Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing

Zhenyuan Ruan Shihang Li [†] Kaiyan Fan Seo Jin Park [+] Marcos K. Aguilera [‡]
Adam Belay Malte Schwarzkopf [†]
MIT CSAIL [†]*Brown University* [‡]*VMware Research by Broadcom* [+]*USC*

Abstract. Datacenters today waste CPU and memory, as resources demanded by applications often fail to match the resources available on machines. This leads to stranded resources because one resource that runs out prevents placing additional applications that could consume the other resources. Unusable stranded resources result in reduced utilization of servers, and wasted money and energy.

Quicksand is a new framework and runtime system that unstrands resources by providing developers with familiar, high-level abstractions (*e.g.,* data structures, batch computing). Internally Quicksand decomposes them into *resource proclets*, granular units that each primarily consume resources of one type. Inspired by recent granular programming models, Quicksand decouples consumption of resources as much as possible. It splits, merges, and migrates resource proclets in milliseconds, so it can use resources on any machine, even if available only briefly.

Evaluation of our prototype with four applications shows that Quicksand uses stranded resources effectively; that Quicksand reacts to changing resource availability and demand within milliseconds, increasing utilization; and that porting applications to Quicksand requires moderate effort.

1 Introduction

It is rarely the case that the resources demanded by applications (*e.g.,* compute, memory, etc.) exactly match those available on machines in a datacenter. This mismatch wastes performance and capacity, because as one resource becomes a bottleneck, others are left idle, a notion referred to as *resource stranding* [8, 21, 23, 35, 40, 53]. In an ideal world, applications would use exactly the resources they need without being constrained by their physical availability on each machine.

One idea that could help solve this problem is memory disaggregation—the ability to allocate memory from a common pool that spans across machines. Unfortunately, memory disaggregation forces programmers to give up control of whether they are accessing local or remote memory, which sacrifices performance. This problem is severe enough that cloud providers currently limit their use of memory disaggregation to only the coldest memory pages [32].

Recent granular programming models, by contrast, give the programmer control over where resources are consumed. They decompose applications into granular *units* that bundle together portions of compute and memory, referred to as "proclets" [49], "components" [19], "actors" [39], and "functions" [29]. Each unit resides entirely on a specific machine, so it is guaranteed that state contained within it remains local. Units communicate with each other via the network using remote procedure calls (RPCs), so it is clear to programmers when a memory access could be remote. In some systems, units migrate across machines in response to resource pressure [49]. But this fails to address resource stranding, as the migratable units still couple different resources. Granular programming is also often hard to adopt, as programmers must manually decompose their programs into small units.

Quicksand is a new framework and runtime system that unstrands resources by providing a more performant alternative to disaggregation and offers familiar, high-level programming interfaces to application developers. Quicksand takes inspiration from the idea of granular programming, but crucially lets applications allocate resources of different types largely independently across a rack of machines. Quicksand unstrands resources through three key ideas.

First, Quicksand introduces *resource proclets*, a new abstraction that helps developers specify the use of different resource types in a fine-grained and independent manner. Resource proclets are categorized by the type of resource they primarily use and their functionality: a *compute proclet* handles threads of computation, while a *memory proclet* manages data structures and heap data. Resource proclets can be independently scheduled across machines and efficiently communicate with each other via a low-latency, high-bandwidth datacenter network (e.g., 400 GbE). Compute proclets can efficiently access data from remote memory proclets through prefetching, minimizing the performance impact of cross-machine data access. This allows applications to exploit stranded resources wherever they are available.

Second, Quicksand manages the size of resource proclets so they can be efficiently migrated through the network to rebalance resource use across machines. If a resource proclet becomes too large, it impedes fast migration; and if it becomes too small, communication and metadata overheads will 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 resource proclets, so the right number of proclets emerges dynamically.

Finally, Quicksand provides developers with familiar, high-level libraries while internally managing fine-grained resource allocation through resource proclets, freeing developers from

having to reason about low-level resource management. Developers use these libraries in the same way they use familiar data structures and compute abstractions, such as C++ STL containers and MapReduce-style parallel programming. The libraries internally decompose these high-level constructs into resource proclets. Consequently, Quicksand occupies a sweet spot: its libraries control how applications are decomposed and where memory will be accessed locally by leveraging semantic information. This not only hides the complexity of managing low-level granular primitives but also helps avoid unnecessary and costly remote memory accesses.

To illustrate the power of Quicksand, consider a machine learning pipeline, running on datacenter GPUs shared across applications—a common scenario today [28, 57, 61]. The pipeline preprocesses batches of raw images on CPUs in stages (resizing, noise reduction, etc.) before training a model on the GPUs. We want to keep GPUs fully utilized because they are expensive [31, 37, 38], use CPU and memory wherever available as needed, without having to overprovision them for pre-processing stages. Quicksand's resource proclets enable the pipeline to use stranded resources (*e.g.,* denoise images in memory on another server). To keep GPUs saturated, Quicksand splits and merges resource proclets to adapt its CPU and memory use to changing GPU availability.

Quicksand targets rack-scale environments where machines are typically connected with an overprovisioned network that offers high full-bisectional bandwidth. We prototyped Quicksand, and ported four applications to it: an image processing pipeline for machine learning, a latency-critical service from the DeathStarBench suite [17], a data-intensive sorting tool, and a serverless video encoding system from ExCamera [16]. Experiments show that Quicksand achieves 2–8× better performance than Hermit [45], a state-of-the-art memory disaggregation system, and 2–3× better throughput than Nu [49], a prior granular programming system. Additionally, Quicksand enables millisecond-scale fast reaction to changing load and resource availability by splitting and merging proclets, increasing utilization by 25%–60% compared to baselines that necessitate overprovisioning. Application developers enjoy these benefits with moderate programming effort.

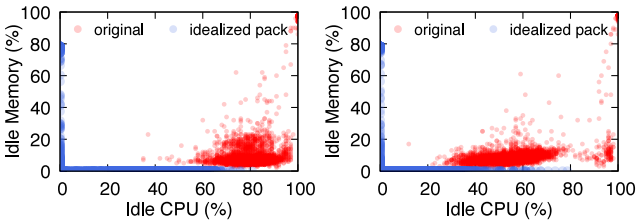
Our prototype has some limitations. It implements only compute and memory proclets, but we envision proclet support for other resource types (e.g., SSDs). Resource proclets are also not inherently fault-tolerant, although existing techniques for fault-tolerant granuluar computing apply [49].

2 Background and Motivation

2.1 Resource Stranding in Datacenters

Resource stranding occurs when hardware resources on machines are underutilized due to a mismatch between the resources demanded by applications and those available, resulting in wasted capacity [8, 21, 35, 40, 53].

We investigate resource stranding by analyzing traces from an Alibaba datacenter [23]. Alibaba colocates batch tasks with high-priority online services to increase utilization. The workload has a strong diurnal pattern [23], so we analyze a demand trough (at t = 500k seconds) and a demand peak (at t = 540k seconds). Figure 1 shows a scatterplot of idle CPU and memory, with each red point representing a specific machine. Even at peak load, machines have an average of 56% idle CPU and 10% idle memory. These idle resources exist because operators overprovision to handle uncertainty about the resources the online services will consume, the first source of stranded resources. This is especially critical for memory, as running out of it leads to crashes. Even though Alibaba already overcommits resources, this approach fails to utilize all available resources, as the level of idle resources is hard to predict and varies over time.



(a) At t = 500k seconds (load trough). **(b)** At t = 540k seconds (load peak).

Figure 1: Resource stranding in an Alibaba datacenter, shown by red points (*original*), arises from overprovisioned resources to accomodate load bursts of online services. Even with theoretically maximal binpacking of batch tasks based on perfect future resource predictions (blue points, *idealized pack*), stranding remains as batch task demands mismatch with available machine resources.

Resource stranding persists even in an idealized scenario without resource overprovisioning. To illustrate this, we simulate an ideal (but impractical) setting where Alibaba can perfectly predict resource needs and densely pack additional batch tasks to maximize utilization. This simulation iteratively fills each machine with batch tasks that have resource requirements and execution times equal to the averages of existing batch jobs in the trace. This continues until no more tasks can be added without exceeding the machine's CPU or memory capacity. This approach is unrealistic because it assumes perfect knowledge of future resource usage, including anticipating load spikes from online services.

Even in this hypothetical setting, some resources remain idle, as shown by the blue points (*idealized pack*) in Figure 1. For instance, under peak load (t = 540k), an average of 14% idle CPU and 3% idle memory persist on machines. These points, clustering along the axes, indicate machines with surplus of one resource type but scarcity of the other. This results from the mismatch between the CPU/memory demand ratios of batch tasks and the availability ratios of those resources on machines, another source of stranded resources.

Resource stranding is a widespread phenomenon: for instance, Microsoft reports up to 30% stranded memory in Azure [8], and Google's datacenters see wasted compute resources because best-effort jobs placed to fill spare compute

encounter out-of-memory events [54].

2.2 Memory Disaggregation

Memory disaggregation can alleviate resource stranding by exposing memory over the network, either from another server or a shared memory pool. Disaggregation can be supported at the kernel layer through paging (*e.g.,* Infiniswap [22], FastSwap [2], Hermit [45]) or at the bus layer through cache coherence (*e.g.,* GenZ [18], CXL [11, 34], LegoOS [51]). Bus-level solutions require new hardware that has not yet been deployed at the time of writing.

Memory disaggregation is appealing as it can be adopted without application changes. However, it faces two major limitations. First, while memory disaggregation facilitates pooling memory across machines, it does not extend this capability to CPUs. Hence, it fails to effectively utilize stranded CPU resources. Second, the transparency of memory disaggregation comes at a performance cost. Any memory access could be local or remote, and remote accesses may introduce delays that stall the CPU. This lack of awareness poses challenges for code optimization, especially problematic when compute intensity (*i.e.,* CPU cycles per byte of data access) is low. For example, Hermit [45] (a state-of-the-art memory disaggregation system) delivers only 9%–11% of the non-disaggregated throughput for a web service application (§7.1).

2.3 Granular Computing Systems

Recent cloud programming trends suggest decomposing applications into granular units, such as "actors" in Ray [39], "proclets" in Nu [49], and "components" in ServiceWeaver [19]. This evolution aligns with the increasing adoption of serverless and microservices paradigms in cloud development. Granular programming can utilize stranded CPU and memory by distributing small, granular units across machines. Unlike transparent memory disaggregation, granular programming explicitly defines unit boundaries; memory accessed within a unit is local, and memory can be accessed remotely only deliberately through remote procedure calls. Hence, developers can optimize their programs by structuring them to operate on local memory when compute intensity is low and exploit remote memory when it is high.

Among granular computing systems, Nu enables fast unit migration across machines to achieve *fungibility*—the ability to use resources interchangeably across machines [49]. However, Nu has major limitations: it still bundles different resources (such as CPU and memory) and fails to avoid resource stranding. Additionally, developing applications in Nu is hard, as developers must manually decompose the application into units and maintain their granularity. Developers may compose units with incorrect granularity, or create initially-granular units that become large over time (*e.g.,* due to memory growth). Improper sizing results in slow migration and imbalanced load, which impacts utilization and performance.

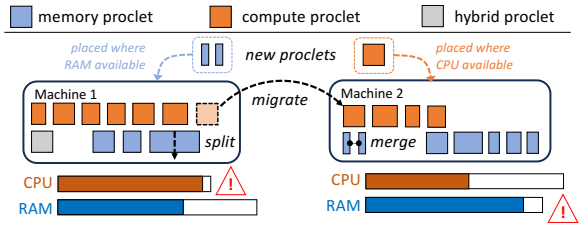


Figure 2: Quicksand places resource proclets where they are available: *e.g.,* memory proclets (blue) on machine 1 with available RAM; and compute proclets (orange) on machine 2 with spare CPU. Quicksand splits, merges, and migrates resource proclets as they grow, shrink, and as resource pressure develops.

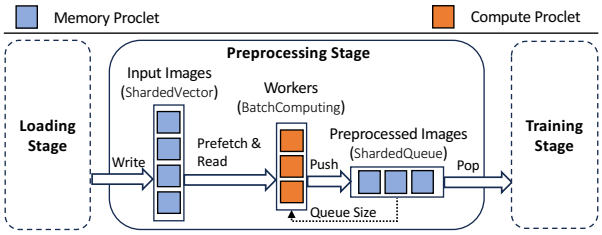
3 Quicksand Overview

Quicksand integrates the principles of resource disaggregation and granular computing in a new system. Quicksand reclaims stranded resources by decoupling resources and aligning allocation with actual use, while making it easy for application developers to use resources wherever they are available.

To dissect the use of different resources, Quicksand introduces *resource proclets*, granular abstractions categorized based on functionality. A compute proclet manages a set of computing threads, while a memory proclet manages megabytes of heap data. Both types of proclet can be dynamically split and merged to ensure a granular size. Each resource proclet functions as an independent unit, with Quicksand's runtime scheduling and migrating it based on application resource demands and availability. Resource proclets efficiently communicate with each other via the datacenter network (when remote) or a fast local data path (when local).

Quicksand places resource proclets on machines based on resource availability. Proclet sizes may change over time. For example, in Figure 2, the application is inserting data into its memory proclets on the left machine, and removing data from its memory proclets on the right machine. Quicksand reacts by dynamically splitting and merging resource proclets to maintain their granularity. In situations where a machine becomes oversubscribed, such as when the left machine's CPU capacity is nearly depleted, Quicksand migrates compute proclets to the right machine where additional CPUs are available. Similarly, Quicksand migrates memory proclets from machines nearing full to those with available space. Additionally, Quicksand supports hybrid proclets (as depicted on the left machine) for scenarios that require a combination of CPU and RAM. These hybrid proclets are also migratable, but resource proclets are easier for Quicksand to schedule because they only require a substantial amount of one resource.

To simplify the use of resource proclets, Quicksand offers developers familiar high-level programming abstractions, freeing them to focus on application logic without thinking about decomposing applications, maintaining granularity, or manually managing resources. Applications link with Quicksand's libraries that provide high-level abstractions: data structures, batch-parallel computations, and building blocks for stateless and stateful services.




```
1 void preprocessing_stage(qs::ShardedVector<Img>& inputs,
2                          qs::ShardedQueue<Img>& outputs) {
3  auto f = [outputs](Img i) { outputs.push(preprocess(i)); };
4  auto scaling_policy = qs::batch::MinimizeQueue(outputs);
5  qs::batch::ForAll(inputs, f, scaling_policy);
6 }
```

Figure 3: An example of an ML training pipeline illustrating the decomposition of the application into resource proclets using Quicksand's high-level abstractions. It highlights the preprocessing stage, where distributed workers process input images read (with prefetching) from a sharded vector and output to a sharded queue. Quicksand's batch computing framework dynamically scales the number of workers based on the output queue size to match the training stage's consumption rate.

Internally, these libraries provide their functionality by orchestrating resource proclets through a common auto-sharding layer, which dynamically splits and merges resource proclets to keep them granular. This layer is built on resource proclets to decouple state and compute, and leverages Quicksand's runtime to promptly respond to changes in resource demand and availability. Crucially, all of this resource management happens transparently to the application developer, who interacts only with the high-level abstractions.

4 Design

We now discuss how developers create Quicksand applications using its high-level abstractions (§4.1); describe the libraries in Quicksand that support these abstractions (§4.2); explain Quicksand's auto-sharding layer, which Quicksand libraries use to map high-level constructs into resource proclets (§4.3), and finally explain resource proclets in depth (§4.4).

4.1 Quicksand Applications

Application developers write a Quicksand program without worrying about how it is decomposed into resource proclets or how resources are managed. Quicksand handles resource allocation, scaling, and load balancing transparently, freeing developers from manual resource management. It facilitates this by offering three high-level abstractions with familiar APIs: sharded data structures, batch computing, and stateless and stateful services. These abstractions allow developers to focus on application logic while Quicksand's runtime efficiently manages the underlying resources.

The sharded data structures provide in-memory distributed data storage, while exposing APIs akin to C++ STL containers. For example, application developers use `emplace_back` to insert data into a sharded vector and use iterators to traverse the vector. The batch computing abstration enables distributed computing over sharded data structures, exposing declarative API functions such as `Map`, `ForAll`, `Reduce`, etc. The service abstraction offers wrappers for building elastic stateless and stateful services. Developers register a handler function for stateless services, or a class that implements application-specific stateful logic for stateful ones.

To showcase Quicksand's high-level abstractions, let's examine an ML training pipeline that runs on shared datacenter GPUs, which often experience changing availability [28, 57, 61]. As shown in Figure 3, the pipeline has three stages: loading, which loads input images from storage into memory for training; preprocessing, which preprocesses the loaded images; and training, where GPUs train a model on the preprocessed images.

Since input images may consume significant memory, the loading stage utilizes Quicksand's `ShardedVector` to efficiently distribute images across machines using sharding, where each shard is associated with a memory proclet. During preprocessing, the pipeline leverages Quicksand's batch computing abstraction, deploying compute proclets across machines to preprocess the images in parallel. Quicksand might schedule compute and memory proclets into different machines to combine their available resources, and it employs transparent prefetching of the vector to efficiently access remote memory. The pipeline then pushes the preprocessing outputs into a sharded queue for consumption by GPUs. This design might increase network traffic between stages, but it allows Quicksand to decouple compute and memory resources, resulting in better resource utilization and scalability.

If too few compute proclets are provisioned, the GPU can be starved for data, resulting in underutilization of expensive GPU resources. Conversely, provisioning too many compute proclets wastes CPU resources and increases memory usage, as the queue will grow if images are preprocessed faster than the GPU can consume them. To strike this balance, Quicksand dynamically scales the number of compute proclets in the preprocessing stage based on the queue length, aiming to keep the queue shallow while ensuring image production matches the GPU's consumption rate (details in §4.2).

Implementing the core logic of the preprocessing stage requires only a few lines of code (Figure 3). This code has no mention of resource proclets or sharding, the developer does not need to think about resource decomposition.

4.2 Quicksand Libraries

Quicksand offers high-level abstractions via a collection of libraries that applications link to. Each library is dedicated to a specific abstraction, mapping high-level constructs into granular resource proclets through its default policies and mechanisms for splitting and merging. The auto-sharding layer (§4.3) initiates the actual splitting and merging, calling upon these policies and mechanisms accordingly. These libraries abstract away the complexities of resource proclet management, allowing developers to focus on application logic rather than resource allocation and decomposition.

```
void emplace_back(T value)
T& at(size_t index)
size_t size()
```
```
void seal()
  - Seals the vector, making it immutable and iterate-able.
void unseal()
  - Unseals the vector, allowing modifications.
const_iterator cbegin(), cend()
  - Provides read-only iteration with prefetching.
```

Figure 4: Quicksand's sharded data structure API (e.g., a subset of the `ShardedVector` shown above) mirrors C++ STL interface, with additional methods to support efficient iteration with prefetching.

```
void ForAll(ShardedDS& input, F f, Policy p)
  - Applies f to all items of a sharded data structure in parallel.
```
```
ShardedDS Map(ShardedDS& input, F f, Policy p)
  - Transforms input items with f in parallel.
```
```
T Reduce(ShardedDS& input, F f, T init, Policy p)
  - Reduces input items with f and init in parallel.
```

Figure 5: A subset of Quicksand's batch computing API for parallel computation over sharded data structures, with support for custom scaling policies via an optional argument.

The sharded data structure library offers general-purpose data structures, including all common C++ STL container types, with an interface similar to the C++ STL. For instance, Figure 4 shows a subset of the `ShardedVector` API.

Leveraging Quicksand's auto-sharding layer, the library distributes data structures' contents across memory proclets based on a data structure-specific sharding key. For instance, the `ShardedVector` shards over vector indices, such that each memory proclet holds data for a contiguous range; a `ShardedMap` shards into ranges of the map's key space, with each memory proclet managing key-value pairs for a designated key range. The library supports iterators with prefetching for efficient traversal.

The memory proclets of a data structure enforce lower and upper bounds on their heap sizes. Upon element insertion, the auto-sharding layer checks if the shard becomes too large (or too small on element removal). In such cases, the library splits the memory proclet or merges it with a neighboring one.

The mechanisms to split and merge are data structure-specific. For example, the `ShardedQueue` in Figure 3 only supports insertion at the back (`push`), so only the last memory proclet can grow. Consequently, a split always occurs at the tail proclet, generating a new, empty memory proclet for subsequent insertions. By contrast, a sharded map is partitioned by key range and any of its proclets can grow. When a proclet goes above its size threshold, it splits its key range in half and selectively moves key-value pairs to the new proclet.

The batch computing library provides common parallel batch computing operators (*e.g.,* `ForAll`, `Map`, `Reduce`) over input and output stored in sharded data structures, see Figure 5. The computations are internally performed by compute proclets that split and merge via the auto-sharding layer to dynamically adjust parallelism based on available resources or user-specified policies.

```
StatelessServ Register(F handler)
  - Registers a stateless service.
StatefulServ<ID> Register<ID>(F handler, State& s)
  - Registers a stateful service with a client ID type and state.
void Invoke(StatelessServ& s, Args args)
  - Invokes a stateless service with arguments.
void Invoke(StatefulServ<ID>& s, ID id, Args args)
  - Invokes a stateful service with a client ID and arguments.
```

Figure 6: Quicksand's service library API for registering and invoking stateless and stateful services.

Recall the ML training example in Figure 3. The program uses the `ForAll` function to parallelize image preprocessing. Initially, it creates a single compute proclet, responsible for processing the entire range of the input `ShardedVector` by applying `f` to each element. When there are idle CPU cores in the cluster, the library splits the input range of the compute proclet, creating a new compute proclet to handle the other half. By iteratively splitting compute proclets, the library can utilize all available CPUs in the cluster. The batch computing library supports custom scaling policies, such as the `MinimizeQueue` policy in Figure 3. Every two milliseconds, it checks the queue length: if below a small threshold, it adds compute proclets. Otherwise, it adjusts the number of compute proclets by a proportional factor of the queue's shrinking rate, aligning production with consumption speed.

The completion time of compute proclets may vary, due to data skew or slow machines. Upon the completion of a fast compute proclet, the auto-sharding layer detects an idle core and reactively subdivides an ongoing proclet. This process effectively automates the mitigation of stragglers.

The service library facilitates the creation of elastic low-latency services, whether stateless or stateful, capable of dynamically responding to fluctuations in load. Figure 6 shows its API for application developers.

The stateless service uses compute proclets through the auto-sharding layer. Each invocation request from clients implicitly has a random 64-bit number, which Quicksand internally uses as the sharding key in the auto-sharding layer to evenly distribute the requests among compute proclets.

The stateful service uses pairs of compute and memory proclets managed by the auto-sharding layer. Each client request explicitly comes with a per-client or per-session identifier, which Quicksand uses as the sharding key to implement sticky routing through the auto-sharding layer.

Internally, both stateless and stateful services parallelize client interactions to scale out by dispatching requests to different compute proclets based on sharding keys.

Both service types adopt the same policy: Quicksand monitors the CPU utilization of each compute proclet at two-millisecond intervals. If the average usage exceeds one core, the library splits it into two, enabling services to scale up in response to load increase. Conversely, a service winding down from a load spike may have compute proclets that underutilize CPU resources; the library reactively merges proclets, adapting a service's resources to its current load. This two-

```
class VectorShard<T> : std::vector<T> {
  using Key = size_t; using Val = T;
  std::pair<Key, VectorShard<T>> Split() { /*...*/ }
  void Merge(VectorShard<T>& other) { /*...*/ }
};
using MyShardedVector<T> = qs::ds::ShardedDS<VectorShard<T>>;
```

Figure 7: A user can implement their own sharded vector by creating a wrapper (`VectorShard`) around `std::vector`, specifying the types of the sharding key and value, and implementing the split/merge functionality for the vector data.

millisecond interval strikes a balance between reactivity and stability. A shorter interval could lead to frequent, unnecessary adjustments due to transient load spikes, while a longer interval might delay responses to real workload changes. When splitting and merging a stateless service's proclets, the library splits or combines the key ranges they are responsible for. For a stateful service, the application developer provides `Split` and `Merge` functions over the service's internal state stored in proclets.

4.2.1 Extensibility

While Quicksand's libraries cover common use cases for cloud applications, users (*i.e.,* developers) can extend them for additional functionality. For instance, they could augment the sharded data structure library with new data structures, or the batch computing library with new operators (*e.g.,* flatmap, joins, or streaming-based transformations). The service library likely needs no extension as its stateless and stateful APIs cover all intended use cases, with applications supplying the business logic.

The interface of Quicksand libraries facilitates this extensibility. For example, a user can implement their own version of sharded vector in a few lines of code, as shown in Figure 7. The user wraps `std::vector`, adding type definitions for the sharding key and value, and provides implementations for `Split` and `Merge`, which split a vector shard's elements and combine two vectors' elements, respectively. Then, the user defines a new, application-facing type (`MyShardedVector`) by aliasing to the `ShardedDS` class provided by Quicksand's auto-sharding layer, parameterized with the shard implementation (`VectorShard`).

4.3 The Auto-Sharding Layer

Quicksand's libraries use the auto-sharding layer to shard high-level programming constructs into resource proclets and maintain their granularity. The auto-sharding layer accomplishes three tasks: (1) it keeps track of how sharding keys correspond to shards encapsulated within resource proclets, (2) it directs the execution of a given function to the appropriate shard, and (3) it splits or merges shards based on a user-supplied policy using the shard-specific split or merge functionality.

The auto-sharding layer is implemented by Quicksand's `AutoSharder` class, which requires a sharding key type, a proclet type to be sharded, and two policy functions `ShouldSplit` and `ShouldMerge`. These functions assess whether a shard's current state calls for splitting or merging. In turn, `AutoSharder` provides a `Route` method to direct a function invocation to the right shard, given a sharding key, a hint, and the function lambda. The hint indicates whether the shard might need merging or splitting.

Internally, `AutoSharder` keeps a map from sharding key ranges to resource proclets. This map is centrally stored in a mapping proclet, but each `AutoSharder` instance caches it to reduce load. The `Route` method checks the local map and dispatches the lambda to the appropriate shard. If the local cache is stale, the shard rejects the dispatch, in which case `Route` consults the central map and retries. After executing the lambda, the `Route` method uses the hint in combination with `ShouldSplit` or `ShouldMerge` to possibly split or merge the proclet shard using the shard-specific functions.

As an example, the sharded vector in Figure 7 can be built with `AutoSharder` using the vector index as the sharding key and memory proclet as the proclet type. The `ShouldSplit` and `ShouldMerge` functions check that the shard's heap size falls within a desired range. The sharded vector's `push_back` method routes execution of `std::vector`'s `push_back` to the last shard, while the `at` method routes execution to the appropriate shard based on the index key.

4.4 Resource Proclets

Quicksand defines three resource proclet types. The intent is that each resource proclet type primarily consumes one type of resource (compute, memory, etc). The actual definitions of resource proclets are based on their functionality: a compute proclet executes computations through a lambda API, while a memory proclet stores state through a read/write API. §7.4.1 shows that these abstractions in practice indeed meet the intent of using a single resource: compute proclets have a CPU-to-memory ratio greater than 0.1 core per MiB, so they mainly consume CPU; and memory proclets have a ratio of less than 0.001 core per MiB, so they mainly consume memory. Quicksand's runtime monitors the resource usage of resource proclets, issuing warnings if usage exceeds the thresholds. Quicksand also defines a hybrid proclet that provides both compute and memory functionality to support colocation of compute and memory when necessary for performance; it has a more balanced CPU-to-memory ratio (between 0.001–0.1 core per MiB). Appendix §A.2 shows the full API.

4.4.1 Compute Proclet

Compute proclets are conceptually akin to computing threads; developers use them to implement an application's computational logic. A compute proclet still consumes some memory to hold thread stacks and a small heap for temporary data. To create a compute proclet, Quicksand provides two functions: `MakeCompProc(Fn)` simply executes a given function, and `MakeCompProc(Range, Fn)` runs the function over a specified range of data. The range variant is useful for splitting (and merging similarly): upon splitting a compute proclet, the range is split into half, and the new compute proclet takes over the other half. 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 split, merge, and migrate during their execution, enabling Quicksand applications to utilize idle CPUs and react to compute pressure.

Compute proclets support both ephemeral and long-running computations. A compute proclet can create more compute proclets and wait for their completion. However, a compute proclet cannot directly access another compute proclet's memory. Instead, compute proclets communicate with memory proclets (via the network or the local data path) to share computation results, which we discuss next.

4.4.2 Memory Proclet

Memory proclets are conceptually akin to small heap regions (a few MBs); developers use them to keep application data. A memory proclet may consume a small amount of CPU to read and write data. Quicksand offers the `MakeMemProclet` function to create a memory proclet encapsulating a user-defined shard. This function returns a smart pointer to the memory proclet that exposes the `Read` and `Write` functions. Quicksand's libraries use these functions (and their asynchronous counterparts) to operate on the memory proclet's contents. As memory proclets can be remote, `Read` returns a copy of the indexed element.

Quicksand supports *sealing* memory proclets to make them read-only. Sealed memory proclets allow for efficient runtime prefetching, as there is no risk of reading stale data even with concurrent accesses. When a memory proclet is sealed and code invokes `Iterate` on it, the Quicksand runtime leverages the semantic information provided by the high-level abstractions to predict access patterns. It then prefetches subsequent elements in the background, effectively hiding access latency for remote data. Finally, memory proclet supports splitting and merging by invoking the `Split` and `Merge` functions of the encapsulating shard.

4.4.3 Hybrid Proclet

Hybrid proclets colocate a computing thread with heap data, providing compute-data locality for applications with low compute intensity. However, a hybrid proclet offers the least placement flexibility, as it may consume both nontrivial amounts of CPU and memory resources. Consequently, Quicksand discourages the use of hybrid proclets via constrained APIs. A hybrid proclet cannot be created directly; it can be realized only by promoting an existing memory proclet using an explicit `Hybridize` call. Once hybridized, it provides the `Offload` method to execute a C++ lambda within the hybrid proclet (computation shipping). A hybrid proclet can be demoted to a memory proclet by calling `Dehybridize`.

5 Implementation

Our Quicksand prototype has 10k lines of C++ code, which implement resource proclets atop Nu [49].

We co-designed the Quicksand runtime with resource proclets, leveraging information from the thread scheduler and slab memory allocator to precisely monitor per-proclet resource usage, with minimal overhead (<1%). Each machine's runtime periodically reports resource usage and idle resources to Quicksand's centralized controller, facilitating centralized decision-making on proclet placement and migration.

To safely and efficiently support resource proclet splitting and merging, we employ RCU locks [36]. Proclet access operations acquire the RCU reader lock, while splitting/merging events acquire the writer lock. This design minimizes the impact on application performance by shifting most synchronization overheads to infrequent splitting and merging events.

Initiation of splitting and merging, managed by the auto-sharding layer (§4.3), involves updating the sharding mapping stored in the centralized mapping proclet. Stale clients efficiently discover updates by fetching the latest mapping from an append-only log at the centralized proclet. This log sequentially records all changes made to the mapping.

We optimized our data structure library (§4.2) to minimize network overhead. For instance, our sharded data structures offer batching, grouping mutations (*e.g.,* `push_back` operations in `ShardedVector`) locally before flushing to memory proclets using large network packets. This significantly improves the performance of small-object operations. In addition, the library supports concurrent flushing, adapting parallelism to align with the application's production rate. Similarly, it supports concurrent prefetching for iterating over data structures.

6 Application Case Studies

We now describe our experience implementing and porting four applications for Quicksand. They cover batch and latency-critical applications, a range of compute intensities, memory usage, and resource proclet usage (Figure 8). Thanks to Quicksand's high-level abstractions, all applications required moderate programming effort, in terms of lines of code and development time; porting existing applications (social network and video encoding) involved modifying 3%–5% of the code and building new applications (ML training and sorting) required adding about 20 Quicksand-related lines of code. Figure 3 illustrates the decomposition of the ML training application, while Figure 9 presents the others. Sorting is the only application that uses hybrid proclets.

ML training pipeline. We implemented the compute-intensive batch application in Figure 3 as an OpenCV-based pipeline with a data loading stage, a CPU-based preprocessing stage, and a GPU-based training stage. We use images from the ImageNet ILSVRC2012 dataset [50] as our training data (35.3 GiB). Our implementation has 300 LoC, of which 20 LoC relate to Quicksand. We leverage Quicksand's `ShardedVector` to store input images, `ShardedQueue` for preprocessed images, and the batch computing library for parallel preprocessing. The batch computing abstraction dynamically scales workers based on the output queue size, matching the training stage's consumption rate. Since our testbed lacks GPUs, we simulate the training stage by consuming images at a rate of 0.2 milliseconds per image, based on the reported

Apps	Type	Dataset	Comp. Intensity	Mem. (GiB)	Abstractions Used	# resource proclets: comp.+mem.+hybrid	QS-related LoC/ Total LoC	Development Time (days)
ML Training	Batch	ILSVRC 2012 [50]	High	35.3	`Sharded{Queue, Vector}` `BatchComputing`	158 + 1,340 + 0	21 / 307	1
Social Network	Latency-critical	soc-delicious [13]	Low	33.4	`ShardedMap` `StatelessService`	91 + 19,034 + 0	98 / 1800	2
Sorting	Batch	TritonSort [46]	Low	112.0	`ShardedPartitions` `BatchComputing`	182 + 4,108 + 4,108	22 / 98	1
Video Encoding	Batch	bunny_480p [9]	High	17.5	`ShardedVector` `BatchComputing`	168 + 625 + 0	203 / 6083	4

Figure 8: Our four case study applications leverage (different) high-level abstractions provided by Quicksand, resulting in moderate porting and development efforts. Their memory usage and the number of resource proclets vary over time, with peak values reported.

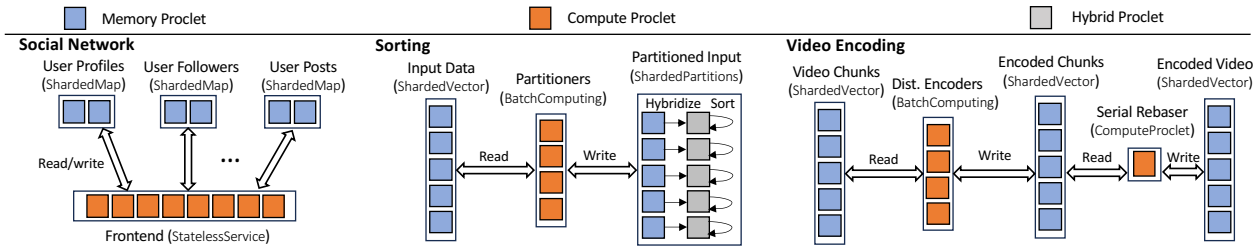


Figure 9: Quicksand's high-level abtractions automatically decompose applications into resource proclets and maintain their granular sizes.

rate of image classification models [14, 44].

Social Network. We ported the SocialNetwork application from DeathStarBench [17]. This is a latency-critical, Twitter-like web service, originally built using twelve microservices. It has a low compute intensity and uses 33.4 GiB memory. The original implementation employs an NGINX web frontend for processing client requests, which accesses user data stored in Redis. This architecture naturally separates CPU-intensive (frontend) and memory-intensive (data store) components, aligning well with Quicksand's model of resource decoupling. Consequently, it is straightforward to port this application to Quicksand, and gain benefits such as resource fungibility and elasticity that are not present in the original implementation.

Our port structures the application into a stateless front-end service layer and several stateful key-value stores, using Quicksand's `StatelessService` and `ShardedMap` abstractions. The stateless service wraps all the business logic originally handled by the microservices, while the sharded map stores user data originally handled by Redis. Our port has 1.8k LoC compared to the 6.8k LoC in the original application. This reduction is primarily due to the elimination of boilerplate code required for interacting with frameworks like gRPC, Redis, etc., which were used in the original implementation. In our port, 98 LoC are Quicksand-related, while the remaining code is application logic. By contrast, an already-granular implementation with Nu [49] (2.2k LoC) required developers to manually implement substantial functionality (such as decomposition, shard resolution, etc.) in the application code. Quicksand provides this functionality out-of-the-box and adds extra functionality (such as load-adaptive dynamic sharding) despite reducing the code size by 20%.

Sorting. We built an in-memory distributed sorting tool that sorts 400 million 100B key-value pairs, similar to TritonSort [46]. This is a batch application with low compute intensity and uses 112 GiB of memory. Our implementation has 307 LoC, of which 21 LoC are Quicksand-related. As depicted in Figure 9, our implementation uses Quicksand's batch computing library and custom data structures. It has two stages: *(i)* shuffling input data into key-based ranges; *(ii)* parallel sorting of these ranges. The first stage uses the batch computing library to concurrently read input data from a `ShardedVector` (with prefetching) and insert them into a `ShardedPartitions` data structure. This data structure uses the input as the sharding key and maintains an unordered vector of key-value pairs within each shard. Upon splitting, it uses the quickselect algorithm [25] to evenly partition these pairs into two ranges. After shuffling, sorting each partition (*i.e.,* the data of each shard) produces the final result. Given the low compute intensity of sorting, we improve locality by promoting memory proclets into hybrid proclets for fast in-place sorting, rather than sorting using compute proclets.

Video encoding. We ported the distributed video encoder from ExCamera [16], a batch application with high compute intensity designed for serverless platforms like AWS Lambda [6]. ExCamera has two stages: 1) a parallel encoding stage that encodes all video chunks, and 2) a sequential rebasing stage that stitches all encoded chunks together in sequence. Since ExCamera's code is no longer maintained or runnable, we created a faithful baseline implementation. Our port, as shown in Figure 9, utilizes Quicksand's `ShardedVector` and batch computing abstractions. It initially loads video chunks from the `bunny_480p` dataset (with a memory footprint of 17.5 GiB) into the input `ShardedVector`. It then uses the batch computing library to read chunks in parallel from the input vector, encodes them, and writes the outputs into another `ShardedVector` for encoded chunks. Finally, a single

compute proclet rebases all chunks from the encoded vector into the final encoded video. The port changes 203 LoC of the application's total 6k LoC.

7 Evaluation

Our evaluation focuses on answering following questions:

1. Does Quicksand unstrand resources better than existing disaggregation and granular programming systems (§7.1)?

2. How does Quicksand respond to changes in resource availability (§7.2) and resource demand (§7.3)?

3. Does the resource proclet abstraction effectively separate the use of different resources (§7.4.1)?

4. Does Quicksand's rapid scaling and fine granularity improve utilization and performance (§7.4.2 and §7.4.3)?

Setup. Our testbed has eight machines, each equipped with an Intel Xeon E5-2680 v4 CPU and 64 GB memory, connected by a 100 GbE network, running Linux kernel v5.10. One machine hosts Quicksand's cluster controller. For the ML pipeline experiments, we use one machine as the GPU server and the remaining six machines as the CPU servers (for preprocessing). For the social network experiments, one machine serves as the load generator and the remaining six machines as servers; and for the sorting and video encoding experiments, all seven machines run the application.

7.1 Using Stranded Resources

We first investigate how Quicksand performs on machines with stranded resources (*e.g.,* servers with ample free CPU but little memory or vice-versa). We use the ML training pipeline, which has high compute intensity, and the social network application, whose compute intensity is low. To comprehensively evaluate Quicksand's ability to handle various cases of stranded resources, we divide the same amount of total resources between machines in different ways (Figure 10). 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.

We compare Quicksand to two existing systems: Hermit [45], a state-of-the-art memory disaggregation framework, and Nu [49], a recent granular programming framework. Hermit is not designed to scale past the CPU resources of one machine (§2.2). To overcome this limitation, we give Hermit an advantage by running one Hermit instance per machine and evenly distributing tasks among them. This requires extra programming effort that counteracts Hermit's transparency promise. We compare the performance of all systems on the unbalanced setups to an ideal, balanced setup, where the aggregate resources are the same but distributed over only three machines, each with ample CPU and memory. Since Hermit is designed for memory disaggregation, we also evaluate it on a *memory-disaggregated* setup, which partitions the same amount of total resources into three CPU servers with minimal memory and three pure memory servers.

	Machine 1-3		Machine 4-6	
	# CPUs	RAM (GiB)	# CPUs	RAM (GiB)
Ideal	26	12	N/A	N/A
CPU-unbalanced	6	6	20	6
Mem-unbalanced	13	1	13	11
Both-unbalanced	6	11	20	1
Mem-disaggregated	26	1	0	11

Figure 10: To create resource stranding, we partition resources (78 CPUs, 36 GiB memory) in different ways: CPU-unbalanced, memory-unbalanced, and both-unbalanced. We also consider a memory-disaggregated setup and an ideal balanced setup.

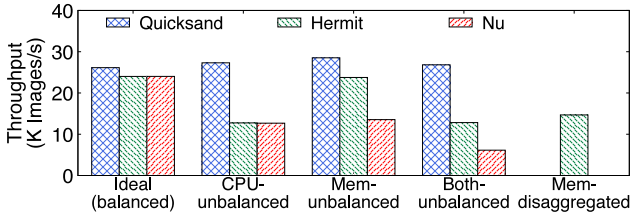


Figure 11: The ML pipeline on Quicksand combines resources from different machines, achieving throughput similar to the ideal scenario. Quicksand has 1.2–2× higher throughput than Hermit in unbalanced setups, as Hermit cannot use stranded CPU resources and suffers from kernel paging overheads when combining memory resources; and Quicksand's throughput exceeds Nu's by 2–4× in unbalanced setups as Nu always bundles CPU and memory resources.

A good result for Quicksand would show performance similar to Hermit and Nu on the ideal (balanced) setup, and better performance than these systems in the unbalanced setups. With high compute intensity, a strong result would show Quicksand coming close to the ideal setup's performance; while when compute intensity is low, resource proclet network's overheads dominate, and a good result would show limited performance loss compared to the ideal setup.

Figure 11 shows the results for the compute-intensive ML pipeline. In the ideal setup, each machine uses all 26 cores and ≈11 GiB memory, achieving throughput of 26k images/sec. In unbalanced setups, Quicksand achieves similar throughput despite the resource imbalance[1], thanks to its effective semantic-based data prefetching mechanism. Hermit and Nu achieve 92% of the throughput of Quicksand in the ideal setup—the overhead arises because they spawn a large number of compute threads, causing lower cache locality and extra context switches. By contrast, Quicksand splits and merges compute proclets to adapt to the available CPU resources. In the mem-unbalanced setup, Hermit combines stranded memory resources across machines through paging, but paging is expensive and limits Hermit to 83% of Quicksand's throughput. Hermit cannot combine stranded CPU resources, as its modified baseline evenly distributes tasks; Quicksand therefore achieves 2× higher throughput in the CPU-unbalanced

[1] In fact, Quicksand's throughput in unbalanced setups is up to 10% *above* the ideal setup; this is an artifact of lighter last-level cache contention.

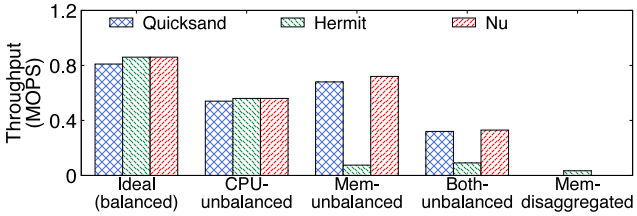


Figure 12: The social network application's low compute intensity poses a challenge in using stranded resources. Yet, Quicksand attains 40%–84% of the ideal throughput in severely imbalanced setups, outperforming Hermit, which achieves only 4%—11% due to poor locality from paging. Nu matches Quicksand's performance as it allows developers to control memory placement to optimize locality.

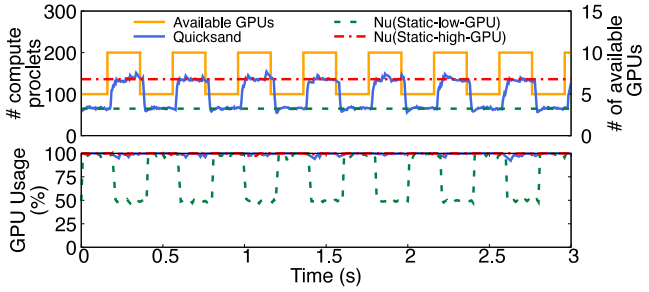
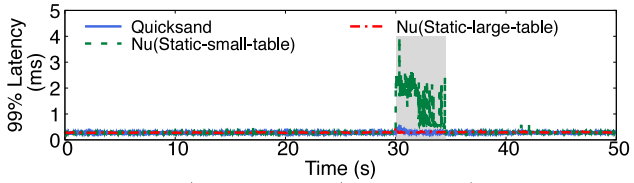


Figure 13: Quicksand quickly adapts the ML pipeline's preprocessing rate to the changing GPU availability by scaling the number of compute proclets. Compared to two Nu-based baselines that statically provision CPU resources, Quicksand achieves either 25% higher GPU utilization or uses 25% fewer CPU cores.

and both-unbalanced setups. Nu cannot combine stranded CPU and memory, as its proclets bundle different types of resources; Quicksand therefore outperforms Nu by 2–4×.

Figure 12 shows the results of the social network application with low compute intensity, where it is challenging to hide the overhead of accessing data on remote machines. In the ideal setup, each machine uses all 26 cores and ≈11 GiB memory, and all three systems achieve similar throughput (810k ops/sec). Quicksand's performance is slightly lower than the baselines as its auto-sharding layer adds a ≈6% overhead. All systems also achieve similar throughput in the CPU-unbalanced setup because the application has a low compute intensity. When there is memory imbalance (memory-unbalanced and both-unbalanced), it is crucial to optimize data locality to achieve good performance. Hermit's transparent paging prevents programmer control over memory placement and results in poor locality: Hermit achieves 9–11% of the ideal throughput in unbalanced setups (4–8× worse than Quicksand), and merely 4% in the memory-disaggregated setup. By contrast, Nu allows programmers to control placement to optimize locality and achieves 40–84% of the ideal throughput, similar to Quicksand.

These results demonstrate that Quicksand successfully uses stranded resources, and matches or outperforms state-of-the-art disaggregation and granular computing frameworks.



	Number of shards	Avg. shard size (MiB)	Avg. mem. usage (GiB)
Quicksand	1,086–19,034	1.7	17.25
Static-small-table	1,053	14.0	14.75
Static-large-table	19,928	1.4	27.5

Figure 14: In the social network application, Quicksand dynamically splits and merges memory procets of the hash table, ensuring granularity. This allows Quicksand to mitigate memory pressure (gray region) with rapid proclet migration, resulting in stable low tail latency. The Nu-based *static-small-table* baseline statically provisions a small hash table, resulting in larger shards that are slower to migrate, thereby experiencing disruptions. The *static-large-table* baseline, by contrast, statically overprovisions a large hash table, achieving low latency but using 37% more memory than Quicksand.

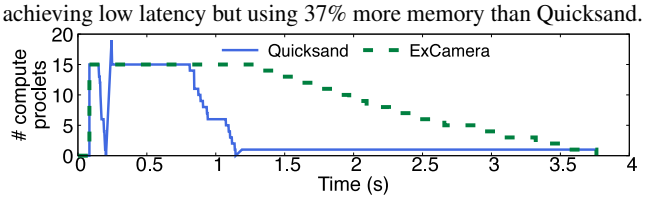


Figure 15: In the video encoding application, Quicksand adapts to the changing parallelism of the application by reducing the number of compute proclets, saving 60% CPU compared to ExCamera.

7.2 Adapting to Changing Resource Availability

We now study how well Quicksand reacts to changes in the availability of scarce resources.

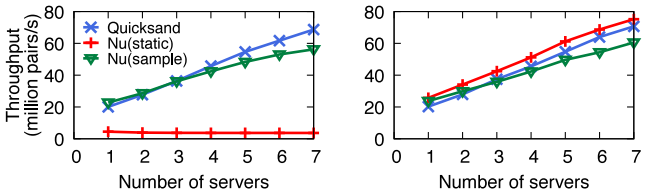
7.2.1 Reacting to Changing GPU Availability

In this experiment, we vary the number of GPUs available for the training stage of the ML pipeline between five and ten every 200ms, mimicking the burstiness reported by Park et al. [44]. We measure the amount of CPU cores allocated by Quicksand for the pre-processing stage. We compare Quicksand to two baselines that statically provision CPUs, implemented using Nu: *static-high-GPU* provisions CPUs for the peak number of available GPUs (ten), while *static-low-GPU* provisions for the minimum number (five). A good result for Quicksand would demonstrate it saturating available GPUs, comparable to *static-high-GPU*, without wasting CPUs.

Figure 13 shows that Quicksand rapidly scales compute proclets by splitting or merging them to match changing GPU availability. As a result, Quicksand achieves GPU saturation, while *static-low-GPU* achieves only 75% average GPU utilization. Quicksand consumes an average of 105 cores, but *static-high-GPU* consumes 140 cores. This shows that Quicksand effectively adapts to rapid changes in GPU availability.

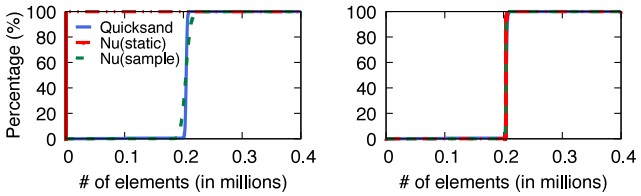
7.2.2 Reacting to Memory Pressure

We now investigate if Quicksand can quickly react to memory pressure to avoid performance disruption by splitting, merging, and migrating resource proclets across machines.



(a) Normal distribution. **(b)** Uniform distribution.

Figure 16: Sorting normally-distributed data, Quicksand achieves higher throughput than the *static* baseline because it avoid stragglers through dynamic splitting. For both normally and uniformly distributed data, Quicksand achieves higher throughput than the *sample* baseline because it avoids the extra sampling phase.



(a) Normal distribution. **(b)** Uniform distribution.

Figure 17: The CDF of shard sizes in the sort benchmark. Quicksand balances shard sizes well by dynamically splitting resource proclets. The *sample* baseline also balances shard sizes but requires an extra phase. The *static* baseline experiences serious load imbalance with a normal distribution of input due to its static partitioning.

We run the latency-critical social network application on all available machines with constant load (1.5 MOPS), and launch a memory antagonist that keeps allocating new pages at one machine, creating memory pressure. The memory usage in the social network application mainly comes from its sharded hash table, so the experiment focuses on this data structure. We compare Quicksand to two baseline systems that statically provision the number of shards, implemented using Nu: *static-small-table* provisions for a small hash table with a few shards; while *static-large-table* baseline overprovisions for a large hash table with many shards. A good result for Quicksand would show it achieving constant, low latency despite memory pressure and consuming less memory than the *static-large-table* baseline.

Figure 14 shows the results. Between 0–30 seconds, there is no memory pressure, so all setups achieve low and stable 99th percentile latency (≈0.3 ms). The memory antagonist starts at t=30 seconds and all setups react by migrating shards during the gray region. Both Quicksand and *static-large-table* experience only a small latency increase as they can quickly migrate their small (<2 MiB) shards. However, Quicksand uses 37% less average memory than *static-large-table*, as it rightsizes the number of shards through dynamic splitting and merging. *Static-small-table*, by contrast, ends up with a few large shards (≈14 MiB) that are slow to migrate, and therefore experiences latency spikes. After resolving memory pressure through migration, the latency returns to normal. These results show that Quicksand effectively mitigates memory pressure while keeping good tail latency and memory usage.

7.3 Adapting to Changing Resource Demand

We now study if Quicksand adapts to changing resource demand caused by phased program behavior or load imbalance.

7.3.1 Phased Application Behavior

The video encoding application has multiple concurrent stages that encode chunks, followed by a serial stage that stitches together encoded chunks. We run the ExCamera[6,16] encoding scheme[2] and compare our performance to the baseline. A good result for Quicksand would show its resource usage matching the application's available parallelism.

Figure 15 shows the results. The concurrent encoding stages in the beginning (0–0.8s) have 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 Quicksand scales back up, it briefly overshoots the right level of parallelism and scales back down to 15 compute proclets. After 0.8s, the parallelism gradually decreases as encoding tasks complete. In Quicksand, compute proclets write their outputs to a sharded vector built from memory proclets; once they finish encoding, they terminate. 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 implementation couples compute and state; the encoding lambdas must remain alive until the stitching lamdba consumes their output, leading to many allocated but idle CPUs between 1.2–3.7s. Even though Quicksand and ExCamera complete the job at the same time, Quicksand saves 60% CPU resources, represented by the area between the two lines.

This result demonstrates that Quicksand can quickly adapt to phased application behavior to improve resource utilization without impacting performance.

7.3.2 Load Imbalance

We study Quicksand's ability to adapt to load imbalance, which causes poor utilization and stragglers [10]. We use the sorting application with an input that follows either a skewed normal distribution or a uniform distribution without skew. We compare Quicksand with two baselines implemented using Nu. The *sample* baseline handles the skewed input by running an initial phase that samples 0.5% of the input to learn its distribution and choose appropriate pivots to partition the data across machines. The *static* baseline ignores skew and partitions by uniformly spaced pivots. Quicksand avoids the need for sampling, as its auto-sharding dynamically splits memory proclets when they become too large. We measure the sorting throughput and shard size distribution as we scale the number of machines. A good result for Quicksand would show that it adapts to skew and achieves balanced partitions without the cost of an additional sampling phase.

Figure 16a shows the sorting throughput for the normal

[2] In ExCamera's notation, this means encoding chunks of 6 frames in parallel, and then combining them in strings of 16 chunks [16].

Apps	Compute proclet			Memory proclet		
	CPU (cores)	Mem (MB)	Ratio (core/MB)	CPU (cores)	Mem (MB)	Ratio (core/MB)
ML training	1.0	0.7	1.43	0.0038	31	0.00012
Social network	0.58	5.4	0.11	0.00044	1.3	0.00034
Sorting	0.34	1.3	0.26	0.022	32	0.00068
Video encoding	0.75	4.1	0.18	0.0016	12	0.00013

Figure 18: The CPU and memory usage for compute and memory proclets, averaged over time and across all proclets within each category, along with their CPU-to-memory ratios. Hybrid proclets, used only in the sorting application, have an average use of 0.28 cores and 32 MB memory, resulting in a CPU/memory ratio of 0.0087.

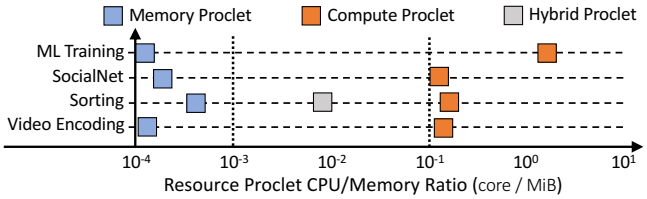


Figure 19: Different types of resource proclets vary significantly in CPU/memory ratio (*x*-axis, in log-scale). Compute proclets range from 0.1 to 10 in ratio, hybrid proclets range from 0.001 to 0.1, and memory proclets range from 0.0001 to 0.001.

distribution. *Static* performs and scales poorly due to stragglers caused by load imbalance, while *sample* and Quicksand perform and scale well. Figure 17a explains the load imbalance of *static*: a few partitions have most of the data, while others have little data. *Sample* and Quicksand have balanced partitions of 200K elements and thus balance load well across machines. Quicksand performs better than *sample* as it avoids the need for an extra sampling phase, saving 1.4 seconds.

Figures 16b and 17b show the results for a uniform distribution. Here, *static* is optimal because it partitions the input well without sampling work. Quicksand is slightly slower because splitting resource proclets incurs overhead. *Sample* is the slowest due to the extra sampling phase.

These results show that Quicksand efficiently adapts to load imbalance by splitting resource proclets dynamically.

7.4 Drill-Down Experiments

7.4.1 The Effectiveness of Resource Proclets

We now investigate the effectiveness of resource proclets in separating the usage of different types of resources. Figure 18 presents the resource consumption for each application's resource proclets. For each type of resource proclet, it shows the CPU and memory usage (averaged over time and across all proclets within that category) along with the CPU/memory ratio (*i.e.,* average CPU usage to average memory usage).

Different types of resource proclets exhibit drastically different CPU/memory ratios, as shown in Figure 19. Compute proclets primarily utilize CPU resources with small memory usage, exhibiting CPU/memory ratios between 0.1 and 10;

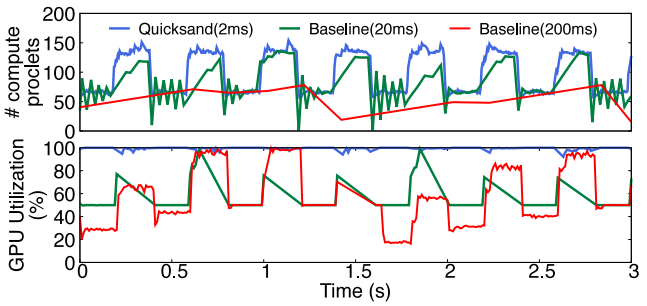


Figure 20: Quicksand's rapid scaling (2 ms) is crucial for saturating GPUs in the ML training pipeline. Slowing down reaction time to 20 ms or 200 ms causes GPU utilization to drop as it takes too long to provision enough CPUs to keep the GPUs busy.

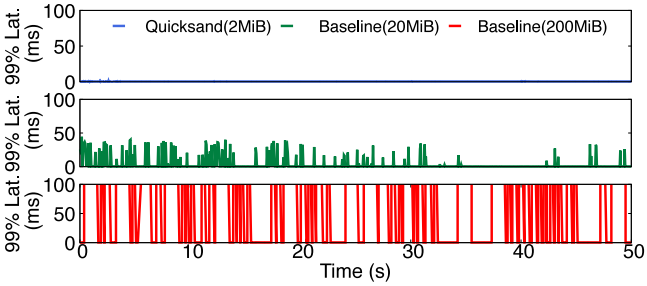


Figure 21: Fine granularity of memory proclets in Quicksand (2 MiB) is crucial for low tail latency in the social network application. Larger memory proclets (20 MiB or 200 MiB) causes intermittent latency disruption due to the impact of splitting larger proclets.

this indicates a significant bias towards CPU in its resource usage. With compute proclets, Quicksand is able to unstrand 1 CPU core in exchange for consuming 2 MiB of memory.

Memory proclets, by constrast, predominately utilize memory resources with low CPU consumption, exhibiting CPU/memory ratio ranging from 0.0001 to 0.001—four to five orders of magnitude lower than compute proclets; this indicates a significant bias towards memory in its resource usage. With memory proclets, Quicksand is able to unstrand 1 GiB memory with an average CPU consumption of 0.3 cores.

Hybrid proclets fall between memory proclets and compute proclets, with CPU/memory ratio ranging from 0.001 to 0.1.

This shows that compute proclets and memory proclets effectively separate the use of CPU and memory resources.

7.4.2 Benefit of Rapid Scaling

We now investigate the importance of Quicksand's rapid scaling to react to changes in resource availability. We use the ML pipeline and the same setup as in §7.2.1, but slow down Quicksand's reaction time from 2 ms to 20 ms or 200 ms to measure the impact of fast scaling on utilization.

Figure 20 shows the result. Unlike the original Quicksand, baselines with slower reaction time fail to saturate GPUs as they scale the number of compute proclets too slowly to match the changing GPU availability. This shows Quicksand's rapid scaling is crucial for harnessing changing resources.

7.4.3 Benefit of Fine Granularity

Finally, we examine the necessity to keep resource proclets fine-grained. We run the social network application on all available machines with constant load (1.5 MOPS). For latency-critical applications, by default Quicksand splits memory proclets above a 2 MiB threshold. Larger proclets impact tail latency in two ways: first, splitting them takes longer, causing a longer stall; second, larger proclets imply fewer proclets, thus stalling a proclet impacts a larger proportion of requests. To quantify the impact, we compare the original Quicksand to modified versions that use 20 MiB and 200 MiB as the split threshold.

Figure 21 shows the result: original Quicksand has a stable 99th percentile latency, but coarser-grained proclets suffer latency disruptions during splitting. These results show that Quicksand's fine granularity is crucial for tail latency.

8 Discussion

8.1 Applicability and Limitations of Quicksand

Quicksand is ideal for cloud applications that benefit from flexible resource allocation and fine-grained scaling. However, some traditional single-machine applications, like high-performance computing simulations that depend on low-latency memory access, may be difficult to adapt without performance loss. Similarly, applications optimized for specific hardware, such as those requiring direct access to accelerators or GPUs, may face challenges. Legacy systems with monolithic architectures may also need significant refactoring to fully utilize Quicksand.

8.2 Relation between Quicksand and Disaggregation

Unlike recent hardware-based resource disaggregation systems [5, 26, 34, 51, 55, 59], which rely on specialized interconnects like CXL [11] or GenZ [18], Quicksand enables practical resource disaggregation on commodity hardware through its software-based approach. By decomposing applications into independent, granular resource proclets, Quicksand disaggregates compute and memory, unlike pure memory disaggregation systems [2, 22, 45]. Quicksand's abstractions simultaneously give library developers more control over data locality than transparent memory disaggregation. While hardware-based solutions may ultimately offer lower latency, Quicksand represents a practical intermediate step that delivers many disaggregation benefits on today's commercial datacenter hardware.

9 Related Work

Data Analytics Frameworks. Distributed frameworks like Spark [58], MapReduce [12], and Dryad [27] offer high-level APIs (e.g., DataFrames) to automate task decomposition for data analytics, but they are ill-suited for other domains like sharded services. While providing some resource management, they operate at coarser granularities and timescales, coupling resources and lacking the flexibility to respond dynamically to resource fluctuations. As a result, they cannot leverage stranded resources as effectively as Quicksand, which uses fine-grained proclets and millisecond-scale adaptability.

Autoscaling Frameworks. Slicer [1] and Shard Manager [33] help developers add sharding to applications, and add or remove shards in response to load. However, they target long-running, large-scale systems with big shards and minute-scale reaction times. Instead, Quicksand targets millisecond-scale reaction times to unstrand short-lived resources.

Resource Decoupling. Resource decoupling, as demonstrated by Monotasks [42, 43], involves breaking down jobs into single-resource tasks, akin to resource proclets. However, Monotasks primarily aim to facilitate performance analysis of Spark jobs, which leads to distinct design choices and implications: Monotasks cannot combine stranded resources across machine and operates at a coarser time granularity.

VM Consolidation. VM consolidation is a classic method for enhancing resource utilization by packing VMs onto physical machines [24, 30, 54]. However, it is inefficient due to resource overprovisioning and fragmentation [49]. StopGap [41] introduces VM splitting and fusion to improve consolidation, but it is restricted to stateless services (relying on reconfiguring the load-balancer to support splitting and fusion), and second-scale reaction times (as VMs are heavyweight).

Resource Harvesting. Resource harvesting aims to improve resource utilization by using idle resources that may be available only intermittently, using evictable VMs [3, 7, 20] and subsequent improvements [4, 47, 52, 56, 60]. 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.

10 Conclusion

We presented Quicksand, a new system to unstrand datacenter resources. Quicksand introduces resource proclets, which decouple the consumption of resources of different types, automatically regulate resource granularity, and support fast migration. Quicksand offers familar high-level abstractions to programmers while hiding the complexity of resource decomposition into resource proclets in extensible libraries.

Our results show that Quicksand successfully unstrands resources other systems cannot utilize, while also promptly responding to changes in resource availability and demand. Quicksand is available as open-source software [48].

Acknowledgements

We thank our shepherd Kay Ousterhout; the anonymous reviewers; and members of the PDOS group at MIT and the ETOS group at Brown University for their helpful feedback. We appreciate CloudLab [15] for providing the experiment platform. This work was funded in part by a Facebook Research Award; Google Faculty Awards; the DARPA FastNICs program under contract #HR0011-20-C-0089; the NSF under awards CNS-2104398 and CNS-2045170; VMware; and by ACE, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. "Slicer: Auto-Sharding for Datacenter Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. "Can Far Memory Improve Job Throughput?" In: *European Conference on Computer Systems (EuroSys)*. 2020.

[3] *Amazon EC2 Spot Instances*. URL: https://aws.amazon.com/ec2/spot (visited on 08/20/2024).

[4] Pradeep Ambati, Iñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[5] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. "Programming Fully Disaggregated Systems". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2023.

[6] *AWS Lambda*. URL: https://aws.amazon.com/lambda/ (visited on 08/20/2024).

[7] *Azure Spot VM*. URL: https://azure.microsoft.com/en-us/services/virtual-machines/spot (visited on 08/20/2024).

[8] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. "Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms". In: *IEEE Micro* (2023).

[9] *Big Buck Bunny 480P Video*. URL: https://media.xiph.org/video/derf/y4m/big_buck_bunny_480p24.y4m.xz (visited on 08/20/2024).

[10] Emilio Coppa and Irene Finocchi. "On Data Skewness, Stragglers, and MapReduce Progress Indicators". In: *ACM Symposium on Cloud Computing (SoCC)*. 2015.

[11] *CXL Consortium*. URL: https://www.computeexpresslink.org (visited on 08/20/2024).

[12] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[13] *Delicious online social network dataset*. URL: https://networkrepository.com/soc_delicious.php (visited on 08/20/2024).

[14] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. "IOS: Inter-Operator Scheduler for CNN Acceleration". In: *CoRR* (2020). arXiv: 2011.01302.

[15] 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: *USENIX Annual Technical Conference (ATC)*. 2019.

[16] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing using Thousands of Tiny Threads". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[18] *Gen-Z Consortium*. URL: https://en.wikipedia.org/wiki/Gen-Z_(consortium) (visited on 08/20/2024).

[19] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. "Towards Modern Development of Cloud Applications". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2023.

[20] *Google Cloud Preemptible VM Instances*. URL: https://cloud.google.com/compute/docs/instances/preemptible (visited on 08/20/2024).

[21] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. "The cost of a cloud: research problems in data center networks". In: *ACM SIGCOMM Computer Communication Review* (2009).

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[23] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces". In: *IEEE/ACM International Symposium on Quality of Service (IWQoS)*. 2019.

[24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[25] C. A. R. Hoare. "Algorithm 65: Find". In: *Communications of the ACM* (1961).

[26] *Intel Rack Scale Design Architecture*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf (visited on 08/20/2024).

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: European Conference on Computer Systems (EuroSys). 2007.

[28] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[29] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *CoRR* (2019). arXiv: 1902.03383.

[30] *Kubernetes*. URL: https://kubernetes.io (visited on 09/20/2022).

[31] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. "Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines". In: *Proceedings of Machine Learning and Systems* (2022).

[32] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. "Software-Defined Far Memory in Warehouse-Scale Computers". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[33] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. "Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2021.

[34] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2023.

[35] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. "Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud". In: *Proceedings of the VLDB Endowment (PVLDB)* (2023).

[36] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. "Read-copy update". In: *AUUG Conference Proceedings*. AUUG, Inc. 2001.

[37] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. "Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2022.

[38] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. "Analyzing and Mitigating Data Stalls in DNN Training". In: *Proceedings of the VLDB Endowment (PVLDB)* (2021).

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[40] Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. “TMC: Near-Optimal Resource Allocation for Tiered-Memory Systems”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2023.

[41] Vlad Nitu, Boris Teabe, Leon Fopa, Alain Tchana, and Daniel Hagimont. “StopGap: elastic VMs to enhance server consolidation”. In: *Proceedings of the Symposium on Applied Computing*. 2017.

[42] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. “Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks”. In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[43] Kay Ousterhout, Christopher Canel, Max Wolffe, Sylvia Ratnasamy, and Scott Shenker. “Performance Clarity as a First-Class Design Principle”. In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2017.

[44] Seo Jin Park, Joshua Fried, Sunghyun Kim, Mohammad Alizadeh, and Adam Belay. “Efficient Strong Scaling Through Burst Parallel Training”. In: *Proceedings of Machine Learning and Systems*. Edited by D. Marculescu, Y. Chi, and C. Wu. 2022.

[45] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. “Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2023.

[46] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. “TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System”. In: *ACM Transactions on Computer Systems (TOCS)* (2013).

[47] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi A. Noghabi, and Jian Huang. “BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2022.

[48] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. *Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing*. URL: https://github.com/NSDI25-Quicksand/Quicksand (visited on 10/15/2024).

[49] Zhenyuan Ruan, Seo Jin Park, Marcos Aguilera, Adam Belay, and Malte Schwarzkopf. “Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2023.

[50] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* (2015).

[51] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[52] Supreeth Shastri and David Irwin. “HotSpot: automated server hopping in cloud spot markets”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[53] M. Taubenblatt and A. Tantawi. “Quantifying Composable Data Center Utilization”. In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022.

[54] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. “Borg: The next Generation”. In: *European Conference on Computer Systems (EuroSys)*. 2020.

[55] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang, and Qinfen Hao. “CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers”. In: *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023.

[56] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. “SmartHarvest: Harvesting idle CPUs safely and efficiently in the cloud”. In: *European Conference on Computer Systems (EuroSys)*. 2021.

[57] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. “Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent”. In: *USENIX Annual Technical Conference (ATC)*. 2023.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[59] Yiying Zhang. “Make It Real: An End-to-End Implementation of A Physically Disaggregated Data Center”. In: *ACM SIGOPS Operating Systems Review* (2023).

[60] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. “History-based harvesting of spare cycles and storage in large-scale datacenters”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[61] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. “HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

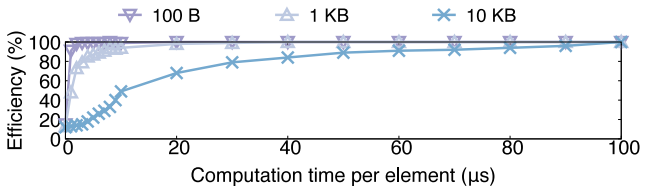


Figure 22: Quicksand's efficiency (y-axis) when compute proclets operate on memory proclets at different servers, as a function of compute intensity, normalized to the performance of local computation in a single machine with the same total resources. Quicksand's efficiency increases with compute intensity, requiring 3 μs of computation per 100B element to reach 99% efficiency.

A Appendix

A.1 Impact of Application Compute Intensity

We study the efficiency of Quicksand when its resource proclets separate computation from data across servers, *i.e.,* when a compute proclet operates on remote memory proclet. While prefetching can mitigate some remote access overhead, it introduces its own costs and becomes less effective as compute intensity decreases. Intuitively, the more CPU cycles spent per memory proclet access, the better Quicksand should amortize the overheads of remote access. We thus conduct a sensitivity analysis of performance under different compute intensities.

Our microbenchmark uses 26 compute proclets that iterate through remote data in separate sharded vectors and compute 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 quantify efficiency with respect to a baseline that computes on local memory with the same total resources (26 cores, 16 GiB memory). 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 100 μs. A good result would show Quicksand achieving near-ideal efficiency at a broad range of compute intensities.

Figure 22 shows the results. When the element size is 100 B, Quicksand requires 3 μs of computation per element to reach 99% efficiency. For larger elements, higher compute intensity is required to amortize the communication cost: it requires 30 μs and 100 μs for 1 KB and 10 KB elements respectively to reach 99% efficiency.

These results suggest that Quicksand can decouple compute and memory efficiently at moderate compute intensities (microseconds of computation per remote access).

A.2 Resource Proclet API

Type	API	Description
Compute Proclet	`CompProc<Fn> MakeCompProc(Fn)`	Makes a compute proclet that runs `Fn`.
	`CompProc<RangedFn> MakeCompProc(Range, Fn)`	Makes a compute proclet that runs `Fn` over a `Range` of data.
	`CompProc<RangedFn> Split()`	Splits the current compute proclet.
	`void Merge(CompProc<RangedFn>&)`	Merges a compute proclet.
Memory Proclet	`MemProc<Shard> MakeMemProc<Shard>(Args)`	Makes a memory proclet that hosts a `Shard` object
	`Shard::Val Read(Fn)`	Reads data via running `Fn`.
	`void Write(Fn, Shard::Val)`	Writes data via running `Fn`.
	`void Seal()`	Makes the memory proclet read-only; enables prefetching.
	`void Unseal()`	Makes the read-only memory proclet writeable; disables prefetching.
	`Iterable<Shard> Iterator()`	Returns iterator that iterates over shard contents in an impl-defined order.
	`HybridProc<Shard> Hybridize()`	Promotes the memory proclet into a hybrid proclet.
	`MemProc<Shard> Split()`	Splits the current memory proclet.
	`void Merge(MemProc<Shard>&)`	Merges a memory proclet.
Hybrid Proclet	`std::invoke_result_t<Fn> Offload(Fn)`	Ships the computation `Fn` to the hybrid proclet.
	`MemProc<Shard> Dehybridize()`	Demotes the hybrid proclet back into a memory proclet.
	`HybridProc<Shard> Split()`	Splits the current hybrid proclet.
	`void Merge(HybridProc<Shard>&)`	Merges a hybrid proclet.

Figure 23: Quicksand's compute proclet API enables the execution of functions (optionally over a specified data range), while the memory proclet API enables reads, writes, and iterations. When sealed, memory proclets permit safe prefetching. A memory proclet can be promoted into a hybrid proclet, allowing for computation offloading. Compute proclets with ranges, memory proclets, and hybrid proclets support splitting and merging.