migrate across servers to use that resource anywhere in the cluster, even if it is available for only tens of milliseconds, or stranded, such as free memory on a server without spare CPUs. Resource proclets can be combined together into full distributed applications through easy-to-use programming abstractions and an efficient, cluster-wide runtime.

Quicksand, our prototype, provides high-level C++ abstractions for state, compute, and familiar programming models, but dynamically decomposes computation and state. Experiments with batch sorting, an ML training pipeline, and a latency-critical social network service show that Quicksand supports a variety of workloads, achieves high performance, and effectively uses otherwise-stranded resources.

1 Introduction

Data centers today have low resource utilization, with several sources reporting CPU and memory utilization below 60% [11, 47]. This is costly for both cloud providers and users: providers get only part of the potential revenue for their expensive hardware, while users pick up the tab with higher prices. This underutilization happens because of a mismatch between applications’ resource needs and reservation abstractions. On one hand, application needs vary in quantity and ratio of resources, often at high frequency, as external load changes [5, 25] and applications go through phases (e.g., a sorting application may oscillate between I/O-heavy data movement and CPU-intensive in-memory sorting [35]).

On the other hand, application resource reservations are based on abstractions—such as tasks, containers, or virtual machines—that reserve resources in large fixed chunks (GBs of memory or a CPU core at a time) for long periods (minutes or hours), and must include any extra capacity needed to handle bursts. This mismatch causes overprovisioned resources (idle reserved resources), and stranded resources (free resources without other resources, making them hard to use, such as free memory without free CPU) [23, 42, 47].

In this paper, we describe resource proclets, a new execution abstraction that helps data center applications use different types of server resources independently, and at a fine granularity (tens of milliseconds and megabytes of memory), so applications can use stranded resources and minimize overprovisioning. Resource proclets break down monolithic applications into small, granular units that each consume primarily one dominant resource type (viz., memory and compute). A resource proclet has a small size, so that it can migrate between machines quickly on modern datacenter networks. For example, a memory proclet might contain only a shard (say, of 4MB) of a large vector of images, while a compute proclet might contain a small heap and stack (say, with 100 KB) and do substantial compute work.

Resource proclets allow applications to use stranded resources, as resource proclets decouple resource consumption. Since resource proclets reserve and free resources quickly, a server may have idle resources available only briefly (e.g., for tens of milliseconds). Resource proclets can use such resources productively by migrating to servers with idle gaps from others that are busy. Resource proclets also help applications react to load changes, since the application can create new resource proclets to harness more resources with fine granularity on additional machines.

We realize resource proclets in Quicksand, a system that offers compute and memory proclets. Compute proclets can operate over the data in memory proclets even if they run on a different machine, and both types of proclets can migrate around the cluster freely.

Making resource proclets effective required us to address three key challenges. First, resource proclets are a low-level abstraction, while developers prefer to work with higher-level data structures and compute pools. Quicksand addresses this by separating application developers from library developers, and giving them different abstractions. Quicksand provides library developers with frameworks, a collection of APIs that allows the developers to make use of resource proclets without interacting with them directly. Application developers can then use these libraries in the same way they
would familiar data structures and compute abstractions, such as C++ STL containers and MapReduce. Thus, writing Quicksand applications looks and feels similar to classic, single-machine programming.

Second, Quicksand needs to use the right number of resource proclets: too few and the resource proclets become too large, which impedes fast migration; too many and resource proclets become too small, causing communication overheads to dominate. To address this issue, Quicksand splits and merges resource proclets to keep them granular: splitting a memory proclet partitions its state into two shards (e.g., by key range of a hash table), and splitting a compute proclet breaks up its work (e.g., loop iterations) into two parts. Merging does the reverse and combines proclets, so the right number of proclets emerges dynamically.

Third, leveraging idle and stranded resources at fine granularity requires an efficient runtime that migrates proclets across servers and routes communication between them. Quicksand provides a generic auto-sharding layer that provides location transparency, as well as a centralized controller for proclet placement and efficient remote access based on kernel-bypass networking.

To illustrate the power of Quicksand, consider a system with expensive GPUs that are shared over time by different applications. One of them is a machine learning pipeline in which a batch of raw images undergo several stages of pre-processing on CPUs (resizing, noise reduction, color correction, etc.) before training a model on the GPUs. Because GPUs are expensive, keeping them fully utilized is critical [26, 29–31], so we must ensure that pre-processing always produces enough images to feed the GPUs. Furthermore, we want to minimize the amount of CPUs and memory reserved for the pre-processing, to keep their utilization high also. Quicksand achieves these goals: as the availability of GPUs changes, Quicksand quickly scales CPU and memory resources by adding or removing the corresponding proclet type, to ensure the GPUs are always busy. Furthermore, as CPU and memory resources fluctuate, Quicksand migrates proclets to escape overloaded servers and utilize even brief periods of idle compute and memory elsewhere.

We implemented a prototype of Quicksand, which provides both low-level components for building abstractions atop resource proclets, and high-level APIs that make the underlying decomposition into resource proclets transparent to an application. Our prototype has many distributed data structures (hash tables, vectors, queues, etc) built from resource proclets, and also provides interfaces for developers to realize parallel batch computation, and easily build stateful and stateless services from resource proclets.

We ported four applications to Quicksand: an image processing pipeline for machine learning, a latency-critical service from DeathStarBench’s [17], a compute-intensive sort tool, and a serverless video encoding system from ExCamera [15]. Experiments show that Quicksand’s abstractions are general enough to support these workloads, and that Quicksand is able to harness otherwise stranded or idle resources at fine granularity. We also find that Quicksand offers an attractive platform to develop applications that react quickly—within tens of milliseconds—to changing load and resource availability by splitting and merging resource proclets.

In summary, our key contributions are the following:

1. Resource proclets, a new abstraction for decomposing applications into small units of state and compute that split, merge, and migrate independently on a cluster of machines and use resources wherever available.
2. Quicksand, a system that simplifies developing applications using resource proclets, by providing sharded data structures, parallel computation pools, and services.
3. An efficient cluster-wide runtime that reacts to changing load and resource availability by creating, destroying, and migrating resource proclets between machines with little disruption for applications.
4. Evaluation of our Quicksand prototype with four workloads, demonstrating that resource proclets increase utilization when resources are stranded, react quickly to changing application needs and resource availability, and achieve good performance.

Our prototype has some limitations. It requires developers to specify policies for splitting and merging resource proclets that capture application objectives (e.g., short queue length, low tail latency), although these policies are pluggable. Quicksand only implements compute and memory proclets, but we envision proclet support for other resource types (e.g., GPUs). Quicksand’s decoupling works particularly well when a workload’s compute intensity is high, while workloads with lower compute intensity require some locality between their proclets to achieve high performance.

## 2 Background and Motivation

Datacenters provide a shared infrastructure that hosts many types of resource-hungry workloads whose resource consumption vary over time. In theory, this variety creates an opportunity to improve resource utilization when there is some independence of consumption across applications and time (i.e., different applications use different resources at a time). In practice, achieving high utilization is hard because resources are bundled together within servers, resource consumption can change rapidly, applications consume different types of resources together, and applications are difficult to move across servers.

Consider our motivating example of a machine learning pipeline that pre-processes images on CPUs and uses the result to train a model on GPUs that are shared over time with other applications. These goals are hard to achieve today: systems that reserve fixed amounts of CPU and memory will undershoot or overshoot as the availability of GPUs changes [31], while systems that try to dynamically scale
CPU and memory take too long to adapt (see below). And, as available CPU and memory resources also vary, we want to use them wherever available. An ideal solution needs to rapidly scale CPU and memory resources when GPU availability changes, and quickly move the consumption of CPU and memory across servers.

**Autoscaling frameworks.** Slicer [1] and Shard Manager [27] help developers add sharding to applications, and add or remove shards in response to load. They target long-running, large-scale systems with large shards (web service front-ends, storage systems, web caches, etc.), with a reaction time in the order of minutes. We need reaction times in tens or hundreds of milliseconds for fine-grained components that may be short-lived.

**Resource decoupling.** Monotasks [35, 36] propose the idea of decomposing a job into (mono)tasks that each consume a single resource, thereby decoupling resource consumption similarly to resource proclets. Monotasks have different design choices and goals: their primary aim is to make it easier for users to reason about performance, and they target larger time granularity than resource proclets, making them insufficient to fill brief periods of idle resources to improve resource utilization.

**Resource disaggregation.** Resource disaggregation improves the utilization of resources by moving them outside the confines of a server into a shared resource pool. This idea helps to avoid overprovisioning and stranded resources, but to be efficient it requires new hardware architectures that are not yet deployed [13, 19, 28, 45]. Memory disaggregation can be implemented on current hardware with software assistance, using techniques such as paging [21, 22] or new programming constructs [43, 52]. These approaches are specific to the memory resource, and they incur high overhead (with paging) or require significant developer effort (with the new programming constructs). Overall, resource disaggregation and resource decoupling are dualistic ideas: disaggregation separates resource offering, while decoupling separates resource consumption.

**Resource harvesting.** Resource harvesting aims to improve resource utilization by using idle resources that may be available only intermittently, using evictable VMs [3, 7, 18] and many subsequent improvements [4, 41, 46, 48, 50]. This idea targets secondary applications that execute opportunistically but need not run if the resources are unavailable or disappear. Quicksand targets primary applications and uses more fine-grained execution abstractions than VMs.

**Specific applications.** Some prior work focuses on improving specific applications. Data pipelines for ML can be optimized through custom techniques [16, 20, 51]. Likewise, data analytics query processing can benefit from state separation [2, 49], separating the working state and progress of a query from the machine executing it to facilitate migration. Quicksand targets general datacenter applications, rather than a single kind of application.

**Parallel Programming.** Many programming frameworks make it easier for developers to exploit parallel hardware resources, either within a machine [10, 39] or across machines in a distributed setting [8, 9, 32, 33]. Quicksand builds upon these ideas, but brings a new focus of decoupling resource by types and decomposing applications into granular units.

**Logical Processes.** Nu [42] proposes the idea of proclets as fungible and granular units of work. While Quicksand adopts a similar terminology as Nu (resource proclets vs. procets), Nu is hard to use: app developers must write code on top of Nu’s proclet interface, which is low-level and requires a fixed choice of proclet size (Nu’s proclets cannot grow or shrink). Determining this size requires developers to have a deep understanding of the external load and resource needs of the application. Nu also cannot use stranded resources, as Nu’s proclets couple together compute and state. Quicksand, by contrast, decouples resources with resource proclets, and reacts to changing load by adjusting proclet size dynamically through splitting and merging, and provides high-level frameworks for developers to program against. However, Quicksand borrows some low-level building blocks from Nu, such as fast migration and communication between proclets, and offers “hybrid proclets”, which couple compute and state like Nu’s proclets do.

### 3 Quicksand Overview

Quicksand’s goal is to raise utilization in datacenters by decoupling resources and aligning allocation with actual use, while making it easy for developers to write applications that make use of resources wherever they are available.

Figure 1 gives an overview of Quicksand: applications link with libraries that provide distributed data structures, batched parallel computation, and building blocks for latency-critical services. These libraries are implemented within high-level frameworks that Quicksand provides and which manage and orchestrate resource proclets. Resource proclets decouple state and compute and allow fast reaction to varying resource needs and availability via Quicksand’s runtime.
## Design

This section first introduces Quicksand’s high-level frameworks built atop resource proclets. These frameworks provide APIs to library developers, who in turn implement libraries that applications use. We then explain Quicksand’s auto-sharding layer, which is used by all frameworks for sharding resource proclets. Finally, we describe resource proclets and how frameworks use them.

### 4.1 High-Level Frameworks

Quicksand’s high-level frameworks provide common functional components of datacenter applications: in-memory distributed data storage (e.g., caching layers, message queues), parallel batch computation (e.g., MapReduce), and stateless and stateful latency-critical services (e.g., web serving, key-value stores). Our goal in designing the frameworks is to provide useful building blocks for library developers. These building blocks provide generic support for creating, splitting, and merging resource proclets using default policies and mechanisms. Thus, each framework chooses a resource proclet type, (default) split/merge policies to decide when to split/merge resource proclets, and split/merge mechanisms to determine how to split/merge resource proclets. The actual splitting and merging is the responsibility of the auto-sharding layer (§4.2), which is the same for all frameworks but uses framework-specific policies and mechanisms.

Figure 2 provides an overview of the three frameworks of Quicksand, which we next describe in detail.

| Framework            | Sharding key  | Proclets                      | Default split policy   | Split mechanism           | Default merge policy       | Merge mechanism
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>DS-dependent</td>
<td>Memory proclet/hybrid proclet</td>
<td>Container too big</td>
<td>Partition container and key range</td>
<td>Container too small</td>
<td>Combine containers and key ranges</td>
</tr>
<tr>
<td>Batch computing</td>
<td>Task index</td>
<td>Compute proclet</td>
<td>Remaining tasks &amp; idle CPU in the cluster</td>
<td>Partition task range</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Service (stateful)</td>
<td>Session/user ID</td>
<td>Hybrid Proclet</td>
<td>Use more than a core</td>
<td>Partition ID range and state</td>
<td>Use less than half a core</td>
<td>Merge ID ranges and states</td>
</tr>
<tr>
<td>Service (stateless)</td>
<td>Random ID</td>
<td>Compute Proclet</td>
<td>Use more than a core</td>
<td>Partition ID range</td>
<td>Use less than half a core</td>
<td>Combine ID ranges</td>
</tr>
</tbody>
</table>

**Figure 2.** Quicksand’s high-level frameworks for composing efficient distributed applications. The frameworks automatically maintain the right granularity of each shard through adaptive splitting and merging at runtime.

### 4.1.1 Sharded Data Structures

The data structure framework provides support for building sharded data structures atop memory proclets, with default implementations for all C++ STL container types. This data structure framework builds on Quicksand’s auto-sharding layer to internally divide data structures’ data across memory proclets based on a data structure-specific sharding key. For instance, a sharded vector shards over vector indices, such that each memory proclet holds data for a contiguous range. A sharded map, by contrast, shards into ranges of the map’s key space, and each memory proclet holds key-value pairs for a key range.

Library developers add new data structures using the framework without having to deal with the details of memory proclets. Consider the toy example of a vector of integers (avoiding generics for simplicity) in Figure 3. The library developer extends `std::vector<int>`, adding type definitions for the sharding key and value, and provides implementations for `Split` and `Merge`, which split a vector’s elements and combine two vectors’ elements, respectively. Then, the library developer defines a new, application-facing...
Figure 3. To create a vector of integers with Quicksand, a library developer writes a wrapper (ShardedIntVector) around std::vector that provides the split/merge functionality for the vector data, which Quicksand stores in memory proclcts. qs::ShardedDS is provided by Quicksand’s data structure framework and handles the details of sharding and proclet decomposition.

```cpp
// Written by library developer
class ShardedIntVector : std::vector<int> {  
public:  
    using Key = size_t;  
    using Val = int;  
    std::tuple<Key, ShardedIntVector> Split() { /*...*/ }  
    void Merge(ShardedIntVector& other) { /*...*/ }  
};  
using IntVector = qs::ShardedDS<ShardedIntVector>;
```

Figure 4. A dataframe application written with a MapReduce implementation based on Quicksand’s batch computing framework. The vectors and map are data structures sharded into memory proclcts, and the Map and Reduce operations run in compute proclcts.

```cpp
// Written by application developer
int main() {  
    IntVector v;  
    v.push_back(42);  
}
```

that split and merge via the auto-sharding layer to control the degree of parallelism. The framework includes a MapReduce implementation in its library; library developers might add other operations (e.g., Filter, FlatMap, joins, or streaming-based transformations).

Figure 4 shows how an application uses the MapReduce implementation. In the map phase, the program iterates over an element-wise zipped view of the two input vectors and computes an output for each input pair. The reduce phase counts the number of occurrences over the set of output values, and stores the results in a sharded unordered map.

Internally, both phases of the program parallelize using compute proclcts. The map stage first creates a compute proclet that takes in a zipped view of the two sharded vectors and applies the map function (expensive_merge) on each element. As the vectors are large and already sharded, Quicksand’s auto-sharding layer splits the initial compute proclet into many compute proclcts, each responsible for a range of vector elements, to leverage the cluster’s idle CPUs. The compute proclcts operate in parallel and produce intermediate output from the map phase in a sharded data structure. The reduce phase then starts a new compute proclet, splits it, and processes the map phase output in parallel, storing its results in m, an empty map created via qs::MakeMap(). Each phase utilizes Quicksand’s auto-sharding layer to keep track of each compute proclet’s range and its current progress.

Split and Merge Policy and Mechanism. Compute proclcts’ processing time may vary, either due to data skew or due to external factors like a slow server. When some compute proclcts complete, the auto-sharding layer identifies idle CPU resources and applies an externally-induced split on remaining compute proclcts to mitigate stragglers. Slow compute proclcts cut their remaining range of elements to be processed in half, and create new compute proclcts that takes over the other half.

Latency-Critical Services. This framework provides support to build scalable latency-critical services, which receive client requests and return a reply. Unlike the other frameworks, the service framework is typically used directly by application developers, rather than extended by library developers, as the provided stateless and stateful service class is sufficient for most services. A service can be stateless or
stateful, depending on whether the reply is a deterministic function of the request alone or the reply depends on internal state kept by the service. Accordingly, Quicksand includes classes for stateless and stateful services in the framework library. The stateless service is parameterized by a hard-coded lambda function (provided by the app developer) that produces the response. The stateful service is parameterized by a more general service class (also provided by the app developer), which can implement application-specific RPC interfaces and store state in the shard’s hybrid proclet heap.

The stateless service class uses compute proclets and Quicksand’s auto-sharding layer. Client requests have a random 64-bit number, which Quicksand uses as a key in the auto-sharding layer to distribute the requests among compute proclets. The stateful service class uses hybrid proclets managed by the auto-sharding layer. Client requests have a per-client or per-session identifier, which Quicksand uses to implement sticky routing through the auto-sharding layer.

Internally, both stateless and stateful services parallelize interactions with clients to scale the service and keep tail latency low by dispatching requests to different proclets. Quicksand’s auto-sharding layer provides request routing, and the service framework offers C++ client classes that embed proclet location caching.

**Split and Merge Policy and Mechanism.** Both stateless and stateful services use the same split policy: once the proclet uses more than one CPU, a shard splits itself into two. Conversely, a service winding down from a load spike may have proclets that underutilize their CPU resource, and the auto-sharding layer reacts to this by merging proclets, adapting a service’s resources to its current load. When splitting and merging a stateless service’s shards, Quicksand splits or combines the key range they are responsible for. For a stateful service, the app developer must provide Split and Merge functions for splitting and merging the internal data structures of the service.

### 4.2 The Auto-Sharding Layer

Quicksand’s high-level frameworks use the auto-sharding layer to distribute their computation or state by managing resource proclets. Each framework defines its own sharding class (e.g., ShardedDS, StatelessService, etc.), which interacts with Quicksand’s components for auto-sharding.

Consider the example of ShardedDS in Figure 6. The library developer instantiates ShardedDS with the library class they provide that implements the Split and Merge functions (e.g., ShardedIntVector in Figure 3). ShardedDS re-exports the library class’s API, but shards the data structure into memory proclets. Library developers use Split to split a local data structure with familiar C++ techniques (e.g., cutting an std::vector in half). Quicksand takes the split output and moves it to a new memory proclet transparently. Conversely, Merge combines two shards, with Quicksand handling the data movement. A proclet can trigger a Split or Merge itself or receive such requests from external sources. A library can also implement ShouldSplit and ShouldMerge functions to implement custom granularity policies—e.g., a data structure may bound its shard size, a batch computation may set a minimum number of elements per batch, and a service may split on a queuing delay threshold.

Quicksand’s sharding layer’s AutoSharder class maintains shard mappings, resolves keys to shards, and monitors split/merge with default policies. AutoSharder shards over the key range in the Shard::Key type, and makes each proclet responsible for a part of that range. Quicksand stores the mapping of key ranges to proclets in a centralized shard mapping proclet. The AutoSharder communicates each split and merge to the sharding mapping proclet to maintain a consistent view, and caches a local view at the client. A Quicksand application may contain multiple shard mapping proclets, each for a different use case. For example, one shard mapping proclet may manage memory proclets in a distributed hash table, while another tracks compute proclets for a parallel processing pipeline over that hashtable’s contents, and a third realizes a key-value store service over the hashtable.

C++ code using AutoSharder caches shard mappings locally. When a client makes a request based on a stale mapping view, the request might be routed to an incorrect shard. Shards reject requests they are not responsible for, and upon rejection, a client fetches the latest mappings from the shard mapping proclet and re-issues the request.

### 4.3 Resource Proclets

Quicksand supports two types of resource proclets, compute proclets and memory proclets, as well as hybrid proclets. Figure 5 shows their APIs, which we discuss in the following.

**Compute Proclets** express computation on one CPU. Each compute proclet forms a thread of computation that can be independently migrated across machines. Although Quicksand assumes that the code a compute proclet primarily consumes CPU time, it has stack memory and may allocate on the heap if needed. Because compute proclets are small, Quicksand migrates a compute proclet under 100µs.

To construct a compute proclet, Quicksand provides two APIs: Run(Lambda, Args...) starts the function passed with the arguments specified inside a compute proclet, and Run(Range, Lambda, Args...) does the same but maintains metadata about the input range processed. The range-based API is useful upon splitting a compute proclet, the range is split into two (with default Split implementation customizable by library developers) and the new compute proclet takes over part of the range.

Like popular serverless platforms, the compute proclet API enables developers to execute computations without thinking about server infrastructure or locality. Unlike serverless functions, compute proclets can migrate during their execution, enabling Quicksand applications to utilize stranded CPUs and react to compute pressure.
Table 1. Quicksand's Proplet API.

<table>
<thead>
<tr>
<th>Proplet Type</th>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Proplet</td>
<td>Run(Fn)</td>
<td>Runs function Fn on compute proplet.</td>
</tr>
<tr>
<td></td>
<td>Run(Range, Fn)</td>
<td>Runs Fn over a range of input on compute proplet.</td>
</tr>
<tr>
<td>Memory Proplet</td>
<td>Read(Fn) -&gt; [uint8_t]</td>
<td>Runs function Fn on proplet’s heap; copies out data.</td>
</tr>
<tr>
<td></td>
<td>Write(Fn, [uint8_t])</td>
<td>Runs function Fn to mutate proplet’s heap; may alloc/free.</td>
</tr>
<tr>
<td></td>
<td>Seal()</td>
<td>Makes memory proplet read-only; enables prefetching.</td>
</tr>
<tr>
<td></td>
<td>Unseal()</td>
<td>Makes read-only memory proplet writable; disables prefetching.</td>
</tr>
<tr>
<td></td>
<td>Iterate() -&gt; Iterable</td>
<td>Returns iterator that iterates over proplet contents in impl-defined order.</td>
</tr>
<tr>
<td>Hybrid Proplet</td>
<td>arbitrary RPC API</td>
<td>implementation-defined.</td>
</tr>
</tbody>
</table>

Figure 5. Quicksand’s compute proplet API starts functions, while the memory proplet API serves reads, writes, and iteration. When sealed, memory proclets allow safe prefetching. Run, Read, and Write have asynchronous counterparts (not shown).

```cpp
// Provided by Quicksand
1 template <typename Shard> class ShardedDS {
2 public:
3   void push_back(Shard::Val v)
4     requires PushBackable<Shard> {
5       MemoryProcletPtr mp = sharder_.last();
6       // ... handle mapping cache misses via sharder_ ...
7       mp.Write([v] (Shard &s, auto v) {
8         s.push_back(v);
9         if (s.ShouldSplit()) {
10            // split, inform sharder */
11        }, v);
12     }, v);
13   };
14   Shard::Val at(size_t pos) requires Indexable<Shard> {
15     MemoryProcletPtr sharder_.find(pos);
16     // ... retry and error handling code ...
17     return mp.Read([Shard &s, auto i] { return s[i]; }, pos);
18   }
19   // other variants for Insertable, Emplaceable, etc.
20 private:
21   qs::AutoSharder<Shard::Key> sharder_;
22};
```

Figure 6. The data structure framework’s ShardedDS uses Quicksand’s auto-sharding functionality (AutoSharder) to locate, split, and merge proclets, and realizes each shard as a memory proplet. The Shard type parameter is initialized, e.g., as ShardedIntVector in Figure 3, which implements a C++ concept PushBackable.

Compute proclets support both ephemeral and long-running computations. A compute proplet can create more compute proclets and wait for their completion. However, a compute proplet cannot directly access another compute proplet’s memory. Instead, compute proclets use memory proclets to share computation results, which we discuss next.

A Memory Proplet is an independently migratable heap. Quicksand overrides malloc and free within a memory proplet so that they allocate from its heap. Memory proclets’ dominant resource is memory, but they also consume a minimal amount of compute to support reads and writes.

Quicksand provides a MakeMemProplet(capacity) API to create a memory proplet with an initial allocation of the specified capacity. This API returns a memory proplet pointer that exposes Read(read_fn) and Write(update_fn) APIs. The internal side of Quicksand’s high-level frameworks uses these APIs (and their asynchronous counterparts) to operate on the memory proplet’s contents. For example, consider the code from ShardedDS in Figure 6: after obtaining the memory proplet pointer for a key via the AutoSharder, the push_back and at methods access state stored in a memory proplet. When invoking Write(update_fn) on line 8, the update function calls push_back on the underlying shard (e.g., a ShardedIntVector, which amounts to calling it on std::vector). The update function could also allocate and free memory, and modify it in arbitrary ways. In our example, the update function invokes ShouldSplit to check if the memory proplet has grown too large and needs to split. The update function can optionally return a value: this is helpful for supporting operations such as compare-and-swap.

The Read(read_fn) API call on line 18 calls at on the underlying type to fetch part of the allocated memory. Because memory proclets can be remote, Read makes a copy of the accessed state. Quicksand can support copy elision when accessing a local memory proplet.

By default, the Quicksand runtime performs concurrency control to mediate concurrent memory proplet read and write accesses. In particular, Quicksand wraps the shard type (e.g., ShardedIntVector) in another wrapper type that contains a mutex and locks it on an access. For use cases that require custom synchronization (e.g., advanced data structures), Quicksand provides an alternative wrapper that lets the memory proplet implement its own synchronization.

Last but not least, Quicksand supports sealing a memory proplet to make it read-only. Sealed memory proclets allow for efficient runtime prefetching, since there is no risk of reading stale data from the prefetched cache even with concurrent access. When a memory proplet is sealed and code invokes Iterate on it, the Quicksand runtime prefetches subsequent elements in the background to hide access latency. To support custom iteration order, developers can specify an Iterate method in the shard type implementation. This is useful when the memory proplet’s heap requires iteration in a particular order (e.g., an ordered set).

Hybrid Proclets have the least constrained API, but also offer the least placement flexibility, as they may use arbitrary amounts of computation and memory. Hybrid proclets are
similar to the proclets in Nu [42], but unlike Nu’s statically-partitioned proclets. Quicksand’s hybrid proclets have Split and Merge interfaces that developers can implement to support auto-sharding.

5 Implementation

Our prototype has 8.5k lines of C++ on top of Nu [42].

**Sharded data structures.** The Quicksand default data structure library provides distributed counterparts for all C++ STL container types. It uses classic STL containers inside the memory proclets (for example, a shared vector contains memory proclets hosting std::vector). To build new data structures, Quicksand provides a sharded data structure base class that implements shard management, request routing, error retry, batching, and read prefetching. Thus, library developers can focus on the implementation of one shard, by providing an in-memory data structure and its split/merge mechanism. For example, we implemented a shared vector in 180 LOC by wrapping std::vector and adding a Split and Merge implementation.

Library developers need only provide a single-threaded implementation, as Quicksand has a concurrency control layer to handle concurrent reads, writes, splits, and merges for data structure shards. We optimized this layer to minimize overhead, e.g., by using an RCU read lock to guard against splitting while accessing a shard. Library developers can also opt out of the concurrency control layer and provide thread-safe data structure implementations.

Splitting, merging, and migration. To make good split and merge decisions, we co-design our resource proclets with the Quicksand runtime for high-fidelity signals. The runtime collects resource usage, with each server periodically reporting its usage to a central controller. Quicksand uses this information to trigger splits and merges. For instance, the auto-sharding layer splits a compute proclet if it uses a full CPU and the cluster has idle CPU cycles.

Quicksand’s runtime also supports fast resource proclet migration. It uses a slab allocator per proclet to help the runtime identify the state to migrate. To keep memory proclets fine grained with a reasonable metadata overhead, our prototype splits a memory proclet if its heap grows beyond 16 MB. Library developers can configure this threshold if their use case benefits from more or fewer splits, based on a tradeoff of fungibility and migration overhead.

Limitations. Our Quicksand prototype has some limitations. Our runtime can monitor affinity among compute and memory proclets, but we haven’t implemented locality-aware scheduling or migration to optimize for locality between compute and memory proclets yet. Our prototype also does not yet enforce usage limits on non-dominant resource types in resource proclets (e.g., limiting compute time in memory proclets).

6 Application Case Studies

We implemented and ported four applications into Quicksand. They cover both batch and latency-critical applications and a range of compute intensity, memory usage, proclet usage, and Quicksand’s framework usage (Figure 7).

**Social Network.** We ported the SocialNetwork application from DeathStarBench [17] to Quicksand. SocialNetwork is a latency-critical, Twitter-like web service, originally built using twelve microservices. It has a low compute intensity and uses 33.4 GiB of memory. Our port builds on the Nu-based version of SocialNetwork, which structures the application into a statically-provisioned stateless layer and statically-partitioned key-value store [42]. Porting this to Quicksand required changing 60 LOC: we replace the static stateless layer with Quicksand’s shardable stateless service and replace the static key-value stores with Quicksand’s shardable hashtable. Since the original application is relatively simple, porting to Quicksand required changing few changes, even though it moved substantial functionality from the application into the framework (e.g., shard resolution) and added extra functionality (e.g., load-adaptive dynamic sharding).

**Sorting.** We built an in-memory batch distributed sorting job that sorts 400 million 100B key-value pairs, similar to TritonSort [40]. This is a batch application with low compute intensity and uses 112 GiB of memory, and implemented in around 300 LOC. Like typical distributed sorts, our implementation consists of two stages: (i) a shuffle of the input data into ranges by key; (ii) sorting the ranges in parallel. For the first stage, we built a ShardedPartitioner using Quicksand’s data structure framework, which uses the input key as the sharding key. For each shard, the ShardedPartitioner maintains a vector of unordered key-value pairs. Upon splitting, it uses the quickselect algorithm [24] to evenly partition the key-value pairs into two disjoint ranges. Shuffling input amounts to replacing the key-value pairs into the ShardedPartitioner. We realize the partitioner with hybrid proclets instead of memory proclets so the second stage, which has low compute intensity, runs directly on the shards.

**ML training pipeline.** We built an OpenCV-based pipeline that includes a GPU-based data preprocessing stage and a GPU-based training stage for our motivating workload from §1. This is an example of a compute-intensive batch application. We use images from ImageNet ILSVRC2012 dataset [44] as our training data, leading to 33.4 GiB memory usage. The pipeline’s output is much larger, but can be generated just-in-time as the GPUs consume images for training.

The preprocessing stage has 300 LOC: we load the input data into a sharded vector, and use the batch computing framework to perform image transformations. The outputs are written into a shared queue that is connected to the GPU-based training stage. We implement a custom compute proclet split policy based on the queue length signal (20 LOC), adjusting the number of compute proclets used in the
Figures 7, 8, and 9. Our four case study applications use different high-level Quicksand frameworks. Their memory usage and the number of resource proclets vary over time; we report the peak values measured in our evaluation.

preprocessing stage to match the consumption rate of the training stage. Since our testbed lacks GPUs, we simulate the training stage by consuming images from the pipeline at a rate of 0.25 milliseconds per image sample, roughly matching with the reported rate of image classification models [14, 37].

**Video encoding.** We ported the distributed video encoder from ExCamera [15] as another example of a batch application with high compute intensity. ExCamera is designed for the serverless AWS Lambda platform [6], and has two major components: a C++-based functional encoder and an orchestrator that spawns and schedules lambda instances to run encoders in parallel. ExCamera’s code is no longer maintained or runnable, so we created a faithful baseline implementation. Our port of that baseline to Quicksand adds or modifies 200 LOC of the original 6K LOC. We load the video chunks (17.5 GiB) into a sharded vector, and apply Quicksand’s batch computing framework to invoke the functional video encoder in parallel.

### 7 Evaluation

Our evaluation focuses on answering the following:

1. Can Quicksand combine stranded resources from different machines to increase resource utilization? (§7.1)
2. Can Quicksand adapt to applications’ changing resource needs to reserve only the necessary resources? (§7.2)
3. Can Quicksand adapt to changing resource availability to ensure high performance and utilization? (§7.3)
4. Can Quicksand adapt to skewness of input data to balance load and avoid stragglers? (§7.4)
5. How does porting applications to Quicksand impact their performance? (§7.5.2)

**Setup.** Our testbed has eight machines, each equipped with an Intel Xeon E5-2680 v4 CPU and 64 GB DDR4 2133MHz memory, connected by a 100 GbE network, running Ubuntu Linux 22.10 with kernel v5.10. In line with prior work [34], we enable hyper-threads, but disable CPU C-states, dynamic CPU frequency scaling, transparent huge pages, and kernel mitigations for transient execution attacks.

We dedicate one machine to hosting Quicksand’s cluster controller (inherited from Nu). For the ML training pipeline, we use one machine as the GPU server (for training) and the remaining six machines as the CPU servers (for preprocessing). For the social network experiments, we use one machine as the load generator and the remaining six machines as servers. For sorting and video encoding experiments, we use all seven machines to run the application.
7.1 Using stranded resources

We first investigate if Quicksand can effectively use machines with stranded resources (e.g., servers with ample free CPU but little memory or vice-versa). We use the ML training pipeline (whose compute intensity is high) and the social network application (whose compute intensity is low). To create resource stranding, we divide the resources needed by the workload (78 CPUs, 36 GiB memory) between machines in different ways: in the “CPU-unbalanced” setup, three machines have few available CPU cores, but memory is equally distributed; in the “memory-unbalanced” setup, three machines have little memory, but available cores are equally distributed; in the “both-unbalanced” setup, three machines have few CPU cores and plenty of memory, while others have plenty of available CPU cores, but little available memory. To understand if Quicksand successfully uses stranded resources, we compare Quicksand’s performance on these unbalanced setups to an ideal setup, where the aggregate resources are the same, but distributed over only three machines that each have ample CPU and memory. In all cases, the workload uses nearly 100% of the available resources.

A good result for Quicksand would show close-to-ideal performance when compute intensity is high, as Quicksand can decouple memory and CPUs using resource proclets. When compute intensity is low, resource proclet RPC overheads dominate, and a good result for Quicksand would show limited performance loss compared to the ideal setup.

Figure 8 shows the results of the compute-intensive ML training pipeline. In the ideal setup, each machine uses all 26 cores and ≈19 GiB memory, achieving throughput of 16,251 images/sec. In unbalanced setups, Quicksand achieves similar throughput regardless of resource distribution.

Figure 9 shows the results of the social network application with low compute intensity. In the ideal setup, each machine uses all 26 cores and ≈11 GiB memory, achieving 810k operations/sec. Despite a low compute intensity, Quicksand achieves 40%–84% performance in unbalanced setups. These results demonstrate that Quicksand can use stranded resources productively, particularly for applications with high compute intensity.

7.2 Adapting to changing resource needs

We investigate if Quicksand can quickly scale an application’s resources to adapt to changes in external load or phased application behavior. We use the SocialNetwork application, which has varying load, and the video encoding application, which has phased behavior.

For SocialNetwork, we switch the offered load back and forth between 500k and 1M operations/second every 500 ms, creating a bursty load. We compare Quicksand’s response to an ideal system that reacts immediately and perfectly matches the resource needs. A good result would show Quicksand quickly scaling the number of compute proclets to match the load while retaining low tail latency. Figure 10 shows the results. Quicksand comes close to the ideal: it takes ≤100ms to scale the number of compute proclets to the correct level, and retains low tail latency throughout.

The video encoding application has multiple stages that encode chunks in parallel, followed by a serial stitching stage that combines encoded chunks together. We run the ExCamera baseline and the Quicksand-based version. A good result for Quicksand would show its resource usage matching the application’s available parallelism.

Figure 11 shows the results. The parallel encoding stage in the beginning (0–0.8s) has high parallelism, and both ExCamera and Quicksand use 15 CPUs. Quicksand’s gap at 0.2s occurs because our prototype performs synchronization between two consecutive parallel stages; when it scales back

\[1\] In ExCamera’s notation, this means encoding chunks of 6 frames in parallel, and then combining them in strings of 16 chunks [15].
We investigate if Quicksand reacts to changes in availability with a single core at 1.2s, which combines data from the preprocessing stage to match the training stage.

Therefore, Quicksand decreases its resource usage as parallelism decreases. Quicksand starts the serial stitching stage with a single core at 1.2s, which combines data from the sharded vector. By contrast, ExCamera’s serverless-based implementation couples compute and state; the encoding lambdas must remain alive until the stitching lambda consumes its output, leading to significant idle CPU resources between 1.2–3.7s. The area between two lines represents CPU overprovisioning reclaim by Quicksand.

In summary, these results demonstrate that Quicksand can quickly adapt to the rapid change in resource needs to provide high performance and improve resource utilization.

7.3 Adapting to changing resource availability

We now study Quicksand’s ability to adapt to input data skew, which is known to cause load imbalance, poor utilization, and stragglers [12]. We use the sorting application with an input that follows either a normal distribution (which is skewed) or a uniform distribution (no skew). We compare Quicksand with two baselines implemented using Nu [42]. The sample-based baseline handles the skewed input by running an extra computation phase that samples 0.5% of the input to learn its key distribution and choose appropriate key pivots to partition the input across machines. The static baseline ignores skew and uses uniformly-spaced key pivots for partitioning. Quicksand need not sample, since its auto-sharding layer splits proclets on data insertion when they become too large. We measure the sorting throughput and shard size distribution as we scale the number of machines, for all setups.

In summary, this experiment shows that Quicksand can adapt to skewed data with little overhead by splitting compute proclets dynamically.

7.4 Adapting to skewness of input data

We examine the efficiency of Quicksand when it combines compute and memory on different servers. Intuitively, the more intense the computation, the more easily Quicksand amortizes the overhead of using remote resources. We thus study performance under different levels of compute intensity. We use the sorting application with 26 compute proclets that iterate through remote data in separate sharded vectors and perform computation on it. The total memory is ≈15 GiB. We run this microbenchmark on two imbalanced servers (20 cores with 1 GiB memory, and 6 cores with 15 GiB memory). We compare

7.5 Drill-down experiments

7.5.1 Impact of compute intensity. We examine the efficiency of Quicksand when it combines compute and memory on different servers. Intuitively, the more intense the computation, the more easily Quicksand amortizes the overhead of using remote resources. We thus study performance under different levels of compute intensity. We run a microbenchmark with 26 compute proclets that iterate through remote data in separate sharded vectors and perform computation on it. The total memory is ≈15 GiB. We run this microbenchmark on two imbalanced servers (20 cores with 1 GiB memory, and 6 cores with 15 GiB memory). We compare
We consider vector elements of size 100 B, 1 KB, and 10 KB; and vary the amount of computation per element from 0 μs to 50 μs. A good result would show Quicksand performing close to the baseline under high compute intensity.

Figure 15 shows the results. When the element size is 100 B, it only requires 2 μs of computation per element to achieve >99% efficiency. When element size is higher, more compute intensity is required to amortize the overheads; thus, it requires 40 μs and 200 μs for 1 KB and 10 KB elements respectively to achieve >99% efficiency. This suggests that not much compute intensity is needed to decouple compute and memory efficiently.

7.5.2 Comparison with Nu. To assess the cost of Quicksand’s new abstractions, we now evaluate Quicksand’s performance against a strong baseline, Nu [42], on a workload and environment without splitting, merging, or migrating resource proclets. This is a fair comparison as Quicksand and Nu share the same runtime (TCP, RPC, and threading). We again run the SocialNetwork benchmark, in Quicksand and Nu, but on a constant load, without resource imbalance, and without change in resource availability. We measure throughput as the number of machines increases. Ideally, Quicksand would achieve the same throughput as the Nu-based implementation in this setting.

Figure 16 shows the result. Quicksand matches Nu’s throughput and scalability on this workload. There is a constant ≈6% overhead in Quicksand, which comes from Quicksand’s auto-sharding infrastructure.

8 Conclusion
We presented Quicksand, a system that can improve resource utilization through resource proclets, a new programming abstraction that combines fast migration with the ability to automatically regulate proclet granularity and decouple the
consumption of resources of different types. Quicksand provides high-level abstractions built on top of resource proclets that are familiar to programmers while hiding the complexity of decomposition. Our results show that Quicksand responds quickly to load changes, reducing the need for overprovisioning, and takes advantage of machines with imbalances in compute and memory, recapturing their stranded resources.

We will open-source Quicksand.

References


