Unleashing True Utility Computing with Quicksand

Zhenyuan Ruan
MIT

Shihang Li
Brown University

Kaiyan Fan
MIT

Marcos K. Aguilera
VMware Research

Adam Belay
MIT

Seo Jin Park
Google

Malte Schwarzkopf
Brown University

Abstract

Today's clouds are inefficient: their utilization of resources like CPUs, GPUs, memory, and storage is low. This inefficiency occurs because applications consume resources at variable rates and ratios, while clouds offer resources at fixed rates and ratios. This mismatch of offering and consumption styles prevents fully realizing the utility computing vision.

We advocate for *fungible* applications, that is, applications that can distribute, scale, and migrate their consumption of different resources independently while fitting their availability across different servers (e.g., memory at one server, CPU at another). Our goal is to make use of resources even if they are transiently available on a server for *only a few milliseconds*. We are developing a framework called Quicksand for building such applications and unleashing the utility computing vision. Initial results using Quicksand to implement a DNN training pipeline are promising: Quicksand saturates resources that are imbalanced across machines or rapidly shift in quantity.

ACM Reference Format:
Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. 2023. Unleashing True Utility Computing with Quicksand. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3593856.3595893

1 Introduction

It has been over 60 years since the proposal of "utility computing" [22]—the notion that compute resources are consumed on a pay-per-use model, like electricity. Even though public clouds claim the mantle of utility computing, its vision has not been fully realized yet. In fact, modern computing is not just one utility but multiple utilities, such as CPUs, GPUs, memory, and storage. Cloud providers offer these resources through cloud instances (VMs or containers or serverless functions) at large granularities (e.g., GBs of memory, a full CPU core, etc). Consequently, users can only rent resources bundled together at fixed ratios that often fail to match an application's exact needs. For example, an AWS Lambda instance allocates CPU proportional to the memory configured [8], but the user might use it only as an in-memory data cache that requires little CPU [60]. While there is a large menu of instance configurations, none of the choices may fit the resource usage *exactly*, as datacenter applications often have varying consumption [7, 55]. In other words, there is a mismatch between the cloud offering and the user consumption in resource granularity, ratio, and dynamism.

This mismatch hurts both providers and users: providers cannot multiplex applications to fully utilize the available hardware and must deal with stranded resources, while users must pay for resources they do not use and suffer from performance problems when the consumption exceeds what they expected. While one could hope cloud providers will modify their offerings to better match user needs, doing so would require fundamental changes to the cloud architecture [64].

The goal of this work is to liberate users from the challenging resource provisioning task and enable providers to fully utilize resources even if they are stranded or only available for a few milliseconds. To achieve this goal, we propose a new system called Quicksand, which asks users to build applications slightly differently from current practice, by using abstractions that disentangle the use of resources. With Quicksand, developers decompose their applications into different migratable components that each specialize in consuming a specific type of resource. The resulting application is *fungible*: it can use each resource type wherever present—even on different machines for different resources—while achieving good performance. Applications can adjust their consumption of each resource according to resource needs and availability in the moment, while the cloud places the corresponding application components on servers with the required resources.

For example, consider a batch image processing application that takes a large sequence of images, pre-processes each image on CPUs, and uses the resulting output to train a model on GPUs. Normally, such an application would run on a dedicated set of training machines with a fixed ratio of CPUs, GPUs, and memory. These resources are at risk of

HotOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595893

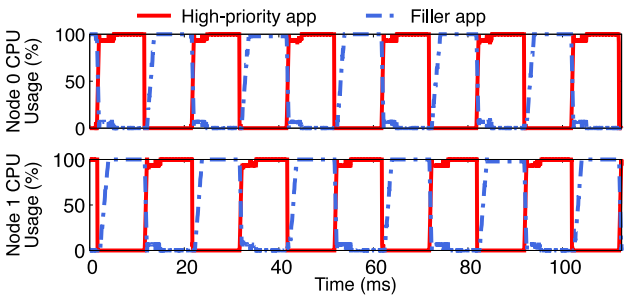

Figure 1: Migration of work at millisecond granularity is possible: the filler application migrates across machines every 10ms to harness periods of idle CPU on the other machine.

being imbalanced, potentially leaving expensive GPUs idle. With Quicksand, the application can adapt its demand to the available resources: if training GPUs are idle or more GPUs become available, the cloud can add additional CPUs for pre-processing to keep them saturated.

There are many challenges in following this approach. To be performant, communication costs must be low enough to maintain performance compared to a traditional design. To be efficient, the system must support the rapid migration of application components to fill in brief resource usage gaps. To be adaptive, the system must react quickly in response to the change in resource consumption and availability. To be usable, the programming abstraction must be easy to adopt, while hiding the complexity of resource migration and scaling behind simple APIs. An efficient implementation must optimize the entire system stack, including the kernel, runtime, and programming abstractions.

2 Background and Motivation

Public clouds today offer resources through instances (e.g., VMs or serverless functions) that are statically provisioned and bundle resources at fixed ratios. For example, an AWS EC2 instance has fixed amounts of CPU, memory, network bandwidth, etc; and an AWS Lambda function allocates CPU cores in a fixed ratio to memory. This scheme couples the allocation of different resources, creating hard problems for both cloud providers and users: cloud providers must fit fixed-size instances to fixed-size physical machines, while users must accurately predict their application resource needs and find the nearest fitting instance configuration.

Application resource needs are hard to predict because they are complex and dynamic. An application rarely uses *exactly* all the resources available on its instance: there is often a bottleneck resource while others remain underutilized [47]. Moreover, resource consumption shifts over short periods in response to the workload. For example, a latency-sensitive application may have spikes in CPU use due to fluctuation in incoming requests, and many batch processing applications have well-known compute-heavy and I/O-heavy phases [18]. Such dynamic behavior makes it hard for cloud providers to effectively oversubscribe resources, as they must guarantee the promised resources without knowing their future consumption. Thus, they often have to conservatively provision resources for an estimated peak [52], resulting in under-utilization; many datacenters report average CPU and memory utilization below 60% [25, 58].

In order to achieve both high resource efficiency and performance, applications have to be able to adaptively scale in response to variable resource needs and availability. Unfortunately, existing solutions react too slowly (tens of seconds or minutes) [1, 6, 17, 25, 32] and fail to independently scale an individual resource (e.g., spawning a new lambda function allocates both CPU and memory). In an ideal world, by contrast, even a few milliseconds of idleness of a single resource could be productively exploited by other workloads that need precisely that resource. We believe that the hardware to achieve this already exists today thanks to fast datacenter networks; what is missing are better software techniques to enhance the fungibility of an application, so parts of the application can run and migrate wherever resources are available—a problem that we tackle in this paper.

We illustrate the power of fungibility through a motivating experiment. This experiment has two servers, each running an instance of a high-priority application (e.g., a latency-critical service). The high-priority application has phased behavior: every 10ms, it goes from consuming no CPU to consuming all the cores on the machine, and reverts to no CPU consumption after another 10ms. The two application instances on different machines are shifted in time: when one instance is idle, the other is consuming all cores, and vice-versa. We want to run another CPU-intensive application to fill the idle CPU cycles in the two machines, but in a classic cloud setting, this filler application could at best leverage *one* machine, leaving 50% of the other machine idle. In contrast, by making the application fungible, we can effectively harness the resources on both machines: as CPU cycles become scarce on one machine, the application quickly migrates to the other machine where the CPU is idle.

To design fungible applications, our starting point is the Nu system [50]. Nu introduces the concept of logical processes, which breaks down the traditional monolithic UNIX process into smaller, independently schedulable units called *proclets*. Each proclet has a heap for storing state and threads for performing computation. Proclets expose an object-oriented interface and can communicate with each other through method invocation. Since proclets are granular, Nu can quickly react to resource pressure by rapidly migrating proclets to machines with spare resources. Nu adopts a distributed runtime that spans all machines to facilitate migration and avoid

cold starts. It takes only a few milliseconds to migrate a proclet with 10MiB of state, orders of magnitudes faster than migrating a VM or even a process.

Nu's quick migration mechanism is a crucial step towards utility computing. In our motivating experiment, we structured the filler application as proclets with small state. When the CPU utilization spikes on the machine running the filler application, its proclets migrate to the other machine. We measure the speed of this migration and the goodput achieved by the filler application. Figure 1 shows the results: the filler application migrates in less than 1 ms between machines to fill in gaps in CPU utilization. This result shows that it is possible to harness tiny periods of idle resources via rapid migration, and that such an approach can increase both utilization and performance.

However, Nu's rapid proclet migration mechanism alone is far from enough to achieve utility computing. First, Nu fails to unleash stranded resources as proclets still bundle different types of resources, making it hard to effectively combine resources from different machines. For example, a server may have much idle CPU but little free memory, while another server has little CPU but plenty of available memory. In this scenario, it may be impossible to fit proclets in either machine, even though in aggregate there are sufficient idle CPUs and free memory. *This problem calls for a new abstraction that further decouples resource consumption.*

Second, Nu is hard to use as it requires significant developer effort to manually decompose an application into granular pieces. While developers are accustomed to writing applications with high-level programming abstractions (e.g., data structure libraries, map-reduce frameworks, etc.), Nu only offers a low-level proclet interface. *This problem calls for a higher-level abstraction that abstracts away the details of application decomposition.*

Finally, fast migration is possible only for fine-grained proclets. A fine-grained proclet may evolve into a coarse-grained one amidst changing loads and resource demands. For example, a proclet that stores a hash table shard can grow significantly as data gets inserted into it. Nu's proclets lack an ability to keep their own granularity small, thereby making migration slow as they grow. *This problem calls for an adaptive repartitioning mechanism to preserve granularity.*

3 A Path Toward Utility Computing

We now propose a potential path toward realizing the vision of utility computing by building upon Nu's proclets. First, we introduce *resource proclets*, new types of proclets each designed to consume a specific resource (e.g., memory, compute, storage, etc.). Second, we propose higher-level programming abstractions, such as sharded data structures and distributed thread pools, that decompose resource usage into resource proclets while providing familiar APIs. Finally, we propose new adaptive mechanisms to split and merge proclets as resource consumption varies.

3.1 Decoupling Resources

Nu adopts *hybrid proclets* that bundle many types of resources, making it hard to independently map resource demand to the resources available on different machines. To address this issue, we propose different kinds of resource proclets, each tailored to a specific resource. For example, memory proclets store in-memory data, compute proclets perform computation, storage proclets keep persistent data, etc. The APIs of these resource proclets are specialized for their underlying resource type. For instance, compute proclets expose `Run(lambda)` to support computation; memory proclets offer `NewPtr<T>(args...)` to allocate distributed pointers that work across proclets for accessing in-memory objects; and storage proclets provide `ReadObject(id)` and `WriteObject(id)` for accessing storage objects.

Resource proclets can interact with each other by invoking their APIs. For example, a compute proclet can consume data from a memory proclet by dereferencing distributed pointers. Quicksand's runtime provides location transparency and optimizes performance; it automatically uses cheap function calls to handle local interactions and remote procedure calls (RPCs) to handle remote interactions.

3.2 High-Level Programming Abstractions

Our resource proclets are low-level constructs that require developers to manually decompose applications, which can be challenging. To make this easier, we will also provide higher-level programming abstractions for developers that hide the complexity of resource proclets.

For memory, we can offer a number of sharded data structures (vector, set, map, queue, etc.) built atop a general sharding library, inspired by existing work in distributed programming [9]. This library partitions data into disjoint ranges based on the sharding key, with each range stored within a separate memory proclet. An index memory proclet maintains a map of sharded ranges to data proclets, allowing users to access elements in these data structures transparently without awareness of which machines are currently storing them. Additionally, we can provide C++-like iterators for seamless iteration of elements across multiple shards. Iterators also provide rich semantic hints, enabling effective data prefetching to reduce the cost of accessing remote shards.

For compute, we can provide a distributed thread pool abstraction where the underlying threads are sharded across compute proclets. The heaps within each shard are left empty, except for any objects temporarily allocated by threads. We can provide a set of commonly used parallel computation APIs (e.g., map, reduce, etc.), enabling users to easily compose memory and compute proclets together. For example,

users can pass data structure iterators to a `map` API; this uses compute proclets to execute a function over each element stored within memory proclets.

Similarly, we can also provide high-level abstractions for other resource proclets, abstracting away the datails of proclet decomposition. For example, for storage, we can offer a flat storage abstraction [40] that automatically spreads fine-grained storage proclets across multiple machines to combine their capacity and IOPS.

3.3 Adaptive Proclet Splitting and Merging

It is crucial to keep resource proclets fine-grained so that Quicksand can quickly migrate them to respond to changes in resource demand or availability. Granular proclets also reduce the complexity for the scheduler to binpack proclets onto machines [39]. To ensure fine granularity, Quicksand has a splitting mechanism for each resource proclet type.

For memory proclets that store data structure shards, Quicksand enforces a maximum size based on a target migration latency. If a shard becomes oversized, Quicksand splits it into two shards by invoking a data-structure-specific `split` function. This technique can also be applied to storage proclets to keep the desired granularity.

A compute proclet, used for parallel computation, can also be oversized when it has more tasks than its CPU resource supports. In this case, Quicksand can split it by dividing its task queue. Splitting occurs only if there are enough CPU resources in the cluster for the new proclet, thus avoiding the creation of an excessive number of compute proclets.

Resource proclets can also become undersized. For example, after removing many key-value pairs from a sharded hash table, memory proclets can have many more hash table buckets than KV pairs, resulting in low memory efficiency. Quicksand can respond by invoking a data-structure-specific `merge` function to combine the adjacent shards into a single memory proclet. Undersized compute proclets can also occur, such as in a data pipeline where producer proclets generate data faster than it can be absorbed by an external sink. Quicksand can react by merging producer compute proclets to match the production rate with the consumption rate.

Splitting/merging resource proclets may briefly disrupt application performance as it blocks new proclet method invocations until it completes. However, Quicksand minimizes the performance impact by ensuring resource proclets are granular so that splits and merges are always fast.

4 Case Study: DNN Training

We illustrate the benefits of Quicksand through a case study in DNN image training, a commonly used cloud workload. Here, a sequence of images must be pre-processed by CPU servers (decompression, data cleaning, augmentation, etc.) before being fed into GPU servers for training a DNN model.

Running these jobs efficiently in today's cloud is challenging [21, 37, 38, 53]. Users struggle to provision CPU and GPU resources without starving one of them. On the one hand, GPUs are the most expensive resource, so in theory users would benefit from adding CPU servers for preprocessing in order to keep GPU servers saturated. On the other hand, provisioning too many CPU servers wastes CPU cycles and increases cost. The optimal amount of resources varies over time as GPU availability changes [5] and training parallelism varies [43].

Second, datacenter providers face difficulties in binpacking jobs into available physical servers in which resource imbalance is common [36]. To illustrate the challenges, consider a scenario of preprocessing a large number of in-memory images with two machines. Suppose one machine has abundant CPU cores but limited memory, and the other has abundant memory but limited CPU cores. It is difficult for existing systems to effectively combine the resources from both machines, forcing them to either run out of memory or underutilize CPUs.

Quicksand provides efficient abstractions to address these issues. To saturate the resources of both machines in the preprocessing stage, we store input images into a sharded vector, therefore decomposing data into granular memory proclets that can be independently scheduled to utilize idle memory across the machines. Similarly, to perform the preprocessing computation, we create a compute proclet that can split on demand to harness free CPU cores across machines. Compute proclets load images from memory proclets using the vector iterator interface, enabling Quicksand to effectively prefetch data to improve locality.

To balance CPU and GPU resources, we apply a sharded queue to connect the CPU-based preprocessing stage (i.e., the producer of the queue) with the GPU-based training stage (i.e., the consumer). The queue can absorb bursts in producer output by storing it in memory proclets that can split and migrate. Quicksand splits or merges preprocessing compute proclets to match the data consumption rate of GPU training, ensuring GPU saturation without wasting CPU resources.

Using an early prototype of Quicksand, we built a simple DNN training pipeline and obtained promising results. The pipeline has an OpenCV-based image preprocessing stage atop Quicksand. For the training stage, we emulated GPUs by adding a delay to consume data from the queue, as we have not yet implemented GPU proclets.

Figure 2 demonstrates that Quicksand can efficiently combine resources from different machines, even when they are heavily imbalanced. In this experiment, we fixed the total amount of CPU and memory resources and divided them between two machines with different configurations (memory-unbalanced, CPU-unbalanced, and both-unbalanced). We

	Machine 1	Machine 2	Time [s]
Baseline	46 cores 13 GiB RAM	N/A	26.1
CPU-unbalanced	6 cores 6.5 GiB RAM	40 cores 6.5 GiB RAM	26.4
Mem-unbalanced	23 cores 1 GiB RAM	23 cores 12 GiB RAM	26.6
Both-unbalanced	6 cores 12 GiB RAM	40 cores 1 GiB RAM	26.5

Figure 2: Quicksand efficiently combines resources from different machines. With the same amount of total resources, a Quicksand-based DNN training preprocessing pipeline can always achieve similar performance to a single machine when its resources are split across two machines, despite severe imbalance.

ran the preprocessing stage on Quicksand with these configurations and compared its performance with the ideal baseline configuration of a single machine with the same total resources (which avoids the overhead of utilizing remote resources). Our evaluation results show that Quicksand schedules compute and memory proclets to the appropriate machines regardless of the setup, achieving both high resource utilization and near-optimal performance. For example, in the scenario where both CPU and memory are unbalanced, Quicksand correctly schedules most of the memory proclets to the memory-heavy node and most of the compute proclets to the CPU-heavy node. Thanks to Quicksand's data prefetcher, preprocessing images from remote memory proclets is as fast as preprocessing local images.

Figure 3 shows that Quicksand can dynamically adapt to varying GPU resources by rapidly scaling the number of preprocessing compute proclets, ensuring GPU saturation at all times. In this experiment, we vary the number of available GPUs between four and eight every 200 milliseconds. We see that Quicksand's mechanisms can adjust the amount of CPU resources quickly, taking 10–15ms to split or merge compute proclets after learning of a change in GPU resources.

5 Discussion and Research Directions

How to design the scheduling policy? There are several challenges to developing effective scheduling policies for Quicksand. First, idle resources must be detected quickly so they can be filled with available resource proclets. Queueing delay could be one such signal to detect idle cores [12], but more techniques are needed for memory, storage, etc. Second, the scheduler has to find a balance between reaction time and quality. On the one hand, the scheduler has to react quickly in response to sudden changes in resource usage. On the other hand, the scheduler has to make high-quality decisions; this inevitably prolongs the reaction time, especially since the fine granularity of resource proclets leads to a large amount of scheduling units. A promising direction is to operate scheduling at two levels: fast local decisions to absorb

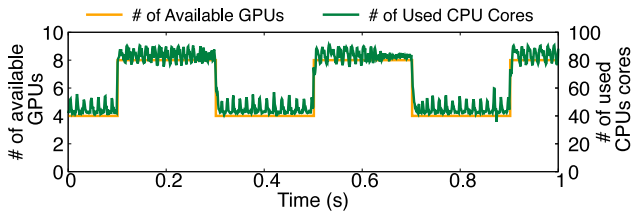

Figure 3: Quicksand dynamically adapts to changing GPU resources by rapidly scaling the number of compute proclets, reaching new equilibriums in 10-15 ms.

usage spikes and slow global decisions that reflect long-term shifts in usage, as observed from runtime traces, compiler hints, and ML models. This could balance the need for fast reaction with the need for optimal long-term placements.

How can we maintain locality? When compute intensity is low, communication costs between resource proclets on different machines might outweigh the utilization benefits of allocating different resource types independently. It may be possible to further optimize communication so that short remote method invocations are efficient. However, another strategy could be to place resource proclets that frequently communicate with one another on the same machine, provided available resources permit it. This is feasible as the runtime can easily capture the affinity information and report it to the scheduler for making colocation decisions. Finally, the I/O path could be used to further reduce the burden of communication by offloading some computation into smart NICs or smart SSDs, which can operate closer to the data they are processing [35, 49].

What other resources can we decouple? We focused on compute, memory, and GPU proclets, but we believe resource proclets could be used to decouple other resources, such as storage, accelerators, or the network. Resources may have sub-resources that are consumed separately and can be decomposed too: for example, storage has capacity and IOPS; compute has cycles and last-level cache usage. Some sub-resources may be inherently tied together (e.g., storage IOPS and bandwidth), which may preclude decoupling. These situations may not allow full utilization of all sub-resources. We could consider different storage classes as different resources, each with its own proclet type: fast flash disks are increasingly used as slow cheap memory, while slower flash or hard disks are used to keep persistent data.

Each type of resource proclet needs its own placement policies and fast migration mechanisms—developing these will pose interesting technical challenges (e.g., how to migrate resource proclets across GPUs rapidly).

How can the hardware help? Faster networks could speed up resource proclet migration. Having a DIMM-attached NIC [3] and copying data directly to CPU cache lines or registers [27] might further boost performance. Offloading transport [24], RPC [33], and serialization [46] to hardware

could reduce communication costs, allowing for proclets with lower compute intensity to run more efficiently on different machines. Finally, new coherent memory disaggregation technologies like CXL [15] offer new design opportunities for our runtime to achieve better performance. For example, with the capability of accessing remote memory transparently, we can speed up resource proclet migration by postponing the copying of data. Another example is optimizing remote proclet communication by avoiding serialization and directly passing data pointers.

How can the OS and runtime help? Our memory proclet migration currently hits kernel bottlenecks on page pinning and memory mapping—optimizing these could further speed up migration. Another opportunity lies in tighter coordination with the memory allocator, allowing Quicksand to skip freed memory during migration. Migration policies will benefit from hints from the OS and user libraries about what accelerators (GPU, TPU, DPU, FPGA) operate on what memory regions. More efficient runtimes could speed up resource proclet creation, execution, and migration. Achieving efficient fault and security isolation among granular resource proclets requires us to rethink the right OS abstraction; existing abstractions, such as process or VM, are too heavyweight.

How can compilers and profilers help? Compilers and profilers could help developers decompose their code into resource proclets, by providing semi-automated guidance based on static and dynamic analyses. With a more constrained programming model, compilers can also help validate the fine granularity of resource proclet decomposition. Profilers will also help identify non-performant resource proclets, while runtime analyses will guide migration policy.

6 Related Work

Improving Datacenter Utilization. Resource disaggregation is a trending approach to improve datacenter utilization. Existing work has demonstrated the feasibility of disaggregating memory [4, 23, 45, 51, 61], storage [29–31, 34], and accelerators [10, 13, 57, 59]. Quicksand has a complementary approach: rather than disaggregating resources, it makes applications fungible. Another line of work improves utilization by harvesting idle server resources; this includes CPUs [63], memory [19], and storage [48]. However, resource harvesting is limited to best-effort restartable applications, as the harvested resources can be forcibly reclaimed anytime under resource pressure.

Resource Decoupling. Existing work such as Monotasks [41, 42] also decouples a job into pieces that each dominantly consumes a single type of resource, similar to Quicksand's resource proclets. However, their main design goal is to achieve a better performance visibility rather than a better resource efficiency through disaggregation.

Live Migration. Existing work live-migrate jobs across machines to balance load [14] and optimize cost [54] over a long time horizon. They conduct migration at coarse granularity—such as VM [14, 26, 28], container [44, 56], and process [11]—which takes more than hundreds of milliseconds. Slow migration impacts the service level agreement of applications and makes it practical only in latency-insensitive scenarios. Quicksand builds atop Nu [50], a recent system that supports fine-grained migration at the proclet level. Nu enables sub-millisecond migration with little performance disruption, thereby enabling Quicksand to fill in small gaps in resource utilization.

Training and Data Analytics Pipelines. Existing ML training pipelines [2, 20, 62] and data analytics pipelines [21, 53] also decouple state from compute. Compared to Quicksand, they are less general and flexible: each of them targets a specific application scenario and a specific machine resource distribution. With decoupling, these pipelines also support independent auto-scaling at each layer to balance performance; for example, Cachew (an ML training pipeline) [21] is able to automatically scale the data preprocessing stage to keep training GPUs/TPUs saturated. Compared to Quicksand, they target a much longer timescale; for example, Cachew takes minutes to reach an equilibrium whereas Quicksand takes only tens of milliseconds (§4).

7 Conclusion

Utility computing is in reality multiple utilities that are bundled together by cloud providers at large granularities in an imperfect offering. As applications consume these resources at variable rates and ratios, resources are often underutilized, either because applications allocate them but do not consume them (resource overprovisioning) or because the cloud cannot offer the resources in a usable form (resource stranding). We are trying to address this problem with Quicksand, by making applications fungible so that they can be distributed, scaled, and migrated to fit the availability of individual resource for even a few milliseconds, despite changing resource demands by applications and resource availability in the cloud. Early experience with a DNN training pipeline shows that this approach is promising.

Acknowledgements

We thank the anonymous reviewers, as well as members of the PDOS group at MIT and the ETOS group at Brown University, for their helpful feedback. We appreciate CloudLab [16] for providing the experiment platform. This work was funded in part by a Facebook Research Award, Google Faculty Awards, the DARPA FastNICs program under contract #HR0011-20-C-0089, the NSF under awards CNS-2104398 and CNS-2045170, and VMware.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. "Slicer: Auto-Sharding for Datacenter Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pages 739–753.

[2] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. "POLARIS: The Distributed SQL Engine in Azure Synapse". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pages 3204–3216.

[3] Mohammad Alian and Nam Sung Kim. "NetDIMM: Low-Latency Near-Memory Network Interface Architecture". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pages 699–711.

[4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. "Can Far Memory Improve Job Throughput?" In: *European Conference on Computer Systems (EuroSys)*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020.

[5] *Amazon EC2 Spot Instances Pricing*. https://aws.amazon.com/ec2/spot/pricing/.

[6] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. "Providing slos for resource-harvesting vms in cloud platforms". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pages 753–768.

[7] Dan Ardelean, Amer Diwan, and Chandra Erdman. "Performance Analysis of Cloud Applications". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018, pages 405–417.

[8] *AWS Lambda: Memory and computing power*. https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[9] Benjamin Brock, Aydın Buluç, and Katherine Yelick. "BCL: A Cross-Platform Distributed Data Structures Library". In: *International Conference on Parallel Processing (ICPP)*. ICPP '19. Kyoto, Japan: Association for Computing Machinery, 2019.

[10] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A Cloud-Scale Acceleration Architecture". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Oct. 2016.

[11] *Checkpoint/Restore In Userspace (CRIU)*. https://criu.org/Main_Page.

[12] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. "Overload Control for µs-Scale RPCs with Breakwater". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. OSDI'20. USA: USENIX Association, 2020.

[13] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38 (Mar. 2018), pages 8–20.

[14] 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)*. 2005, pages 273–286.

[15] *CXL Consortium*. https://www.computeexpresslink.org.

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

[17] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. "Dhalion: Self-Regulating Stream Processing in Heron". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pages 1825–1836.

[18] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. "Caladan: Mitigating Interference at Microsecond Timescales". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. OSDI'20. USA: USENIX Association, 2020.

[19] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. "Memory-Harvesting VMs in Cloud Platforms". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pages 583–594.

[20] Sunny Gakhar, Joyce Cahoon, Wangchao Le, Xiangnan Li, Kaushik Ravichandran, Hiren Patel, Marc Friedman, Brandon Haynes, Shi Qiao, Alekh Jindal, and Jyoti Leeka. "Pipemizer: An Optimizer for Analytics Data Pipelines". In: *Proceedings of the VLDB Endowment (PVLDB)*. Sept. 2022.

[21] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. "Cachew: Machine Learning Input Data Processing as a Service". In: *USENIX Annual Technical Conference (ATC)*. Carlsbad, CA: USENIX Association, July 2022, pages 689–706.

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

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX Association, Mar. 2017, pages 649–667.

[24] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pages 202–215.

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

[26] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. "Post-Copy Live Migration of Virtual Machines". In: *SIGOPS Oper. Syst. Rev.* 43.3 (July 2009), pages 14–26.

[27] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. "The nanoPU: A Nanosecond Network Stack for Datacenters." In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2021, pages 239–256.

[28] Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger. "Efficient Live Migration of Virtual Machines Using Shared Storage". In: *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. VEE '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pages 41–50.

[29] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. "Flash Storage Disaggregation". In: *European Conference on Computer Systems (EuroSys)*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016.

[30] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. "ReFlex: Remote Flash ≈ Local Flash". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pages 345–359.

[31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA: USENIX Association, Oct. 2018, pages 427–444.

[32] *Kubernetes Horizontal Pod Autoscaling*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[33] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. "Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pages 36–51.

[34] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and

Antony Rowstron. "Understanding Rack-Scale Disaggregated Storage". In: *Workshop on Hot Topics in Storage and File Systems (HotStorage)*. Santa Clara, CA: USENIX Association, July 2017.

[35] 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: *ACM Symposium on Operating Systems Principles (SOSP)*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pages 137–152.

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

[37] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. "Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA: USENIX Association, July 2022, pages 579–596.

[38] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. "Analyzing and Mitigating Data Stalls in DNN Training". In: *Proc. VLDB Endow.* 14.5 (Mar. 2021), pages 771–784.

[39] 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: *ACM Symposium on Operating Systems Principles (SOSP)*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pages 521–537.

[40] Ed Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. "Flat Datacenter Storage". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Oct. 2012.

[41] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. "Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks". In: *ACM Symposium on Operating Systems Principles (SOSP)*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pages 184–200.

[42] Kay Ousterhout, Christopher Canel, Max Wolffe, Sylvia Ratnasamy, and Scott Shenker. "Performance Clarity as a First-Class Design Principle". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pages 1–6.

[43] Seo Jin Park, Joshua Fried, Sunghyun Kim, Mohammad Alizadeh, and Adam Belay. "Efficient Strong Scaling Through Burst Parallel Training". In: *Proceedings of Machine Learning and Systems (MLSys)*. Edited by D. Marculescu, Y. Chi, and C. Wu. Volume 4. 2022, pages 748–761.

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

[45] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. "Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX Association, Apr. 2023, pages 181–198.

[46] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. "Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pages 199–205.

[47] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. "TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System". In: *ACM Trans. Comput. Syst.* 31.1 (Feb. 2013).

[48] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. "BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA: USENIX Association, July 2022, pages 17–33.

[49] Zhenyuan Ruan, Tong He, and Jason Cong. "INSIDER: Designing in-Storage Computing System for Emerging High-Performance Drive". In: *USENIX Annual Technical Conference (ATC)*. USENIX ATC '19. Renton, WA, USA: USENIX Association, 2019, pages 379–394.

[50] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. "Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX Association, Apr. 2023, pages 1409–1427.

[51] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory". In: *Symposium*

on Operating Systems Design and Implementation (OSDI). USENIX Association, Nov. 2020, pages 315–332.

[52] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. "Autopilot: Workload Autoscaling at Google". In: *European Conference on Computer Systems (EuroSys)*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020.

[53] *Scaling data ingestion for machine learning training at Meta*. https://engineering.fb.com/2022/09/19/ml-applications/data-ingestion-machine-learning-training-meta/.

[54] Supreeth Shastri and David Irwin. "HotSpot: Automated Server Hopping in Cloud Spot Markets". In: *ACM Symposium on Cloud Computing (SoCC)*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, pages 493–505.

[55] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. "Discovering and Exploiting Program Phases". In: *IEEE Micro* 23.6 (2003), pages 84–93.

[56] Radostin Stoyanov and Martin J Kollingbaum. "Efficient live migration of linux containers". In: *High Performance Computing: High Performance International Workshops (ISC)*. Springer. 2018, pages 184–193.

[57] *Training on TPU Pods*. https://cloud.google.com/tpu/docs/training-on-tpu-pods.

[58] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-Scale Cluster Management at Google with Borg". In: *European Conference on Computer Systems (EuroSys)*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015.

[59] *VMware vSphere Bitfusion Documentation*. https://docs.vmware.com/en/VMware-vSphere-Bitfusion/index.html.

[60] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache". In: *USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA: USENIX Association, Feb. 2020, pages 267–281.

[61] 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: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Nov. 2020, pages 261–280.

[62] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. "On-Demand State Separation for Cloud Data Warehousing". In: *Proc. VLDB Endow.* 15.11 (July 2022), pages 2966–2979.

[63] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. "Faster and Cheaper Serverless Computing on Harvested Resources". In: *ACM Symposium on Operating Systems Principles (SOSP)*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pages 724–739.

[64] Yiying Zhang, Ardalan Amiri Sani, and Guoqing Harry Xu. "User-Defined Cloud". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. May 2021.