Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes

Zhenyuan Ruan Seo Jin Park Marcos K. Aguilera [‡] Adam Belay Malte Schwarzkopf [†]
MIT CSAIL [‡]*VMware Research* [†]*Brown University*

Abstract. Data centers waste significant compute and memory resources today because they lack resource *fungibility*: the ability to reassign resources quickly and without disruption. We propose *logical processes*, a new abstraction that splits the classic UNIX process into units of state called *proclets*. Proclets can be migrated quickly within data center racks, to provide fungibility and adapt to the memory and compute resource needs of the moment. We prototype logical processes in Nu, and use it to build three different applications: a social network application, a MapReduce system, and a scalable key-value store. We evaluate Nu with 32 servers. Our evaluation shows that Nu achieves high efficiency and fungibility: it migrates proclets in $\approx 100\mu s$; under intense resource pressure, migration causes small disruptions to tail latency—the 99.9th percentile remains below or around 1ms—for a duration of 0.54–2.1s, or a modest disruption to throughput (<6%) for a duration of 24–37ms, depending on the application.

1 Introduction

Compute and memory are valuable and expensive resources in data centers today, but they are inefficiently utilized [46,75]. A key reason for this inefficiency is a lack of *fungibility*—the ability to reassign resources quickly and without disruption between different users and across different machines. Without fungibility, resources are stranded and over-provisioned for fear of running short, even as resource consumption naturally fluctuates in data center applications [2, 7, 18, 34, 39].

Existing systems fail to provide fungibility because current abstractions for compute work and memory state (VMs, containers, processes) are too coarse-grained (§2). To address this problem, we introduce the abstraction of a *logical process*. Logical processes provide fungibility, while retaining a familiar programming model similar to traditional processes. A logical process consists of many smaller *proclets*, atomic units of state and compute that can be independently migrated under resource pressure to achieve fungibility. Like a traditional process, a logical process has its own address space, isolated from other processes. But unlike a traditional process, a logical process can spread across many machines in datacenter racks as a result of the migration of its proclets. Intuitively, logical processes break down the monolithic nature of processes into many proclets. A proclet consists of a heap (state) and a set of user-level threads and their execution contexts (stacks and register values). A logical process runtime system responds to spikes in load by migrating proclets quickly to a machine with spare resources.

To realize logical processes and proclets, we had to address three challenges. First, migration speed must be fast and react to resource pressure before resources are exhausted. Second, communication between proclets and proclet migration must impose little overhead or disruption on the application, especially if migration itself consumes resources when resources are short. Third, the programming model of logical processes and proclets must support practical data center applications.

We responded to these challenges as follows. First, we divided process state into proclets, which are small relative to an entire process, so they can be migrated orders of magnitude faster than VMs or processes. Second, we optimized our software stack to take full advantage of modern datacenter networks (at 100–400 Gbit/s). This pushed performance far enough for proclets to migrate in $\approx 100\mu s$. We also scaled proclets across machines with minimal communication overheads by using a single program image across machines and an optimized RPC stack. Third, we used a global address space to provide a programming model that is process-like and intuitive, making it possible to statically check types, and enabling computation shipping by passing function pointers between proclets.

We prototyped logical processes and proclets in Nu, a system that provides a C++ class API and a Caladan-based [28] user-level threading and kernel-bypass networking runtime. Nu targets the *pod-scale* environment with tens of racks: hundreds of machines connected with overprovisioned network that provides high full-bisection bandwidth (100–400 Gbit/s) and low latency (10–20 μs). We implemented three applications using Nu. The first is a version of the DeathStarBench social network application [29], originally implemented using microservices. The Nu version of this application is simpler, shorter, and has an order of magnitude better performance than the microservice version, while preserving scalability. The second application is k-means clustering on Phoenix MapReduce [63], representing a compute-intensive workload well-suited for parallelism. Phoenix MR exploits thread parallelism in NUMA machines, but the Nu version scales across multiple machines, yet also delivers single-machine performance close to the original implementation. The third application is a scalable key-value store implemented in Nu as a hash table whose buckets are distributed across multiple proclets.

We evaluate Nu in a setup of 32 servers with 100 GbE NICs that are connected through a ToR switch. Our evaluation shows that Nu achieves high efficiency and fungibility: it reacts quickly to resource pressure and migrates proclets

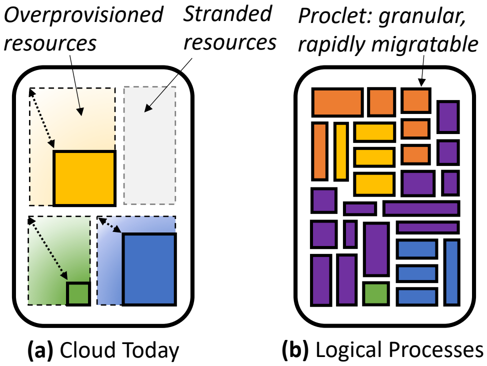

Overprovisioned resources
Stranded resources
Proclet: granular, rapidly migratable
(a) Cloud Today
(b) Logical Processes

Figure 1: (a) Resources are wasted as they are overcommitted for peak use (gradient) or stranded as additional tasks do not fit (gray). (b) Proclets permit tighter packing, which fits more tasks (orange, purple) that can be migrated away quickly under resource pressure.

without disruption to the application workload. Nu migrates proclets in ≈100μs and its migration exceeds the rate at which the Linux kernel can allocate memory (≈ 7GB/s), so Nu handles even intense resource pressure. Under this memory pressure, the social network app adds 122μs to the 99.9th-percentile client latency for a period of 0.86s; the key-value store app adds 52μs for 2.1s; and k-means loses 2.9% throughput for 37ms. Under intense compute pressure, disruption is higher, but still short-lived: the key-value store adds 1,053μs for 0.54s; and k-means loses 5.8% throughput for 24ms. Finally, Nu's logical processes are efficient in the absence of resource pressure, and match or exceed the performance of strong baselines on one or more servers.

Nu has some limitations. First, logical processes require developers to structure applications as proclets. This may not be feasible for every application (§3.3), but we have shown that it is feasible for three very different applications. Second, Nu currently considers only two resources, memory capacity and compute load. We expect other resources can be added (network, caches, memory bandwidth, etc.), but that remains future work. Third, Nu targets pod-scale deployments where network bandwidth is plentiful and latencies low.

Nu will be available as open-source software.

2 Motivation: Resource Fungibility

Cloud computing originally promised to deliver utility computing, with fine-grained, pay-per-use sharing of compute resources, rather than fixed-size machines that customers must purchase and own [4,31]. But, almost two decades later, the operational reality is different: although end-users can readily rent resources, cloud providers still provision and offer these resources in fixed-size units and over long time horizons.

As we argue next, a key problem in this setting is the lack of fungibility—the ability to reassign resources quickly and without disruption between different users and across different machines. Users today submit requests for fixed allocations (number of cores, memory, etc.) as determined by so-called "instances" (or "slots", "tasks"). These allocations tend to overestimate actual resource use, which fluctuates at sub-second time scales. Providers bin-pack instances onto the available servers [33,35,70,75,76]. This is inefficient because users must size instances for peak rather than typical usage, leaving substantial resources idle most of the time. Providers can reclaim some of these wasted resources by overbooking and scheduling best-effort instances in them [2,44,75,83]. But this practice is disruptive, as machines can get intermittently overloaded, leading to performance degradation (e.g., high tail latencies), which is particularly problematic for latency-sensitive workloads [28,44]. In response, the cluster manager must kill some best effort instances to free up resources. But doing so is also disruptive because the work done by a killed instance is wasted and must be redone. Moving the instance usually is not an option as it requires an expensive VM or process migration that can take seconds or minutes because the state to be moved is large, and it requires the cluster manager to find a destination machine that has sufficient resources to take over the entire (indivisible) instance.

In other words, today's cloud is not fungible as illustrated in Figure 1(a). Resources can only be reassigned on relatively long timescales, larger than the timescales over which resource consumption fluctuates. The underlying reason for this problem is that current abstractions for compute work and memory state—VMs, containers, and processes—are too coarse-gained.

A more efficient design would avoid disruption and reassign resources quickly and at fine granularity. This would make it easy for providers to increase utilization by densely packing instances across machines while rebalancing and migrating work as necessary (e.g., in a pod), instead of killing instances under resource pressure. In addition, this would eliminate the burden on users to predict and specify peak per-machine resource usage for each instance, allowing them to instead pay for resources as they are used.

Our approach to fungibility. To provide fungibility, we revisit the *process*, a core OS abstraction that dates back to the 1960s. Traditionally, a process is an instance of a computer program that runs on one machine, consisting of memory and a set of threads. Our work extends this idea across machines to provide a similar abstraction called a *logical process*.

Logical processes are inspired by logical disks [59,73]. Much like a logical disk, a logical process combines together disparate physical resources—in this case machines rather than disks. A logical process automatically scales to use additional machines when more capacity is needed, and can recover from machine failures.

A logical process achieves fungibility through two key ideas (Figure 1(b)). First, a logical process divides program state into fine-grained partitions called *proclets*. Second, proclets are migrated quickly between machines in a pod in response to memory or compute resource pressure. Each proclet runs on one machine at a time, and proclets communicate with each other through efficient message passing.

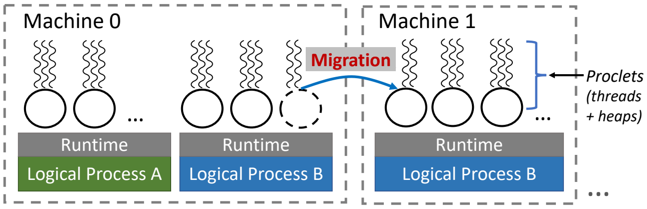


Figure 2: Logical processes spread across machines. Each logical process is comprised of proclets that include a heap (shown as a circle) and threads (shown as squiggles). Proclets rapidly migrate to other machines in response to resource pressure.

Because proclets are fine-grained, migrations complete quickly, causing minimal performance disruption. Inspired by prior work that shows that decomposition into small units simplifies placement [51,57], proclets' fine granularity makes packing them onto machines simple and avoids complex and time-consuming bin-packing on allocation or migration.

Alternative approaches. There are a few other approaches to improving fungibility, but they have drawbacks. One can migrate VMs [20,77], containers [22], or processes [49], but migration is slow due to the state size. Another approach, which maintains the process abstraction, is to use distributed shared memory (DSM) to spread a normal process across machines [82]. But DSM systems experience high coherence overheads with shared memory, leading to poor performance. PGAS [5,17,54] is a type of DSM that can avoid such overheads, but its applicability is limited to parallel applications.

Another approach to fungibility is to adopt new programming models to distribute the application into smaller units, as with distributed objects [6,13,21,30,69,81], microservices, and serverless functions. These models depart significantly from the familiar process abstraction, and they are built on top of traditional, coarse-grained instances that limit their fungibility. They also have high RPC messaging overheads (and cold start delays for serverless functions [71]) that grow in cost as their units become smaller. Alternatively, parallel programming frameworks [8,15,23,26,50,80] partition work via rigid compute patterns (e.g., partition-aggregate, actors). This constrains the programming model and requires data to be statically placed on machines.

Finally, some techniques provide fungibility but in limited form. Far memory systems [32, 66, 78] can incrementally extend the memory of a process, but they perform well only when the remote memory is cold. Request load balancing can make compute fungible, but it is mostly suited for stateless or read-only services. These two techniques are complementary to logical processes and can be combined with them.

3 The Logical Process Abstraction

A *logical process* exists across one or several machines and contains a collection of proclets. Proclets are fine-grained partitions of program state that form units of migration. Proclets can be individually migrated between machines to relieve resource pressure (Figure 2).

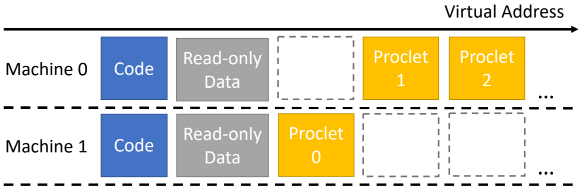


Figure 3: The address space layout of a logical processing running on two machines. Read-only code and data is mapped everywhere, while proclets are mapped in exactly one machine at a time.

A proclet consists of a heap and a set of threads that can access the heap concurrently via shared memory. A proclet never shares its heap memory directly with other proclets. Instead, each proclet has an associated *root object*, which defines a remote method interface that other proclets use to access its state. This approach allows developers to build full programs from proclets in a natural, object-oriented way. The root object may store references (pointers) to ordinary local objects stored on the proclet's heap.

The number of machines allocated to a logical process can change over time in response to shifts in the resources available on each machine. Each machine handling logical processes runs a separate runtime instance. The runtime provides location-transparent communication between proclets, detects resource pressure, migrates proclets between machines, and cleanly handles failures.

Developing software for logical processes is similar to normal UNIX processes. Code can spawn threads, use synchronization primitives to coordinate access to shared memory, and allocate memory using standard APIs like `malloc` or `new`. But there are two major differences. First, developers must partition their program state into proclets. Second, in most cases, developers must use runtime APIs instead of calling system calls or performing I/O directly. We describe the logical process abstraction in more detail in the following.

3.1 Address Spaces and Cache Coherence

A logical process uses an identical address space layout on each machine. This simplifies migration, as pointers remain valid across machines without swizzling. Runtime instances coordinate to keep their layout synchronized during initialization and whenever new proclets are created.

Figure 3 shows an example address space layout for a logical process running on two machines. Code and shared data segments are mapped read-only on all machines. Consequently, the machines must be binary-compatible, but not necessarily identical architectures (e.g., AMD and Intel x86 CPUs). Read-only data can store large static arrays, tables, and other inputs that all proclets might need. Proclets' heaps, on the other hand, are readable and writable, only mapped on one machine at a time, and only ever accessible by the owning proclet. This contrasts with distributed shared memory [3,10,25,40,52,67,68], which typically provides cache

```
1  struct Accumulator {
2    Accumulator(int val) : val_(val) {}
3    void Add(int n) { std::scoped_lock l(mu_); val_ += n; }
4    int Get() { std::scoped_lock l(mu_); return val_; }
5    std::mutex mu_; int val_;
6  };
7
8  void mainfunc() {
9    // Creates two proclets with root class Accumulator.
10   auto p1 = make_proclet<Accumulator>(10);
11   auto p2 = make_proclet<Accumulator>(10);
12   // Invokes Get() on p1; prints 10.
13   std::cout << p1.Run(&Accumulator::Get);
14   // Invokes a closure on p1; prints 15.
15   std::cout << p1.Run(+[](Accumulator &a) { a.Add(5);
         return a.Get(); });
16   // Invokes Get() asynchronously; prints 25.
17   auto f1 = p1.RunAsync(&Accumulator::Get);
18   auto f2 = p2.RunAsync(&Accumulator::Get);
19   std::cout << f1.get() + f2.get();
20   // Adds p2's value to p1 by invoking a closure on p1.
21   p1.Run(+[](Accumulator &a, proclet<Accumulator> p) {
22     auto v = p.Run(&Accumulator::Get); a.Add(v); }, p2);
23   // Arguments statically type checked; DOESN'T COMPILE!
24   // p1.Run(&Accumulator::Add, 10, 20);
25   // Proclets are freed when mainfunc() gets out of scope
26 }
```

Figure 4: Code sample for logical processes. Proclets have a root class with methods containing the app logic. Proclets can create other proclets, run methods synchronously or asynchronously, and run closures. Closures can take proclets as arguments to chain execution.

coherence across machines. In other words, *no proclet can share memory with another proclet*. Instead, proclets communicate via remote method invocation, which passes arguments by copying if the proclets are co-located on a machine or by network transfer if they are on different machines.

This design avoids writable shared memory across machines and aligns well with current datacenter networks, which provide high throughput and low latency, but lack hardware support for cache-coherent memory across machines. Additionally, this design enables fault isolation, as it allows one proclet to fail independently from others on different machines. A failure can cause a proclet's memory to disappear at any time, and these errors can be cleanly reported via remote method return codes. This allows us to use standard distributed systems techniques (e.g., replication) to make critical proclets fault-tolerant.

Proclet migrations occur atomically and each proclet runs on exactly one machine at a time. Consequently, cache coherence is available within proclets, but not across proclets. This design allows for a normal programming environment inside proclets, including synchronization across threads (via spinlocks, mutexes, etc.) when they access shared memory within a single proclet's heap.

3.2 Programming Model

Developers write an application as a set of proclet root classes. As in traditional object-oriented programming, each class defines methods and fields. Methods implement the proclet's application logic and expose the API for the proclet to be invoked by other proclets. Fields specify state internal to the proclet, although additional state can be allocated dynamically in the heap at runtime. Figure 4 shows a running example in C++.[1] Lines 1–6 define Accumulator as the root class for a simple proclet that keeps a value val_ and exposes two methods Add and Get to increment and retrieve the value. Here, the methods are one-liners, but in real applications they constitute most of the code.

When a logical process starts up, the runtime launches a main proclet. This proclet typically creates other proclets by calling function make_proclet with their root classes and constructor parameters. In the example, the main proclet invokes function mainfunc (for brevity we do not show the main proclet, only mainfunc), which in lines 10–11 creates two proclets with root class Accumulator.

Proclets communicate only via remote method invocations and closures. With remote method invocations, a proclet calls the methods of the root object of another proclet, either synchronously using function Run(), or asynchronously using function RunAsync(), which returns a future. Lines 13 and 17–18 show a synchronous and two asynchronous invocations of Get on proclets. The two asynchronous invocations run concurrently to hide latency.

With closures, a proclet can implement function shipping [36, 38, 65, 66, 78, 79], and ship a function that invokes methods—interspersed with its own processing logic—on the root object of another proclet. Line 15 shows a closure that invokes Add and Get on the same proclet. This execution incurs a single roundtrip to the server hosting the proclet, even though it invokes two methods. Shipping code to data in this manner can improve efficiency.

The remote runtime may execute methods and closures on the same proclet concurrently on different threads. Hence, the example uses a mutex mu_ to protect val_ against concurrent execution of Add and Get.

Naming and reference counting. Proclets need to know about each other before they can communicate. We adopted a proclet naming scheme based on smart *proclet pointers*. These pointers provide safety, convenience, and reference counting through a shared_ptr-like interface.

Unlike standard RPC frameworks, remote methods or closures can take proclet pointers as arguments. Thus, code can pass handles to proclets to other proclets by passing them as parameters, similar to delegating capabilities. This feature permits a remote method or closure to chain together the execution of multiple proclets while performing computation in between. For example, line 22 shows a closure on proclet p1 that takes proclet p2 as a parameter; the closure first calls p2's Get method, followed by p1's Add method.

Proclet pointers are valid within the entire logical process, even across machines, and the runtime frees a proclet when it loses its last reference. In the example, proclets p1 and p2 are freed automatically when mainfunc() goes out of scope.

We considered using global strings as proclet names, but

[1] A logical process can be implemented in other languages too.

never needed them in building applications. A logical process is tightly coupled, and we found that passing smart pointers is more convenient than hard-coding strings. Typically, initialization code creates several proclets, and passes around their smart pointers, so access is handed over directly in the code.

Type checking. Because a logical process uses an identical program image across machines, static type checking of argument types is sufficient for remote invocations. This contrasts with standard RPC frameworks (e.g., Thrift or gRPC), which additionally have to perform dynamic type checking, incurring runtime overheads and requiring extra error handling. Line 24 thus fails to compile because of too many arguments.

Raw pointers into a proclet's heap are never allowed as arguments; we made this choice to discourage incorrect code that attempts to share memory between proclets. On the other hand, smart pointers are supported, and passing them as arguments causes the objects they own to be copied.

Unlike standard RPC frameworks, proclet invocations allows remote methods and closures to take function pointers and closures as arguments. This is possible as all machines in the logical process map the code segment at the same address.

Network I/O outside a logical process. Logical processes perform their I/O through abstractions provided by the runtime, rather than POSIX syscalls and I/O abstractions. This allows proclets to be machine-independent and migrate between machines without having to move hard-to-migrate local kernel state (e.g., the TCP state machine). In particular, the runtime maintains TCP network connections to clients, which can be either other logical processes or normal processes. These connections allow clients to communicate with specific proclets inside a logical process—or to spread load across groups of stateless proclets—and will forward client requests if the destination proclet has migrated. Similar to existing libraries for distributed request routing [1, 53], the runtime informs client libraries about the proclet's new location, so that the client knows to expect the response on another network connection and to send future requests there. In a pod-scale setting in a datacenter, client and server code are under the control of the same entity, and custom I/O libraries (e.g., for request routing and load balancing) are commonplace [1, 11, 72].

In rare cases, proclets that need to use local resources directly can instruct the runtime to "pin" them to a machine. Such proclets lose their ability to migrate and reduce resource fungibility, so developers should pin proclets only if absolutely necessary.

3.3 Porting Applications to Logical Processes

In principle, any application that can partition its state into fine-grained units can be ported to a logical process (each unit becomes a proclet). This aligns well with existing cloud applications that already partition their state (e.g., microservices, FaaS, distributed frameworks, etc.), though sometimes at a coarser granularity than proclets. There are two main considerations when dividing a logical process into proclets: the proclet granularity and its scope.

Proclet granularity. Choosing the right size for proclets is important. If proclets are too large, resource fungibility suffers. If they are too small, communication overheads increase as remote invocations become more frequent. Developers must choose a sweet spot that provides sufficient fungibility without significant overheads. §6.4.2 shows empirical measurements of proclet performance at different state sizes and invocation compute intensities; in practice, proclets of a few MiB state size work well.

Proclet scope. The next consideration is how to decide what functionality goes into a proclet. One approach is *functional splitting*, which equates a proclet to a logical functional unit in the application (a module, a microservice, a package, etc). Well-known software engineering practices suggest how to choose appropriate units [47, 56]: the unit should include functionality that is intuitively related, that can be described simply, and that can be encapsulated through a compact and easy-to-understand API. The latter property ensures that the interface between proclets is also compact. Another approach is to use *sharding*. Since a functional unit may be much larger than the ideal proclet size, it may help to shard (partition) the unit. For example, consider a large chaining hash table. Each hash bucket of this data structure becomes a separate proclet and stores the proclet pointer in the hash array. To operate on a key in the hash table, the code makes the appropriate method invocation to the corresponding proclet. This results in a distributed key-value store, as proclets are spread across machines, but maintains the hashtable API.

Limitations. Some applications are hard to decompose into proclets, such as applications that manipulate large amounts of state that is not easily divisible (e.g., video encoders, architecture simulation, sorting, and graph processing). For these examples, decomposition may still be possible, but it requires new algorithmic approaches [27, 41, 45, 48, 64].

Other applications may require functionality that is physically tied to a hardware resources, such as a GPU or an FPGA. In these cases, proclets that interact with the hardware may need to be pinned, thus reducing the fungibility of the logical process.

Our prototype also has some non-fundamental limitations. It requires the use of C++, and though the runtime provides many OS services (timers, external and internal network I/O, synchronization, threads, memory allocation, etc.), it does not yet support all services. Despite these limitations, §5 describes how we ported three very different applications to logical processes.

3.4 Security and Threat Model

A logical process has the same isolation properties as a UNIX process—viz., its memory is isolated from other processes, but its threads share an address space—but applies this model across multiple machines. Even though proclets lack shared memory, there is no hardware memory isolation (e.g., via the MMU) between the proclets *within* a logical process to enforce this. We made this choice for performance reasons and it matches the isolation model of UNIX processes. On the other hand, memory isolation is guaranteed across *different* logical processes: each local logical process instance runs in a different UNIX process and is isolated from other logical process instances on the machine.

Address space layout randomization (ASLR) and stack canaries are important defenses against buffer overflow attacks. Although ASLR might at first glance seems incompatible with logical processes' global address space, it works as long as the loader maintains the same randomized address space layout on each machine. Stack canaries also work, as proclets cannot share stack memory and the implementation can maintain a different secret canary value for each proclet.

Finally, logical processes trust the network to provide confidentiality and integrity. This is necessary to make remote method invocation and migration efficient by sending raw data and pointers. Modern datacenter NICs have hardware encryption engines that ensure these properties.

3.5 Fault Tolerance

A proclet may be replicated to tolerate failures. Replication creates backup copies of the proclet's heap, which the runtime places at the same virtual address in different machines. To keep the backup heaps in sync, the primary replica serializes the invocation requests and forwards them to the backup replicas. (This requires proclet operations to be deterministic.) Operations on a replica are totally ordered without overlap within each proclet—a choice that trades off some performance for strong consistency. To reduce replication latency, the primary overlaps execution with the backups, but the primary only returns from an invocation once the backups finish.

When the system detects the failure of the primary (e.g., due to an RPC time out), it atomically promotes a backup to the primary. To keep the same replication factor, it also adds a new backup by pausing the proclet and copies its heap from the new primary to the new backup replica.

4 The Nu Runtime System

We built Nu, a prototype runtime that provides the logical process abstraction and runs inside a normal Linux environment. Nu shares some architectural and implementation building blocks with Caladan [28]. Caladan was a good fit for Nu because it provides a user-level threading package with overheads low enough to hide microsecond-scale latency. For example, if a thread blocks waiting for a remote proclet invocation to return, the runtime can quickly context switch to another runnable thread with little overhead. Caladan uses work-stealing to balance these threads across cores, which reduces tail latency [62]. Caladan also provides an optimized kernel-bypass, user-level TCP/IP networking stack to further reduce proclet communication and migration costs.

Nu adds ≈10,000 lines of C++ code to Caladan. This includes efficient communication infrastructure, a new memory management layer to handle multiple heaps, a well-optimized proclet migration system, and a controller to track the location of proclets. In the following, we describe these components.

4.1 Serialization and Communication

Nu serializes arguments to remote invocations using `cereal`, an efficient, header-only library for serialization [16]. Cereal has a compact binary serialization format that supports most STL types, but prohibits raw pointers and references (`shared_ptr` and `unique_ptr` are still supported). We modified `cereal` so that it can serialize function and proclet pointers. To optimize use of `cereal`, Nu maintains a buffer pool for serialized outputs and eliminates extra data copies.

Nu uses C++ templates to internally produce code at compile time for serialization and deserialization of remote method arguments. This contrasts with RPC frameworks like Thrift, which require code generation and an interface description language. As a result, developers call remote methods without boilerplate, and they benefit from static type checking.

We took several steps to optimize remote method invocations. First, Nu opens one TCP connection on each core for each outgoing machine. These connections use specific 5-tuples, so they have matching flow-level affinity with the core they are associated with, enabling cache-aware steering [42, 60]. This design increases the number of open connections, but Caladan easily scales to 10,000 connections, much more than needed for our pod-scale target environment. Second, Nu applies adaptive batching to combine remote method invocation payloads (requests and responses) into larger TCP transfers without impacting latency [9]. We modified Caladan to use jumbo frames to increase the benefit of this batching. Third, each connection operates as a closed queuing system, limiting the maximum number of requests in flight. This provides flow control and prevents unbounded memory consumption under overload. Finally, when the caller and callee proclets are in the same machine, Nu substitutes the RPC with a fastpath: a local call without any RPC overheads.

4.2 Memory Management

Nu uses a custom slab allocator to manage each proclet's heap. It includes a per-core object cache to increase scalability, similar to most modern multicore memory allocators [12, 14]. C++ allows a custom definition of `operator new()` that Nu uses to override memory allocations. Nu keeps track of which proclet each thread is associated with and directs allocations to the correct heap. In the future, we plan to explore specialized proclet allocators too. For example, an arena allocator could benefit short-lived proclets because it need not free objects until the proclet terminates, reducing overheads.

4.3 Migration

Nu migrates proclets across binary-compatible machines under resource pressure. Nu separates migration mechanism from policy.

Mechanism. To migrate a proclet, the runtime first sets a migration flag, causing method invocations to the migrating proclet to be rejected and retried. Next, it preemptively pauses and saves register state for all the proclet's running threads to ensure that the data is not mutated during migration. Then, it moves proclet data, including heap, stack, and register state, to the new destination. Finally, the runtime clears the migration flag and contacts the controller to update the location of the proclet, ensuring pending and future method invocations are routed to the new destination (§4.4). We co-designed Nu's RPC layer with migration, and it routes the results of method invocations on migrated proclets back to the caller.

We optimized Nu's migration datapath. To improve TCP throughput, we used parallel connections and jumbo frames. We found that Linux's `mmap` (used for creating the proclet space at the destination machine) was a bottleneck. So we modified the Linux kernel to pre-zero freed pages. After this optimization, we found that Nu could migrate at line rate on 100GbE. When we tried 200GbE, `mmap` again became a bottleneck—in this case due to the Linux kernel's physical frame allocation speed. As a workaround, Nu instead uses `mmap` to prefault a small pool of memory at the destination server. Then, on migration, Nu performs `mremap` on that memory, to reuse prior frame allocations. Further Linux kernel optimizations might avoid the need for this remapping.

The CPU overhead of migration is moderate in our current prototype: it only takes three hyperthreads to saturate 100GbE and five hyperthreads to saturate 200GbE.

Policy. Nu provides an extensible migration policy interface that dictates which proclets to move and where to move them under resource pressure. Many sophisticated policies are possible, including policies that react to several types of resource pressure (e.g., CPU load, cache pressure, memory capacity, memory bandwidth, network bandwidth, etc.), and policies that co-locate frequently communicating proclets to improve locality. Currently, our prototype ignores locality and focuses only on CPU load and memory capacity, two resources often subject to pressure, but we plan to extend it in the future.

Because Nu's migration is fast, we found that even the simplest policies work well (§6). In particular, Nu needs no sophisticated algorithms to predict future resource use, but rather simply migrates proclets at the last moment, when resources are nearly exhausted. To determine when migrations are needed, a monitoring thread in the runtime polls resource use. For memory, it monitors the amount of free memory and begins migrating once it falls below a threshold (e.g., 1 GiB). Similarly, for CPU it monitors system core utilization and begins migrating when a threshold of cores are busy. A better alternative might track the queueing delay of runnable threads, allowing Nu to distinguish actual overload from cases where all cores are busy but not overloaded [19]. We plan to investigate this in the future.

Nu migrates one proclet at a time until resource pressure is eliminated. To determine which proclet to migrate, Nu uses this formula (where P is the set of proclets on the machine):

$$\arg\max_{p \in P} \left(\frac{RESOURCE\_USE(p)}{MIGRATION\_TIME(p)} \right)$$

$RESOURCE\_USE()$ measures a proclet's use of the resource under pressure, and $MIGRATION\_TIME()$ models the migration time of a proclet by considering the size of its heap, as well as the number of threads it must pause and transfer, and the size of thread stacks. This maximizes the pressure alleviation rate, allowing Nu to optimize for its response speed. Nu's runtime collects metrics in real time to enable it to accurately estimate this rate.

To determine the migration destination, Nu queries a global cluster controller, which monitors resource use across servers and returns possible destinations (described next).

4.4 Controller

Nu has a controller that makes cluster-wide decisions, such as proclet placement and virtual address allocation, and tracks information, such as proclet location and resource use. Nu assumes that the controller is highly available. Although our prototype controller is centralized, high availability can be achieved through primary-backup replication or simple recovery: the controller keeps only soft state, so it can always restore its state by querying the servers.

Placing proclets. The controller periodically probes servers' available resources. It uses this information to decide where to place a proclet on creation or migration. Currently, it uses a simple policy that spreads proclets evenly across machines.

Allocating virtual address segments. Proclets must use non-overlapping virtual addresses. Therefore, Nu divides the virtual address space into an array of segments. These segments are large enough (4 GiB by default) to leave room for a proclet's heap to grow. The controller keeps lists of allocated and unallocated segments. On proclet allocation, the local runtime contacts the controller to obtain an unallocated segment.

Resolving proclet location. The controller keeps a location map from the starting logical address of each proclet to the IP of the machine hosting the proclet. Each local runtime maintains a cache of the location map that contains the proclets it has recently accessed. This eliminates the need for method invocations to communicate with the controller in the common case, moving the controller off the critical path for the steady-state application traffic. When a proclet migrates, the controller updates the map. This causes caches to become stale, so a local runtime may send a method invocation to the wrong machine. When this happens, the remote machine returns an error. The local machine then handles the error by invalidating its cache entry and contacting the controller to find the new machine location.

Workload	# proclets	Memory	Compute Intensity [time/invoc.]	Proclet size
SocialNetwork	12 (microservices) + 65,536 (hashtable)	113 GiB	1–100 μs (variable)	31 KiB–8 MiB
In-memory KVS	65,536	138 GiB	1 μs (low)	2 MiB
Phoenix k-means	720 (workers) + 1,024 (hashtable)	8.4 GiB	map: 4.6 ms, reduce: 677 μs	31 KiB–19 MiB

Figure 5: Characteristics of the three case study applications (KVS is an in-memory key-value store, and Phoenix is a MapReduce framework).

4.5 Replication

Nu provides traditional primary-backup replication for proclets. This works by forwarding proclet operations from a primary to backup replicas, akin to traditional state machine replication (SMR). One challenge specific to Nu is that a proclet operation can invoke sub-operations on other proclets. The backup replicas will invoke the same sub-operations as the primary, but side-effect causing invocations must occur only once, and replicas must see the same results as the primary's operations. Nu supports a variant [58] of RIFL [37]'s duplicate detection. Proclets assign a unique ID of the form ⟨proclet id⟩+⟨epoch⟩+⟨sequence number⟩ to each proclet-to-proclet invocation. Primaries forward their sub-operation invocation results to the replicas, and replicas reuse the results (identified by the unique ID). Returning the saved results instead of re-executing sub-operations ensures all replicas have the same heap state. As with an unreplicated system, if a primary crashes in the middle of an operation, its sub-operations are re-executed if the unfinished operation is retried.

When Nu's controller detects a failed primary, it promotes a backup to be the new primary and updates its location map with the new primary. However, runtimes may have the old primary in their caches, which could cause a "split brain" situation if the old primary continues to serve requests. A standard epoch-based approach [43,55] can help Nu avoid this problem: each reconfiguration increments an epoch counter and backup proclets reject operations with outdated epochs.

5 Application Case Studies

We implemented three applications on Nu, which cover a range of proclet sizes, communication patterns, and compute intensities (Figure 5). All applications use a Nu-enabled hashtable library. The hashtable partitions the key space with a hash function and uses proclets as data shards. A root proclet has a vector of proclet pointers to these shards and shares them with client proclets to allow direct communication.

SocialNetwork (a part of the DeathStarBench suite [29]) is a multi-tier, interactive web service, originally built as 12 microservices. Its overall complexity is high, with a fan-out communication pattern and many microservices that have low compute intensity, making it sensitive to both tail latency and RPC overheads. We ported SocialNetwork to a logical process, turning each microservice into a proclet. However, we found that its compute intensity was sometimes too low and that it lacked autoscaling support; both limit its overall scalability. Therefore, we also built a version of SocialNetwork that is better structured for a logical process: this version merges SocialNetwork's small, stateless microservices into a single root class, and scales by spawning it as proclets across machines. Both versions have ≈1,000 LOC, compared to 6,843 LOC in the original, which highlights the simplifications afforded by logical processes. Our implementation replaces the external stores used by microservices (Memcached and Redis) with a backend based on our hashtable, and leverages proclet closures to support Redis-like local operations. We also modified our external I/O subsystem to interact with unmodified Thrift-based clients. This is possible because any root proclet can handle any request as root proclets are stateless.

KV Store is a key-value store composed of the Nu-enabled hashtable library and an additional 200 LOC. It is a stateful application that is latency-sensitive and uses significant memory, making it hard to migrate. On each machine, the runtime's external I/O subsystem receives requests from external clients and steers them to the right proclets. The store has low compute intensity (1μs/invoc.), but large proclet state (2 MiB/proclet).

K-means is a workload from Phoenix MapReduce [63]. Phoenix MR is a NUMA-oriented, shared-memory MapReduce framework designed for single-machine operation. We run k-means—an algorithm that requires multiple iterations—in a Nu-based Phoenix MR port, using proclets to scale across machines. We modified Phoenix's task scheduler to replace worker threads with worker proclets, ship closures to the workers, and shuffle data between mappers and reducers via our hashtable (changing 548 out of Phoenix's 3,066 LOC). K-means is compute-intensive (0.7–4.6ms/invoc.), but has smaller proclet state (31 KiB–19 MiB/proclet). Overall, we found it easy to modify Phoenix MR to work in a distributed setting. Our version follows the same partition-aggregate communication pattern that makes distributed MapReduce frameworks sensitive to stragglers in k-means.

6 Evaluation

We evaluate Nu with three applications, as well as microbenchmarks that measure the impact of specific design decisions. Our evaluation seeks to answer four questions:

1. Can migration in Nu prevent performance disruption during intense resource pressure? (§6.1)
2. How does porting applications to Nu impact their performance? (§6.2)
3. How well does Nu scale with the number of servers? (§6.3)
4. What is the effect of compute intensity, as well as that of our key design decisions, on Nu's performance? (§6.4)

# of Machines	Controller	Proclet servers	Clients
SocialNetwork	1	26	5
KV Store	1	15	16
K-means	1	30	1

Figure 6: The way machines are allocated for each application.

Setup. Except §6.4.2, all other experiments run on a cluster of 32 physical servers in CloudLab [24]. The servers are `c6525-100g` instances (24-core AMD 7402P at 2.80GHz, 128 GB RAM, Mellanox ConnectX-5 NIC), connected by a 100 GbE network. We run §6.4.2 on `c6525-100g` servers, `c6525-25g` servers (i.e., the variant with 25GbE NICs), and our local servers with a 200 GbE network.

Servers run Ubuntu Linux 20.04 with kernel v5.10 patched to pre-zero free pages (§4.3). We disable ASLR, as Nu does not support it yet.

6.1 Application Performance Under Resource Pressure

Nu's proclet-centric design enables fine-grained, rapid migration. The key goal of this design is to achieve high application performance even under resource pressure. To evaluate this, we expose Nu and our applications (§5) to compute and memory resource pressure and measure the application performance as proclets migrate to other machines. We skip SocialNetwork for compute pressure, as this application can handle it with a standard front-end load balancer. We run experiments using 32 machines, one of which serves as the controller (§4.4). The remaining machines are either proclet servers or clients, and we partition them appropriately for the application (Figure 6). To evaluate Nu's ability to manage disruptions under demanding load conditions, we generate enough client load to use ≈70% of CPU capacity across all proclet servers. Then, we induce resource pressure on one proclet server, causing it to migrate its proclets to the other servers.

In these experiments, memory pressure comes from an antagonist process that allocates memory as fast as Linux's virtual memory subsystem permits (≈ 7 GB/s measured in our machine with 4K pages). Once the memory usage of the machine goes above the threshold, Nu starts to migrate proclets to free memory. A good result would show Nu migrating proclets sufficiently quickly to keep up with the allocation rate of the antagonist, without disrupting application performance. To assess the benefit of rapid migration, we compare Nu against a baseline that emulates MigrOS [61], a recent RDMA-based live migration system. To emulate MigrOS, we throttle Nu's migration speed to 600 MB/s on average with a 200 ms initial delay. Since migration speed is slower than the antagonist's memory allocation, the machine starts swapping. We swap to a fast device: Linux `brd`, a block device backed by RAM. (The common alternative—killing processes—is even more disruptive, wastes work, and yields no meaningful baseline.)

Figure 7a shows the 99.9th percentile latency of client requests in the SocialNetwork application. At t=3.9s, the antagonist starts allocating memory, and once Nu's runtime detects that the free memory size goes below 1 GiB (a configurable threshold) at 4.9s, it starts migrating proclets to another machine. During the migration, client-perceived latency increases by less than 19%. At t=5.7s, all proclets have migrated and latency recovers. Figure 7b shows the same experiment with the baseline (Nu emulating MigrOS's migration speed). Since it migrates memory slower than the antagonist requests, memory runs out at t=5s and Linux starts swapping. Thus, the 99.9th latency increases from 639μs to 206ms. The antagonist finishes at t=10s, and latency eventually recovers as memory use drops. Figure 8 summarizes the results for the same experiment on KVS and k-means, which show a similar trend (graphs in §A.1).

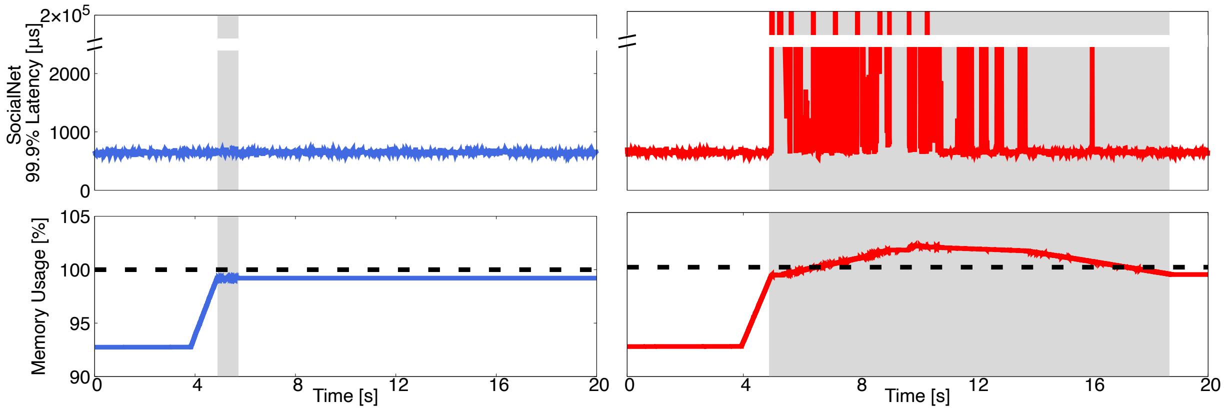
Compute pressure is harder to handle well than memory pressure as the CPU use can spike instantly. Figure 9 shows that Nu experiences a higher performance impact when faced with an antagonist that suddenly uses half the available CPU cores. However, disruption is still short-lived as Nu resolves pressure rapidly through fast migration. By contrast, the performance impact on the baseline lasts ≈15× longer.

These results show that Nu frees resources quickly under pressure, migrating proclets faster than Linux can allocate memory. Consequently, the pressuring workload (here, the antagonist) neither runs out of resources nor slows down, and the applications experience only modest tail latency increases. This means that Nu-based applications can use spare resources without risk: Nu can always migrate proclets if other workloads need the resources.

6.2 Comparison with Existing Implementations

Nu seeks to provide logical processes that match or exceed the performance of current architectures even in the absence of resource pressure. Although Nu allows distributed operation, local proclet invocations would ideally match the performance of computing on a single machine. We therefore compare the performance of Nu-based applications to baseline implementations without logical processes on a single machine. We measure tail latency under varying load for long-running services (SocialNetwork and KVS), and throughput for k-means. A good result would show Nu matching the baseline on NUMA-optimized, compute-intensive applications (e.g., Phoenix k-means), and it would outperform the baseline on RPC-based applications because Nu's fastpath avoids RPC overheads.

Figure 10 shows the results. Nu matches or exceeds the baseline's performance in all cases. For SocialNetwork (Figure 10a), Nu serves about 850k requests/second with sub-millisecond 99.9th percentile latency. The baseline implementation, which runs microservices in Docker containers and uses Thrift RPCs, scales to only 8,000 operations/second, with a 9–60 ms 99.9th-ile latency (very left of the graph). Nu outperforms the baseline because its fastpath avoids the overheads



(a) Nu (fast migration).

(b) Baseline (slow migration).

Figure 7: SocialNetwork runs alongside a memory antagonist that starts at 3.9s. When the memory usage reaches the high watermark, Nu starts migrating proclets rapidly (the gray window). By matching the allocation speed of the antagonist, Nu keeps the memory usage flat and resolves the pressure in 0.86s. SocialNetwork's 99.9$^{\text{th}}$-ile latency is unaffected. By contrast, the baseline fails to migrate fast enough and starts swapping, which leads to 206ms latency (322×). After the antagonist finishes, the memory usage and the latency gradually return to normal.

Workload [Disruption Effect]	**Baseline (Slow migration)**		**Nu (Fast migration)**	
	Duration	Effect	Duration	Effect
SocialNetwork 99.9$^{\text{th}}$ Lat. [μs]	9.14s	206ms (322×)	0.86s	761μs (1.2×)
KV Store 99.9$^{\text{th}}$ Lat. [μs]	60.94s	1.64s (>10,000×)	2.10s	85μs (2.6×)
K-Means Tput. [# iters/s]	0.73s	3.25 (-33%)	37ms	4.71 (-2.9%)

Figure 8: Under memory pressure, proclet migration with Nu sees shorter disruption and better performance during disruption than migration at state-of-the-art process live-migration speed (baseline).

Workload [Disruption Effect]	**Baseline (Slow migration)**		**Nu (Fast migration)**	
	Duration	Effect	Duration	Effect
KV Store 99.9$^{\text{th}}$ Lat. [μs]	7.96s	874μs (26.5×)	0.54s	1086μs (32.9×)
K-Means Tput. [# iters/s]	0.65s	2.41 (-50.3%)	24ms	4.57 (-5.8%)

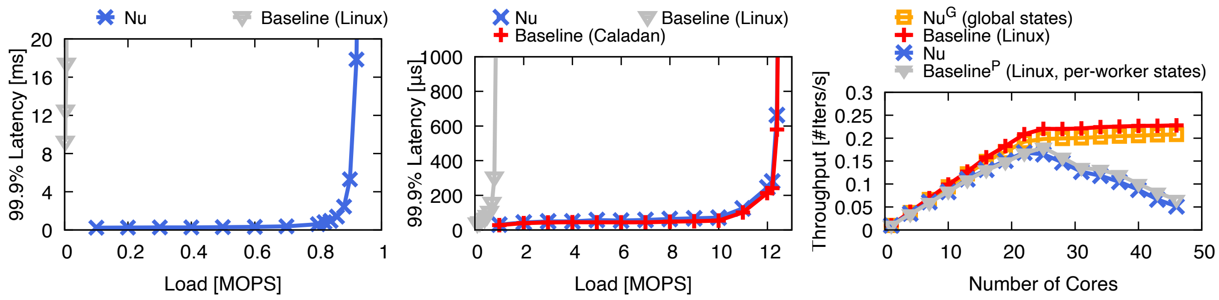
Figure 9: Under compute pressure, Nu sees short disruption and acceptable performance during proclet migration.

of loopback RPCs (serialization and network syscalls) with a single-machine. For KV Store, Nu outperforms memcached on Linux by 15×, serving 12M operations/second to memcached's 800k at sub-millisecond latency (Figure 10b). Crucially, Nu performs as well as the same KV Store running on Caladan [28], which also uses kernel-bypass networking and a user-level threading runtime. Finally, Nu matches Phoenix MR's performance (Figure 10c). Phoenix MR is designed for scalability on a single NUMA machine, and exploits shared memory for performance, so it is a strong baseline. The k-means workload requires sharing the intermediate clustering result across all workers. In a shared-memory setting, this shared state can be a global variable (as in the baseline), but in a distributed framework would involve per-worker copies. Since Nu supports migration, it must be prepared to operate distributedly and keep per-worker (per-proclet) copies, which amplifies the application's cache footprint on a single machine. We therefore compare two Nu setups: per-worker copies (label Nu) and global state (Nu^{G}), and add a modified baseline with per-worker states (Baseline^{P}). Nu^{G} measures the overhead of Nu's infrastructure with pinned (unmigratable) proclets. The overall results show that Nu's proclet invocations on a single machine are fast enough to match the performance of single-machine baselines.

6.3 Scalability

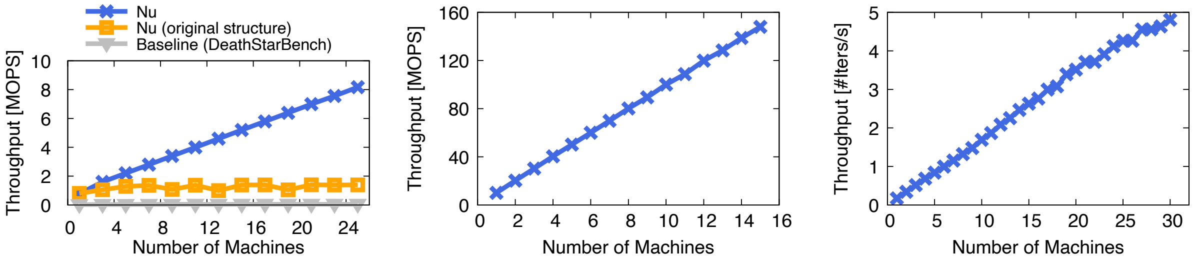
Next, we consider how Nu scales as the proclets of a logical process are spread across many machines. For each of our three applications, we run an experiment where the runtime assigns its proclets round-robin across servers. We consider two versions of the SocialNetwork application: the one from §6.2, which we wrote with logical processes and proclet decomposition in mind; and a second version that mirrors the exact microservice decomposition in DeathStarBench. We measure throughput for equal-sized input, i.e., a strong-scaling setup. Because Nu's local proclet invocation is faster than remote invocation, the single-machine setup has a substantial efficiency advantage, which makes linear scalability difficult to achieve. An ideal result would therefore show scalability close to linear as the number of machines increases.

We show the results in Figure 11. Nu scales well in all three applications, and achieves nearly linear scalability for KV Store and k-means. The SocialNetwork application is the most challenging to scale (Figure 11a). A direct port from the original microservice architecture to Nu (where each microservice becomes one proclet) results in many proclets with methods that have low compute intensity. When invoked remotely, calls to these methods can be costly, while the ad-



(a) SocialNetwork **(b)** KV Store **(c)** K-means

Figure 10: Nu-based applications match or outperform baselines on a single machine. For SocialNetwork, Nu's fastpath helps it outperform the expensive RPCs in the baseline; in KV Store, Nu matches Caladan's [28] performance; and for k-means, Nu matches the baseline depending on how state is represented: as a global shared array (typical in a NUMA setting) or as per-worker arrays (as in distributed settings).



(a) SocialNetwork **(b)** KV Store **(c)** K-means

Figure 11: Nu scales well as the number of machines increases. The Nu-native SocialNetwork application, which merges the baseline's stateless microservices, scales better than the direct port of the baseline because its proclets better amortize the cost of remote invocations. K-means scales sub-linearly as the overhead of broadcasting each iteration's intermediate results increases with the number of machines.

ditional resources of a remote machine speed up the more compute-intensive invocations. On balance, Nu's throughput still increases with the number of machines—from 786k to 1.37M ops/sec—but less so than when the application is decomposed into proclets that have sufficient compute intensity (786k to 8.44M ops/second). However, both Nu-based implementations perform one to two orders of magnitude better than the DeathStarBench baseline (45k ops/sec). The KVS implementation on Nu scales very well (Figure 11b) as it relies on client-side request steering (in response to hints from Nu's runtime) to direct clients' requests to the right machine, which then makes a local proclet invocation. K-means (Figure 11c) has high compute intensity, which makes scaling easy.

We conclude from these results that Nu's logical processes scale well when proclets are distributed across machines if a proclet's methods have sufficient compute intensity. §6.4.1 evaluates the impact of compute intensity on Nu's efficiency.

6.4 Design Drill-Down

6.4.1 Impact of Compute Intensity

We now examine the efficiency of Nu's mechanism for proclet method invocation (§3.2). Intuitively, the more compute an invocation does, the easier it is to amortize the overheads of the invocation (serialization, networking, and threading); yet, the lower these overheads are, the better Nu's performance becomes. Our experiment is a sensitivity analysis in which we vary the compute duration in a proclet's method between 0.1 and 10μs, and we measure the aggregate invocation throughput. We use sufficient threads to maximize throughput, saturating the machine that runs the proclet. We consider two cases for Nu: two proclets in the same machine (local), and proclets in different machines (remote). We compare the performance of Nu against three common mechanisms to invoke a task: (*i*) a function call in a Linux process; (*ii*) an RPC using Thrift, a popular open-source RPC framework [74]; and (*iii*) an RPC using a modified Thrift that uses Caladan [28] to reduce TCP and threading overheads. We measure throughput in a closed-loop setting. A good result would show performance of Nu close to local function calls for local invocations, and at least as good as Thrift for remote invocations.

The results in Figure 12 show that when the invocation is local, Nu's performance tracks closely that of Linux function calls. This happens because of Nu's fastpath for local invocations. When invocation is on a remote proclet, compute intensity (invocation duration) matters. For short invocations (0.1μs), Nu is ≈13× worse than local function calls, but 2.4× better than Caladan-based Thrift (and 29.4× better than Thrift on Linux). As the invocation becomes more compute-intensive, these gaps close: for a 10μs task, Nu's remote invocation achieves 85% of local function call throughput. We conclude that locality matters for remote proclet invocations with low compute intensity, but that Nu delivers near-single-

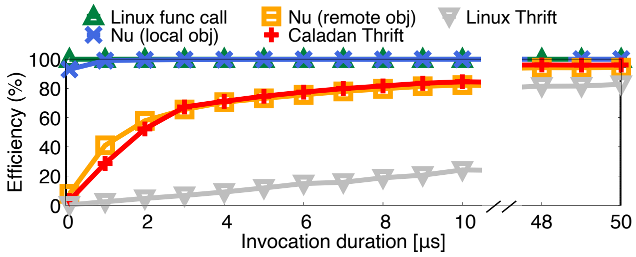


Figure 12: Efficiency (*y*-axis) of Nu invocations as a function of compute intensity (invocation duration), normalized to Linux function call throughput. Nu's local proclet invocation matches the performance of a function call, and Nu outperforms Thrift RPCs by 2.4–29.4× for remote invocations when compute intensity is low.

machine performance for tasks with compute intensity as low as 10μs.

6.4.2 Migration Time and Bandwidth

We now measure the time it takes to migrate a proclet in Nu. The experiment migrates proclets of varying size to another machine and measures the migration time. We vary the test proclet's memory size by adjusting its heap size, from 64 KiB to 16 MiB. The proclet has a single thread with a small stack (64 bytes). A good migration latency would be ≈100μs for modest-sized proclets—orders of magnitude faster than traditional resource balancing mechanisms. For larger proclets, we expect the latency to approach network transfer time.

Heap Size	**Migration Time [μs]**		
	25 GbE	100 GbE	200 GbE
64 KiB	21	21	9
1 MiB	343	111	61
2 MiB	683	216	108
16 MiB	5,452	1,512	771

Figure 13: Nu migrates proclets with different heap sizes (64 KiB–16 MiB) faster with increasing network speeds (25/100/200GbE).

Figure 13 shows the results. With 100 GbE (i.e., the network setting used for all other experiments), Nu migrates small proclets (up to 1 MiB) in under 125μs. This corresponds to a bandwidth of 3–9 GB/s. For larger proclets (2 MiB–16 MiB), the latency varies from 200μs to 1,500μs, which corresponds to a bandwidth of ≈11 GB/s, close to the 100 GbE line rate. The results of 25 GbE and 200 GbE show similar trends. Proclet migration benefits from higher network bandwidth; for example, with 200 GbE, Nu only takes ≈100μs to migrate a 2 MiB proclet. We conclude that Nu migrates proclets quickly and that its migration uses the network efficiently.

6.4.3 Controller Performance

To understand whether Nu's controller can become a performance bottleneck, we benchmark it as a standalone component to measure its capacity. Depending on the type of control message, the controller achieves 0.79-0.96 million msg/s. This is two to three orders of magnitude higher than the real load demand (542–21,450 msg/s) we measured in the end-to-end experiments (§6.1). This makes sense as Nu's runtime caches the proclet location resolution result, thereby moving the controller off the critical path of steady-state application traffic. The controller is only involved in the control plane of initial proclet location resolution and migration.

6.4.4 Proclet Replication

Nu allows replicating proclets for fault-tolerant operation. Replication imposes overhead because it forwards all invocations of a proclet to a backup in a different machine (§3.5). We measure the invocation throughput of calling 8,192 remote replicated proclets, as we vary the compute intensity as in §6.4.1. These invocations do not have sub-operations. The baseline is the same setup without replication. A good result would show a modest loss of throughput with replication.

	Compute Intensity [μs]				
Throughput [MOPS]	0.1	1	10	20	30
with replication	13.18	10.56	2.52	1.52	1.12
without replication	21.04	14.86	3.56	1.97	1.37

Figure 14: Replicated proclets achieve 63–82% of unreplicated throughput, depending on compute intensity.

Figure 14 shows the results. Throughput drops by 37% with a 0.1μs compute intensity, but this drop gradually shrinks to 18% as compute intensity grows to 30μs. Replication adds ≈1.2μs to each operation to invoke the backup proclet, an overhead that gets amortized at larger compute intensities. This result shows that fault-tolerance for critical proclets is feasible and need not come at severe performance cost.

7 Conclusion

We presented logical processes, a new abstraction that decomposes an application into proclets, which are small units of state and compute. Logical processes and proclets solve a key hindrance to increasing datacenter resource utilization: the lack of microsecond-granularity fungibility in resource use.

We found that logical processes and our Nu prototype improve fungibility by making applications granular and migrating proclets quickly under resource pressure. For three applications, Nu matches the performance of strong baselines, scales well, and migrates their proclets within hundreds of microseconds with little disruption to application performance.

Nu is available as open-source software at https://github.com/Nu-NSDI23/Nu.

Acknowledgements

We thank our shepherd Dejan Kostić, the anonymous reviewers, Irene Zhang, Akshay Narayan, and members of the MIT PDOS group for their helpful feedback. We appreciate Cloudlab [24] for providing the experiment platform. This work was supported in part by a Facebook Research Award and a Google Faculty Award.

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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, November 2016. USENIX Association.

[2] 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 *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 735–751, 2020.

[3] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, apr 2010.

[5] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. Upc++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973, 2019.

[6] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[7] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[9] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 168–176, March 1990.

[11] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[12] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128. ACM Press, 2000.

[13] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the Emerald programming language. In *ACM SIGPLAN conference on History of programming languages*, pages 11:1–11:51, June 2007.

[14] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the slab allocator to many CPUs and arbitrary resources. In Yoonho Park, editor, *Proceedings of the General Track: 2001 USENIX Annual Technical Conference (USENIX ATC)*, pages 15–33, 2001.

[15] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 16. ACM, 2011.

[16] cereal: A C++11 library for serialization, 2021. `https://github.com/USCiLab/cereal`.

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In

Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, page 519–538, New York, NY, USA, 2005. Association for Computing Machinery.

[18] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An Alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for μs-scale rpcs with breakwater. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 299–314, 2020.

[20] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[21] Common object request broker archictecture (CORBA). https://www.omg.org/spec/CORBA.

[22] Checkpoint/restore in userspace (CRIU). https://www.criu.org.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI)*, San Francisco, CA, December 2004. USENIX Association.

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

[25] Michael J. Feeley, William E. Morgan, Frédéric H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In Michael B. Jones, editor, *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 201–212. ACM, 1995.

[26] Message P Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, USA, 1994.

[27] 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)*, March 2017.

[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 281–297, November 2020.

[29] 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)*, pages 3–18, April 2019.

[30] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, 2020.

[31] Martin Greenberger. *Management and the Computer of the Future*. Wiley, 1962.

[32] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667, March 2017.

[33] 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 *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[34] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek R. Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial latency-sensitive services. In

Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 519–532, 2018.

[35] Kubernetes. https://kubernetes.io.

[36] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 627–643, 2018.

[37] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, October 2015.

[38] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[40] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, November 1989.

[41] Yilong Li, Seo Jin Park, and John Ousterhout. MilliSort and MilliQuery: Large-scale data-intensive computing in milliseconds. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 593–611, April 2021.

[42] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444. USENIX Association, 2014.

[43] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical report, Massachusetts Institute of Technology, 2012.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[45] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, page 340–349, Arlington, Virginia, USA, 2010. AUAI Press.

[46] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892. IEEE, 2017.

[47] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 2008.

[48] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, January 2010.

[49] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[50] 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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577, Carlsbad, CA, October 2018.

[51] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[52] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference (ATC)*, pages 291–305, July 2015.

[53] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford,

Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[54] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, aug 1998.

[55] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 305–319, Philadelphia, Pennsylvania, USA, June 2014.

[56] John Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018.

[57] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.

[58] Seo Jin Park. *Achieving both low latency and strong consistency in large-scale systems*. PhD thesis, Stanford University, Stanford, California, October 2019.

[59] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 109–116. ACM Press, 1988.

[60] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert Tappan Morris. Improving network connection locality on multicore systems. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 337–350. ACM, 2012.

[61] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. MigrOS: Transparent Live-Migration support for containerised RDMA applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 47–63. USENIX Association, July 2021.

[62] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 325–341. ACM, 2017.

[63] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, February 2007.

[64] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A balanced large-scale sorting system. *ACM Transactions on Computer Systems (TOCS)*, 31(1), February 2013.

[65] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[66] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–332, 2020.

[67] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 174–185, October 1996.

[68] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–306, October 1994.

[69] Malte Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, University of Cambridge Computer Laboratory, February 2016.

[70] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery.

[71] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[72] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. FlightTracker: Consistency across Read-Optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423. USENIX Association, November 2020.

[73] David Teigland and Heinz Mauelshagen. Volume managers in linux. In Clem Cole, editor, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 185–197. USENIX, 2001.

[74] Apache thrift. https://thrift.apache.org.

[75] 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 *Proceedings of the 15th European Conference on Computer Systems*, 2020.

[76] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[77] VMware VirtualCenter user's manual, 2003. https://www.vmware.com/pdf/VirtualCenter_Users_Manual.pdf.

[78] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–280, November 2020.

[79] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship compute or ship data? Why not both? In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 633–651, April 2021.

[80] 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 inmemory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[81] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. Customizable and extensible deployment for mobile/cloud applications. In Jason Flinn and Hank Levy, editors, *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 97–112, 2014.

[82] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. Giantvm: A type-ii hypervisor implementing many-to-one virtualization. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[83] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 379–391, New York, NY, USA, 2013. Association for Computing Machinery.

A Appendix

In appendix, we include the end-to-end performance results under resource pressure that were not included in §6.1 due to the space constraint.

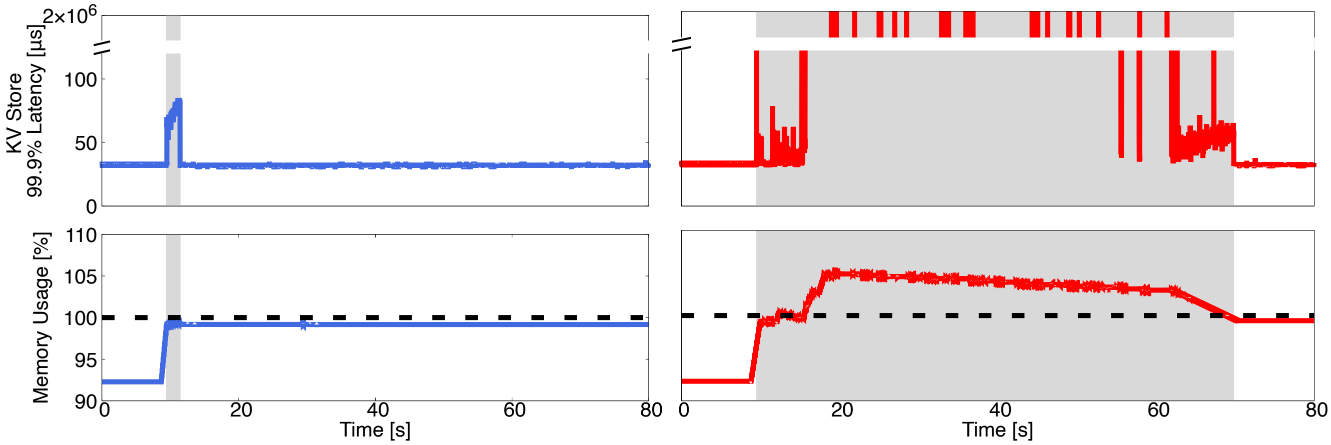
A.1 Application Performance Under Memory Pressure

Figure 15 presents the 99.9th tail latency of KV store under memory pressure. The results are similar to the SocialNetwork results (Figure 7). Figure 16 shows the K-Means performance under memory pressure using the throughput metrics as it is a batch application. Nu achieves 97% throughput during migration, whereas the baseline only achieves 67% throughput. Here we do not show the memory utilization as K-Means has a tiny per-machine memory footprint. The baseline has a low performance mainly because of the long task pausing time caused by slow migration.

A.2 Application Performance Under Compute Pressure

In the next experiment, we evaluate Nu's performance under compute pressure. Compute pressure is harder to handle well than memory pressure, since Nu's solution to resource pressure—proclet migration—itself consumes compute resources. The antagonist process in this experiment is a synthetic CPU-spinning workload that occupies half the CPU cores on the machine, reducing the compute resources available both to the application and to Nu's proclet migrations. In addition, the CPU load of the antagonist can spike instantly; this is different from the memory load which only increases gradually. Therefore, we would expect a higher impact on application performance than when Nu migrates proclets under memory pressure. A good result for Nu would show that the application still achieves acceptable performance, even if degraded for some (ideally short) time.

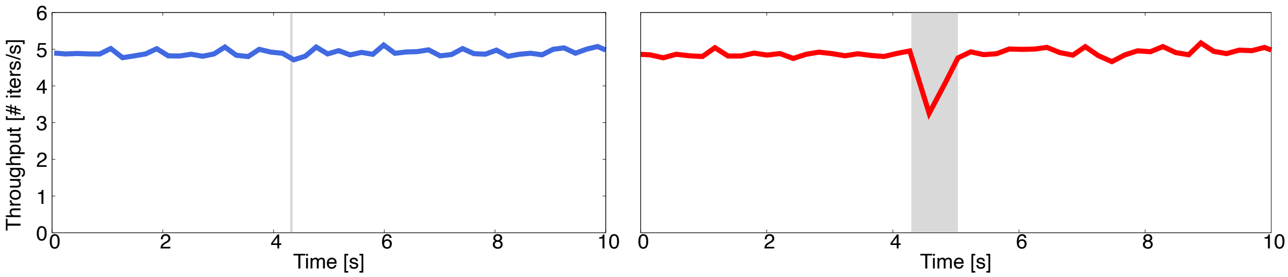
Figure 17a shows Nu's results. At t=4.9s, the compute pressure starts on one machine, taking away half of the application cores, and Nu immediately starts migrating proclets to reduce load on the machine. 99.9th latency increases from 33 μs to 1086 μs. This latency spike makes sense as the machine's compute resources are degraded by 50% and Nu needs additional compute to migrate proclets. The latency gradually decreases as proclets migrate and the other machine starts serving client requests, and soon recovers back to 33 μs as the migration ends at t=5.44s. In contrast, for the baseline, the latency disruption lasts 7.96s, which is 15X of the Nu's 0.54s duration. Figure 18 shows the result of K-means. Nu takes 24ms to resolve the pressure and achieves 94.2% throughput



(a) Nu (fast migration).

(b) Baseline (slow migration).

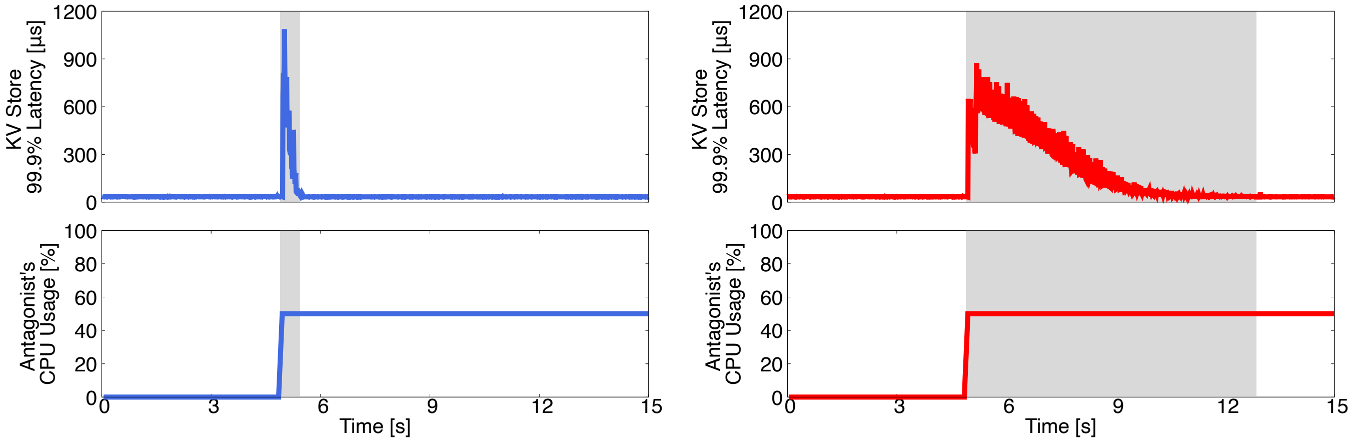
Figure 15: For KV store under memory pressure, Nu is able to maintain 99.9th tail latency under 85μs as it migrates proclets faster than the memory allocation speed of the antagonist. In contrast, the baseline suffers from poor tail latency ($\approx 2\times10^6\mu$s) since it cannot keep up with the allocation speed and has to swap memory.



(a) Nu (fast migration).

(b) Baseline (slow migration).

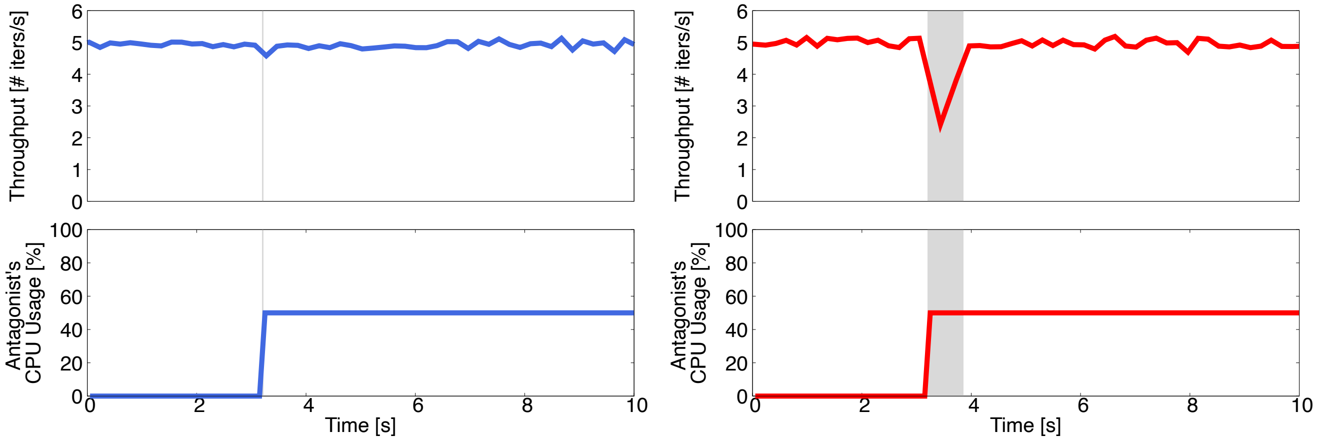
Figure 16: For K-means under memory pressure, Nu maintains stable throughput during migration, whereas the baseline only achieves 67% throughput. We do not show the memory utilization here as K-means has a tiny per-machine memory footprint.



(a) Nu (fast migration).

(b) Baseline (slow migration).

Figure 17: Under compute pressure, the KV store server becomes overloaded and the client-perceived 99.9th tail latency spikes from 33 μs to ≈1 ms. Nu only takes 0.54 s to fully recover the performance, while the baseline requires 7.96 s (≈15X).



(a) Nu (fast migration).

(b) Baseline (slow migration).

Figure 18: For K-means under compute pressure, Nu takes 24ms to resolve the pressure and achieves 94.4% throughput during migration. In contrast, the baseline takes 0.65s and only acheives 49.7% throughput.

during migration. In contrast, the baseline takes 0.65s and only achieves 49.7% throughput.

These results demonstrate that Nu's logical processes react quickly to CPU pressure. Some performance degradation is unavoidable, but the impact is short-lived: after a sub-second delay, proclets migrations relieve the resource pressure.