No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing

Xingda Wei[1,2], Fangming Lu[1], Tianxia Wang[1], Jinyu Gu[1], Yuhan Yang[1], Rong Chen[1,2], and Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory

Abstract

Serverless platforms essentially face a tradeoff between container startup time and provisioned concurrency (i.e., cached instances), which is further exaggerated by the frequent need for remote container initialization. This paper presents Mitosis, an operating system primitive that provides fast remote fork, which exploits a deep codesign of the OS kernel with RDMA. By leveraging the fast remote read capability of RDMA and partial state transfer across serverless containers, Mitosis bridges the performance gap between local and remote container initialization. Mitosis is the first to fork over 10,000 new containers from one instance across multiple machines within a second, while allowing the new containers to efficiently transfer the pre-materialized states of the forked one. We have implemented Mitosis on Linux and integrated it with Fn, a popular serverless platform. Under load spikes in real-world serverless workloads, Mitosis reduces the function tail latency by 89% with orders of magnitude lower memory usage. For serverless workflow that requires state transfer, Mitosis improves its execution time by 86%.

1 Introduction

Serverless computing is an emerging cloud computing paradigm supported by major cloud providers, including AWS Lambda [23], Azure Functions [91], Google Serverless [44], Alibaba Serverless Application Engine [30] and Huawei Cloud Functions [58]. One of its key promises is *auto-scaling*—users only provide serverless functions, and serverless platforms automatically allocate computing resources (e.g., containers[1]) to execute them. Auto-scaling makes serverless computing economical: the platform only bills when functions are executed (no charge for idle time).

However, *coldstart* (i.e., launching a container from scratch for each function) is a key challenge for fast auto-scaling, as the start time (over 100 ms) can be orders of magnitude higher than the execution time for ephemeral serverless functions [37, 94, 121]. Accelerating coldstart has become a hot topic in both academia and industry [41, 122, 94, 17, 102, 37, 20]. Most of them resort to a form of 'warmstart' by *provisioned concurrency*, e.g., launching a container from a cached one. However, they require non-trivial resources when scaling functions to a distributed setting, e.g., each machine should deploy many cached containers.

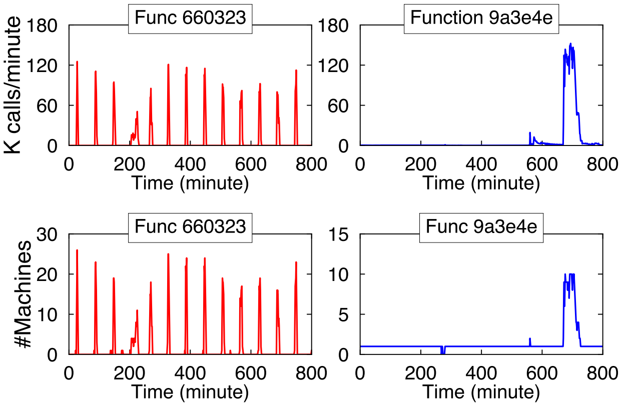


Figure 1. *The timelines of call frequency (top) and sufficient resource provisioning (bottom) for two serverless functions in a real-world trace from Azure Functions [102].*

Unfortunately, scaling functions to multiple machines is common because a single machine has a limited function capacity to handle the timely load spikes. Consider the functions sampled from real-world traces of Azure Functions [102]. The request frequency of function `9a3e4e` can surge to over 150 K calls per minute, increased by 33,000× within one minute (see the top of Figure 1). To avoid stalling numerous newly arriving function calls, the platform should immediately launch sufficient containers across multiple machines (see the bottom part of Figure 1). Due to the unpredictable nature of the serverless workload, it is challenging for the platform to decide the number of cached instances for the warmstart. Hence, there is "no free lunch" for such resources: commercial platforms require users to reserve and pay for them to achieve better performance (i.e., lower response time), e.g., AWS Lambda Provisioned Concurrency [12].

Even worse, dependent functions that run in separate containers cannot directly transfer states. Instead, they must resort to message passing or cloud storage for state transfer, which introduces data serialization/de-serialization, memory copy and storage stack overheads. Recent reports have shown that these may count up to 95% of the function execution

---

[1] We focus on executing serverless functions with containers in this paper, which is widely adopted by existing platforms [122, 123, 54, 64].

time [71, 53]. Unfortunately, transferring states between functions is common in serverless workflows—a mechanism to compose functions into more complex applications [4, 2]. Though recent research [71] bypasses these overheads for local state transfer (i.e., functions that run on the same machine) by co-locating local functions in the same container, it is still unclear how to do so in a remote setting.

We argue that *remote fork* (forking containers across machines like a local fork) is a promising primitive to enable both efficient function launching and fast function state sharing. First, the fork mechanism has been shown efficient in both performance and resource usage for launching containers on a single machine: one cached container is sufficient to start numerous containers with 1 ms time [17, 37, 36]. By extending the fork mechanism to remote, one active container is sufficient to start numerous containers efficiently on all the machines, achieving *no provisioned concurrency* in a distributed setting. Second, remote fork provides transparent intermediate state sharing between remote functions: the code in the container created by the fork can access the pre-materialized states of the forked container transparently bypassing message passing or cloud storage.

However, state-of-the-art systems can only achieve a conservative remote fork with Checkpoint/Restore techniques (C/R) [7, 117]. Our analysis reveals that they are not efficient for serverless computing, i.e., even slower than coldstart due to the costs of checkpointing the memory of parent container into files, transferring the files through the network and accessing the files through a distributed file system (§3). Even though we have utilized modern interconnects (i.e., RDMA) to reduce these costs, the software overhead of checkpointing and distributed file accesses still make C/R underutilize the low latency and high throughput of RDMA.

We present MITOSIS, an operating system primitive that provides a fast *remote fork* by deeply co-designing with RDMA. The key insight is that the OS can directly access the physical memory on remote machines via RDMA-capable NICs (RNICs) [115], which is extremely fast thanks to bypassing remote OS and remote CPU. Therefore, we can realize remote fork by imitating local fork through mapping a child container's virtual memory to its parent container's physical memory without checkpointing the memory. The child container can directly read the parent memory in a copy-on-write fashion using RNIC, bypassing the software stacks (e.g., distributed file system) introduced by traditional C/R.

Leveraging RDMA for remote fork with kernel poses several new challenges (§4.1): (1) fast and scalable RDMA-capable connection establishment, (2) efficient access control of the parent container's physical memory and (3) efficient parent container lifecycle management at scale. MITOSIS addresses these challenges by (1) retrofitting advanced RDMA feature (i.e., DCT [1]), (2) proposing a new connection-based memory access control method designed for remote fork and (3) co-designing container lifecycle management with the help of serverless platform. We also introduce techniques including generalized lean container [94] to reduce containerization overhead for the remote fork. In summary, we show that remote fork can be made efficient, feasible and practical on commodity RNICs for serverless computing.

We implemented MITOSIS on Linux with its core functionalities written in Rust as a loadable kernel module. It can remote-fork $10,000$ containers on 5 machines within 0.86 second. MITOSIS is fully compatible with mainstream containers (e.g., runC [13]), making integration with existing container-based serverless platforms seamlessly. To demonstrate the efficiency and efficacy, we integrated MITOSIS with Fn [123], a popular open-source serverless platform. Under load spikes in real-world serverless workloads, MITOSIS reduces the 99^{th} percentile latency of the spiked function by 89% with orders of magnitude lower memory usage. For a real-world serverless workflow (i.e., FINRA [14]) that requires state transfer, MITOSIS reduces its execution time by 86%.

Contributions. We highlight the contributions as follows:

- **Problem**: An analysis of the performance-resource provisioning trade-off of existing container startup techniques, and the costs of state transfer between functions (§2).
- **MITOSIS**: An RDMA-co-designed OS remote fork that quickly launches containers on remote machines without provisioned concurrency and enables efficient function state transfer (§4–5).
- **Demonstration**: An implementation on Linux integrated with Fn (§6) and evaluations on both microbenchmarks and real-world serverless applications demonstrate the efficacy of MITOSIS (§7). MITOSIS is publically available at `https://github.com/ProjectMitosisOS`.

2 Background and Motivation

2.1 Serverless computing and container

Serverless computing is a popular programming paradigm. It abstracts resource management from the developers: they only need to write the application as *functions* in a popular programming language (e.g., Python), upload these *functions* (as container images) to the platform, and specify how to call them. The platform can *auto-scale* according to function requests by dynamically spawning a container [54, 123, 59, 94, 22, 30, 91, 44, 22, 70][2] to handle each call. The spawned containers will also be automatically reclaimed after functions return, making serverless economical: the developers only pay for the in-used containers.

Container is a popular host for executing functions. It not only packages the application's dependencies into a single image that eases the function deployment, but also provides lightweight isolation through Linux's `cgroups` and `namespaces`, which is necessary to run applications in a multi-tenancy environment. Unfortunately, enabling container

---

[2] Serverless platform may use virtual machines to run functions, which is not the focus of this paper.

Table 1: *A comparison of startup techniques for autoscaling n concurrent invocations of one function to m machines.* Local *means the resources for the startup are provisioned on the function execution machine. The function is a simple python program that prints 'hello world'.*

	Coldstart	**Caching**	**Fork**	**Checkpoint/Restore**	**Remote fork**
	[9, 119]	[63, 123, 94, 102, 122]	[37, 17, 36]	[120, 37, 117, 20]	MITOSIS
Local startup performance	Very slow ($100\,ms$)	Very fast ($< 1\,ms$)	Fast ($1\,ms$)	Medium ($5\,ms$)	Fast ($1\,ms$)
Remote startup performance	Very slow ($1,000\,ms$)	N/A	N/A	Slow ($24\,ms$)	Fast ($3\,ms$)
Overall resource provisioning	$O(1)$	$O(n)$	$O(m)$	$O(1)$	$O(1)$

introduces additional function startup costs and state transferring costs due to container bootstrap and segregated function address spaces, respectively.

2.2 Startup and resource provisioning costs

Coldstart performance cost. Starting a container from scratch, commonly named as '*coldstart*', is notoriously slow. The startup includes pulling the container image, setting up the container configurations and initializing the function language runtime. All the above steps are costly, which take even more than hundreds of milliseconds [37, 94]. As a result, coldstart may dominate the end-to-end latency of ephemeral serverless functions [37, 94, 119, 33]. For example, Lambda@Eedge reports that 67% of its functions run in less than 20 ms [33]. In comparison, starting a Hello-world python container with runC [13]—a state-of-the-art container runtime—takes 167 ms and 1783 ms when the container image is stored locally and remotely, respectively (see Table 1).

Warmstart resource cost due to provisioned concurrency. A wealth of researches focus on reducing the startup time of coldstart with '*warmstart*' techniques [94, 17, 37, 102, 113, 42, 119, 131, 106]. However, they must pay more resource provisioning cost (see Table 1):

Caching [63, 64, 123, 41, 122, 94, 17, 102]. By caching finished containers (e.g., via Docker `pause` [8]) instead of reclaiming them, future functions can reuse cached ones (e.g., via Docker `unpause`) with nearly no startup cost (less than 1 ms). However, Caching consumes large in-memory resources: the resource provisioned—number of the cached instances ($O(n)$) should match the number of concurrent functions (n), because a paused container can only unpause for one function. Given the unpredictability of the number of function invocations (e.g., load spikes in Figure 1), it is challenging for the developers or the platform to decide how many cached instances are required. Thus, Caching inevitably faces the trade-off between fast startup and low resource provisioning, resulting in huge cache misses.

Fork [37, 17, 36]. A cached container (*parent*) can call the `fork` system call (instead of `unpause`) to start new containers (*children*). Since fork can be called multiple times, each machine only requires one cached instance to fork new containers. Thus, fork reduces resource provisioned of Caching—cached containers from $O(n)$ to $O(m)$, where m is the number of machines that require function startup. However, it is still proportional to the number of machines (m) since fork cannot generalize to a distributed setting.

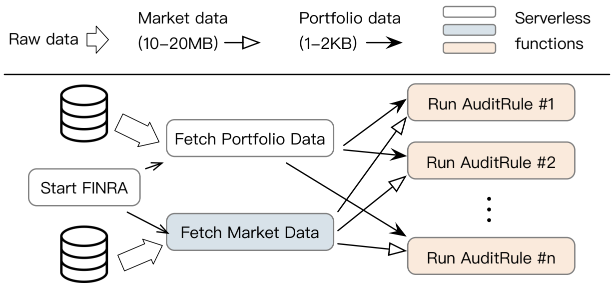


Figure 2. *The workflow graph of a real-world serverless application, Financial Industry Regulatory Authority, FINRA [14].*

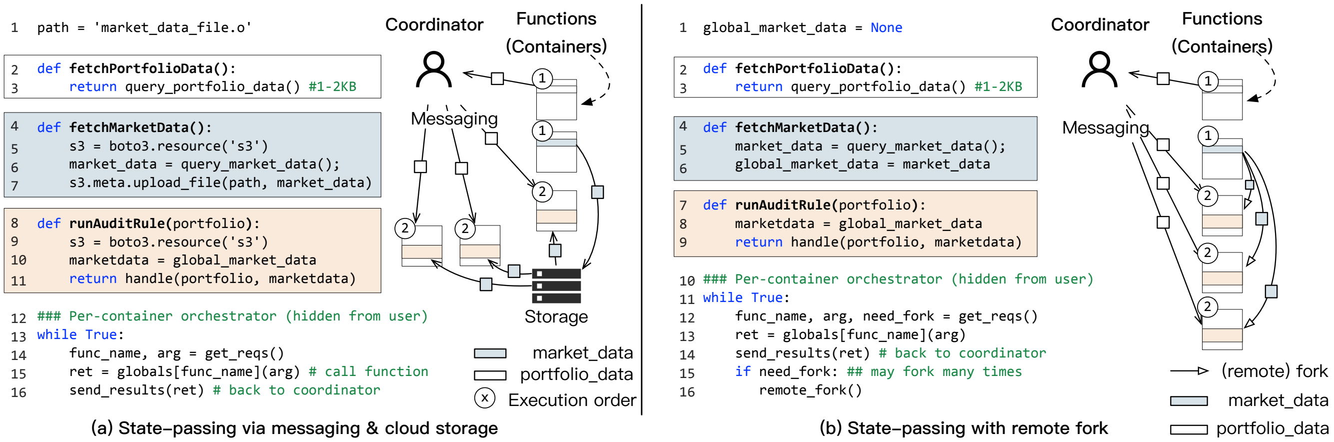
Checkpoint/Restore (C/R) [120, 37, 117]. C/R starts containers from container checkpoints stored in a file. It only needs $O(1)$ resource (the file) to warmstart, because the file can be transferred through the network if necessary. Though being optimal in resource usage, C/R is orders of magnitude slower than Caching and fork. We analyze it in §3 in detail.

2.3 (Remote) state transfer cost

Transferring states between functions is common in serverless workflows [36, 17, 95, 64, 4, 2]. A workflow is a graph describing the producer-consumer relationships between functions. Consider the real-world example FINRA [14] shown in Figure 2. It is a financial application that validates trades according to the trade (Portfolio) and market (Market) data. Upstream functions (the ones that produce states), i.e., `fetchPortfolioData` and `fetchMarketData` first read data from external sources. Afterward, they transfer the results to many downstream functions (the one that consumes states), i.e., `runAuditRules` to process them concurrently for a better performance.

Functions run in different containers can only transfer states either by copying them through the network via message passing or exchanging them at a cloud storage service. Figure 3 (a) shows a simplified code for running FINRA on AWS Lambda. For small states transfers (less than 32KB, e.g., Portfolio), Lambda piggybacks the states in messages exchanged between the coordinator and the function containers [131]. For large ones (Market), functions must exchange them with S3—Lambda's cloud storage service.

Transferring states via messages and cloud storage inevitably faces the overheads of data serialization, memory



```
1  path = 'market_data_file.o'
2  def fetchPortfolioData():
3      return query_portfolio_data() #1-2KB
4  def fetchMarketData():
5      s3 = boto3.resource('s3')
6      market_data = query_market_data();
7      s3.meta.upload_file(path, market_data)
8  def runAuditRule(portfolio):
9      s3 = boto3.resource('s3')
10     marketdata = global_market_data
11     return handle(portfolio, marketdata)
12 ### Per-container orchestrator (hidden from user)
13 while True:
14     func_name, arg = get_reqs()
15     ret = globals[func_name](arg) # call function
16     send_results(ret) # back to coordinator
```

(a) State–passing via messaging & cloud storage

```
1  global_market_data = None
2  def fetchPortfolioData():
3      return query_portfolio_data() #1-2KB
4  def fetchMarketData():
5      market_data = query_market_data();
6      global_market_data = market_data
7  def runAuditRule(portfolio):
8      marketdata = global_market_data
9      return handle(portfolio, marketdata)
10 ### Per-container orchestrator (hidden from user)
11 while True:
12     func_name, arg, need_fork = get_reqs()
13     ret = globals[func_name](arg)
14     send_results(ret) # back to coordinator
15     if need_fork: ## may fork many times
16         remote_fork()
```

(b) State–passing with remote fork

Figure 3. *(a) A simplified code of FINRA (see Figure 2) on existing serverless platforms. (b) A simplified code of using (remote) fork to transfer states between FINRA functions.* `globals` *records a mapping between function name and its pointer.*

copies, and cloud storage stacks, causing up to a 1,000X slowdown [53, 71]. To cope with the issue, existing work proposes serverless-optimized messaging primitives [17] or specialized storage systems [110, 69, 96], but none of the mentioned overhead is completely eliminated [71]. Faastlane [71] co-locates functions in the same container with *threads* so that it can bypass these overheads with shared memory accesses. However, threads cannot generalize to a distributed setting. Faastlane fallbacks to message passing if the upstream and downstream functions are on different machines. SPRIGHT [97] achieves a similar effect by retrofitting eBPF. However, they don't support efficient data sharing across nodes.

3 Remote Fork for Serverless Computing

We show the following two benefits of *remote fork* to address the issues mentioned in the previous section.

Efficient (remote) function launching. When generalizing the FORK primitive to a remote setting, a single *parent* container is sufficient to launch subsequent *child*[3] containers across the cluster, similar to C/R (see Table 1). We believe $O(1)$ resource provisioning is desirable for the developers/tenants since they only need to specify whether they need resource for warmstart, instead of how many (e.g., the number of machines for forking or cached instances [12] for Caching).

Fast and transparent (remote) state transfer. The FORK primitive essentially bridges the address spaces of parent and child containers. The transferred states are pre-materialized in the parent memory, so the child can seamlessly access them with shared memory abstraction with no data serialization, zero-copy (for read-only accesses[4]) and cloud storage costs. Meanwhile, the *copy-on-write* semantic in the FORK primitive avoids the costly memory coherence protocol in traditional distributed shared memory systems [75, 57].

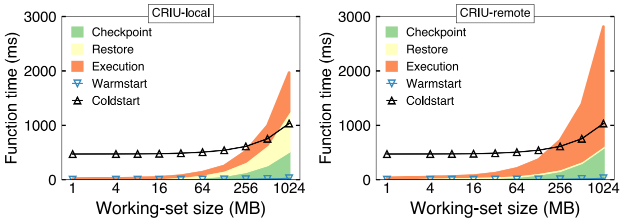


Figure 4. *Analysis of using C/R for remote fork.* **Setup***: CRIU-local: CRIU with a local file system (e.g., tmpfs), which uses RDMA to transfer files between machines. CRIU-remote: CRIU with an RDMA-accelerated distributed file system (e.g., Ceph [89]).*

Figure 3 (b) presents a concrete example of using fork to transfer market data in FINRA (see Figure 2). In this setup, all functions are packaged in the same container, which has an orchestrator dispatching function requests to user-implemented functions (lines 11–14).[5] We further assume the coordinator issuing requests to the orchestrators is fork-aware (§6.1): based on the function dependencies in the workflow graph (e.g., Figure 2), it will request the orchestrator to fork children if necessary (line 12). After the orchestrator finishes `fetchMarketData` (line 13), it forks (lines 15–16) to run downstream functions (`runAuditRule`), which can directly access the `global_market_data` pre-materialized by the parent (line 8).

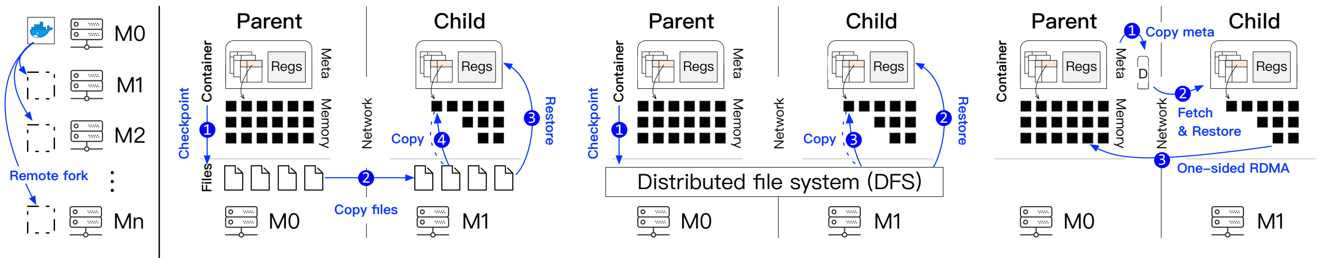
Challenge: remote fork efficiency. To the best of our knowledge, existing containers can only remote fork with a C/R-based approach [108, 32]. To fork a child, the parent first *checkpoints* its states (e.g., register values and memory pages) by copying them to a file, and then *transfers* the file to the child—either using a remote file copy—see *CRIU-local* in Figure 5 (a), or a distributed file system (see *CRIU-remote* in Figure 5 (b)). After receiving the file, the child *restores* the parent's execution by loading the container states from the checkpointed file. Note that C/R may load some states (i.e., memory pages) on-demand for better performance [120].

---

[3] We may also call the kernel/machine hosting the parent/child container as *parent/child* in this paper without losing generality.

[4] In the case of the traditional fork. MITOSIS further optimizes with one-sided RDMA (§4), allowing zero-copy even for read-write accesses.

[5] This setup is common in serverless platforms [70, 71, 2].



Goal (a) C/R with tmpfs (CRIU–local) (b) C/R with a DFS (CRIU–remote) (c) Mitosis

Figure 5. *An overview of different approaches to achieve ultra-fast remote fork, including (a) C/R with a local filesystem (e.g., tmpfs), (b) C/R with a fast distributed filesystem (e.g., Ceph [5]), and (c)* Mitosis*.*

Unfortunately, the C/R-based remote fork is not efficient enough for serverless computing. Figure 4 (a) shows the execution time of serverless functions on a remote machine using CRIU [7]—the state-of-the-art C/R on Linux (with careful optimizations, see §7 for details) to realize CRIU-local and CRIU-remote. The synthetic function randomly touches the entire parent's memory. We observe that C/R-based remote fork can even be 2.7 × slower than coldstart if it accesses 1 GB remote memory. We attribute it to one or more of the following aspects.

Checkpoint container memory. CRIU takes 9 ms (resp. 518 ms) and 15.5 ms (resp. 590 ms) to checkpoint 1 MB (resp. 1 GB) memory of the parent container using local or distributed file systems, respectively. The overhead is dominated by copying the memory to the files: unlike the local fork, the child's OS resides on another machine and thus, lacks direct memory access capability to the parent's memory pages.

Copy checkpointed file. For CRIU-local, transferring the entire file from the parent to the child takes 11–734 ms for 1 MB–1 GB image (compared to the 0.61–570 ms execution time), respectively. The whole file copy is typically unnecessary since serverless functions typically access a partial state of the parent container [120] (see also Figure 16 (b)).

Additional restore software overhead. CRIU-remote enables on-demand file transfer[6]: it only reads the required remote file pages during page faults. However, the execution time is 1.3–3.1× longer than CRIU-local because each page fault requires a DFS request to read the page: the DFS latency (100 μs) is much higher than local file accesses. More importantly, the latency is much higher than one network round-trip time (3 μs) due to the software overhead.

4 The Mitosis Operating System Primitive

Opportunity: kernel-space RDMA. Remote Direct Memory Access (RDMA) is a fast networking feature widely deployed in data-centers [115, 47, 43]. Though commonly used in the user-space, RDMA further gives the kernel the ability to read/write the *physical memory* of remote machines [115] bypassing remote CPUs (i.e., one-sided RDMA READ), with low latency (e.g., 2 μs) and high bandwidth (400 Gbps).

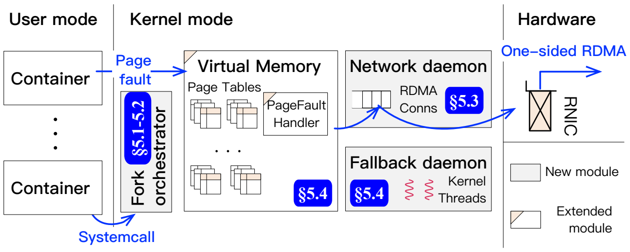


Figure 6. *The* Mitosis *architecture.*

Approach: imitate fork with RDMA. Mitosis achieves an efficient remote fork by imitating the local fork with RDMA. Figure 5 (c) shows an overview. First, we copy the parent's metadata (e.g., page table) to a condensed descriptor (§5.1) to fork a child (❶). Note that unlike C/R, we don't copy the parent's memory pages to the descriptor. The descriptor is then copied to the child via RDMA to recover the parent's metadata, similar to `copy_process` in the local fork (❷). During execution, we configure the child's remote memory accesses to trigger page faults, and the kernel will read the remote pages accordingly. The fault handler is triggered naturally in an on-demand pattern, which avoids transferring the entire container state. Meanwhile, Mitosis directly uses one-sided RDMA READ to read the remote physical memory (❸), bypassing all the software overheads.

Architecture. We target a decentralized architecture—each machine can fork from others and vice versa. Note that we do not require dedicated resources (e.g., pinned memory) to fork containers, thus, non-serverless applications can co-run with Mitosis. We realize Mitosis by adding four components to the kernel (see Figure 6): The *fork orchestrator* rehearsals the remote fork execution (§5.1 and 5.2). The *network daemon* manages a scalable RDMA connection pool (§5.3) for communicating between kernels. We extend OS's *virtual memory subsystems* to utilize the remote memory with RDMA (§5.4). Finally, *fallback daemon* provides RPC handlers to restore rare remote memory accesses that cannot utilize RDMA.

Security model. We preserve the security model of containers, i.e., the OS and hardware (RNIC) are trustworthy while malicious containers (functions) may exist.

---

[6] CRIU lazy migration [6] also supports on-demand transfer. However, it is not optimized for RDMA and is orders of magnitude slower than our evaluated CRIU-remote (210 vs. 42 ms) for the python hello function.

4.1 Challenges and approaches

Efficient and scalable RDMA connection setup. Though RDMA is fast (e.g., 2 μs), it is traditionally only supported in the connection-oriented transport (RC) [35, 83, 126, 125, 105, 127, 124], where connection establishment is much slower (e.g., 4 ms [11] with a limited 700 connections/second throughput). Caching connections to other machines can mitigate the issue, but it is impractical when RDMA-capable clusters have scaled to more than 10,000 nodes [43].

We retrofit DCT [1], an underutilized but widely supported advanced RDMA feature with fast and scalable connection setups to carry out communications between kernels (§5.3).

Efficient remote physical memory control. Mitosis exposes the parent's physical memory to the children for the fastest remote fork. However, this approach introduces consistency problems in corner cases. If the OS changes a parent's virtual–physical mappings [77, 80, 78, 79] (e.g., swap [78]), the children will read an incorrect page. User-space RDMA can use memory registration (MR) [93] for the access control. However, MR has non-trivial registration overheads [49]. Further, kernel-space RDMA has limited support for MR—it only supports MR on RCQP (with FRMR [90]).

We propose a registration-free memory control method (§5.4) that transforms RNIC's memory checks to connection permission checks. We further make the checks efficient by utilizing DCT's scalable connection setup feature.

Parent container lifecycle management. For correctness, we must ensure a forked container (parent) is alive until all its successors (including children forked from the children) finish. A naive approach is letting each machine track the lifecycles of the successors of its hosting parents. However, it would pose significant management burdens: a parent's successors may span multiple machines, forming a distributed *fork tree*. Meanwhile, each machine may have multiple trees. Consequently, each machine needs extensive communications with the others following paths in the trees to ensure a parent can be safely reclaimed.

To this end, we onload the lifecycle management to the serverless platform (§6.3). The observation is that serverless coordinators (nodes that invoke functions via fork) naturally maintain the runtime information of the forked containers. Thus, they can trivially decide when to reclaim parents.

5 Design and Implementation

For simplicity, we first assume one-hop fork (i.e., no cascading) and then extend to multi-hops fork (see §5.5).

API. We decouple the fork into two phases (see Figure 7): The user can first call `fork_prepare` to generate the parent's metadata (called *descriptor*) related to remote fork. The descriptor is globally identified by the local unique `handle_id` and `key` (generated and returned by the prepared call) and the parent machine's RDMA address. Given the identifier, users can start a child via `fork_resume` at another machine (can be the same as the parent, i.e., local fork).

```
// Prepare the container descriptor at the parent machine
status_t fork_prepare(uint64_t *handler_id, uint64_t *key);

// Resume from a parent descriptor at the child machine
status_t fork_resume(char *addr, uint64_t handler_id, uint64_t key);
```

Figure 7. *The major* Mitosis *remote fork system calls.*

Compared to the traditional one-stage fork system call, a two-phase fork API (prepare and resume)—similar to `pause` and `unpause` in Caching is more flexible for serverless computing. For example, after preparing and recording the parent's identifier at the coordinator, it can later start children without communicating with the parent machine.

Visibility of the parent's data structures. By default, Mitosis exposes all the parent's data structures—including virtual memory and file descriptors, to the child after `fork_prepare`. Mitosis could introduce APIs to let the application limit the scope of the exposure, but currently, we find it unnecessary: parents must trust the children since they are from the same application.

5.1 Fork prepare

`fork_prepare` will generate a local in-memory data structure (*container descriptor*) capturing the parent states, which contains (1) cgroup configurations and namespace flags—for containerization, (2) CPU register values—for recovering the execution states, (3) page table and virtual memory areas (VMAs)—for restoring the virtual memory, and (4) opened file information—for recovering the I/O. We follow local fork (e.g., Linux's `copy_process()`) to capture (1)–(3) and CRIU [7] for (4). Since deciding when to reclaim a descriptor is challenging, we always keep the prepared parents (and their descriptors) alive unless the serverless platform explicitly frees them (i.e., via `fork_reclaim`).

Though the descriptor plays a similar role as C/R checkpointed file, we emphasize one key difference: the descriptor only stores the page table, not the memory pages. As a result, it is orders of magnitude smaller (KB vs. MB) and orders of magnitude faster to generate and transfer.

5.2 Fork resume

`fork_resume` resumes the parent's execution state by fetching the parent descriptor and then restoring from it. We now describe how to make the above two steps fast. For now, we assume the child OS has established network connections capable of issuing RPCs and one-sided RDMAs to the parent. The next section describes the connection setup.

Fast descriptor fetch with one-sided RDMA. A straightforward implementation of fetching the descriptor is using RPC. However, RPC's memory copy overhead is non-trivial (see Figure 18), as the descriptor of a moderate-sized container may consume several KBs. The ideal fetch is using one one-sided RDMA READ, which requires (1) storing the parent's descriptor into a consecutive memory area and (2) informing the child's OS of the memory's address and size in advance.

The first requirement can be trivially achieved by serializing the descriptor into a well-format message. Data serialization has little cost (sub-millisecond) due to the simple data structure of descriptor. For the second requirement, a naive solution is to encode the memory information in the descriptor identifier (e.g., `handler_id`) that is directly passed to the resume system call. However, this approach is insecure because a malicious user could pass a malformed ID, causing the child to read and use a malformed descriptor. We adopt a simple solution to remedy this: Mitosis will send an authentication RPC to query the descriptor memory information with the descriptor identifier. If the authentication passes, the parent will send back the descriptor's stored address and payload so that the child can directly read it with one-sided RDMA. We chose a simple design because the overhead of an additional RPC (several bytes) is typically negligible: reading the descriptor (several KBs) will dominate the fetch time.

Fast restore with generalized lean containers. With the fetched descriptor, child OS uses the following two steps to resume a child to the parent's execution states: (1) Containerization: set the `cgroups` and `namespaces` to match the parent's setup; (2) Switch: replace the caller's CPU registers, page table, and I/O descriptors with the parent's. The switch is efficient (finishes in sub-milliseconds): it just imitates the local fork—e.g., unmapping the caller's current memory mapping and mapping the child's virtual memory to the parents by copying parent's page table to the child . On the other hand, containerization can take tens of milliseconds due to the cost of setting `cgroups` and `namespaces`.

Fortunately, fast containerization has been well-studied [94, 17, 27, 112]. For instance, SOCK [94] introduces *lean container*, which is a special container having the minimal configurations necessary for serverless computing. It further uses pooling to hide the cost of container bootstrap, reducing its time from tens of milliseconds to a few milliseconds. We generalize SOCK's lean container to a distributed setting to accelerate the containerization of the remote fork. Specifically, before resuming a remote parent, we will use SOCK to create an empty lean container that satisfies the parent's isolation requirements. Afterward, the empty container calls Mitosis to resume execution. Since the container has been properly configured with SOCK, we can skip the costly containerization.

5.3 Network daemon

The network daemon aims to reduce the costs of creating RDMA connections (commonly called *RCQP*) on the critical path of the remote fork. Meanwhile, it also avoids caching RCQPs connected to all the servers to save memory.

Solution: Retrofit advanced RDMA transport (DCT). The essential requirement behind the goal is that we need QP to be connectionless. RDMA does provide a connectionless transport—unreliable datagram (UD), but it only supports messaging, so we can just use it for RPC.

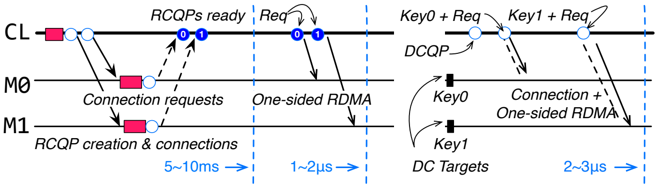


Figure 8. *A comparison of a client (CL) using two RCQPs and DCQP to communicate with two machines (M1 and M2).*

We find dynamic connected transport (DCT) [1]—a less studied but widely supported RDMA feature suits remote fork well. DCT preserves the functionality of RC and further provides a connectionless illusion: a single DCQP can communicate with different nodes. The target node only needs to create a *DC target*, which is identified by the node's RDMA address and a 12B *DC key*.[7] After knowing the keys, a child node can send one-sided RDMA requests to the corresponding targets without connection—the hardware will piggyback the connection with data processing and is extremely fast (within 1μs [11, 67]), as shown in Figure 8.

Based on DCT, the network daemon manages a small kernel-space DCQP pool for handling RDMA requests from children. Typically, one DCQP per-CPU is sufficient to utilize RDMA [11]. However, using DCT alone is insufficient because the child needs to know the DCT key in advance to communicate with the parent. Therefore, we also implement a kernel-space FaSST RPC [67] to bootstrap DCT. FaSST is a UD-based RPC that supports connectionless. With RPC, we piggyback the DCT key associated with the parent in the RPC request to query the parent's descriptor. To save CPU resources, we only deploy two kernel threads to handle RPCs, which is sufficient for our workloads (see Figure 13 (b)).

Discussion on DCT overheads. DCT has known performance issue due to extra reconnection messages. Compared with RC, it causes up to 55.3% performance degradations for small (32B) one-sided RDMA READs [67]. Nevertheless, the reconnection has no effect on the large (e.g., more than 1 KB) transfer because transferring data dominates the time [11]. Since the workload pattern of Mitosis is dominated by large transfers, e.g., reading remote pages in 4KB granularity, we empirically found no influence from this issue.

5.4 RDMA-Aware virtual memory management

For resume efficiency, we directly set the page table entries (PTE) of the children's mapped pages to the parent's physical addresses (PA) during the resume phase. However, the original OS is unaware of the remote PA in the PTE. Thus, we dedicate a remote bit in the PTE for distinction. In particular, the OS will set the remote bit to be 1 and clear the present bit of the PTE during the switch process at the resume phase. Afterward, child's remote page access will trap in the kernel

---

[7] The key consists of a 4 B NIC-generated number and 8 B user-passed key.

Table 2: *A summary of page fault handling related to remote fork at the child categorized by whether the virtual address (VA) is mapped to remote and whether the remote physical address (PA) is stored.*

Example	VA mapped	Parent PA in PTE	Method
Stack grows	No	No	Local
Code in .text	Yes	Yes	RDMA
Mapped file	Yes	No	RPC

after the switch. Consequently, Mitosis can handle them in the RDMA-aware page fault handler. Note that we don't change the table entry data structure: we utilize an ignored PTE bit (i.e., one in $[58:52]$ [60]) for the remote bit.

RDMA-aware page fault handler. Table 2 summarizes how we handle different faults related to remote fork. If the fault page has not mapped to the parent, e.g., stack grows, we handle it locally like a normal page fault. Otherwise, we check whether the fault virtual address (VA) has a mapped remote PA. If so, we use one-sided RDMA to read the remote page to a local page. Most child pages can be restored via RDMA because serverless function typically touches a subset of the previous run [120, 37]. In case of a missed mapping, we fallback to RPC.

Fallback daemon. Each node hosts a fallback daemon that spawns kernel threads to handle children's paging requests, which contains the parent identifier and the requested virtual address. The fallback logic is simple: After checking the validity of the request, the daemon thread will load the page on behalf of the parent. If the load succeeds, we will send the result back to the child.

Connection-based memory access control and isolation. Direct exposing the parent's physical memory improves the remote fork speed. Nevertheless, we need to reject accesses to mapped pages that no longer belong to a parent and properly isolate accesses to different containers. Since we expose the memory via one-sided RDMA in a CPU-bypassing way, we can only leverage RNIC for the control.

Mitosis proposes a connection-based memory access control method. Specifically, we assign different RDMA connections to different portions of the parent's virtual memory area (VMA), e.g., one connection per VMA. If a mapped physical page no longer belongs to a parent, we will destroy the connection related to the page's VMA. Consequently, the child's access to the page will be rejected by the RNIC. The connections are all managed in the kernel to prevent malicious users from accessing the wrong remote container memory.

To make connection-based access control practical, each connection must be efficient in creation and storage. Fortunately, the DCQP satisfies these requirements well. At the child-side, each connection (DC key) only consumes 12B—different DC connections can share the same DCQP. Meanwhile, the parent-side DC target consumes 144B. Note that creating DCQPs and targets also has overheads. Yet, they are logically independent of the parent's memory. Therefore, we use pooling to amortize their creation time (several ms).

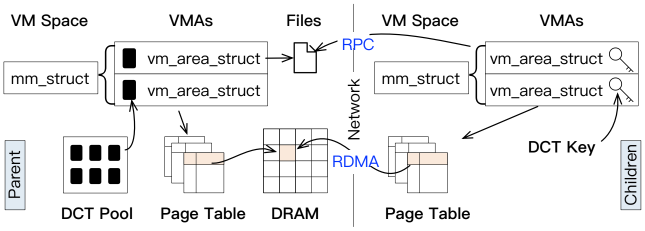


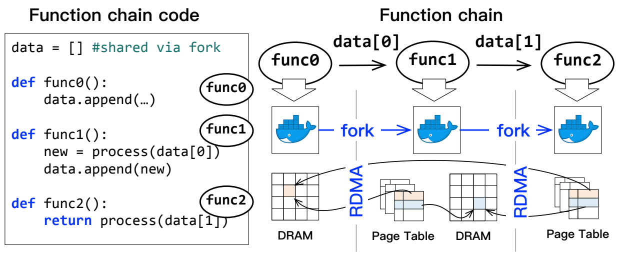
Figure 9. *An illustration of the extended virtual memory subsystems to map children's virtual addresses to remote memory. The memory space is divided into a list of virtual memory area (VMA)s, each managed by a `vm_area_struct`. DC target pool and DCT keys are used by connection-based memory access control.*

Figure 9 shows the DCT-based access control in action. Upon fork preparation, Mitosis assigns one DC target—selected from a target pool—to each parent VMA. The pool is initialized during boot time and is periodically filled in the background. The DC keys of these targets are piggybacked in the parent's descriptor so that the children can record them in their VMA during resume. Upon reading a parent's page, the child will use the key corresponding to the page's VMA to issue the RDMA request. With this scheme, if the parent wants to reject accesses to this page, it can destroy the corresponding DC target.

Connection-based control has false positives: after destroying a VMA's assigned target, all remote accesses to it are rejected. Assigning DC targets in a more fine-grained way (e.g., multiple targets per VMA) can mitigate the issue at the cost of increased memory usage. We found it is unnecessary because VA–PA changes are rare at the parent. For example, swap never happens if the OS has sufficient memory.

Security analysis. Compared with normal containers, Mitosis additionally exposes its physical memory to remote machines via RDMA. Nevertheless, since remote containers must leverage their kernels to read the exposed memory, a malicious container cannot read others states as long as its kernel is not compromised. Besides this, the inherent security issues of RDMA [111, 99, 128] may also endanger Mitosis. While such security threats are out of the scope of our work, it is possible to integrate orthogonal solutions [111, 99, 128, 115] to improve the security of Mitosis.

Optimizations: prefetching and caching. Even with RDMA, reading remote pages is still much slower than local memory accesses [35] ($3\,\mu s$ vs. 100 ns). Thus, we apply two standard optimizations: *Prefetching* prefetches adjacent remote pages upon page faults. Empirically, we found a prefetch size of one is sufficient to improve the performance of remote fork at a small cost to the runtime memory (see Figure 15). Thus, Mitosis only prefetches one adjacent page by default. *Caching* caches the finished children's page table (and the read pages) in the kernel. A later child forking the same parent can then reuse the page table in a copy-on-write way to avoid reading



```
data = [] #shared via fork

def func0():
    data.append(…)

def func1():
    new = process(data[0])
    data.append(new)

def func2():
    return process(data[1])
```

Figure 10. *An illustration of multi-hops remote fork.*

the touched pages again. This is essentially a combination of local-remote fork. To avoid extra memory cost, we only keep the cached page table for a short period (usually several seconds) to cope with load spikes (e.g., see Figure 1).

5.5 Supporting multi-hops remote fork

MITOSIS supports multi-hops fork: a child can be forked again with its children possibly on the third machine. It is similar to one-hop fork except that we need to further track the ownership of remote pages in a fine-grained way. As shown in Figure 10, the pages behind `data[1]` and `data[0]` resides on two different machines. A naive approach would be maintaining a map to track the owner of each virtual page. However, it would consume non-trivial storage overhead. To reduce memory usage, MITOSIS encodes the owner in the PTE: we dedicate 4 bits in the PTE's ignored bits to encode the remote page machine—supporting a maximum of 15-hops remote fork (up to 15 ancestors)

6 Bringing MITOSIS to Serverless Computing

This section describes how we apply MITOSIS to FN [123]—a popular open source serverless platform. Though we focus on FN, we believe our methodology can also apply to other serverless platforms (e.g., OpenWhisk [122]) because they follow a similar system architecture (see Figure 11).

Basic FN. Figure 11 shows an overview of FN. It handles the function request that is either an invocation of a single function, or an execution of a serverless workflow (e.g., see Figure 2). A dedicated *coordinator* is responsible for scheduling the executions of these requests. The function code must be packed to a container and uploaded to a Docker registry [34] managed by the platform.

To handle the invocation of a single function, the coordinator will direct the request to an *invoker* chosen from a pool of servers. After receiving the request, the invoker spawns a container with Caching to accelerate startups to execute the function. Note that FN hides the mapping of request to user-function (e.g., 12–16 in Figure 3 (a)) with function development kit (*FDK*): i.e., the user only needs to provide the code for the function, not the code that dispatches the requests to the function. Thanks to this abstraction, we can extend FDK to add the fork capabilities.

To execute a workflow, the coordinator will first decompose the workflow into single-function calls (one for each workflow graph node), then schedule them based on the dependency relationship. In particular, the coordinator will only execute a downstream function (e.g., `defrunAuditRule` in Figure 2) after all its upstream functions (`fetchPortfolioData` and `fetchMarketData`) finish.

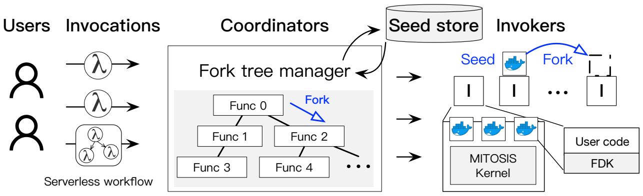


Figure 11. *Integrating* MITOSIS *to* FN. *The gray boxes are our added (or extended) components.*

6.1 Fork-aware serverless platform

Being aware of MITOSIS, the platform can leverage parents that have prepared themselves via `fork_prepare` (we term them as *seeds* in this paper) to accelerate function startup and state transfer. Besides, it is also responsible for reclaiming the seeds. Based on the use cases, we further categorize seeds into two classes. 1) For seeds that are used for boosting function startups, the frequency of reclamation is low. Hence, we name them *long-lived* seeds and use a coarse-grained reclamation scheme (§6.2). 2) For seeds that are used for state transfer, they only live during the lifecycle of a serverless workflow. We name them *short-lived* seeds and use a fine-grained fork tree-based mechanism to free them (§6.3).

The steps to accelerate FN with MITOSIS are: (1) Extend the FN coordinator to send prepare/resume requests to the invoker to fork containers if necessary and (2) Instrument FDK so that it can recognize the new (fork) requests from the coordinator (e.g., line 12–16 in Figure 3 (b)). Since the extensions to the FDK are trivial, we focus on describing the extensions to the coordinator.

Fork-aware coordinator. For a single function call, the coordinator first looks up an available (long-lived) seed. The locations of seeds are stored at a *seed store*. If one seed is available, it sends a fork resume request to the invoker. Otherwise, we fallback to the vanilla function startup mechanism.

During workflow execution, the coordinator dynamically creates short-lived based on state transfer relationship. Specifically, it will tell the invoker to call `fork_prepare` if it executes an upstream function in the workflow. The prepared results are piggybacked in the reply of the function. Afterward, the coordinator can use `fork_resume` to start downstream functions, which transparently inherit the pre-materialized results of the upstream one.

Note that one function may have multiple upstream functions (e.g., *run AuditRule* in Figure 2). For such cases, we require the user to specify which function to fork by annotating the workflow graph or fuse the upstream functions.

6.2 Long-lived seed management

Deployment. We deploy long-lived seeds as cached containers because they naturally load the function's working set into the memory. If the invoker decides to cache a container, it will call `fork_prepare` to generate a seed. Note that we must also adjust FN's cache policy to be fork-aware. For example, FN always caches a container if it experiences a coldstart, which is unnecessary considering MITOSIS because the fork can accelerate startups more resource-efficiently. Therefore, we only cache the first container facing coldstart across the platform. Moreover, we also detect whether a container is a multi-hop one, i.e., forked from a long-lived seed. We don't cache such containers as they are short-lived seeds.

Seed store. To find the seed information, we record a mapping between function name and the corresponding seed's RDMA address, the `handle_id` and `key` (the latter two are returned by `fork_prepare`) at the coordinator. We also record the time when the seed was deployed, which is necessary to prevent the coordinator forking from a near-expired cache instance. The seed store can be co-located with the coordinator or implemented as a distributed key-value store.

Reclamation. Similar to Caching, the long-lived seeds are reclaimed by timeout. Unlike Caching, seeds can have a much longer keep-alive time (e.g., 10 minutes vs. 1 minute) since they consume orders of magnitude smaller memory. The coordinators can renew the seed if it doesn't live long enough for the forked function.

6.3 Fork tree and short-lived seed management

Fork tree granularity and structure. Each serverless workflow has a dedicated fork tree stored and maintained at the coordinator executing it. The upper-layer nodes in the tree correspond to the upstream functions (parents) in the workflow and the lower-layer nodes represent the downstream functions (children). Each node encodes the container IDs and locations, which is sufficient for the coordinator to reclaim the corresponding seed.

Fork tree construction and destroy. The construction of the fork tree is straightforward: After the coordinator forks a new child from a short-lived seed, it will add the seed to the tree. When all functions in the tree finish, MITOSIS will reclaim all the nodes except for the root node: the root node can be a long-lived seed and MITOSIS will not reclaim it.

Fault tolerance. The fork tree should be fault-tolerant to prevent memory leakage caused by dangling seeds. Replicating the tree with common replication protocols (e.g., Paxos [74]) can tolerate the failure, but adds non-trivial overheads during the workflow execution. Observing that serverless functions have a maximum lifetime (e.g., 15 minutes in AWS Lambda [3]), we use a simple timeout-based mechanism to tolerate the failures. Specifically, invokers will periodically garbage collect short-lived seeds if they run beyond the function's maximum allowed runtime.

6.4 Limitation

First, fork still needs a long-lived seed to quickly bootstrap others. If no seed is available, we can leverage existing approaches that optimize coldstart (e.g., FaasNET [119]) to first start one. Second, fork only enables a read-only state transfer. Yet, it is sufficient for serverless workflow—the dominant function composition method. Finally, fork cannot transfer states between multiple upstream functions. Thus, MITOSIS must fuse these upstream functions into one or fallback to messaging (see `Portfolio` in Figure 3) for such cases. We are addressing this limitation by further introducing a *remote merge* primitive to complement the remote fork.

7 Evaluation

Experimental setup. We conduct all our experiments on a local cluster with 24 machines. Each machine has two 12-core Intel Xeon E5-2650 v4 processors and 128GB of DRAM. 16 machines are connected to two Mellanox SB7890 100Gbps switches with two 100 Gbps ConnectX-4 MCX455A InfiniBand RNICs. We use them as invokers to execute the serverless functions. Nodes without RDMA are left as coordinators.

Comparing targets. The evaluating setups of MITOSIS and its baselines are listed as follows. Note that we apply our generalized lean container (§5.2) to all the systems to hide the cost of containerization.

1. ***Caching*** is the de facto warmstart technique that provides a near-optimal function startup.
2. ***CRIU-local*** leverages CRIU [7] to implement remote fork (see Figure 5 (a)) and stores all files in an in-memory local filesystem (tmpfs). The file is transferred via our optimized transfer library with one-sided RDMA. We also apply existing on-demand restore optimization [120].
3. ***CRIU-remote*** leverages CRIU and a distributed file system for the remote fork (see Figure 5 (b)). We use Ceph [89]—a state-of-the-art production DFS that embraces RDMA. We also apply optimizations from CRIU-local: in-memory storage and on-demand restore.
4. ***FaasNET*** [119] optimizes the container image pulling of coldstart with function trees. We evaluate an optimal setup of FaasNET (for performance) that pre-provisions the images at all the invokers.[8]
5. **MITOSIS** is configured with on-demand execution and reads all pages from remote with a prefetch size of one.
6. **MITOSIS+*cache*** is the version of MITOSIS that always caches and shares the fetched pages among children. It essentially fallbacks to the local fork.

---

[8] The setup has been confirmed by the FaasNET authors.

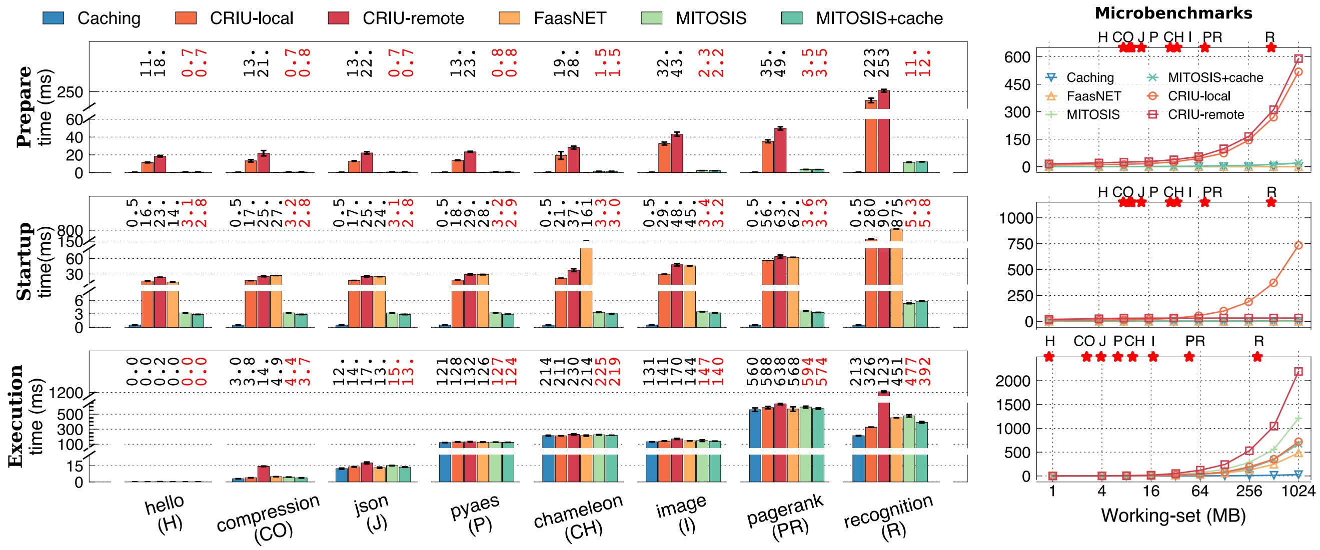


Figure 12. *(a) End-to-end latency comparisons of* MITOSIS *and baselines. (b) Analyses of different phases using microbenchmarks. Note that the working set of the execution is smaller than the prepare and startup because child only touches a subset of the parent's memory.*

Functions evaluated. We chose functions from representative serverless benchmarks (i.e., ServerlessBench [131], FunctionBench [68], and SeBS [31]), which cover a wide range of scenarios, including simple function (*hello*/H—print 'Hello world'), file processing (*compression*/CO—compress a file), web requests (*json*/J—(de)serialize json data, *pyaes*/P—encrypt messages, *chameleon*/CH—generate HTML pages), image processing (*image*/I—apply image processing algorithms to an image), graph processing (*pagerank*/PR—execute the pagerank algorithm on a graph) and machine learning (*recognition*/R—image recognition using ResNet). These functions are written in python—the dominant serverless language [33]. Besides, we also use a synthetic *micro-function* that touches a variant portion of the memory to analyze the overhead introduced by MITOSIS. It is written in C to minimize the language runtime overhead interference.

7.1 End-to-end latency and memory consumption

Figure 12 shows the results of end-to-end latency: the left subfigure is the time of different phases of the functions during remote fork, and the right is each phase's result on microfunction. The function request is sent by a single client. To rule out the impact of disk accesses, we put all the function's related files (e.g., images used by *image*/I) in tmpfs.

Prepare time. The prepare time is the time for the parent to prepare a remote fork. For CRIU-local and CRIU-remote, it is the time to checkpoint a container. For variants of MITOSIS, it is the `fork_prepare` time. Caching and FaasNET do not have this phase because they do not support fork.

MITOSIS is orders of magnitude faster in preparation than CRIU-local and CRIU-remote. On average, it reduces the prepare time by 94%. For example, MITOSIS prepared a 467 MB *recognition*/R container in 11 ms, while CRIU-local and CRIU-remote took 223 ms and 253 ms, respectively. The variants of CRIU are bottlenecked by copying the container state from the memory to the filesystems.

Startup time. We measure the startup time as the time between an invoker receiving the function request and the time the first line of the function executes. As shown in the middle of Figure 12, caching is the fastest (0.5 ms) because starting a cached container only requires a simple unpause operation. MITOSIS comes next, it can start all the functions within 6 ms. It is up to 99%, 94%, and 97% (from 98%, 86%, and 77%) faster than CRIU-local, CRIU-remote, and FaasNET, respectively. The startup time of MITOSIS is dominated by the generalized lean container setup time since reading the descriptor with RDMA is extremely fast with our fast descriptor fetch protocol.

The startup of CRIU-local is dominated by copying the entire file (shown in Figure 12 (b)). Using CRIU-remote avoids transferring the file, but the overhead of communicating with the DFS meta server (from 23–90 ms) is still non-trivial. Compared to CRIU-remote, MITOSIS can directly read the container metadata (descriptor) from the remote machine's kernel. Finally, the startup cost of FaasNET (coldstart) is dominated by the runtime initialization of the function, as we skipped the image pull process of it. The overhead depends on the application characteristics. For example, *recognition*/R requires loading a ResNet model from PyTorch, which takes 875 ms. Other techniques can skip the loading process since the model has been loaded in the parents or the cached containers.

Note that the results of CRIU-remote and FaasNET are not significantly higher in the startup microbenchmark (Figure 12 (b)). For CRIU-remote, it is because the time (40ms) is relatively small compared to CRIU-local (>191ms for working-set larger than 256MB). For FaasNET, we use a native lan-

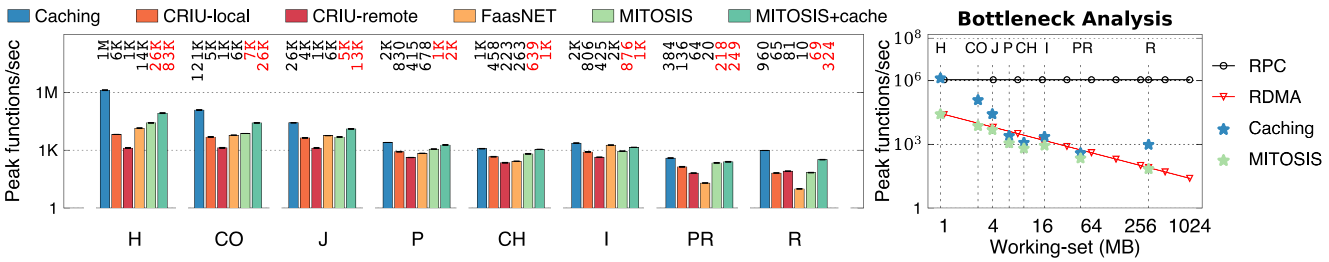


Figure 13. *(a) Peak throughput comparisons of* MITOSIS *and baselines. (b) Bottleneck analysis of* MITOSIS *using a single parent seed.*

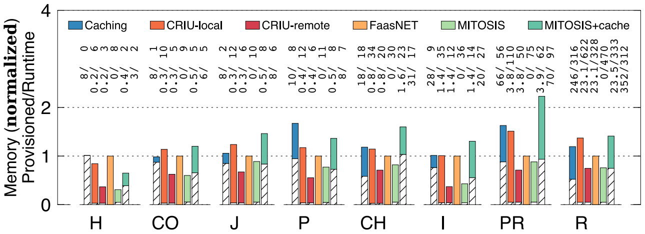


Figure 14. *The per-function memory usage (in MB) for each technique before running (hatched) and during runtime (colored).*

guage in the microbenchmark (C), so it doesn't suffer from the runtime initialization and library loading costs of the application functions in Figure 12 (a).

Execution time. For function execution, MITOSIS is up to 2.24×, 1.46× and 1.14× (from 1.04×, 1.04×, and 1.02×) slower than Caching, CRIU-local and FaasNET, respectively, except for *hello*/H. The overhead is mainly due to page faults and reading remote memory, which is proportional to the function working set (see Figure 12 (b)). Consequently, the overhead is most significant in *recognition*/R that reads 321 MB of the parent memory: MITOSIS is 2.24× (477 vs. 213 ms) and 1.46× (477 vs. 326 ms) slower than Caching and CRIU-local, respectively. CRIU-local is faster since it reads files from the local memory (tmpfs). To remedy this, MITOSIS+cache reduces the number of remote memory accesses by reading from the local cached copies of the remote pages. It improves performance by up to 17%, making MITOSIS close to or better than CRIU-local and FaasNET during execution. Note that Caching is always optimal (i.e., faster than FaasNET and CRIU-local) because it has no page fault overhead. Finally, MITOSIS is up to 3.02× (from 1.02×) faster than CRIU-remote thanks to bypassing DFS for reading remote pages.

Memory consumption. Figure 14 reports the amortized per-machine memory consumed for each function categorized by provisioned memory (before running) and runtime memory. An ideal serverless platform should use minimal provisioned memory for each function. On average, MITOSIS only consumes 6.5% of the provisioned memory (one cached instance across 16 machines) while Caching requires at least 16 instances. CRIU-local/remote consumes a slightly lower memory (77% on average) than MITOSIS, because it reuses the local OS's shared libraries to prevent storing them in the checkpointed files. This works at the cost of requiring storing all the function's required libraries on all the machines, otherwise the restored container will fail. For the same reason, MITOSIS consumes a slightly larger runtime memory (8% on average) than CRIU-remote. Yet, its runtime memory is smaller than CRIU-local because the CRIU-local will read the entire file before it can execute the function.

7.2 Bottleneck analysis and throughput comparisons

Bottleneck analysis. Using a single seed function is ideal for resource usage. However, the parent-side network bandwidth (RDMA) and two RPC threads can become the bottleneck. Meanwhile, MITOSIS is also bottlenecked by the aggregated client-side CPU resources processing the function logic. The peak client-side performance for each function is the peak throughput of running functions with Caching.

Figure 13 (b) analyzes the impact of the above factors. We utilize all 16 invokers to achieve the peak throughput. For H, CO, J, and R, RDMA is the bottleneck. For example, *recognition*/R touches 321 MB of the parent's memory, so the RDMA (200 Gbps) can only serve (ideal) 80 forks/sec. Thus, MITOSIS achieves 69 reqs/sec and is lower than Caching (960 reqs/sec). In contrast, if the children CPU is the bottleneck, MITOSIS is similar to Caching (P, CH, I, and PR). For example, Caching can only execute 384 reqs/sec for *pagerank*/PR. In comparison, RDMA can handle an ideal 544 PR forks/sec (the working set is 47 MB). Thus, MITOSIS can achieve a slightly lower throughput (249 reqs/sec). Finally, the RPC would never become the bottleneck: two kernel threads can handle up to 1.1 million reqs/sec, which is always faster than RDMA for working set from 1 MB to 1 GB.

Throughput comparison. Figure 13 (a) further compares the peak throughput of different approaches. Note that we exclude the prepare phase of CRIU—otherwise, it will be bottlenecked by this phase. MITOSIS is up to 8.0× (from 2.1×) faster than CRIU-local, thanks to avoiding the whole file during the restore phase. Compared with CRIU-remote, MITOSIS is also up to 20.4× (from 2.1×) faster except for R (69 vs. 81): CRIU-remote reads a smaller amount of remote memory because it reuses local copies of the shared libraries. R has the largest working set, so it is mostly affected by the network. For the others, MITOSIS is faster as it bypasses the overhead of DFS.

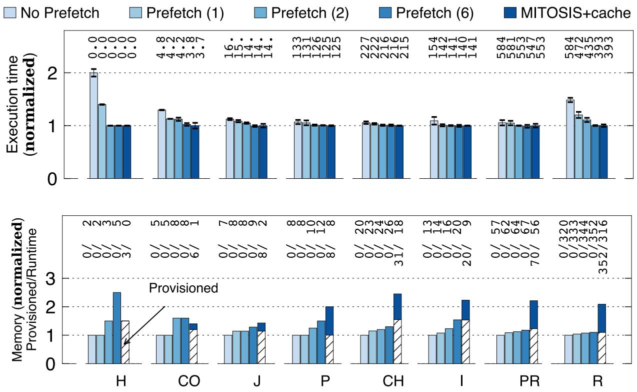


Figure 15. *Effects of the number of pages prefetched per-fault on (a) execution time (in ms) and (b) memory consumption (in MB).*

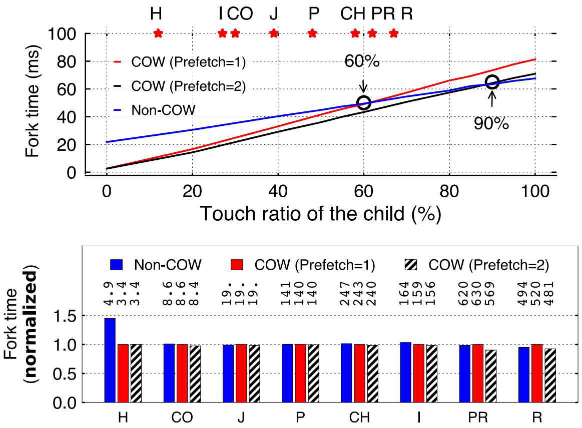


Figure 16. *Effects of COW to latencies on (a) the micro-function (with a 64 MB parent working set) and (b) serverless functions.*

We omit the comparison between MITOSIS and Caching, which has been studied in the bottleneck analysis.

7.3 Effects of prefetching

We next explore how the prefetch number affects MITOSIS in Figure 15 (a). As we can see, prefetching can significantly improve the execution time of functions: prefetching 1, 2, and 6 pages improve the average time by 10%, 16%, and 18% (up to 30%, 50%, and 50%), respectively. More importantly, a small prefetch size (6) can achieve a near-identical performance as the optimal, i.e., no remote access, (MITOSIS+cache). Note that for small prefetch size the cost to the throughput is negligible, so we omit the results.

Prefetching has additional runtime memory consumption: as shown in Figure 15 (b), prefetching 1, 2, and 6 consumes average 1.1×, 1.3×, and 1.5× (up to 1.15×, 1.6×, and 2.5×) more memory than no prefetching. Therefore, we currently adopt a prefetch size of 1 to reduce runtime memory usage.

7.4 Effects of copy-on-write (COW)

MITOSIS reads the child's pages in an on-demand way (copy-on-write). This section presents the benefits and costs of COW compared to a non-COW design—the child will read all the parent's memory before executing the functions.

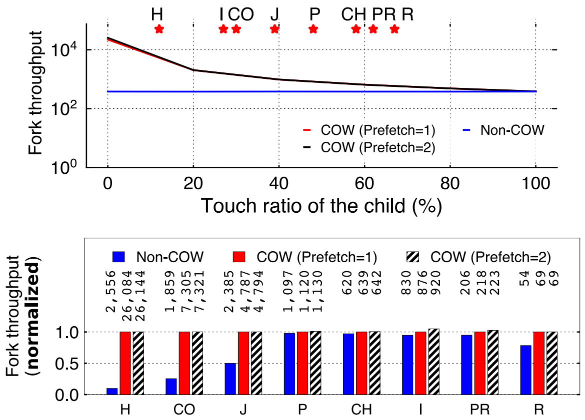


Figure 17. *Effects of COW to peak thpt on (a) the micro-function (with a 64 MB parent working set) and (b) serverless functions.*

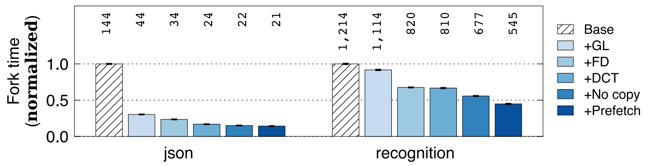


Figure 18. *Effects of optimizations applied by* MITOSIS.

Latency. Figure 16 reports the latency results. The benefit of COW in latency depends on the amount of the parent's memory touched by the child (touch ratio): the cross points in the microbenchmark are 60% and 90% when the prefetch size is 1 and 2, respectively. For larger prefetch size, the cross point is close to 100%. Non-COW has a longer startup time due to extra remote memory reading, but it is more efficient in reading pages with RDMA because it can batch multiple paging requests [66]. Nevertheless, serverless functions typically have a moderate touch ratio (i.e., $< 67\%$). Therefore, COW has averages of 8.7% (from 0.6% to 44%) and 3.7% (from -5% to 31%) lower latency than Non-COW when the prefetch size is 1 and 2, respectively.

Throughput. Figure 17 further reports the throughput results. Unlike latency, COW is always faster in throughput (except for 100% touch ratio) because non-COW will issue more RDMA requests. Consequently, COW is 1.03X–10.2X faster than Non-COW on serverless functions.

7.5 Effects of optimizations

Due to space limitation, Figure 18 briefly shows the effects of optimizations introduced in §5 on the end-to-end fork time using a short function (*json*/J) and a long function (*recognition*/R). First, generalized lean container (+GL) reduced a fixed offset of the latency (100 ms) to all the functions compared with a baseline of using runC [13]. Compared with RPC, fast descriptor fetch with one-sided RDMA (+FD) further contributes 10% and 25% latency reduction for both

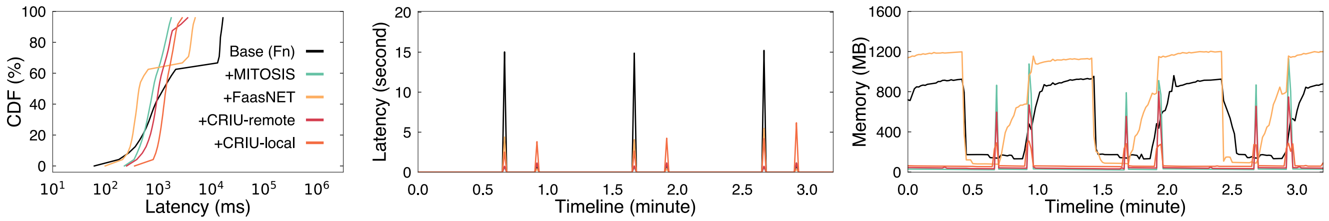


Figure 19. *(a) The latency CDF, (b) average latency, and (c) memory consumption timelines on image processing* (I) *under spikes.*

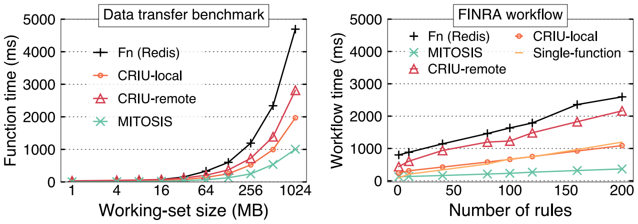


Figure 20. *(a) The state-transfer performance between two functions and (b) performance of FINRA.*

functions. The improvement is more obvious for R because its descriptor is much larger (1.3 MB vs. 31 KB). Using DCT instead of RC reduced a 10–20 ms to the functions, and directly exposing the physical memory with RDMA instead of copying them (+no copy) further reduced the fork time by 12% and 20% for J and R, respectively. Finally, prefetching (+prefetch) shortens the time by 9% and 15%.

7.6 State-transfer performance

Microbenchmark. We use the data-transfer testcase (5) in ServerlessBench [131] to compare different approaches to transfer states between two remote functions. As shown in Figure 20 (a), Mitosis is up to 1.4–5× faster than Fn, which leverages Redis to transfer data between functions, when transferring 1 MB–1 GB data. Note that we exclude the data (de)serialization overhead (by skipping the phase) and coldstart overhead (by pre-warming the containers) in Fn. Otherwise, the gap between Fn and Mitosis would become larger. Compared to CRIU-local/remote, Mitosis is faster thanks to the design for a fast remote fork (see §7.1).

Application: FINRA. We next present the performance of Mitosis on FINRA [14], whose workflow graph is shown in Figure 2. We manually fuse the `fetchPortfolioData` and `fetchMarketData` into one function to fully leverage remote fork for Mitosis and CRIU variants. For Fn, functions use Redis to transfer states. Figure 20 (b) reports the end-to-end latency w.r.t the number of instances of `runAuditRule`, where FINRA spawns about 200 instances [10]. We select the market data from seven stocks, resulting in a total 6 MB states transferred between functions.

As we can see, Mitosis is 84–86%, 47–66% and 71–83% faster than the baseline Fn, CRIU-local and CRIU-remote, respectively. Note that we have pre-warmed Fn to prevent the effects of coldstart—which is unnecessary for Mitosis. Fn is bottlenecked by Redis (27 ms) and data serialization and de-serialization (600 ms). Mitosis has no such overhead and it further makes state transfer between machines optimal via RDMA. Moreover, Mitosis can scale to a distributed setting with little COST [88]—it can outperform a single-function sequentially processing all the rules (Single-function). This is because Mitosis can concurrently run functions across machines with minimal cost transferring data between them.

7.7 Performance under load spikes

Finally, we evaluate the performance of Mitosis under load spike using *image*/I on the real-world traces (`660323` [102]). Figure 19 (a) summarizes the latency CDFs. The 99^{th} percentile latency of Fn+Mitosis is 73.64% and 89.08% smaller than Fn+FaasNET and Fn, respectively, thanks to avoiding the coldstart with remote fork. Nevertheless, its median latency is 1.85× longer than FaasNET (799 ms vs. 430 ms), because FaasNET leverages Caching and has a 65.1% cache hit during spikes. However, Caching incurs non-trivial memory consumption: Fn (and Fn+FaasNET) will cache a container for 30 seconds if it is a coldstart, resulting in a significant amount of memory usage (see Figure 19 (c)). In comparison, Mitosis only caches a single seed and saves orders of magnitude memory during the idle time. For example, at time 2.3 min, Mitosis only consumes 29 MB memory per-machine, which is 3% and 2% of Fn (914 MB) and Fn+FaasNET (1,199 MB), respectively.

8 Discussion

Seed placement and selection policies. We currently choose a random placement policy. A better policy may further consider network topology and system-wide load balance. Meanwhile, we simply choose the first container experiencing coldstart as the long-lived seed, yet, a better selection policy should further consider the status of the running container. For instance, recent works have discovered that containers may need multiple invocations to warm up properly [28, 107], e.g., to JIT a function written in a managed language. Therefore, choosing a properly warm-up container as the seed can significantly improve the function performance after the fork. As these policies are orthogonal to Mitosis, we plan to investigate them in the future.

Frequency and cost of fallbacks. The frequency of fallbacks can significantly impact the performance of remote forks. During our experiment, we encountered no fallbacks because the parent (cached container) had loaded all the children's memory. However, fallbacks can happen in corner cases (e.g., swapping). The per-page overhead is 22× (65 vs. 3 μs) due to the cost of RPC and loading the page from disk (SSD). Currently, one fallback handler can process 16 K paging requests per second, so it will not become a bottleneck.

The benefits of implementing MITOSIS in the kernel. We choose to implement MITOSIS in the kernel for performance considerations. First, a user-space solution cannot directly access the physical memory of the container, so it pays the checkpointing overhead (see §3). Moreover, the kernel can establish RDMA connections more efficiently (see KRCore [11]), and the kernel-space page fault handler is much faster than the user-space fault handler.

9 Related Work

Optimizing serverless computing. MITOSIS continues the line of research on optimizing serverless computing, including but not limited to accelerating function startups [94, 17, 106, 37, 117, 101, 119], state transfer [110, 69, 96, 71, 17, 86], stateful serverless functions [132, 63], transactions [84], improving the cost-efficiency [134, 42, 100, 76, 98, 40, 38], and others [109, 133, 65, 36, 64, 15, 114, 85, 130, 136]. While most of these works are orthogonal to MITOSIS, we believe they can also benefit from our work. In particular, we propose using the remote fork abstraction to simultaneously accelerate function startups and state transfer, which is critical to all serverless applications. We also compare our work extensively to its closest related approaches in §2. Moreover, while the implementation of Linux fork may not be optimal in some scenarios [129, 24, 135], it has been shown to be suitable for serverless functions [17, 37]. Thus, we generalize the fork abstraction to accelerate functions running across machines.

Checkpoint and restore (C/R). C/R has been investigated by OSes for a long time [39, 82]. e.g., KeyKOS [51], EROS [104], Aurora [116] and others [52, 72, 7, 137, 118, 21, 26, 48]. Aurora [116] leverages C/R to realizing efficient single level store, it introduces techniques including system shadowing for efficient incremental checkpointing. MITOSIS eliminates checkpointing in the context of remote fork via OS-RDMA co-design. VAS-CRIU [118] also noticed the overhead of C/R introduced by filesystems. It leverages multiple independent address spaces (MVAS) [50] to bypass the filesystem for C/R on a single machine. We further use kernel-space RDMA to build a global distributed address space and scale fast C/R to a distributed setting.

Remote fork (migrations). Besides using C/R for remote fork [108, 32], MITOSIS is also inspired by works on virtual machine fork (SnowFlock [73]) and migrations [18, 29, 45, 56, 55, 92, 81], just to name a few. For example, the MITOSIS container descriptor is inspired by the VM descriptor used in SnowFlock, which only captures the critical metadata used for instantiating a child container at the remote side. We further consider the opportunities and challenges when embracing RDMA for remote fork in the context of serverless computing. We believe our techniques can benefit existing works not utilizing RDMA.

RDMA-based remote paging and RDMA multicast. Reading pages from remote hosts via RDMA is not a so new technique in modern OSes [19, 46, 16, 87, 103]. For example, Infiniswap [46] leverages RDMA to build a fast swap device for memory disaggregation. Remote regions [16] proposes a remote file-like abstraction to simplify exposing an application's memory with RDMA. MITOSIS further builds efficient remote fork by reading remote pages in a "copy-on-write" fashion with RDMA.

MITOSIS exhibits a pull-based RDMA multicast communication pattern, where multiple children pull from the same parent's memory during load spikes. Push-based RDMA multicast has been extensively studied in the literature [25, 61, 62]. For example, RDMC [25] proposes a binomial pipeline protocol where a sender can efficiently push data to a group of nodes using RDMA. We believe MITOSIS can further benefit from research on pull-based RDMA multicast.

10 Conclusion

We present MITOSIS, a new OS primitive designed for fast remote fork by co-designing with RDMA. MITOSIS possesses two key attributes for serverless computing. (1) Startup efficiency: MITOSIS is orders of magnitude faster than coldstart while consuming orders of magnitude fewer resources than warmstart (with comparable performance). (2) State transfer efficiency: functions can directly access pre-materialized states from the forked function. Extensive evaluation using real-world serverless applications confirmed the efficacy and efficiency of MITOSIS on commodity RDMA-capable clusters. While we focus on serverless computing in this paper, we believe MITOSIS also shines with other tasks, e.g., container migrations.

Acknowledgment

We sincerely thank our shepherd Christopher Rossbach and the anonymous reviewers, whose reviews and suggestions greatly strengthened our work. We also thank Wentai Li, Qingyuan Liu, Zhiyuan Dong, Dong Du, Nian Liu, Sijie Shen, and Xiating Xie for their valuable feedback. This work was partly supported by the National Key Research & Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China (No. 62202291, 62202292, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 22511106200), as well as research grants from Huawei Technologies and Shanghai AI Laboratory. Corresponding author: Rong Chen (`rongchen@sjtu.edu.cn`).

References

[1] Dynamically connected transport. https://www.openfabrics.org/images/2018workshop/presentations/303_ARosenbaum_DynamicallyConnectedTransport.pdf, 2018.

[2] Apache OpenWhisk Composer. https://github.com/apache/openwhisk-composer, 2022.

[3] AWS Lambda FAQs. https://aws.amazon.com/en/lambda/faqs/, 2022.

[4] AWS Step Functions. https://aws.amazon.com/step-functions/, 2022.

[5] Ceph - a scalable distributed storage system. https://github.com/ceph/ceph/tree/luminous-release, 2022.

[6] CRIU Lazy migration. https://criu.org/Lazy_migration, 2022.

[7] CRIU Website. https://www.criu.org/Main_Page, 2022.

[8] docker container pause. https://docs.docker.com/engine/reference/commandline/container_pause/, 2022.

[9] Docker Website. https://www.docker.com/, 2022.

[10] FINRA adopts AWS to perform 500 billion validation checks daily. https://aws.amazon.com/solutions/case-studies/finra-data-validation/, 2022.

[11] KRCORE: a microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association.

[12] Provisioned concurrency for lambda functions. https://aws.amazon.com/cn/blogs/aws/new-provisioned-concurrency-for-lambda-functions/, 2022.

[13] runc. https://github.com/opencontainers/runc, 2022.

[14] United States Financial Industry Regulatory Authority. https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/, 2022.

[15] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (2020), R. Bhagwan and G. Porter, Eds., USENIX Association, pp. 419–434.

[16] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., NOVAKOVIC, S., RAMANATHAN, A., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 775–787.

[17] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 923–935.

[18] AL-KISWANY, S., SUBHRAVETI, D., SARKAR, P., AND RIPEANU, M. Vmflock: virtual machine co-migration for the cloud. In *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011* (2011), A. B. Maccabe and D. Thain, Eds., ACM, pp. 159–170.

[19] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 14:1–14:16.

[20] AO, L., PORTER, G., AND VOELKER, G. M. Faasnap: Faas made fast using snapshot-based vms. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 730–746.

[21] ARMENATZOGLOU, N., BASU, S., BHANOORI, N., CAI, M., CHAINANI, N., CHINTA, K., GOVINDARAJU, V., GREEN, T. J., GUPTA, M., HILLIG, S., HOTINGER, E., LESHINKSY, Y., LIANG, J., MCCREEDY, M., NAGEL, F., PANDIS, I., PARCHAS, P., PATHAK, R., POLYCHRONIOU, O., RAHMAN, F., SAXENA, G., SOUNDARARAJAN, G., SUBRAMANIAN, S., AND TERRY, D. Amazon redshift re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022* (2022), Z. Ives, A. Bonifati, and A. E. Abbadi, Eds., ACM, pp. 2205–2217.

[22] AWS. Aws fargate. https://aws.amazon.com/cn/fargate/, 2022.

[23] AWS. Aws lambda. https://aws.amazon.com/lambda, 2022.

[24] BAUMANN, A., APPAVOO, J., KRIEGER, O., AND ROSCOE, T. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2019), HotOS '19, Association for Computing Machinery, p. 14–22.

[25] BEHRENS, J., JHA, S., BIRMAN, K., AND TREMEL, E. RDMC: A reliable RDMA multicast for large objects. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018* (2018), IEEE Computer Society, pp. 71–82.

[26] BILAL, M., CANINI, M., FONSECA, R., AND RODRIGUES, R. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. *CoRR abs/2105.14845* (2021).

[27] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. SEUSS: skip redundant paths to make serverless fast. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 32:1–32:15.

[28] Carreira, J., Kohli, S., Bruno, R., and Fonseca, P. From warm to hot starts: leveraging runtimes for the serverless era. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021* (2021), S. Angel, B. Kasikci, and E. Kohler, Eds., ACM, pp. 58–64.

[29] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings* (2005), A. Vahdat and D. Wetherall, Eds., USENIX.

[30] cloud, A. Alibaba serverless application engine. `https://www.aliyun.com/product/aliware/sae`, 2022.

[31] Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., and Hoefler, T. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021* (2021), K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds., ACM, pp. 64–78.

[32] CRIU. CRIU Usage scenarios. `https://criu.org/Usage_scenarios`, 2022.

[33] DATADOG. The state of serverless. `https://www.datadoghq.com/state-of-serverless/`, 2022.

[34] Docker. Docker Registry. `https://docs.docker.com/registry/`, 2022.

[35] Dragojevic, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 401–414.

[36] Du, D., Liu, Q., Jiang, X., Xia, Y., Zang, B., and Chen, H. Serverless computing on heterogeneous computers. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 797–813.

[37] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.

[38] Dukic, V., Bruno, R., Singla, A., and Alonso, G. Photons: lambdas on a diet. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 45–59.

[39] Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing 65*, 3 (2013), 1302–1326.

[40] Fingler, H., Akshintala, A., and Rossbach, C. J. USETL: unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2019, Hangzhou, China, Augsut 19-20, 2019* (2019), ACM, pp. 23–30.

[41] for AWS Lambda Container Reuse, B. P. `https://medium.com/capital-one-tech/best-practices-for-aws-lambda-container-reuse-6ec45c74b67e`, 2022.

[42] Fuerst, A., and Sharma, P. Faascache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. Berger, and C. Kozyrakis, Eds., ACM, pp. 386–400.

[43] Gao, Y., Li, Q., Tang, L., Xi, Y., Zhang, P., Peng, W., Li, B., Wu, Y., Liu, S., Yan, L., Feng, F., Zhuang, Y., Liu, F., Liu, P., Liu, X., Wu, Z., Wu, J., Cao, Z., Tian, C., Wu, J., Zhu, J., Wang, H., Cai, D., and Wu, J. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 519–533.

[44] Google. Google serverless computing. `https://cloud.google.com/serverless`, 2022.

[45] Gu, J., Hua, Z., Xia, Y., Chen, H., Zang, B., Guan, H., and Li, J. Secure live migration of SGX enclaves on untrusted cloud. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017* (2017), IEEE Computer Society, pp. 225–236.

[46] Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), A. Akella and J. Howell, Eds., USENIX Association, pp. 649–667.

[47] Guo, C., Wu, H., Deng, Z., Soni, G., Ye, J., Padhye, J., and Lipshteyn, M. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 202–215.

[48] Guo, Z., Blanco, Z., Shahrad, M., Wei, Z., Dong, B., Li, J., Pota, I., Xu, H., and Zhang, Y. Resource-centric serverless computing, 2022.

[49] Guo, Z., Shan, Y., Luo, X., Huang, Y., and Zhang, Y. Clio: a hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 417–433.

[50] Hajj, I. E., Merritt, A., Zellweger, G., Milojicic, D. S., Achermann, R., Faraboschi, P., Hwu, W. W., Roscoe, T., and Schwan, K. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016* (2016), T. Conte and Y. Zhou, Eds., ACM, pp. 353–368.

[51] Hardy, N. Keykos architecture. *SIGOPS Oper. Syst. Rev. 19*, 4 (oct 1985), 8–25.

[52] Hargrove, P. H., and Duell, J. C. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series* (2006), vol. 46, IOP Publishing, p. 067.

[53] Hellerstein, J. M., Faleiro, J. M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019), www.cidrdb.org.

[54] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016* (2016), A. Clements and T. Condie, Eds., USENIX Association.

[55] Hines, M. R., Deshpande, U., and Gopalan, K. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev. 43*, 3 (jul 2009), 14–26.

[56] Hines, M. R., and Gopalan, K. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009* (2009), A. L. Hosking, D. F. Bacon, and O. Krieger, Eds., ACM, pp. 51–60.

[57] Hong, Y., Zheng, Y., Yang, F., Zang, B., Guan, H., and Chen, H. Scaling out numa-aware applications with rdma-based distributed shared memory. *J. Comput. Sci. Technol. 34*, 1 (2019), 94–112.

[58] Huawei. Huawei clound functions. `https://developer.huawei.com/consumer/en/agconnect/cloud-function/`, 2022.

[59] IBM. IBM Cloud Functions. `https://www.ibm.com/cloud/functions`, 2022.

[60] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. `https://cdrdv2.intel.com/v1/dl/getContent/671200`, 2022.

[61] Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., van Renesse, R., Zink, S., and Birman, K. P. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst. 36*, 2 (2019), 4:1–4:49.

[62] Jha, S., Rosa, L., and Birman, K. Spindle: Techniques for optimizing atomic multicast on RDMA. *CoRR abs/2110.00886* (2021).

[63] Jia, Z., and Witchel, E. Boki: Stateful serverless computing with shared logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 691–707.

[64] Jia, Z., and Witchel, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 152–166.

[65] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019* (2019), ACM, pp. 158–164.

[66] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 437–450.

[67] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.

[68] Kim, J., and Lee, K. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019* (2019), ACM, p. 477.

[69] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. *login Usenix Mag. 44*, 1 (2019).

[70] Knative. `https://knative.dev`, 2022.

[71] Kotni, S., Nayak, A., Ganapathy, V., and Basu, A. Faastlane: Accelerating function-as-a-service workflows. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 805–820.

[72] Laadan, O., and Hallyn, S. E. Linux-cr: Transparent application checkpoint-restart in linux. In *Linux Symposium* (2010), vol. 159, Citeseer.

[73] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009* (2009), W. Schröder-Preikschat, J. Wilkes, and R. Isaacs, Eds., ACM, pp. 1–12.

[74] LAMPORT, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[75] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4 (1989), 321–359.

[76] LI, Q., LI, B., MERCATI, P., ILLIKKAL, R., TAI, C., KISHINEVSKY, M., AND KOZYRAKIS, C. RAMBO: resource allocation for microservices using bayesian optimization. *IEEE Comput. Archit. Lett. 20*, 1 (2021), 46–49.

[77] LINUX. Kernel samepage merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html, 2022.

[78] LINUX. Linux swap. https://www.linux.com/news/all-about-linux-swap-space/, 2022.

[79] LINUX. Page migration. https://www.kernel.org/doc/html/latest/vm/page_migration.html, 2022.

[80] LINUX. Transparent hugepage. https://www.kernel.org/doc/html/latest/vm/transhuge.html, 2022.

[81] LITZKOW, M., AND SOLOMON, M. Supporting checkpointing and process migration outside the unix kernel.

[82] LITZKOW, M., TANNENBAUM, T., BASNEY, J., AND LIVNY, M. Checkpoint and migration of unix processes in the condor distributed processing system. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1997.

[83] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 773–785.

[84] LYKHENKO, T., SOARES, R., AND RODRIGUES, L. Faastcc: Efficient transactional causal consistency for serverless computing. In *Proceedings of the 22nd International Middleware Conference* (New York, NY, USA, 2021), Middleware '21, Association for Computing Machinery, p. 159–171.

[85] LYU, X., CHERKASOVA, L., AITKEN, R. C., PARMER, G., AND WOOD, T. Towards efficient processing of latency-sensitive serverless dags at the edge. In *EdgeSys@EuroSys 2022: Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking, Rennes, France, April 5 - 8, 2022* (2022), A. Y. Ding and V. Hilt, Eds., ACM, pp. 49–54.

[86] MAHGOUB, A., SHANKAR, K., MITRA, S., KLIMOVIC, A., CHATERJI, S., AND BAGCHI, S. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 285–301.

[87] MARUF, H. A., AND CHOWDHURY, M. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 843–857.

[88] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015* (2015), G. Candea, Ed., USENIX Association.

[89] MELLANOX. Bring up ceph rdma - developer's guide. https://community.mellanox.com/s/article/bring-up-ceph-rdma---developer-s-guide, 2021.

[90] MELLANOX. Kernel verbs api update. https://www.openfabrics.org/images/eventpresos/2016presentations/204KernelVerbs.pdf, 2022.

[91] MICROSOFT. Azure functions. https://azure.microsoft.com/en-us/services/functions/, 2022.

[92] MILOJICIC, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process migration. *ACM Comput. Surv. 32*, 3 (2000), 241–299.

[93] NVIDIA. Rdma aware networks programming user manual. https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=34256548, 2022.

[94] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.

[95] PONS, D. B., ARTIGAS, M. S., PARÍS, G., SUTRA, P., AND LÓPEZ, P. G. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019* (2019), ACM, pp. 41–54.

[96] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 193–206.

[97] QI, S., MONIS, L., ZENG, Z., WANG, I., AND RAMAKRISHNAN, K. K. SPRIGHT: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022* (2022), F. Kuipers and A. Orda, Eds., ACM, pp. 780–794.

[98] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. FIRM: an intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 805–825.

[99] Rothenberger, B., Taranov, K., Perrig, A., and Hoefler, T. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 4277–4292.

[100] Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., and Wilkes, J. Autopilot: workload autoscaling at google. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 16:1–16:16.

[101] Saxena, D., Ji, T., Singhvi, A., Khalid, J., and Akella, A. Memory deduplication for serverless computing with medes. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 714–729.

[102] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 205–218.

[103] Shan, Y., Huang, Y., Chen, Y., and Zhang, Y. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 69–87.

[104] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999* (1999), D. Kotz and J. Wilkes, Eds., ACM, pp. 170–185.

[105] Shi, J., Yao, Y., Chen, R., Chen, H., and Li, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 317–332.

[106] Shillaker, S., and Pietzuch, P. *FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing*. USENIX Association, USA, 2020.

[107] Shin, W., Kim, W., and Min, C. Fireworks: a fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 663–677.

[108] Smith, J. M., and Ioannidis, J. *Implementing remote fork () with checkpoint/restart*. Department of Computer Science, Columbia Univ., 1987.

[109] Sreekanti, V., Wu, C., Chhatrapati, S., Gonzalez, J. E., Hellerstein, J. M., and Faleiro, J. M. A fault-tolerance shim for serverless computing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 15:1–15:15.

[110] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J. E., Hellerstein, J. M., and Tumanov, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow. 13*, 12 (jul 2020), 2438–2452.

[111] Taranov, K., Rothenberger, B., Perrig, A., and Hoefler, T. Srdma: Efficient nic-based authentication and encryption for remote direct memory access. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (USA, 2020), USENIX ATC'20, USENIX Association.

[112] Thalheim, J., Bhatotia, P., Fonseca, P., and Kasikci, B. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 199–212.

[113] Thomas, S., Ao, L., Voelker, G. M., and Porter, G. Particle: ephemeral endpoints for serverless networking. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 16–29.

[114] Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., and Xu, G. H. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (2021), A. D. Brown and J. R. Lorch, Eds., USENIX Association, pp. 495–514.

[115] Tsai, S.-Y., and Zhang, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.

[116] Tsalapatis, E., Hancock, R., Barnes, T., and Mashtizadeh, A. J. The aurora single level store operating system. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 788–803.

[117] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 559–572.

[118] Venkatesh, R. S., Smejkal, T., Milojicic, D. S., and Gavrilovska, A. Fast in-memory CRIU for docker containers. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019* (2019), ACM, pp. 53–65.

[119] Wang, A., Chang, S., Tian, H., Wang, H., Yang, H., Li, H., Du, R., and Cheng, Y. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 443–457.

[120] Wang, K. A., Ho, R., and Wu, P. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 39:1–39:16.

[121] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 133–146.

[122] Website, A. O. https://openwhisk.apache.org, 2022.

[123] Website, F. P. https://fnproject.io, 2021.

[124] Wei, X., Chen, R., and Chen, H. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 117–135.

[125] Wei, X., Dong, Z., Chen, R., and Chen, H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 233–251.

[126] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[127] Xie, X., Wei, X., Chen, R., and Chen, H. Pragh: Locality-preserving Graph Traversal with Split Live Migration. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), pp. 723–738.

[128] Xing, J., Hsu, K.-F., Qiu, Y., Yang, Z., Liu, H., and Chen, A. Bedrock: Programmable network support for secure RDMA systems. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 2585–2600.

[129] Xu, W., Kashyap, S., Min, C., and Kim, T. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2313–2328.

[130] Yang, Y., Zhao, L., Li, Y., Zhang, H., Li, J., Zhao, M., Chen, X., and Li, K. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 768–781.

[131] Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C., and Chen, H. Characterizing serverless platforms with serverlessbench. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 30–44.

[132] Zhang, H., Cardoza, A., Chen, P. B., Angel, S., and Liu, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1187–1204.

[133] Zhang, W., Fang, V., Panda, A., and Shenker, S. Kappa: a programming framework for serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 328–343.

[134] Zhang, Y., Goiri, I. n., Chaudhry, G. I., Fonseca, R., Elnikety, S., Delimitrou, C., and Bianchini, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 724–739.

[135] Zhao, K., Gong, S., and Fonseca, P. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 540–555.

[136] Zhao, L., Yang, Y., Li, Y., Zhou, X., and Li, K. Understanding, predicting and scheduling serverless workloads under partial interference. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021* (2021), B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds., ACM, pp. 22:1–22:15.

[137] Zhong, H., and Nieh, J. Crak: Linux checkpoint/restart as a kernel module. Tech. rep., Citeseer, 2001.