0sim: Preparing System Software for a World with Terabyte-scale Memories

Mark Mansi
markm@cs.wisc.edu
University of Wisconsin – Madison

Michael M. Swift
swift@cs.wisc.edu
University of Wisconsin – Madison

Abstract

Recent advances in memory technologies mean that commodity machines may soon have terabytes of memory; however, such machines remain expensive and uncommon today. Hence, few programmers and researchers can debug and prototype fixes for scalability problems or explore new system behavior caused by terabyte-scale memories.

To enable rapid, early prototyping and exploration of system software for such machines, we built and open-sourced the 0sim simulator. 0sim uses virtualization to simulate the execution of huge workloads on modest machines. Our key observation is that many workloads follow the same control flow regardless of their input. We call such workloads *data-oblivious*. 0sim harnesses data-obliviousness to make huge simulations feasible and fast via memory compression.

0sim is accurate enough for many tasks and can simulate a guest system 20-30x larger than the host with 8x-100x slowdown for the workloads we observed, with more compressible workloads running faster. For example, we simulate a 1TB machine on a 31GB machine, and a 4TB machine on a 160GB machine. We give case studies to demonstrate the utility of 0sim. For example, we find that for mixed workloads, the Linux kernel can create irreparable fragmentation despite dozens of GBs of free memory, and we use 0sim to debug unexpected failures of memcached with huge memories.

CCS Concepts. • **Computing methodologies → Simulation types and techniques**; • **Software and its engineering → Memory management**; *Extra-functional properties.*

Keywords. operating systems; simulation; huge-memory system; memory capacity scaling; data-obliviousness

ACM Reference Format:
Mark Mansi and Michael M. Swift. 2020. 0sim: Preparing System Software for a World with Terabyte-scale Memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378451

1 Introduction

Computer memory sizes have grown continuously since the dawn of computing. Past designs that rigidly limited the maximum amount of memory faced huge difficulties as memory capacity increased exponentially (e.g., IBM's System/360 had an architectural limit of 28MB [11]). This growth trend will continue as technological advances greatly expand the density and decrease the cost of memory. Most recently, Intel's 3D Xpoint memory supports up to 6TB on a two-socket machine [48]. Consequently, multi-terabyte systems may become common, paving the way for future systems with tens to hundreds of terabytes of memory.

There is a pressing need to study the scalability of system software and applications on huge-memory systems (multiple TBs or more) to prepare for increased memory capacity. While tolerable today, the linear compute and space overheads of many common operating system algorithms may be unacceptable with 10-100x more memory. Other system software, such as language runtimes and garbage collectors also need redesigns to run efficiently on multi-terabyte memory systems [64]. In Linux, many scalability issues are tolerable for small memories but painful at larger scale. For example:

- With 4TB of memory, Linux takes more than 30s at boot time to initialize page metadata, which reduces availability when restarts are needed.
- Kernel metadata grows to multiple gigabytes on large systems. On heterogeneous systems, metadata may overwhelm smaller fast memories [26].
- Memory management algorithms with linear complexity, such as memory reclamation and defragmentation, can cause significant performance overhead when memory scales up 10x in size, as we show later.
- Huge pages are critical to TLB performance on huge systems, but memory defragmentation to form huge pages has previously been found to cause large latency spikes for some applications [4, 32, 58, 63].
- Memory management policies built for smaller memories, such as allowing a fixed percentage of cached file contents to be dirty, perform poorly with huge memories when that small percentage comprises hundreds

of gigabytes [59].

We expect system designers to encounter new scalability problems as memory sizes grow to terabyte scales and beyond. But, exploring system behavior with huge memories, reproducing scalability issues, and prototyping solutions requires a developer to own or rent a system at great cost or inconvenience. Cloud offerings for 4TB instances cost over $25 per hour [9, 41, 57]. Larger instances require a three-year contract at an expense of over $780,000 [10, 57].

To this end, we built and open-sourced 0sim ("zero·sim"), a virtualization-based platform for simulating system software behavior on multi-terabyte machines. 0sim runs on a commodity host machine, the *platform*, and provides a user with a virtual machine, the simulation *target*, which has a huge physical memory. 0sim is fast enough to allow huge, long-running simulations and even direct interaction, enabling both performance measurements and interactive debugging of scalability issues.

We employ several novel techniques to fit terabytes of target memory contents into gigabytes of platform memory. First, we observe that many programs are *data oblivious*: they perform the same computation independent of input values. Therefore, we can exercise the target with predetermined input; then, 0sim can compress a 4KB page with predetermined content to 1 bit. Second, current processors may have small physical address spaces to simplify the CPU design. We use software virtualization techniques to ensure that simulated physical addresses are never seen by the platform CPU, but instead are translated by 0sim. Thus, our system can simulate the maximum allowable address space for a given target architecture. Finally, we enable efficient performance measurement within the simulation by exposing hardware timestamp counters that report the passage of target time.

0sim simulates both functional and performance aspects of the target as if measured on a real multi-terabyte machine. 0sim does not aim to be an architecture simulator or to be perfectly accurate, as the target may differ from the platform in many ways including processor microarchitecture. Instead, 0sim simulates system software *well enough* for the important use cases of reproducing scalability problems, prototyping solutions, and exploring system behavior. 0sim sacrifices some accuracy for simulation speed, enabling huge, long-running simulations and interactive debugging.

In this paper, we describe the architecture and implementation of 0sim. We validate 0sim's accuracy and simulation speed with these goals in mind. 0sim can simulate a target system 20-30x larger than the platform with only 8x-100x slowdown compared to native execution for the workloads we tested, with more compressible workloads running faster. For example, we simulate a 4TB memcached target on a 160GB platform and 1TB memcached target on a 30GB platform with only 8x slowdown. By comparison, architecture simulators incur 10,000x or worse slowdown [21].

We perform several case studies demonstrating the usefulness of 0sim: we reproduce and extend developer performance results for a proposed kernel patch; we measure the worst-case impact of memory compaction on tail latency for memcached as a 22x slowdown; we show that for a mix of workloads Linux can incur irreparable memory fragmentation *even with dozens of GBs of free memory*; and that synchronous page reclamation can be made much more efficient at the cost of very small additional delay. Furthermore, we used 0sim to interactively debug a scalability bug in memcached that only occurs with more than 2TB of memory. 0sim is available at https://github.com/multifacet/0sim-workspace.

2 Problem: Scaling with Memory Capacity

0sim addresses a critical need to study how software scales with memory capacity, including making efficient use of memory, addressing algorithmic bottlenecks, and designing policies with huge memory capacity in mind. Moreover, it removes barriers for developers that limit software from being effectively tested and deployed for huge-memory systems.

Computational Inefficiency. Any operation whose execution time increases linearly with the amount of memory may become a bottleneck. For example, the Page Frame Reclamation Algorithm, huge page compaction, page deduplication, memory allocation, and dirty/referenced bit sampling all operate over per-page metadata. If the kernel attempts to transparently upgrade a range of pages to a huge page, running huge page compaction may induce unpredictable latency spikes and long tail latencies in applications. This has led many databases and storage systems to recommend turning off such kernel features despite potential performance gains [4, 32, 58, 63].

Likewise, allocating, initializing, and destroying page tables can be expensive. This impacts the time to create and destroy processes or service page faults. Linus Torvalds has suggested that the overhead of allocating pages to pre-populate page tables makes the `MAP_POPULATE` flag for `mmap` less useful because its latency is unacceptably high [75].

Another example is the Linux kernel's `struct page` [27]. Huge-memory systems may have billions of these structures storing metadata for each 4KB page of physical memory. In a 4TB system, initializing each of them and freeing them to the kernel's memory allocator at boot time takes 18 and 15 seconds, respectively, on our test machines. This has implications for service availability, where the boot time of the kernel may be on the critical path of a service restart.

Memory Usage. Any memory usage that is proportional to main memory size may consume too much space in some circumstances. For example, a machine with a modest amount of DRAM and terabytes of non-volatile memory may find all of its DRAM consumed by page tables and memory management metadata for non-volatile memory [26].

As previously mentioned, for each 4KB page, the Linux kernel keeps a 200-byte `struct page` with metadata. Similarly, page tables for address translation can consume dozens of gigabytes of memory on huge systems [37]. While this space may be a small fraction of total memory, it consumes a valuable system resource, and as discussed above, imposes a time cost for management.

Huge-Memory-Aware Policies. Effective memory management policies for small memories may perform poorly at huge scales. For example, the Linux kernel used to flush dirty pages to storage once they exceeded a percentage of memory, but on huge machines this led to long pauses as gigabytes of data were flushed out [59]. Also, some applications that use huge memories to buffer streaming data have found the kernel page cache to be a bottleneck [30]. In an era of huge and/or non-volatile memories, its not clear what role page caching should play. As high-throughput, low-latency network and storage and huge memories become more common, it is important to reevaluate kernel policies for buffering and flushing dirty data.

Likewise, policies for large contiguous allocations and fragmentation control need to be examined. Many modern high-performance I/O devices, such as network cards and solid-state drives, use large physically contiguous pinned memory buffers [33]. Researchers have proposed large contiguous allocations to mitigate TLB miss overheads [14, 39]. In order to satisfy such memory allocations, the kernel must control fragmentation of physical memory, which has been a problem in Linux [28, 30]. A complementary issue is the impact of internal fragmentation caused by eager paging [39] and transparent huge pages on multi-terabyte systems. These problems are well-studied for smaller systems, but to our knowledge they have not been revisited on huge-memory systems and workloads.

Capacity scalability problems are not unique to operating systems. Other system software, such as language runtimes, also needs to be adapted for huge memory systems. For example, Oracle's Java Virtual Machine adopted a new garbage collector optimized for huge-memory systems [64].

Barriers to Development. Huge-memory systems are uncommon due to their expense. Hence, system software is not well-tested for huge-memory systems that will become common soon. In our work, we often found that software had bugs and arbitrary hard-coded limits that caused it to fail on huge-memory systems. Usually, limitations were not well-documented, and failures were hard to debug due to user-unfriendly failure modes such as unrelated kernel panics or hanging indefinitely. `0sim` makes it easier for developers to test software at scale.

3 Related Work

Simulation is often used when hardware is unavailable (e.g., processor simulators [21]). Unlike other hardware advances, such as increasing processor cores or device speed, simulating memory capacity on existing hardware is challenging due to the amount of state that must be maintained. Alameldeen et al. address this by painstakingly scaling down and tuning the benchmarks and systems they test [8]. While accurate, this methodology is error prone, tedious, and difficult to validate. In Quartz and Simics, simulation size is limited by the size of the host machine [37, 76]. `0sim` is able to run huge simulations on a modest host by leveraging data-obliviousness to store more memory state than the memory or storage capacity of the host. Researchers have simulated fast networks by slowing down the simulated machine's clock [44]. `0sim` similarly adjusts the target's view of the passage of time. David [6] and Exalt [78] simulate large storage systems by storing only metadata and generating contents at read time. This technique is difficult for memory because important metadata is not separated from disposable data in most programs. `0sim` uses data-obliviousness with pre-determined inputs to accomplish a similar result. Several systems virtualize a cluster of machines to produce a single large virtual machine [16, 23, 72, 74]; Firesim uses cloud-based accelerators to scale simulations [52]. In contrast, `0sim` uses virtualization but on a single commodity host machine, which is more accessible to researchers and developers.

Scalability. Extensive prior work has examined different kinds of scalability problems in system software. For example, RadixVM tries to overcome performance issues in highly concurrent workloads due to serialization of memory management operations on kernel data structures [24]. Other studies have suggested that `struct page`, `struct vm_area_struct`, and page tables tend to comprise a large portion of memory management overhead [37]. Java 11 features a new garbage collector that allows it to scale to terabyte-scale heaps while retaining low latency [64]. However, there has been fairly little work aimed at improving the scalability of systems with respect to the memory capacity. `0sim` makes it easy to prototype and test software that address this problem.

Techniques. `0sim`'s design and implementation make use of a number of known software techniques to efficiently overcommit memory. One of our contributions is to show how `0sim` uses these techniques to build a novel approach to simulation. Page compression and deduplication can increase memory utilization in the presence of overcommitment; they are implemented in widely-used software, such as Linux, MacOS, and VMware ESX Server [2, 3, 12, 77]. In Linux, work has been done to increase the achievable memory compression ratio by using more efficient allocators [60] and optimizing for same-filled pages [35]. Remote-memory proposals swap pages out over the network to a remote machine, allowing larger workloads to run locally [38, 43]. Work has also been done on hardware-based memory compression [5] and zero-aware optimizations [36], but these proposals require specialized hardware, unlike `0sim`.

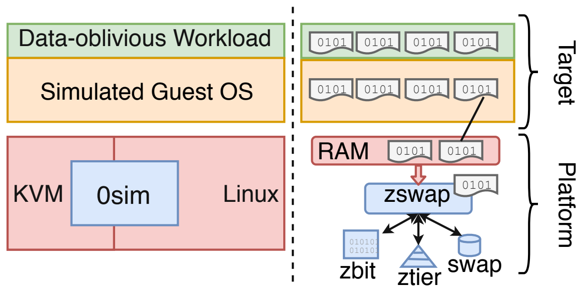


Figure 1. Design of 0sim (left) and Simulation State (right).

4 Design of 0sim

0sim enables evaluation of *system software* on machines with huge physical memories. We emphasize that 0sim is not an architecture simulator; instead it has the following goals:

- Run on inexpensive commodity hardware.
- Require minimal changes to simulated software.
- Preserve performance trends, not exact performance.
- Run fast enough to simulate long-running workloads.

Figure 1 (left) shows an overview of 0sim's architecture. 0sim boots a virtual machine (VM), the *target*, with physical memory that is orders-of-magnitude larger than available physical memory on the host, or *platform*, while maintaining reasonable simulation speed. 0sim is implemented as a modified kernel and hypervisor running on the platform but requires no target changes. *Any unmodified target OS* and a wide variety of workloads can be simulated by executing them in the target (e.g., via SSH). The x86 `rdtsc` instruction can be used in the target to read the hardware timestamp counter (TSC) for simulated time measurement.

0sim trades off simulation speed and ease of use of the system against accuracy by seeking to preserve trends rather than precisely predict performance. 0sim preserves trends in both temporal metrics (e.g., latency) and non-temporal metrics (e.g., memory usage) in the simulated environment. For example, 0sim can be used to compare the performance of two targets to measure the impact of an optimization.

The central challenges facing 0sim are (1) emulating huge memories and (2) preserving temporal metrics. We address (1) using data-oblivious workloads and memory compression. We address (2) by virtualizing the TSC.

4.1 Data-Obliviousness

Simulating huge-memory systems is fundamentally different from simulating faster hardware, such as CPUs or network devices. As previously mentioned, simulating huge memories requires maintaining more state than the platform is capable of holding. 0sim relies on the platform kernel's swapping subsystem to transparently overflow target state to a swap device. However, the platform may not have enough swap space for the state we wish to simulate; even if it did, writing and reading all state from the storage would be painfully slow and would make huge, long-running simulations impractical.

Our key observation is that many workloads follow the same control flow regardless of their input. We call such workloads *data-oblivious*. For example, the memcached in-memory key-value store does not behave differently based on the *values* in key-value pairs – only the keys. Another example is fixed computation, such as matrix multiplication; we can provide matrix workloads with sparse or known matrices. One workload, the NAS Conjugate Gradient Benchmark [13], naturally uses sparse matrices.

Figure 1 (right) depicts the management of target state in 0sim. Providing predetermined datasets to a data-oblivious workload makes it highly amenable to memory compression without changing its behavior. 0sim recognizes pages with the predetermined content (e.g., a zeroed page) and compresses them down to 1 bit, storing them in a bitmap called *zbit*. Pages that do not match the predetermined content can instead be compressed and stored in a highly efficient memory pool called *ztier*. This allows 0sim to run huge workloads while maintaining simulation state on a much more modest platform machine. Moreover, because much of the simulation state is kept in memory, zbit enables much faster simulation than if all state had to be written to a swap device. For example, on our workstations writing 4KB to an SSD takes about 24μs, while LZO compression [62] takes only 4μs.

0sim depends on data-obliviousness for simulation performance. Some interesting workloads are difficult to make data-oblivious, such as graphs and workloads with feedback loops. Nonetheless, to study system software, such as kernels, data-oblivious workloads usefully exercise the system in different ways including different memory allocation and access patterns. Thus, we believe data-oblivious workloads are sufficient to expose numerous problems, and that many of our findings generalize to other workloads. For example, much of the kernel memory management subsystem can be exercised because it is agnostic to page contents. We demonstrate this using several case studies in Section 8. Moreover, preparing systems for data-oblivious workloads benefits non-data-oblivious workloads too.

4.2 Hardware Limitations

Existing commodity systems may not support the amounts of memory we wish to study. For example, one of our experimental platforms has 39 physical address bits, only enough to address 512GB of memory, whereas we want to simulate multi-terabyte systems. This hardware limitation prevents running huge-memory workloads. 0sim overcomes the address-size limitation using shadow page tables [22]: the hypervisor, not hardware, translates the target physical addresses to platform physical addresses of the appropriate width. While not implemented, targets running virtual machines [17] or with 5-level paging, which was announced by

Intel but is not yet widely supported [46], can also be simulated with this technique. Similarly, 0sim supports memory sizes larger than the available swap space by transparently using memory compression in the hypervisor to take advantage of workload data-obliviousness.

4.3 Time Virtualization

Hardware simulators, such as gem5 [21], often simulate the passage of time using discrete events generated by the simulator. However, this is extremely slow, leading to many orders-of-magnitude slowdown compared to native execution. Such slowdowns make it impractical to study the behavior of huge-memory systems over medium or long time scales. Instead, 0sim uses hardware timestamp counters (TSCs) on the platform to measure the passage of time.

Each physical core has an independent hardware TSC that runs continuously. However, there are numerous sources of overhead in the hypervisor, such as page faults, that should not be reflected in target performance measurements. We create a virtual hardware TSC for the target that the hypervisor advances only when the target is running. We accomplish this with existing hardware virtualization support to adjust the target's virtualized TSC. Thus, within the simulation, the hardware reports target time.

5 Implementation

This section describes challenging and novel parts of 0sim's implementation. Note that 0sim only runs in the platform kernel; 0sim can run any unmodified target kernel. We implement 0sim as a modification to Linux kernel 4.4 and the KVM hypervisor. The kernel changes comprise about 4,100 new lines of code and 770 changed lines, and for KVM 400 new lines and 12 changed lines. By comparison, the gem5 simulator is almost 500,000 lines of code [21].

5.1 Memory Compression

We modify Linux's Zswap memory compression kernel module [3] to take advantage of data-obliviousness.

First, we modify Zswap to achieve compression ratios of 2^{15} for data in the common case: we represent zero pages with a single bit in the *zbit* bitmap, indicating whether it is a zero page or not. In practice, page tables and less-compressible pages (e.g., text sections or application metadata) limit compressibility, but ideally 1TB can be compressed to 32MB. In zbit, each page gets a bit. When selected for swapping by the platform, an all-zero target page will be identified by Zswap and compressed down to 1 bit in zbit. When the swapping subsystem queries Zswap to retrieve a page, it checks zbit. If the page's bit is set, we return a zero page; otherwise, Zswap proceeds as normal. Internally, zbit uses a radix tree to store sparse bitmaps efficiently.

Second, we observe that even non-zero pages can still be significantly compressed and densely stored. Zswap uses special memory allocators call "zpools" to store compressed pages. The default zpool, *zbud*, avoids computational overhead and implementation complexity at the expense of memory overhead. It limits the effective compression ratio to 2:1 and stores 24 bytes of metadata per compressed page [3].

We implement our own zpool, *ztier*, that significantly improves over zbud. All memory used by ztier goes toward storing compressed pages. It reduces metadata with negligible computational overhead by making use of unused fields of `struct page` and maintaining free-lists in the unallocated space of pages in the pool. Moreover, it supports multiple allocation sizes, leading to higher space efficiency. Thus, ztier achieves higher effective compression ratios than Linux's zbud at the expense of implementation complexity. For example, using zbud for a 500GB memcached workload on unmodified Zswap requires 294GB of memory. With zbit, zbud requires 15GB of RAM to store the compressed pages. In contrast, ztier consumes less than 6GB.

Overall, with our modifications a target page with predetermined content takes 66 bits of platform memory: 64 bits for a page table entry, 1 bitmap bit, and 1 bit amortized for upper levels of the page tables.

These optimizations allow the platform to keep most target state in memory, but a swap device is still needed since not all simulation state is compressible, and state may still need to overflow to the swap device. The amount of needed swap space depends heavily on the workload. Linux assigns swap space before attempting to insert into Zswap, even though it does not actually write to the swap space if Zswap is used. Thus, we thin-provision the swap space using device mapper [1] to look much larger than it actually is. We found that 1TB of swap space was sufficient for most of our workloads. Workloads with high churn or low compressibility required 2-3TB of swap space.

5.2 Shadow Page Tables

As mentioned in Section 4.2, 0sim uses shadow page tables to decouple the size of the target address space from the amount of physical address bits in the platform processor. The hypervisor reads the guest kernel's page tables and constructs the shadow page tables which are used by the hardware to translate guest virtual addresses to host physical addresses. Hardware never sees guest physical addresses. Thus, the target physical and virtual address spaces are as large as the platform virtual address space (48 bits, rather than 39 bits, on our machine).

When the platform has enough physical address bits, 0sim can optionally use hardware-based nested paging extensions [20]. Nested paging does not have the space overhead of shadow page tables, and is faster because the hypervisor is not on the critical path of address translation. However, the added simulation speed comes at the expense of some accuracy, as 0sim cannot account for the overhead of nested paging, which happens transparently in the hardware. Thus,

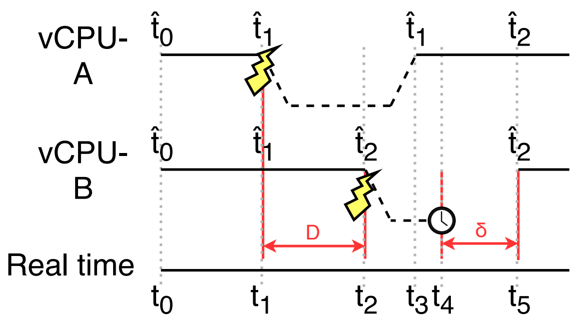


Figure 2. Example of proposed DTS mechanism. Both vCPUs start at target time $\hat{t}_0$ at platform (real) time t_0. At time t_1, vCPU-A pauses due to a trap or interrupt to the hypervisor, but vCPU-B continues to execute. At time t_3, vCPU-A continues, while vCPU-B is paused. At time t_4, vCPU-B is ready to run again but is ahead by D time units, so we delay it by δ time units to give vCPU-A time to reach target time $\hat{t}_2$. At platform time t_5, vCPU-A has caught up, so vCPU-B is allowed to run.

there is a tradeoff between simulation speed and accuracy; for more accuracy, one can disable nested paging extensions.

5.3 Time Virtualization

0sim virtualizes the `rdtsc` x86 instruction, which returns a cycle-level hardware timestamp counter (TSC). Each physical core has an independent TSC, and the Linux kernel synchronizes them at boot time. Most Intel processors have the ability to virtualize the TSC so that the guest TSC is an offset from the platform TSC of the processor it runs on [47]. 0sim adjusts this offset per-vCPU to hide time spent in the hypervisor rather than executing the target.

Virtualization itself has associated overheads. For example, the hypervisor emulates privileged instructions and I/O operations from the target kernel. We modify KVM to hide most virtualization overhead from the simulation. We record platform TSC values whenever a vCPU stops (i.e., when target time pauses). Before the vCPU resumes, we read the platform TSC again and offset the target TSC by the elapsed time.

Preserving timing in multi-core simulations presents an additional challenge because the hypervisor executes simulated cores concurrently on different platform cores. Since vCPUs can be run or paused independently by the hypervisor, their target TSCs may become unsynchronized. This can be problematic if timing measurements may cross cores or be influenced by events on other cores (e.g., synchronization events, responses from server threads). For some use cases, this can cause measurement inaccuracy.

In this section, we propose a solution to this problem, which we call *Dynamic TSC Synchronization* (DTS). While we have implemented DTS in 0sim, we leave it off for all the experiments in this paper because it increases simulation time and our current implementation can sometimes cause instability; more evaluation is needed before it should be used. While we have observed drift in our experiments, we find that 0sim is still accurate enough for many use case, as shown in sections 6 and 8.

DTS works as follows: To prevent excessive drift, 0sim delays vCPUs that run too far ahead to give other cores a chance to catch up. Specifically, let $\hat{t}_v$ be the target TSC of vCPU v. 0sim delays a vCPU c by descheduling it from running for δ time units if it is at least D time units ahead of the most lagging target TSC:

```
$\hat{t}_{min} := \min_v \hat{t}_v$        ▷ Most lagging target TSC
**if** $\hat{t}_{min} < \hat{t}_c - D$ **then** delay $c$ by $\delta$
**else** run $c$
```

Periodic events such as timer interrupts prevent cores from running ahead indefinitely. Figure 2 walks through an example. Generally, simulator speed will decrease as D decreases and as the number of simulated cores increases. Users can adjust the parameters D and δ as needed.

DTS has many desirable properties. Most importantly, it bounds the amount of drift between target TSCs to the threshold D. This means that it is possible to get more accurate measurements by measuring longer, since error does not accumulate. Moreover, it does not cause target time to jump or go backwards.

LAPIC Timer Interrupts. Operating systems commonly use the arrival of timer interrupts as a form of clock tick. For example, the Linux kernel scheduler and certain synchronization events (e.g., `rcu_sched`) perceive the passage of time using `jiffies`, which are measured by the delivery of interrupts. We virtualize the delivery of timer interrupts by delaying their delivery to the target kernel until the appropriate guest time is reached on the vCPU receiving the interrupt. We empirically verified that the target perceives an interrupt rate comparable to the platform.

Limitations. Because many events (e.g., I/O) are emulated by the hypervisor, there is no clear way to properly know how much time they would take on native hardware. Rather than guess or use an arbitrary constant, we opt to make such events take no target time; effectively, the target time is paused while they are handled by the hypervisor. As a result, 0sim is not suitable for measuring the latency of I/O events, though it can measure the CPU latency of events in I/O-bound processes. This is similar to other simulators [21].

Our scheme does not account for microarchitectural and architectural behavior changes from virtualization. Context switching to the platform may have microarchitectural effects that may affect target performance, such as polluting the caches or the TLB of the platform processor. For example, after the hypervisor swaps in a page from Zswap, the target may take a TLB miss, which is not accounted for.

In addition, 0sim does not perfectly preserve hardware events. In particular, Linux uses multiple sources of time, including processor timestamps (e.g., rdtsc), other hardware clocks (e.g., to track time when the processor sleeps), and the rate of interrupt delivery. 0sim only virtualizes processor timestamps and LAPIC timer interrupts.

In practice, we find that 0sim preserves behavior well, as we show in Section 6. However, targets see a higher-than-normal rate of I/O interrupts (e.g., networking or storage), which can lead to poor performance and crashes of I/O intensive workloads.

6 Simulator Validation

Before showing case studies in Section 8, we demonstrate that 0sim is accurate enough to be useful for reproducing scalability issues, prototyping solutions, and exploring system behavior. Since 0sim does not modify the data structures or algorithms of the target, it automatically preserves non-temporal metrics, such as the amount of memory consumed.

6.1 Methodology

0sim runs on a wide range of commodity hardware, from older workstations to servers. The specifications of our test platforms can be found in Table 1. wk-old is a 6-year-old workstation machine with 31GB of DRAM. wk-new is a 4-year-old workstation machine with 64GB of DRAM. Both machines cost around $1000-2000 when originally bought. server is a server-class machine with 160GB of DRAM. These machines all cost orders-of-magnitude less than a huge memory machine or prolonged rental of cloud instances.

We set the CPU scaling governor to performance. Hyperthreads and Intel Turbo Boost are enabled everywhere for consistency because the server testbed we used does not have a way to disable them.

In all simulations, the target OS is CentOS 7.6.1810 with Linux kernel v5.1.4 since the stock CentOS kernel is several years old. We disable Meltdown and Spectre [54, 56] mitigations because they cause severe performance degradation when the host is overcommitted.

To collect metrics from the simulation, we export an NFS server from the platform to the target. This has reasonable performance and does not introduce new performance artifacts. We provide workloads, such as memcached and redis, with all-zero data sets. We modify microbenchmarks, like memhog, to use all-zero values. In all experiments with server applications (e.g., memcached), we run the client program driving the workload in the same VM as the server. Otherwise, I/O virtualization quickly becomes the bottleneck, so measurements are actually measuring aspects of the hypervisor, not the target.

All multi-core simulations have 8 simulated cores on desktop-class machines and 6 simulated cores on server-class machines. We found that even unmodified KVM is unable to boot multi-terabyte virtual machines with more than 6 cores on server when the platform is overcommitted. We believe this is because KVM's emulated hardware devices use real (platform) time, and the overhead of running very huge machines causes hardware protocol timeouts. In the future, we would like to extend 0sim's virtualization of time to address this. In the meantime, we expect 6 simulated cores to be enough to exercise multi-core effects of software.

Our comparison baseline is direct execution (not simulation) on an AWS x1e.32xlarge instance with 3904GB which costs $26.818 per hour [9]. This is the largest on-demand cloud instance we could find. We have run simulations up to 8TB, but here we use a maximum size of 4TB for comparison with the baseline. While 0sim simulates the performance of a *native* execution (excluding time spent in the hypervisor), the AWS instance is a virtual machine, so the overheads of virtualization are present, though it is not overcommitted.

6.2 Single-core Accuracy

We first evaluate the accuracy of 0sim for single-core targets. To measure how well 0sim hides hypervisor activity from the target, we record the differences between subsequent executions of rdtsc. Figure 3a shows the CDF of timestamp differences. For all targets and the baseline, there are three main patterns: first, almost all measurements are less than 10ns, which is as fine-grained as rdtsc can measure and corresponds roughly to the pipeline depth of the processor. Second, there is a set of measurements of about 3μs. These correspond to the page faults from storing the results of the experiment. Finally, some measurements fit neither pattern and correspond to other system phenomena such as target interrupt handlers and times when the target kernel scheduler takes the processor away from the workload. 0sim hides hypervisor activity to closely approximate the baseline behavior on all three simulation platforms.

Conclusion 1: 0sim is able to hide idle hypervisor activity, such as servicing interrupts, from the target at the granularity of tens of nanoseconds up to the 99%-tile of measurements.

To validate that 0sim preserves the accuracy of the target OS, we run a workload that mmaps and sequentially touches all target memory. This causes significant kernel activity, as the kernel must handle page faults and allocate and zero memory. Likewise, significant hypervisor activity must be hidden from the target. Figure 3b shows the resulting time per page. Note that the server platform has much larger caches, aiding its performance. We see that 0sim is able to roughly preserve the time to touch memory, though we note that 0sim is not intended for such fine-grained measurements. The latency for the baseline machine is much higher (about 2.5μs) for 75% of operations because it includes hypervisor activity, such as allocating and zeroing pages, across NUMA nodes. In the simulations, hypervisor activity causes new pages to be cached, hiding inter-NUMA-node latency from the target.

Table 1. Specifications for test platforms.

Machine	OS	CPU	DRAM	Boot Disk	Swap Disk
wk-old	CentOS 7.6.1810 Linux kernel 4.4.0	Intel Core i7-4770K, 3.50GHz Haswell 2013 39-bit physical address 48-bit virtual address	31GB DDR3 1600MHz	1TB HDD SATA 6GBps 7200RPM	2TB HDD SATA 6GBps 7200RPM
wk-new	CentOS 7.6.1810 Linux kernel 4.4.0	Intel Core i7-6700K, 4.00GHz Skylake 2015 39-bit physical address 48-bit virtual address	62GB DDR4 2133MHz	100GB SSD shared with swap SATA 6GBps 555MBps seq read 500MBps seq write	10TB thin-provisioned device backed by 365GB SSD partition
server	CentOS 7.6.1810 Linux kernel 4.4.0	2x Intel E5-2660v3, 3.00 GHz Haswell 2014 46-bit physical address 48-bit virtual address	160GB DDR4 2133MHz	1.2TB HDD SAS 6GBps 10000RPM	10TB thin-provisioned device backed by 480GB SSD
baseline (AWS)	RHEL 8.0.0 Linux kernel 5.1	Intel E7-8880v3, 2.3 GHz Haswell 2015 46-bit physical address 48-bit virtual address	3904GB Frequency Unknown	30GB SSD AWS EBS gp2	N/A

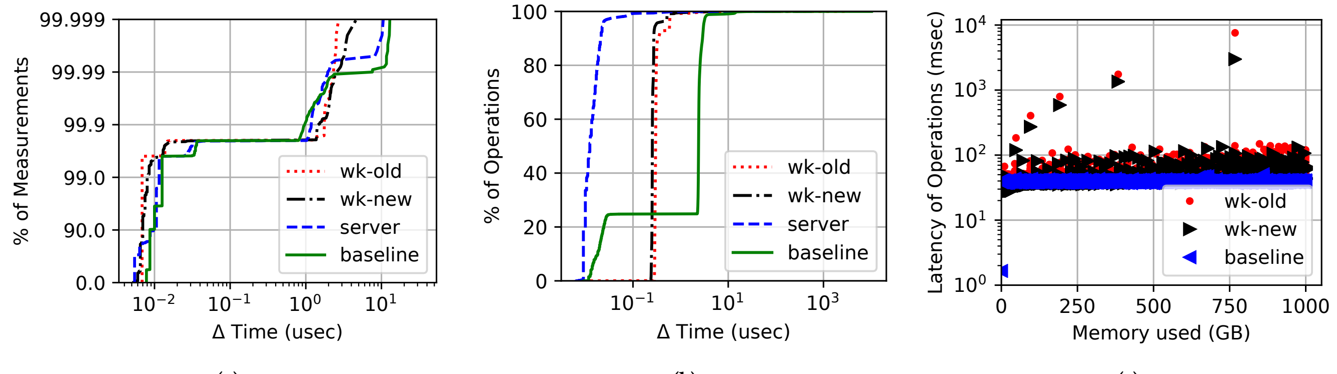


Figure 3. (a) CDF of Δ Time between subsequent calls to `rdtsc` in simulations on different host machines. Note the log scale. (b) CDF of latency to touch and fault pages in simulations on different host machines. (c) Latency of sets of 100 insertions to memcached in simulations on different host machines.

Conclusion 2: In the presence of significant platform activity, `0sim` *is able to preserve timing of events to within 2.5μs, though it does not model NUMA effects.*

To validate that `0sim` can simulate more realistic workloads accurately, we simulate a workload that fills a large memcached server. The keys are unique integers, while the values are 512KB of zeros. We measure the latency of sets of 100 insertions in a 1TB workload. Memcached is implemented as a hashmap, so the time to insert is roughly constant for the entire workload. Figure 3c shows that `0sim` preserves both the constant insertion time of memcached and the latency of requests compared to the baseline. The linear (note the logarithmic scale) trend of points at the top of the figure correspond to memcached resizing its hashmap, which blocks clients because they time-share the single core. This trend is not present in the baseline which is a multi-core machine and does resizing concurrently on a different core.

Conclusion 3: For real applications on single-core targets, `0sim` *is able to accurately preserve important trends, in addition to preserving high-level timing behavior.*

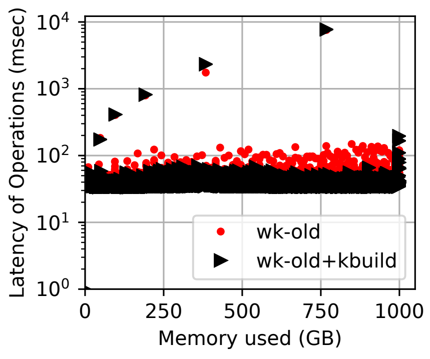
Conclusion 4: `0sim` *produces comparable results when run on different platforms, down to scale of tens of microseconds.*

To evaluate the sensitivity of simulation results to activity on the platform, we rerun the previous memcached experiment with a Linux kernel build running on the platform. Figure 4a shows the results on `wk-old` compared with the corresponding measurements from above. Despite the significant activity on the platform, the target sees similar results and trends to those collected above.

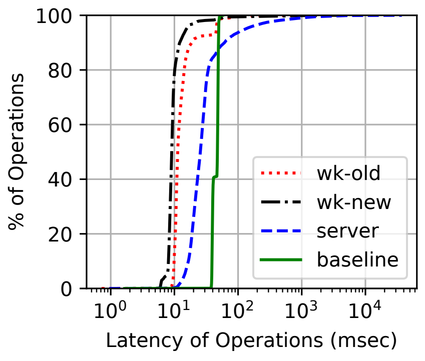
Conclusion 5: `0sim` *masks platform activity well enough to hide significant activity from the target.*

6.3 Multi-core Accuracy

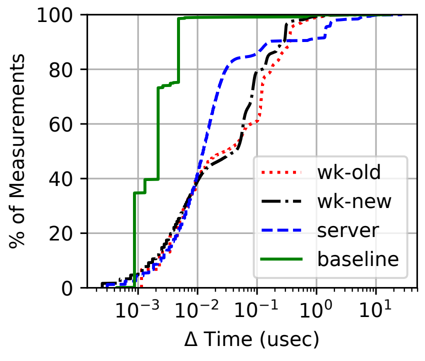
We evaluate `0sim`'s accuracy when running multi-core targets. A memcached client is pinned to one core and measures the latency of requests to a multi-threaded server not pinned to any core. Figure 4b shows a CDF of the measured latencies. The results are noisier than for the single-core simulations, as



(a)



(b)



(c)

Figure 4. (a) Latency per 100 sequential insertions to memcached while host is idle and doing a kernel build. (b) CDF of latency per 100 sequential insertions to memcached in multi-core simulations on different host machines. (c) CDF of Δ Time between subsequent memory accesses in simulations on different host machines for a workload with poor memory locality.

expected. 0sim is able to preserve the constant-time behavior of memcached, even without dynamic TSC synchronization. We believe the tail events for `wk-old` and `wk-new` represent increased jitter from multi-core interactions such as locking. We believe the long tail for `server` is due to the use of Intel nested paging extensions (EPT): nested page faults are hidden from the hypervisor, so we cannot adjust for them, as section 5.2 notes. Note that we measure 100 requests together, accumulating all of their cache misses, TLB misses, guest and host page faults, and nested page walks. One can obtain more accurate measurements by disabling EPT.
Conclusion 6: While 0sim produces less accurate results for multi-core targets, important trends and timing are preserved.

6.4 Worst-case Inaccuracy

To measure the worst-case impact of microarchitectural events, we run a workload with poor temporal and spatial locality that touches random addresses in a 4GB memory range. It incurs cache and TLB misses and both platform and target page faults, which are expensive since the host is oversubscribed. Figure 4c shows that these artifacts result in significant latency visible to the simulation. Around 90% of these events have a latency of 500ns or less, corresponding to the latency of cache and TLB misses in modern processors. Also, the server machine shows fewer such events, corresponding to its larger caches and TLB.
Conclusion 7: Experiments that measure largely microarchitectural performance differences such as TLB and cache misses may be inaccurate on 0sim.

7 Data-obliviousness and Speed

To achieve reasonable simulation performance, 0sim requires that the target have good compressibility. We believe this includes a large class of useful workloads. Table 2 reports the aggregate compressibility ratio for a few example workloads from our experiments, which is the average compressibility of all pages the kernel attempted to insert into Zswap (this is not the same as the ratio of platform memory to target memory). Note that an aggregate compressibility ratio of only 20:1 represents a 95% saving in memory usage over the course

Table 2. Observed aggregate compressibility ratio for various workloads. NAS CG (class E) runs with its natural dataset, a sparse matrix. Metis runs a matrix multiplication workload.

Workload	Platform	Target	Compressibility
memcached	62GB	1TB	215:1
redis	160GB	1TB	231:1
Metis	160GB	4TB	327:1
NAS CG	31GB	500GB	16:1

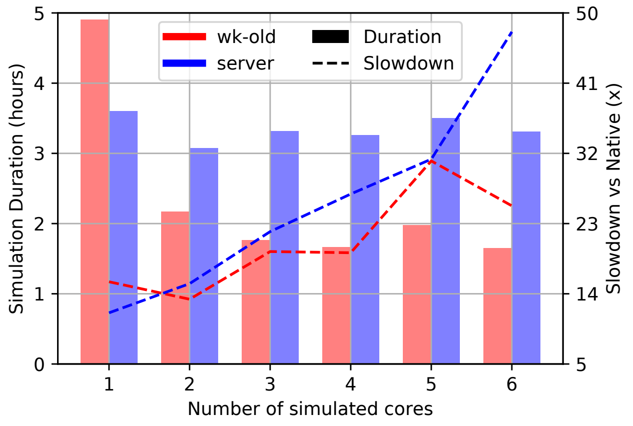


Figure 5. Simulation duration and slowdown compared to native execution (computed from TSC offset) for 1TB memcached workload as number of simulated cores varies.

of the workload. A few results deserve comment. Metis is an in-memory map-reduce framework [51]; unfortunately, it crashes mid-way through huge workloads, highlighting a need to test systems software on huge-memory systems. Also, NAS CG runs unmodified with its standard data set (a sparse matrix), which is compression-friendly but not data-oblivious. Overall, these results show that `0sim` is able to vastly decrease the memory footprint of huge data-oblivious workloads, making huge simulations feasible.

Data-obliviousness is critical to simulation performance and has some role in accuracy. We run an experiment in which we turn off Zswap, relying entirely on swapping. A 1TB memcached workload takes 3x longer to boot and 10x longer to run. Worse, the overhead is so high that despite `0sim`'s TSC offsetting scheme, overhead still leaks into the simulation and leads to target performance degradation proportional to the size of the workload.

Simulation speed depends heavily on the workload, platform, and number of simulated cores. Figure 5 shows the simulation speed of a 1TB memcached workload as the number of simulated cores increases. Generally, overhead increases with the number of simulated cores, but runtime may decrease due to improved workload performance.

In our experiments, we generally observe between 8x and 100x slowdown compared to native multi-core execution. For reference, this is comparable to running the workload on a late-1990s or early 2000s processor [71]. Architecture simulations often incur slowdowns of 10,000x or worse [21]. We found that workloads with heavy I/O are slower due to I/O virtualization.

Simulator performance degrades gracefully as platform memory becomes increasingly overcommitted. Users can balance simulation speed and scalability testing by varying the amount of target physical memory. In practice, we find that simulation size is limited by hard-coded limits and software bugs, rather than memory capacity. For example, KVM-QEMU does not accept parameter strings for anything larger than 7999GB. With engineering effort, such limitations can be overcome. Overall, we find that `0sim` makes huge multi-core simulations of data-oblivious workloads feasible and performant.

8 Case Studies

`0sim` is useful both for prototyping and testing and for research and exploration. We give case studies showing how we debugged scalability bugs in memcached, explored design space issues in Linux memory fragmentation management and page reclamation, evaluated a proposed kernel patchset, and reproduced known performance issues.

8.1 Development

The ability to interact with `0sim` workloads proved invaluable. While running experiments, memcached returned an unexpected out-of-memory error after inserting only two 2TB of data out of 8TB. To understand why memcached was misbehaving, we started an interactive (albeit slow) debugging session on the running memcached instance. We found that memcached's allocation pattern triggered a pathological case in `glibc`'s `malloc` implementation that led to a huge number of calls to `mmap`. This caused allocations to fail due to a system parameter that limits per-process memory regions. Increasing the limit resolves the issue.

This incident demonstrates that `0sim` is useful for finding, reproducing, debugging, and verifying solutions for bugs that only occur on huge-memory systems. A lead developer of memcached was unsure of the problem because they had not tried memcached on a system larger than 1.5TB. We hope that `0sim` can better prepare the systems community for the wider availability of huge-memory systems.

8.2 Exploration

We give two case studies in the Linux kernel demonstrating how `0sim` can be useful for design space exploration.

8.2.1 Memory Fragmentation .

Memory fragmentation at the application level and at the system level has been extensively studied in prior literature [7, 15, 18, 19, 34, 40, 42, 45, 49, 55, 61, 65, 66, 66–69, 73]. However, we are aware of little work addressing the effects of fragmentation in huge-memory workloads. Some have suggested that huge-memory workloads do not suffer extensively from fragmentation [14]. `0sim` is well-suited for studying such workloads and their system-level effects.

The Linux kernel allocator is a variant of the buddy allocator [42, 53, 68] and uses different free lists for different sizes of contiguous free memory regions. Specifically, there is a free list for each *order* of allocation, where the order-n free list contains contiguous regions of 2^n pages. The kernel merges free regions to form the largest possible free memory regions before adding them to the appropriate free list. Thus, if a large percentage of free pages is in the low-order free lists (closer to 0), memory is highly fragmented.

Methodology. We record the distribution of pages across free lists over time in the Linux v5.1.4 physical memory allocator. We simulate a 1TB Redis instance in isolation with snapshotting enabled. Redis periodically snapshots its contents: the Redis process forks, and the child writes its contents to disk, relying on kernel copy-on-write, and terminates, while the parent continues processing requests.

We then simulate a mixed workload: a redis client and server pair are used to represent a typical key-value server found in many distributed applications; a Metis workload represents a concurrent CPU and memory intensive computation; and a `memhog` workload modified to pin memory and be data-oblivious mimics high-performance I/O drivers that use large pinned physical memory regions for buffers [33]. These applications each receive 1/3 of system memory.

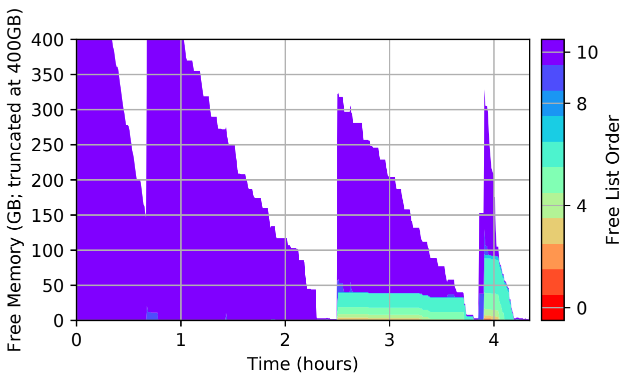


Figure 6. Amount of memory in each kernel page allocator free list in mix workload. The y-axis is truncated to 400GB to show more detail. More red indicates high fragmentation.

Results. Running alone, Redis does not suffer from fragmentation, but in the presence of other workloads it does. Figure 6 shows the amount of free memory in each buddy list throughout the mix workload. More purple (top) indicates more large-contiguous free physical memory regions, whereas more red (bottom) indicates that physical memory is highly fragmented. Each time free memory runs low, fragmentation degrades for subsequent portions of the workload: before time 2.5h, there is little fragmentation, but after 2.5h, almost 40GB of free memory is in orders 8 or lower, and after 4.2h, almost 100GB is in orders 8 or lower. Note that order 9 or higher is required to allocate a huge page. Upon closer inspection, we see that while most regions are contiguous, many individual base pages are scattered around physical memory. These pages represent some sort of "latent" fragmentation that persists despite the freeing and coalescing of hundreds of gigabytes of memory. This suggests that any true anti-fragmentation solution must also deal with this "latent" fragmentation.

The above results deal with *external fragmentation* – that is, fragmentation that causes the waste of unallocated space. While running our experiments, we also discovered that for some workloads, *internal fragmentation* (wasted space *within* an allocation) is a problem at the application-level. Specifically, we observed that in some cases, a 4TB memcached instance could only hold 2-3TB of data due to pathological internal fragmentation. If the size of values inserted does not match memcached's internal unit of memory allocation, memory is wasted. This wastage increases proportionally to the size of the workload, so it becomes problematic for multi-terabyte memcached instances. We had to carefully tune the parameters of memcached to get acceptable memory usage. These observations also suggest that internal fragmentation may be a more important problem on huge-memory systems.

Table 3. Time spent and pages scanned and reclaimed with different reclamation policies and modes.

Mode	CPU Time (s)	Scanned	Reclaimed
Idle	24	25,637,891	6,631,878
Direct	5	2,473,659	706,596
Idle 4x	21	19,594,007	5,657,472
Direct 4x	7	6,382,659	1,695,243

8.2.2 Page Reclamation.

Datacenter workloads may overcommit servers [31] to improve efficiency. A page reclamation algorithm satisfies kernel allocations when memory is scarce. In Linux, *direct reclamation* (DR) satisfies an outstanding allocation that is blocking userspace (e.g., from a page fault), while *idle reclamation* (IR) happens in the background when the amount of free memory goes below a threshold. Reclamation on huge-memory systems can be computationally expensive because it scans through billions of pages to check for idleness, potentially offsetting any gains made from the additional available memory. Google and Facebook both use in-house IR solutions to achieve better memory utilization efficiently [31]. Using 0sim, we measure the amount of time each algorithm spends per reclaimed page and explore policy modifications to the DR algorithm to attempt to make it more efficient.

Methodology. We run a workload that hogs all memory and sleeps. Then, we run a memcached workload that only performs insertions into the key-value store. This causes idle and direct reclamation from the hog workload. We instrument Linux to measure the time spent in idle and direct reclamation and the number of pages scanned and reclaimed.

Results. The top of Table 3 ("Idle" and "Direct") shows that DR costs 7μs per reclaimed page, whereas IR costs 3.5μs per reclaimed page but runs about 5 times longer. This makes sense; DR blocks userspace execution, so it is optimized for latency, rather than efficiency. Thus, it stops as soon as it can satisfy the required allocation, whereas IR continues until a watermark is met.

We hypothesized that DR would run less frequenctly if it were more efficient. We modify DR to reclaim four times more memory than requested. Table 3 (Idle 4x and Direct 4x) shows that direct and IR now both spend about 4μs per reclaimed page. Direct reclaim consumes about 2s more than before but runs about 36% less often, decreasing overall reclaimation time by 1s. This suggests that on continually busy systems, applications that can tolerate slightly longer latency may benefit from our modification.

8.3 Reproducing and Prototyping

0sim can be used to reproduce known scalability issues and prototype fixes for them. We demonstrate this with two case studies in the Linux kernel.

Table 4. Number of chunks and amount of time spent initializing and freeing memory during boot with ktask.

machine	cores	memory	chunks	init time	free time
wk-new	1	1TB	8035	3.5s	2.6s
wk-new	8	1TB	8035	1.1s	1.0s
server	1	4TB	32224	16.2s	13.1s
server	20	4TB	32224	2.5s	3.4s
no-ktask	20	4TB	1	18.2s	15.2s

8.3.1 ktask scalability. The proposed *ktask* patchset parallelizes CPU-intensive kernel-space work [29, 50], such as `struct page` initalization. Using `0sim`, we reproduce and extend developer results for the ktask patchset.

Methodology. We apply the patchset [50] to Linux kernel 5.1 and instrument the kernel to measure the amount of time elapsed during initialization using `rdtsc`. During boot, the structs are first initialized; then, they are freed to the kernel memory allocator, making them available to the system. We also record the number of 32,768-page "chunks" used by ktask, which can be initialized in parallel with each other.

Results. Table 4 shows 3x and 5x improvement in initialization for 1TB machine with 8 cores and a 4TB machine with 20 cores, respectively. This is proportional to the results posted by the patchset author for a real 512GB machine [50]. However, even with ktask, page initialization is still expensive! On a 4TB machine, almost 6 seconds of boot time are consumed, whereas, for example, an availability of 5 "nines" corresponds to about 5 minutes of downtime annually. Some prior discussions among kernel developers [25, 27] have investigated eliminating `struct page` for some use cases, and our results suggest that `struct page` usage in the kernel is unscalable. Memory management algorithms are needed that do not scale linearly with the amount of physical memory.

8.3.2 Memory Compaction. Using `0sim`, we reproduce and quantify memory compaction overheads. Huge pages and many high-performance I/O devices (e.g., network cards) [28, 30] require large contiguous physical memory allocations. The Linux kernel creates contiguous regions with an expensive memory compaction algorithm. Sudden compaction can produce unpredictable performance dips in applications, leading many databases and storage systems to recommend disabling features like Transparent Huge Pages, including Oracle Database [63], Redis [4], Couchbase [32], and MongoDB [58].

Methodology. We measure the latency of memcached requests in the presence and absence of compaction. Compaction usually happens in short bursts, making it hard to reproduce, so we modify the kernel to retry compaction continuously. Measurements with this modification approximate the worst-case during compaction on a normal kernel.

Results. Figure 7 shows the latency of memcached requests in the presence and absence of compaction for a 1 TB

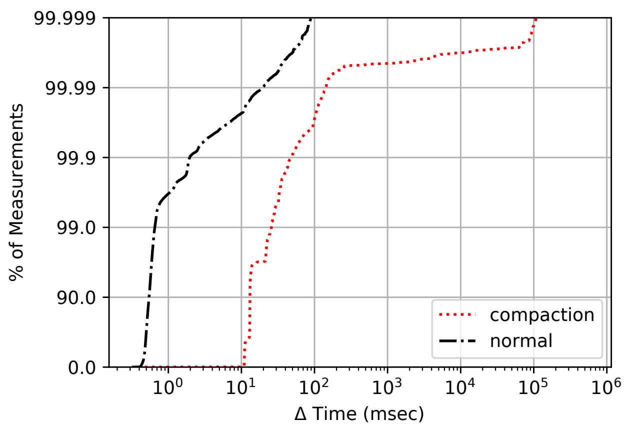


Figure 7. Latency of memcached requests in the presence and absence of continuous compaction. Note the log scale.

workload. Median latency degrades by 22x, while 99.999%-tile latency degrades by 10,000x, leading to occasional very-long latency spikes. Some of the tail effects are due to `0sim` overhead, but Figure 3a shows that this overhead cannot cause such a large effect. This suggests that in a production system, compaction can lead to events that define service tail latency. `0sim` can be used to further explore memory allocation and compaction policies for huge systems.

9 Conclusion

System scalability with respect to memory capacity is critical but under-studied. The memory usage, computational inefficiency, and policy choices of current systems are often unsuitable for huge systems. `0sim` is a simulation platform designed to address this problem. It takes advantage of the data-obliviousness of many workloads to make their memory contents highly-compressible. `0sim` runs on hardware that is easily available to researchers and developers, enabling both prototyping and exploration of system software. It accurately preserves behavior and trends. `0sim` allowed us to debug unexpected behavior. By open-sourcing `0sim`, we hope to enable both researchers and developers to prepare system software for a world with terabyte-scale memories.

Acknowledgments

We would like to thank Mark Hill, Swapnil Haria, Ram Alagappan, Yuvraj Patel, Anjali, and the anonymous reviewers for their insightful and helpful feedback. We would like to thank Bijan Tabatabai, Suhas Pai, Hasnain Ali Pirzada, and Varun Ravipati for their help in exploring workloads and developing infrastructure at various points in this project. We would like to thank GitHub user and memcached maintainer `dormando`, for their aid in debugging the memcached issues mentioned in Section 8.1. This works is supported by the National Science Foundation under grant NSF CNS-1815656.

References

[1] Linux Kernel Documentation: Device Mapper Thin-Provisioning. https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt.

[2] Linux Kernel Documentation: Kernel Samepage Merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[3] Linux Kernel Documentation: Zswap. https://www.kernel.org/doc/Documentation/vm/zswap.txt.

[4] Redis latency problems troubleshooting – Redis. https://redis.io/topics/latency.

[5] Bulent Abali, Hubertus Franke, Dan E. Poff, Robert A. Saccone, Charles O. Schulz, Lorraine M. Herger, and T. Basil Smith. Memory Expansion Technology (MXT): Software support and performance. *IBM Journal of Research and Development*, 45(2):287–301, March 2001.

[6] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. *ACM Trans. Storage*, 7(4):12:1–12:21, February 2012.

[7] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. Fast, Multicore-scalable, Low-fragmentation Memory Allocation Through Large Virtual Memory and Global Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2015.

[8] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, and Daniel J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *Computer*, 36(2):50–57, February 2003.

[9] Amazon Inc. EC2 Instance Pricing – Amazon Web Services (AWS). https://aws.amazon.com/ec2/pricing/on-demand/.

[10] Amazon Inc. Amazon EC2 High Memory Instances with 6, 9, and 12 TB of Memory, Perfect for SAP HANA. https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-sap-hana/, September 2018.

[11] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the IBM system/360. *IBM Journal of Research and Development*, 8(2):87–101, April 1964.

[12] Apple Inc. OS X Mavericks Core Technologies Overview. https://images.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overview.pdf, 2013.

[13] David H Bailey, E. Barszcz, John T Barton, D. S. Browning, R. L. Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Tom A Lasinski, Robert S Schreiber, Horst D Simon, V. Venkatakrishnan, and Sisira K Weeratunga. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991.

[14] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA, 2013.

[15] Leland L. Beck. A Dynamic Storage Allocation Technique Based on Memory Residence Time. *Commun. ACM*, 25(10):714–724, October 1982.

[16] C. Gordon Bell and Ike Nassi. Revisiting Scalable Coherent Shared Memory. *Computer*, 51(1):40–49, January 2018.

[17] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2010.

[18] Anna Bendersky and Erez Petrank. Space Overhead Bounds for Dynamic Memory Management with Partial Compaction. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2011.

[19] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[20] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2008.

[21] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[22] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[23] Matthew Chapman and Gernot Heiser. vNUMA: A Virtual Shared-memory Multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX ATC, 2009.

[24] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys, 2013.

[25] Jonathan Corbet. Persistent memory and page structures. https://lwn.net/Articles/644079/, May 2015.

[26] Jonathan Corbet. Persistent memory support progress. https://lwn.net/Articles/640113/, April 2015.

[27] Jonathan Corbet. ZONE_DEVICE and the future of struct page. https://lwn.net/Articles/717555/, March 2017.

[28] Jonathan Corbet. Improving support for large, contiguous allocations. https://lwn.net/Articles/753167/, May 2018.

[29] Jonathan Corbet. Ktask: Optimizing CPU-intensive kernel work. https://lwn.net/Articles/771169/, November 2018.

[30] Jonathan Corbet. Toward better performance on large-memory systems. https://lwn.net/Articles/753171/, May 2018.

[31] Jonathan Corbet. Proactively reclaiming idle memory. https://lwn.net/Articles/787611/, May 2019.

[32] Couchbase. Disabling Transparent Huge Pages (THP) | Couchbase Docs. https://docs.couchbase.com/server/current/install/thp-disable.html.

[33] Jean-Francois Dagenais. Extra large DMA buffer for PCI-E device under UIO. https://lkml.org/lkml/2011/11/18/462.

[34] D. Julia. M. Davies. Memory Occupancy Patterns in Garbage Collection Systems. *Commun. ACM*, 27(8):819–825, August 1984.

[35] Srividya Desireddy. [PATCH v2] zswap: Zero-filled pages handling. https://lkml.org/lkml/2017/8/16/560.

[36] Magnus Ekman and Per Stenstrom. A Robust Main-Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[37] Jakob Engblom. Simulating six terabytes of serious RAM. https://software.intel.com/en-us/blogs/2016/09/02/simulating-six-terabytes-of-serious-ram, 2017.

[38] Michael Joseph Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[39] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 36(3):118–126, May 2016.

[40] Erol Gelenbe, J. C. A. Boekhorst, and J. L. W. Kessels. Minimizing Wasted Space in Partitioned Segmentation. *Commun. ACM*, 16(6):343–349, June 1973.

[41] Google Inc. Google Compute Engine Pricing - Google Cloud. https://cloud.google.com/compute/pricing#machinetype.

[42] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Proceedings of the Linux Symposium*, volume 1, pages 369–384, January 2006.

[43] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.

[44] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time Warped Network Emulation. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP Poster Session, 2005.

[45] Daniel S. Hirschberg. A Class of Dynamic Memory Allocation Algorithms. *Commun. ACM*, 16(10):615–618, October 1973.

[46] Intel Inc. 5-Level Paging and 5-Level EPT. https://software.intel.com/en-us/download/5-level-paging-and-5-level-ept-white-paper.

[47] Intel Inc. Timestamp-Counter Scaling (TSC scaling) for Virtualization. https://www.intel.com/content/www/us/en/processors/timestamp-counter-scaling-virtualization-white-paper.html.

[48] Intel Inc. Intel's 3D XPoint™ Technology Products – What's Available and What's Coming Soon. https://software.intel.com/en-us/articles/3d-xpoint-technology-products, October 2017.

[49] Mark S. Johnstone and Paul R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management*, ISMM, 1998.

[50] Daniel Jordan. [RFC,v4,00/13] ktask: Multithread CPU-intensive kernel work - Patchwork. https://patchwork.kernel.org/cover/10668661/.

[51] Frans Kaashoek, Robert Morris, and Yandong Mao. Optimizing MapReduce for Multicore Architectures. Technical Report MIT-CSAIL-TR-2010-020, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 2010.

[52] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA, 2018.

[53] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[54] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, S&P, 2019.

[55] Ted G. Lewis, Brian J. Smith, and Marilyn Z. Smith. Dynamic Memory Allocation Systems for Minimizing Internal Fragmentation. In *Proceedings of the 1974 Annual ACM Conference - Volume 2*, 1974.

[56] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, USENIX Security, 2018.

[57] Microsoft Inc. Pricing - Linux Virtual Machines | Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.

[58] MongoDB Inc. Disable Transparent Huge Pages (THP) — MongoDB Manual. https://docs.mongodb.com/manual/tutorial/transparent-huge-pages.

[59] Andrew Morton. Re: [PATCH - RFC] allow setting vm_dirty below 1% for large memory machines. https://lkml.org/lkml/2007/1/9/80.

[60] Andrew Morton. Re: [PATCH v2] z3fold: The 3-fold allocator for compressed pages. https://lkml.org/lkml/2016/4/21/799.

[61] Norman R. Nielsen. Dynamic Memory Allocation in Computer Simulation. *Commun. ACM*, 20(11):864–873, November 1977.

[62] Markus F.X.J. Oberhumer. Oberhumer.com: LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[63] Oracle Inc. Database Installation Guide. https://docs.oracle.com/cd/E11882_01/install.112/e47689/pre_install.htm#LADBI1152.

[64] Oracle Inc. HotSpot Virtual Machine Garbage Collection Tuning Guide. https://docs.oracle.com/en/java/javase/11/gctuning/z-garbage-collector1.html#GUID-A5A42691-095E-47BA-B6DC-FB4E5FAA43D0.

[65] Ashish Panwar, Naman Patel, and K. Gopinath. A Case for Protecting Huge Pages from the Kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys, 2016.

[66] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2018.

[67] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, 2017.

[68] James L. Peterson and Theodore A. Norman. Buddy Systems. *Commun. ACM*, 20(6):421–431, June 1977.

[69] Aravinda Prasad and K. Gopinath. Prudent Memory Reclamation in Procrastination-Based Synchronization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[70] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:*, 39(6):36–38, December 2014.

[71] Karl Rupp. 40 Years of Microprocessor Trend Data. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/.

[72] ScaleMP Inc. ScaleMP - Virtualization for high-end computing. https://www.scalemp.com/.

[73] John E. Shore. On the External Storage Fragmentation Produced by First-fit and Best-fit Allocation Strategies. *Commun. ACM*, 18(8):433–440, August 1975.

[74] TidalScale Inc. Software Defined Servers. https://www.tidalscale.com/technology.

[75] Linus Torvalds. Pre-populating anonymous pages. https://www.realworldtech.com/forum/?threadid=185310&curpostid=185398, June 2019.

[76] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference*, Middleware, 2015.

[77] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI, 2002.

[78] Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-scale Storage Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2014.

A Artifact Appendix

A.1 Abstract

We have put significant effort into building tools that 0sim's installation and usage easy. We open-source 0sim and its tools to allow other researchers and developers to use and improve it in their own work. These tools should enable others to reproduce a subset of our results and conduct their own studies on huge-memory systems.

Our suggested workflow dedicates a machine to 0sim. A user drives 0sim via SSH from another machine such as one's desktop workstation or an instructional lab machine. The tooling we built assumes this setup.

Our artifact consists of three parts that work together: (1) the 0sim simulator itself; (2) the `runner` tool which is a program that runs the various experiments under the configurations presented in the paper. It is also extensible, so other users can modify it for their experiments; and (3) the `jobserver` tool which makes it easy to queue up and run large numbers of simulations on a remote machine or cluster.

A.2 Artifact check-list (meta-information)

- **Program:** 0sim simulator and tooling
- **Run-time environment:** Centos 7 (0sim kernel) + tooling
- **Hardware:** Two machines: (1) server or workstation, $\geq$ 32GB RAM, Intel x86_64 CPU with features `tsc`, `tsc_deadline_timer`, `tsc_adjust`, `constant_tsc`, `tsc_known_freq`, and (2) any other Linux machine with a persistent network connection
- **Execution:** Automated by tooling.
- **Run-time state:** Managed by tooling automatically.
- **Experiments:** Huge-memory Simulations, key-value stores, NAS CG, microbenchmarks (all run by tooling)
- **Metrics:** Simulation fidelity, speed, usability
- **Output:** Specific to experiment; see Table 5
- **How much disk space required (approximately):** On 0sim machine: 50GB in home directory + 1-2TB swap space.
- **How much time is needed to prepare workflow (approximately):** about 1 hour (mostly automated)
- **How much time is needed to complete experiments (approximately):** usually < 14 hours, up to 48 hours, per experiment. Parallelism recommended.
- **Publicly available:** Yes. Open-source on GitHub.
- **Code licenses (if publicly available):** 0sim is GPL; tooling is Apache v2.
- **Workflow framework used:** Custom tooling.
- **Archived:** DOI: 10.5281/zenodo.3560996, but we recommend using the master branch from GitHub.

A.3 Description

A.3.1 How delivered.

All source code is open-source and available on GitHub. Our workflow requires downloading https://github.com/multifacet/0sim-workspace, which includes 0sim and the tooling, to a *local* machine. The tooling downloads and installs 0sim on a remote machine. When cloning to the local machine, only a shallow clone is need, which is fast and requires less than 10MB of space.

A.3.2 Hardware dependencies.

Two machines are used: The "remote" runs 0sim. The "local" runs the tooling which drives the remote over SSH.

- The local machine can be an arbitrary machine but should have a persistent network connection and should be able to compile and run the tools in the `0sim-workspace`. We have only tested the tooling on Linux. The local should also have internet access to download dependencies for building the tools.
- The remote machine requires:
  - Intel x86_64 processor (no AMD support yet)
  - The following common CPU features, as reported by the `lscpu` command: `tsc, tsc_deadline_timer, tsc_adjust, constant_tsc, tsc_known_freq`.
  - 50GB storage space in the home directory for VM image and build artifacts.
  - 1-2TB swap space. SSDs preferred.
  - Internet access (can be behind a proxy) to clone 0sim and the tooling and to install dependencies.
  - If using CloudLab [70], the following profile works well: https://www.cloudlab.us/p/SuperPages/centos-n-bare-metal with Wisconsin cluster `c220g2` instances.

A.3.3 Software dependencies.

The following should be installed on the local machine:

- Linux (tested on Ubuntu). MacOS and Windows may also work, but are untested.
- `ssh`, `openssl`
- Stable Rust 1.37 or later, including `cargo`.

The following should be installed on the remote machine:

- `Centos 7` (kernel version does not matter). Our tooling uses `yum` and other Centos tooling. Centos gives guarantees about supported amounts of RAM, whereas Ubuntu does not as of this writing. RHEL, Fedora, or more recent versions of Centos might also work, but we have not tested them.
- SSH server listening at a well-known port.
- Other dependencies automatically installed by tooling.

A.4 Installation

Please see the `README.md` file of the https://github.com/multifacet/0sim-workspace repository, which contains a detailed step-by-step "Getting Started" guide.

A.5 Experiment workflow

The suggested workflow is documented more extensively in the `README.md` file of the `0sim-workspace` repository, linked above. At a high level, the suggested workflow is as follows:

1. The user writes the script for an experiment by adding a new module to the `runner` tool.

Table 5. Commands for experiments. `{MACHINE}` = IP:PORT of remote SSH server (e.g., `x.y.edu:22`). `{USER}` = username on remote. `{RESULTS}` = file path results. `{FREQ}` = frequency of remote CPU.

Experiment	Section	Commands to Run Experiments and Plot Results	
memcached	6.2	run (1 core)	`./runner exp00000 {MACHINE} {USER} 1024 1 -m`
		run (8 core)	`./runner exp00000 {MACHINE} {USER} 1024 8 -m`
		plot raw	`./plot-memcached_gen_data-time_per_op.py LABEL1:{RESULTS1} ...`
		plot CDF	`./plot-memcached_gen_data-time_per_op-cdf.py LABEL1:{RESULTS1} ...`
locality	6.2, 6.4	run	`./runner exp00002 {MACHINE} {USER} 100000 -l -v 1024 -C 1`
		plot CDF	`./plot-time-elapsed-deriv-cycles-cdf.py linear LABEL1:{RESULTS1} ...`
fragmentation	8.2.1	run (redis)	`./runner exp00007 {MACHINE} {USER} 30 -r --vm_size 1024 -C 1`
		run (mix)	`./runner exp00007 {MACHINE} {USER} 30 -x --vm_size 1024 -C 1`
		plot	`./plot-buddyinfo-over-time.py {RESULTS}`
compaction	8.3.2	setup	`./runner setup00001 {MACHINE} {USER} markm_instrument_thp_compaction`
		run	`./runner exp00003 {MACHINE} {USER} 1024 -C 1 --continual_compaction 1`
		plot	`./plot-time-elapsed-cycles-cdf.py close_to_one \`
			`      LABEL1:{RESULTS1}::{FREQ1} ...`

2. `runner` is executed on the local machine and uses SSH to execute commands on the remote machine, including repeatably setting up the remote machine and starting `0`sim with appropriate configurations.
3. The experiment outputs results on the remote which can be copied somewhere else for processing/analysis.

A.6 Experimental Results Format

After running, output from each experiment can be found in the directory `$HOME/vm_shared/results` on the remote. The name of each file contains important parameters of the experiment and timestamp to make filenames unique. The output for all experiments consists of multiple files, all with the same name but different extensions:

1. The data generated by the experiment (usually `.out`, but some experiments use other extensions, especially if there are multiple generated data files).
2. The parameters/settings of the experiment (`.params`), including the git hash of the workspace.
3. The time to run the experiment (`.time`).
4. Infomation about the platform and target, useful for debugging (`.sim`), including the output of `lscpu`, `lsblk`, and `dmesg`, memory usage, and `zswap` status.

A.7 Evaluation and expected result

We have two goals. First, since `0`sim is a (set of) tool(s), we hope that others will find our tooling useful and ergonomic. We have built our tooling to encourage reproducibility of results, including infrastructure for recording parameters and git hashes of code used to generate results. Second, the specific results in this paper should be reproducible. We have ourselves reproduced some of our results across multiple machines, as shown in section 6. In Table 5 we provide commands for reproducing a subset of our results, including our postprocessing scripts to generate the graphs in the paper. We give the 1TB versions of the experiments, but the parameters scaled up or down.

The first two rows of Table 5 refer to experiments from section 6. The remaining rows refer to experiments from the case studies in section 8. Some of the case studies require specially instrumented kernels installed in the target. This setup can be done with `runner` subcommand `setup00001`, and we have included these commands in the table where needed.

The plotting scripts are in the following repository: https://github.com/multifacet/0sim-plotting-scripts. The resulting plots should match the respective figures from the paper.

A.8 Experiment customization

The `runner` program is capable of running all of the experiments reported in this paper (see the usage message). There are a bunch of flags for each experiment that modify behavior and `0`sim configuration. Additionally, the code is well-documented and written to be easily extensible so users can add their own experiments. See the `exp00000.rs` module of the `runner` for an example of how to write an experiment.

A.9 Notes

Sometimes `0`sim will cause the remote to become unresponsive for large experiments or experiments with many cores. Often, restarting the experiment with more swap space is needed. More troubleshooting and known issues can be found in the README of the workspace repository.

A.10 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging