# Phoenix: Fast, Compressed, In-memory page-fault handling for Serverless MicroVMs

Arnav Mummineni
*University of Michigan*

Adhav Rajesh
*University of Michigan*

Harish Jaisankar
*University of Michigan*

## Abstract

Serverless platforms must rapidly start, pause, and resume large numbers of short-lived function instances, each running inside an isolated MicroVM. While systems such as Firecracker provide strong isolation, they incur high memory overhead because each MicroVM maintains its own full memory image, even though most pages are identical across instances. This limits utilization and slows cold starts. We present Phoenix, a memory-efficient snapshotting and paging system that reduces duplication across Firecracker MicroVMs. Phoenix introduces Incinerator, a fast page-level diff-compression algorithm that identifies similar pages across snapshots and stores compact XOR-based diffs entirely in memory. Integrated with Linux userfaultfd, Phoenix reconstructs pages lazily on demand with microsecond-scale latency, avoiding slow disk I/O during snapshot restores. Across Python and LINPACK workloads, Phoenix reduces diff sizes dramatically and lowers page-fault latency by an order of magnitude compared to disk-backed snapshot loading. Phoenix enables finer-grained preemption and improves resource utilization in VM-based serverless architectures. Phoenix is available at https://github.com/ProjectPhoenixVM/phoenix.

## 1 Introduction

As cloud computing has grown, serverless architectures have become increasingly prominent due to their elasticity, fine-grained billing, and operational simplicity for developers. Serverless providers continuously execute vast numbers of short-lived functions from mutually untrusted clients. This places strict requirements on security, isolation, and performance. Guaranteeing strong isolation is essential for protecting tenants, but doing so introduces large overheads that reduce system throughput and impact the economics of serverless platforms.

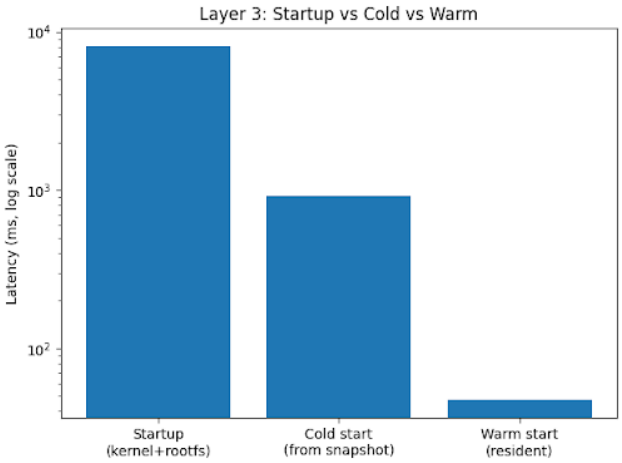
Serverless workloads are characterized by services like Amazon Web Services' Lambda or Microsoft Azure Functions. These facilities are usually used by application developers to run small, simple, single purpose web backend code components. Because many users are running these functions at a given time, serverless providers isolate function invokations using some form of virtulization. This means a single server running the infrastructure for Lambda or Azure Functions can have an extraordinary number of functions and consequently VM instances running at the same time. These reasons motivate extensive systems research in order to improve the problems faced in the serverless world.

Current serverless infrastructures follow one of two architectural models: container-based isolation or virtual-machine-based isolation. Container-based approaches offer low startup latency and minimal computational overhead, but they give up isolation guarantees and are not fully compatible with all binaries. In contrast, virtual machine (VM) based solutions provide robust isolation and support for a wider range of workloads, at the cost of higher resource consumption and slower initialization.

We build our work on Firecracker, a system developed by Amazon enabling hundreds or thousands of small MicroVMs to run on a single machine. Firecracker is used by AWS for serverless functions running with AWS Lambda.

### 1.1 Cold and Warm Starts

General serverless workloads run many different functions by a variety of clients at a given time. These functions run in individual MicroVMs. The number of functions running on a serverless machine is bounded by the machine's resources. This leads us to develop mechanisms that can preempt/suspend certain functions so others can run. This is commonly implemented in industry and in Firecracker with cold and warm states. Functions are served using a pool of running or "warm" machine instances. These instances will continue to exist as the function's workload persists. When demand for a given function is lessened, the system preempts these instances by taking machine snapshots and leaving them in a "cold" state outside of RAM and CPU. Preemption is




usually initiated after a period of inactivity, configured considering the time to remove an instance from the warm pool. This is important because every second a microVM is resident in memory but not running a function, the system is failing to fully utilize the resources available to our machine. Serverless function providers like AWS Lambda seek to achieve 99.9% utilization of machine resources, but existing cold start implementations are too slow to provide the granularity of preemption that is needed to improve utilization. Phoenix extends existing snapshot functionality in Firecracker to improve preemption capability and allow for finer-grained starting and stopping of serverless function instances.

## 1.2 Memory Duplication

The main insight of Phoenix is the fact that a large amount of memory across serverless instances is duplicated: functions utilize the same kernel, runtimes, and libraries. By decreasing memory duplication, we free up memory for more serverless functions to run, leading to more utilization of memory. Phoenix decreases the memory overhead of serverless instances by deduplicating similar memory accross non-executing microVMs.

## 1.3 Firecracker Snapshotting

Firecracker currently contains the capability to create snapshots of VMs which are written to disk and can be loaded and resumed on any machine running the same processor. These snapshots are split into two different files. Snapshot files contain the non-memory state of the VM such as configuration parameters and registers, and as such are very small- a few dozen kilobytes at most. Memory files. on the other hand, are raw dumps of physical memory, including all pages both mapped and unmapped by the VMs operating system; the default memory size of VMs in Firecracker is 128 MiB, so memory files are also 128 MiB. Phoenix builds off of the snapshotting capabilities of Firecracker. Since the majority of memory size and duplication in a VM snapshot is located in the VM's memory, Phoenix focuses only on compressing the size of memory files.

Firecracker has two modes of snapshots, full snapshots and diff snapshots. Full snapshots are complete dumps of a VM's memory state. After a full snapshot is loaded from disk and resumed, a diff snapshot can be taken which only contains data for pages which were written since the VM was resumed. Diff snapshot memory files are the same size as full snapshot memory files, simply containing empty data for pages which have not been modified; we assume this is because Firecracker developers consider disk storage to be inexpensive. The existing diff snapshot strategy could not be reused in Phoenix because Phoenix compresses differences between arbitrary snapshots, rather than just differencing a snapshot to its "parent" snapshot.

## 1.4 Medes

MEDES focuses on compressing duplicate memory pages across the cluster and was a major inspiration for the development of Phoenix. We specifically were motivated by the idea of deduplicating memory between serverless instances as a solution to the preemption problem. MEDES differs from Phoenix because it approaches the problem for a containerized serverless system. Phoenix is a drop-in replacement for Firecracker, where MEDES requires proprietary implementation and does not have the same compatibility with arbitrary binaries that Firecracker provides. In addition to this, we believe their use of off-the-shelf compression algorithms and final results do not provide enough of an improvement to convince users to switch to their implementation. Because Phoenix significantly compresses snapshot differences, Phoenix is able to maintain deduplicated snapshots in memory, which is faster to access on snapshot restore than MEDES, which stores deduplicated snapshots on distributed disk.

# 2 Project Phoenix

Phoenix decreases the memory overhead of serverless instances by creating a new "Ashes" state residing in memory. When an instance manager decides to preempt a MicroVM, instead of creating a full snapshot and putting the function in a "cold state", it has the option to create a deduplicated state which provides startup times an order of magnitude faster than cold starts while still drastically reducing the memory overhead of maintaining a warm pool. Phoenix employs several different strategies to achieve this.

## 2.1 Memory Compression Method

To decrease the amount of memory required to store an Ashes snapshot, Phoenix stores the deltas of snapshots relative to

a base snapshot. As mentioned earlier, serverless workloads have large amounts of reused kernels, libraries, and runtimes. By storing unused memory as a delta relative to a base snapshot with a high degree of common data, Phoenix is able to remove redundant sections of the memory dump.

# 3 Incinerator

A central component of our system is Incinerator, a purpose-built memory snapshot diff-compression algorithm designed for VM-based serverless platforms. Given two memory snapshots, a base snapshot and a derivative snapshot, typically generated by microVMs running similar workloads, Incinerator computes a compact, lossless diff. Using only the base snapshot and this diff representation, Incinerator can reconstruct the derivative snapshot exactly. Because Phoenix aims to provide fast startup times and significant memory usage reductions, Incinerator primarily prioritizes decompression speed and compressed size while still sufficiently optimizing compression speed such that real serverless operations are feasable.

At a high level, Incinerator processes each 4096-byte memory page in the derivative snapshot, identifies a similar page in the base snapshot, computes the byte-wise XOR between the two, and then compresses the resulting page-sized XOR array. The algorithm consists of three major stages.

## 3.1 XOR Compression

Given a base page and its corresponding derivative page, Incinerator computes their byte-wise XOR. When the two pages are similar, most bytes in the XOR array are zero, since the XOR of identical bytes is zero. The resulting 4096-byte XOR array is then compressed using a custom compression algorithm optimized for this domain: fixed-size inputs with a high expected density of zero bytes. A formal description of this compression scheme will be released separately, and the complete implementation is available in `incinerator/src/compression.rs`.

## 3.2 Page Matching

Selecting an appropriate base page for each derivative page is critical: better matches lead to XOR arrays with more zero bytes and therefore smaller compressed diffs. A brute force approach would compare each derivative page with every page in the base snapshot and choose the closest match. However, for 128MiB snapshots this requires $32{,}000^2 = 1.024 \times 10^9$ page comparisons, taking several dozen seconds on our evaluation hardware.

Instead, Incinerator employs a randomized sampling technique that identifies at most 64 candidate pages for comparison. The algorithm then performs exact byte-difference comparisons only on this candidate set and selects the closest match. Empirically, the best candidate found via sampling is typically within 1-2% of the brute-force optimum, while reducing comparisons by several orders of magnitude.

## 3.3 Base Preprocessing

To support fast page matching, Incinerator constructs two data structures: a `HashMap` of all pages in the base snapshot and a `SampledMultiHashMap` of all pages in the base snapshot. The `HashMap` is simple: it maps base snapshot page data (4096-byte arrays) to their indexes in the linear base snapshot address space. The `SampledMultiHashMap` is more complex and inspired by bit-sampling techniques for approximate nearest-neighbor search, the structure consists of 16 independent `SampledHashMap`s. Each `SampledHashMap` is parameterized by 8 byte positions uniformly selected from the half-open range $[0, 4096)$. For any page, the `SampledHashMap` extracts an 8-byte sample at those positions and maps the sample to a `RandomVec` containing up to 4 page indices.

Each `RandomVec` implements reservoir sampling: if fewer than 4 pages share a given sample, all are stored; if more than 4 exist, new pages replace existing entries with probability proportional to the number of pages processed so far.

During `SampledMultiHashMap` lookup, the query page's sample is computed for each of the 16 hashmaps, yielding at most 64 total candidate base pages.

### 3.3.1 Randomized Sampling Evaluation

Let $Q$ denote a derivative page being queried and $B$ the true best-matching base page. Three scenarios arise:

**No good matches:** If no base page is particularly similar to $Q$, then many/all base pages have approximately the same byte difference. In this case, even a suboptimal match causes minimal additional diff size.

**Few good matches:** Suppose $B$ differs from $Q$ by $d$ bytes, where $d$ is small (e.g., a few hundred). The probability that $Q$ and $B$ share the same 8-byte sample in one hashmap is:

$$p = \left(1 - \frac{d}{4096}\right)^8$$

The probability that they collide in at least one of the 16 independent hashmaps is:

$$q = 1 - (1 - p)^{16}$$

For small $d$, $p$ is large, so $q \approx 1$. Since few pages are similar to $Q$, few will share its sample, meaning `RandomVec` is unlikely to evict $B$ via reservoir sampling.

**Many good matches:** If many pages resemble $Q$, then many will share its samples. In this case, Incinerator does not need to select $B$ specifically: any highly similar page yields a small diff.

Across all regimes, the sampling scheme reliably returns a near-optimal match whenever one exists.

### 3.4 Parallelism and Usage Model

Base preprocessing is performed once per base snapshot, enabling the diffs for multiple derivative snapshots to be computed efficiently. Additionally, page matching and XOR compression for different pages are independent and trivially parallelizable. Our reference implementation is single-threaded to reflect realistic data-center deployments, where remaining cores are dedicated to active microVM workloads.

### 3.5 MemoryDiff Representation

Incinerator stores compressed per-page diffs and associated metadata (e.g., base-page indices) in a `MemoryDiff` structure, which can be serialized into a byte array. This serialized representation may optionally be recompressed using general-purpose algorithms such as `zstd` or `lz4` for additional storage reduction. The `MemoryDiff` implementation is available in `incinerator/src/diff.rs`.

A key property of `MemoryDiff` is that each page can be extracted in $O(1)$ time.[1] Thus, pages may be decompressed lazily by an on-demand fault handler without scanning diff data proportional to snapshot size. `MemoryDiff` also has special handling for some pages. Pages whose compressed diff exceeds 4088 bytes (4096 bytes minus metadata) are stored uncompressed. Pages which are exact copies of some page in the base snapshot are identified using the exact `HashMap` and are stored simply as a single index into the base snapshot, taking only 4 bytes. Pages which are entirely zero (more common than you may expect!) are special cased and also only contribute 4 bytes to the `MemoryDiff` structure.

### 3.6 Decompression

To reconstruct a derivative page, the decompressor retrieves its diff and base page index, loads the corresponding base page, decompresses the XOR array, and computes:

$$\text{derivative} = \text{xor} \oplus \text{base}.$$

This follows from the identity $(a \oplus b) \oplus a = b$.

[1] `MemoryDiff` contains one auxiliary structure which grows at a rate of at worst $n/2^{14}$ where $n$ is the number of pages. This structure must be binary searched on all page decompressions so technically the extraction is $O(\log(n))$, but the implied constant is small enough that in practice we expect this structure to have size 1 for all but the largest memory snapshots.

### 3.7 Granularity: Why 4 KiB Pages?

Incinerator operates at page granularity (4096 bytes) for all comparison, compression, and decompression operations. We evaluated both coarser and finer granularities. Larger units (e.g., 8192 bytes) fail to capture cross-page reordering in guest physical memory, while smaller units offer only marginal diff-size improvements at significant computational and memory cost. Page granularity provides the best balance of locality, similarity capture, and efficiency.

## 4 Integration with Firecracker

We integrate Incinerator into Firecracker through Phoenix, a userfaultfd (UFFD)-based page-fault handler responsible for serving memory pages on demand from compressed snapshot diffs. Phoenix maintains a single base snapshot in memory and supports multiple concurrently loaded derivative snapshots, each represented compactly using Incinerator's `MemoryDiff` structures.

### 4.1 System Overview

Phoenix exposes a REST API through which external management components (e.g., orchestration layers or Firecracker control tooling) interact with the system. The REST interface supports two primary operations:

1. *Load Derivative Snapshot:* Given a path on disk, Phoenix reads a derivative snapshot and compresses it relative to the base snapshot using Incinerator. The resulting diff is retained in memory and indexed by a user-provided snapshot identifier.

2. *Create Fault-Handler:* Phoenix spawns a dedicated fault-handler thread and associated UNIX-domain socket capable of servicing UFFD events for a specified derivative snapshot.

This design allows Phoenix to simultaneously manage numerous MicroVM images while storing only one full snapshot in memory: the base image.

### 4.2 Firecracker Integration

A Firecracker MicroVM communicates with Phoenix through the socket associated with its fault-handler thread. Once connected, Firecracker configures userfaultfd to route all page faults in the MicroVM's address space to the Phoenix handler.

When a page fault occurs, the UFFD subsystem delivers a fault event to the corresponding thread. The handler determines the faulting guest-physical page number, retrieves the corresponding compressed diff entry from the `MemoryDiff` structure, and invokes Incinerator's decompression pipeline to reconstruct the 4 KiB page. The reconstructed page is then

copied directly into the MicroVM's memory via UFFD's `UFFDIO_COPY` interface.

## 4.3 Operational Model

Since each derivative snapshot is stored solely as a diff against the shared base snapshot, Phoenix enables substantial memory consolidation across MicroVMs running similar workloads. Each MicroVM requires only:

- the base snapshot (shared across all MicroVMs), and
- the diff pages needed to reconstruct its private memory on demand.

Page reconstruction is performed lazily, triggered only when the MicroVM attempts to access memory not yet materialized. This allows the system to pay the cost of decompression only for pages actually touched by the workload, and to avoid loading unused pages altogether.

Overall, Phoenix provides a transparent, memory-efficient integration layer between Firecracker and Incinerator, enabling large numbers of lightweight MicroVMs to coexist with dramatically reduced memory duplication.

# 5 Evaluation

## 5.1 Experimental Setup

All experiments were conducted on a Framework Laptop 16 equipped with an AMD Ryzen 7 7840HS processor, 32 GiB of RAM, and a 2 TB WD Black SN850X NVMe SSD. The machine ran Arch Linux with all packages up to date as of December 1, 2025. The SN850X is a high-end consumer SSD with performance characteristics exceeding those of typical data-center block devices. Consequently, our reported numbers should be interpreted as optimistic upper bounds on real-world performance.

We evaluate both Incinerator and Phoenix using two classes of programs executed inside Firecracker MicroVMs: a simple Python workload and a matrix-multiplication workload based on the LINPACK benchmark. Each program was run inside a 128 MiB MicroVM, and memory snapshots were collected at execution points corresponding to the steady-state behavior of each workload.

## 5.2 Workload Descriptions

### 5.2.1 Simple Python Runtime Workload

Our first workload is a short Python script whose primary purpose is to induce the loading of the CPython runtime and standard library into the MicroVM's memory. The program performs a short sequence of prints, waits for user input, and then continues execution:

```
#!/usr/bin/python3
for i in range(10):
    print(i)
input()
for i in range(10, 20):
    print(i)
```

Although trivial, this program reflects common serverless behaviors: short-lived invocations with minimal application logic but nontrivial runtime footprint. Since most serverless executions are dominated by runtime initialization rather than user code, this benchmark allows us to measure Incinerator's effectiveness on realistic, low-entropy workloads with large shared components across invocations.

### 5.2.2 Matrix Multiplication (LINPACK) Workload

To stress the system with a high-entropy, memory-intensive workload, we use a matrix multiplication program adapted from the LINPACK benchmark by DDPS Lab's serverless FAAS workbench at https://github.com/ddps-lab/serverless-faas-workbench/blob/cf3e1e9c14870788a38dc38c5bb4be9e18fdcf3c/aws/cpu-memory/linpack/lambda_function.py

The workload initializes an $n \times n$ matrix of random floating-point values, computes a matrix–vector multiplication, and reports LINPACK-style MFLOPS and latency statistics (which we do not use in our analysis). The primary purpose of this benchmark is twofold:

1. *Generate many page faults.* Large matrices require repeated access to memory pages that have not yet been materialized in the MicroVM, forcing Phoenix to reconstruct pages on demand.
2. *Increase diff entropy.* Because the matrix is initialized with random data, pages in the derivative snapshot share little similarity with the base snapshot, challenging Incinerator's compression effectiveness.

We evaluate the workload using matrix sizes $n \in \{100, 1000, 1800\}$, where 1800 represents the largest dimension that fits within the 128MiB MicroVM.

## 5.3 Summary of Results

Across both workloads, we measure:

- *Diff-compressed snapshot size:* the reduction achieved relative to storing full derivative snapshots.
- *Page reconstruction latency:* time from UFFD fault receipt to successful `UFFDIO_COPY`.

- *Throughput under fault load:* the effective page-fault servicing rate when running the LINPACK workload.
- *End-to-end performance impact:* how fault-driven lazy loading affects total application runtime.

The simple Python workload demonstrates the regime where Incinerator excels: high similarity between snapshots yields small diffs and very low reconstruction latency. Conversely, the LINPACK workload stresses the system in the opposite regime, exposing worst-case behaviors where pages contain largely random data. Despite this, Incinerator maintains acceptable diff sizes and per-page reconstruction latency, and Phoenix is able to service page faults at a rate sufficient to sustain the workload without catastrophic slowdown.

# 6 Results

## 6.1 Compression

Figures 1 and 2 compare several compression methods across two workloads: the Simple Python program and the LINPACK-based `matmul` program (with matrix size 1800). Both benchmarks use memory snapshots of 128 MiB or 32,768 pages of 4 KiB each, the default configured memory size in Firecracker. All results were collected using our CLI tools, which measure compression and decompression time *excluding* disk I/O. This reflects the intended usage within Phoenix, where snapshots and diffs are stored and accessed entirely in memory.

We evaluate four configurations:

- *Zstd* : normal whole-file compression (no diffing).
- *XDelta* : a delta compression command line tool implementing the VCDIFF compression format [3]
- *Incinerator* : our full diff-compression algorithm.
- *Incinerator + LZ4/Zstd recompression* : optional whole-file recompression applied to the `MemoryDiff` output.

The Zstd baseline is included to illustrate the benefit of diff compression when snapshots share substantial similarity. Because Zstd treats snapshots as opaque byte arrays, it lacks the page-wise semantic alignment that Incinerator exploits.

**One-time vs. startup vs. per-fault costs.** The bar colors in the figures separate the compression and decompression time into different sections depending on where the time cost is paid.

- *Blue (compressed size:)* the size of the snapshot delta
- *Cyan (compression time):* a *one-time* preprocessing cost paid only once for the base snapshot and not paid on each derivative snapshot.
- *Green (compression time):* cost paid for specifically the derivative snapshot.
- *Red (decompression startup time):* the time spent in decompression that must be spent on snapshot load.
- *Yellow (per-page decompression time):* the time spent in decompression during page faults.

*Zstd* and *Xdelta* do not perform any base snapshot preprocessing so their entire compression time is paid on every derivative snapshot. Additionally, vecause *Zstd* and *Xdelta* compress whole files, they cannot be decompressed on a per-page basis and all of their decompression cost must be paid upfront. Because the whole file must be decompressed at snapshot restore, and the existing compression schemes do not benefit from lazy page loading. If we were to give up the ability to load pages lazily, it is faster to simply read all pages from disk upfront than to decompress using either existing compression scheme.

### 6.1.1 Simple Python Program

The Simple workload contains very little dynamic data; most memory belongs to the CPython runtime and standard library. As expected, this leads to extremely high similarity across snapshots.

- *Diff sizes are small:* Incinerator reduces the 128 MiB snapshot to only a few MiB.
- *Compression is fast:* page matching quickly identifies nearly identical pages.
- *Per-page decompression latency is very low*: a few hundred nanoseconds on average.

These results represent the target regime for serverless computing, where many functions share the same runtime image.

### 6.1.2 Matmul (LINPACK 1800) Program

The LINPACK workload initializes large matrices with random data, intentionally creating a worst-case scenario for diff compression.

- *Diff sizes are significantly larger* than in the Simple workload, as expected for high-entropy pages.
- *Incinerator still outperforms Zstd*: although pages are random, the structure of the MicroVM (kernel, CPython, libraries) still contains substantial shared regions.
- *Compression time increases modestly*: page matching must search larger candidate spaces when similarity is lower. However, because each page compares against a fixed upper limit of 64 other pages, the compression time per page does not grow unboundedly.

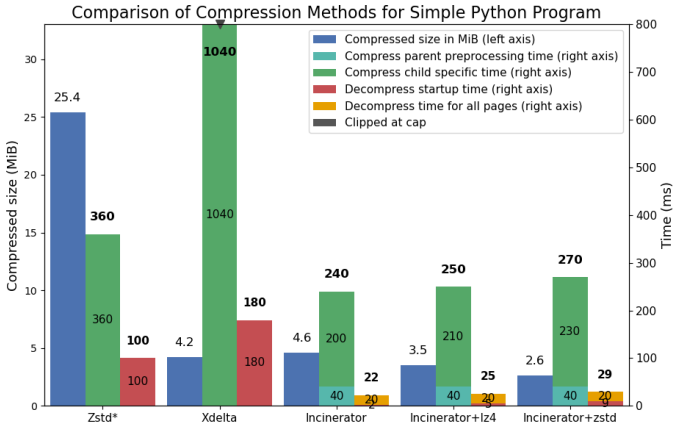



Figure 1: comparison of compression methods on simple python program

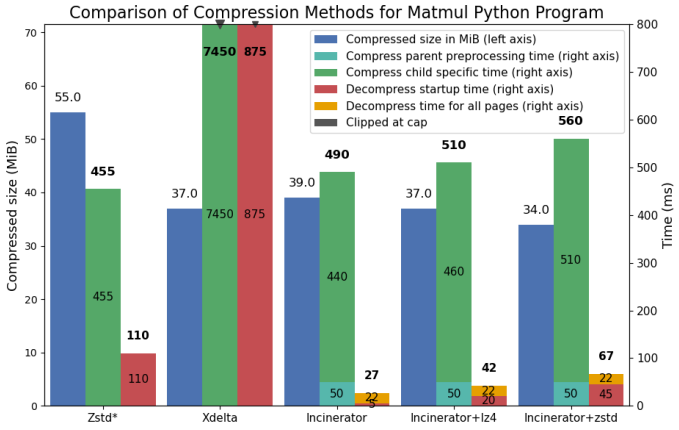



Figure 2: comparison of compression methods on linpack 1800

Even under worst-case memory layouts, Incinerator delivers meaningful compression improvements over non-diffed whole-file compression.

## 6.2 Page Fault Latency

To evaluate Phoenix in a realistic on-demand paging scenario, we compare its average page-fault servicing latency against a baseline implementation in which UFFD loads pages directly from the snapshot file on disk (*file-uffd*).

Before each run, we flush the Linux page cache using: `sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"`. This ensures that the file-based handler triggers actual disk reads rather than reading from the in-memory file cache. This matches the behavior expected in system cold-start situations in real serverless servers, where memory pressure will quickly evict the file's blocks from the file cache.

Table 1 presents our results in comparison to the file-based UFFD handler which read the required memory page from the snapshot file stored on disk. We interpret the data as follows:

| Workload | # Faults | Phoenix | File-UFFD |
|---|---|---|---|
| Linpack 100 | 6236 | 3413ns | 51,701ns |
| Linpack 1000 | 8184 | 4176ns | 52,417ns |
| Linpack 1800 | 13,236 | 3273ns | 36,239ns |

Table 1: Fault count and average UFFD page fault handler time

- Phoenix is consistently an order of magnitude faster than disk-backed UFFD.
- The file-UFFD handler spends the overwhelming majority of its time blocked on disk reads. When the file cache is *not* flushed, its average latency drops to $\approx 3\,\mu$s, confirming that disk latency dominates the fault cost. This also confirms that Phoenix spends only a few hundred nanoseconds decompressing pages with Incinerator,

since it is only a few hundred nanoseconds slower than serving a page directly from the in-memory file cache.

- Even on our very fast consumer NVMe SSD (SN850X), Phoenix outperforms file-UFFD by 10–15×. On typical cloud SSDs or hard disks, we expect the gap to widen further.

Phoenix's performance therefore benefits directly from Incinerator's small diffs and fast per-page decompression. Because all diff data reside directly in memory, page faults avoid disk I/O entirely.

**Impact on full workload runtime.** While we attempted to measure full end-to-end MicroVM workload runtime, precise instrumentation proved challenging. However, given that:

- disk-backed UFFD latency is $> 50\,\mu$s, and
- Phoenix provides $\approx$ 3–4 $\mu$s latency,

we expect end-to-end improvements proportional to the number of cold-start page faults encountered by the workload. For high-fault-count workloads (like LINPACK 1800), Phoenix should provide substantial wall-clock speedups.

**Kernel-based loading.** Although KVM's built-in snapshot loading mechanisms may avoid UFFD overhead, any approach requiring disk reads will still suffer from the same bottleneck identified above. Disk latency is the dominant cost, not UFFD messaging.

## 7 Conclusion and Future Work

Phoenix demonstrates that VM-based serverless platforms can achieve significantly faster cold-start behavior and higher memory efficiency by exploiting the substantial redundancy present across MicroVM snapshots. By introducing Incinerator, a lightweight diff-compression algorithm tailored for high-similarity memory workloads, Phoenix reduces snapshot footprints by orders of magnitude and achieves microsecond-scale page reconstruction latency through in-memory userfaultfd handling. These improvements enable finer-grained preemption of MicroVMs, higher consolidation on serverless hosts, and improved overall resource utilization. We believe that the tradeoff between storing compressed diffs in memory and gaining dramatically lower read latency is favorable for many serverless deployments.

There are several promising directions for future work. First, simple preprocessing of the base snapshot, such as removing zero pages or collapsing duplicate pages, may further reduce storage overhead. Integrating Phoenix more deeply with Firecracker or KVM could reduce userfaultfd overhead and improve end-to-end latency. Another avenue is double-diff compression, where the base snapshot is itself compressed relative to a third snapshot; during reconstruction, the system would decompress the base page first and then apply the derivative diff. Phoenix's REST interface could also be extended with snapshot deletion to reclaim memory when instances are no longer needed.

Beyond these extensions, Incinerator itself may benefit from new matching heuristics, improved compression schemes, or hardware-accelerated diffing. Finally, combining Phoenix with RDMA-based or remote UFFD handlers could allow a cluster to store a single shared base snapshot and serve diffs across machines, reducing memory consumption per host and enabling distributed snapshot management.

Additionally, we believe that the incinerator algorithm may be useful as the basis of a general diff compression algorithm. Incinerator compares derivative pages to all pages in the base, which may identify better diffing opportunities than many existing diff compression algorithms (such as VCDIFF [3]), which consider only one section of the base chosen arbitrarily. We believe space of xor-based diff compression algorithms has yet to be fully explored.

## References


[1] A. Agache, M. Brooker, A. Iordache, T. Leach, M. Pemberton, F. Popa, C. Raiciu, and A. Visan, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proc. 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[2] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, "Memory Deduplication for Serverless Computing with Medes," in *Proc. EuroSys '22*, Rennes, France, Apr. 2022, pp. 1–16. doi: 10.1145/3492321.3524272.

[3] D. Korn, J. P. MacDonald, J. Mogul, and K.-P. Vo, "The VCDIFF Generic Differencing and Compression Data Format," RFC 3284, Request for Comments, RFC Editor, Jul. 2002. doi:10.17487/RFC3284. Available: https://www.rfc-editor.org/info/rfc3284