On-demand-fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications

Kaiyang Zhao
Purdue University
zhao776@purdue.edu

Sishuai Gong
Purdue University
sishuai@purdue.edu

Pedro Fonseca
Purdue University
pfonseca@purdue.edu

Abstract

Fork has long been the process creation system call for Unix. At its inception, fork was hailed as an efficient system call due to its use of copy-on-write on memory shared between parent and child processes. However, application memory demand has increased drastically since the early days and the cost incurred by fork to simply set up virtual memory (e.g., copy page tables) is now a concern, even for applications that only require hundreds of MBs of memory. In practice, fork performance already holds back system efficiency and latency across a range of uses cases that fork large processes, such as fault-tolerant systems, serverless frameworks, and testing frameworks.

This paper proposes On-demand-fork, a fast implementation of the fork system call specifically designed for applications with large memory footprints. On-demand-fork relies on the observation that copy-on-write can be generalized to page tables, even on commodity hardware. On-demand-fork executes faster than the traditional fork implementation by additionally sharing page tables between parent and child at fork time and selectively copying page tables in small chunks, on-demand, when handling page faults. On-demand-fork is a drop-in replacement for fork that requires no changes to applications or hardware.

We evaluated On-demand-fork on a range of micro-benchmarks and real-world workloads. On-demand-fork significantly reduces the fork invocation time and has improved scalability. For processes with 1 GB of allocated memory, On-demand-fork has a 65× performance advantage over Fork. We also evaluated On-demand-fork on testing, fuzzing, and snapshotting workloads of well-known applications, obtaining execution throughput improvements between 59% and 226% and up to 99% invocation latency reduction.

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456258

ACM Reference Format:
Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456258

1 Introduction

The fork system call has a long history [18, 55]. Fork was originally introduced in Unix to create new processes and was praised for its simplicity – the fork system call creates a child process with an exact copy of the entire address space of the parent process [63]. Since then the size of the memory used by applications and the cost of accessing memory tremendously increased, straining fork performance. Eventually fork implementations adopted the copy-on-write technique, which copies page tables *during* the system call invocation but defers the cost of (data) page copying to the fault handlers [65]. The copy-on-write approach significantly increased fork performance and kept fork practical for mainstream applications.

Fork is not just used for process spawning from a shell – fork is used in a wide range of situations, often to exploit the benefits of the *copy-on-write* semantics. For instance, key-value stores, such as Redis [60], use fork to get a *consistent* snapshot of the store, when fault-tolerance is required, while simultaneously serving client requests. Fuzzers and testing frameworks often use fork to avoid initialization overheads [8, 13, 31, 53]. More recently, serverless frameworks have exploited fork to cache computations for fast startup of short-lived lambda functions and to de-duplicate data [14, 25, 68].

However, the implementation of the fork system call is not simple and is no longer efficient given the demands of modern memory-intensive applications [5, 6, 18, 20]. The fork implementation requires duplicating nearly all process kernel state, which includes virtual memory, scheduler parameters, file descriptors, permissions, namespaces, etc. Of all kernel state components copied, the most concerning is the virtual memory – fork becomes significantly slower as the allocated memory of the application increases. Our measurements show that a process using 128 MB of memory can spend more than 0.8 ms executing the fork system call, and a process using 1 GB of memory can spend more than

6 ms. This cost increases linearly with the mapped memory size and may not be acceptable for memory-intensive or latency-sensitive applications.

Developers and system designers are taking notice of this problem and resorting to alternative methods to address the performance limitations of fork, one of which is to use huge pages. Even though huge pages make fork faster, by having less page table entries to copy on fork invocation, huge pages are plagued by high latency variation and latency spikes, high internal fragmentation, and inconvenient usage [42, 46, 57]. We analyze in more detail the limitations of huge pages in §2. This work proposes a fork design to address the emerging performance problems caused by the current fork approach. Our approach specifically targets memory-intensive and latency-sensitive applications and can coexist with the original fork.

This paper proposes On-demand-fork[1], a responsive and efficient fork system call that is specifically designed for memory-intensive and latency-sensitive applications. On-demand-fork extends the copy-on-write technique to *page tables*, by copying page tables in small chunks, as needed and deferred from the fork invocation. In particular, On-demand-fork ensures parent and child share the page tables at fork time and only copies the page tables when handling page faults. We implemented On-demand-fork on the Linux kernel by modifying the virtual memory subsystem. Although page tables are addressed by the CPU using physical addresses and faults are not permitted on page tables during translation, because page tables are only modified by the OS, our page table copy-on-write approach does not require special hardware support or modifications to applications. Hence, On-demand-fork is a drop-in replacement for the traditional Fork having the exact same semantics, which makes deployment practical.

Our experiments using micro-benchmarks show that On-demand-fork significantly reduces the fork system call latency, has reasonable page fault handling cost, and has lower total cost under many memory access patterns. We further evaluated On-demand-fork on real-world applications, including SQLite, Redis and AFL, that have large memory footprints and show that application end-to-end performance benefits significantly from On-demand-fork. On-demand-fork was able to achieve 226% improvements in execution throughput of AFL, 99% shorter fork latency in running SQLite unit tests, and 98% reduction in Redis snapshot blocking time.

This paper makes the following contributions:

- A detailed analysis of the performance characteristics of the (traditional) Fork implementation.
- The design of a responsive and efficient fork system call, On-demand-fork, that generalizes copy-on-write to page tables.
- A Linux-based implementation of On-demand-fork that runs on commodity hardware.
- An evaluation of On-demand-fork using microbenchmarks and real-world applications, demonstrating the high performance and applicability of On-demand-fork.

[1]On-demand-fork source code: https://github.com/rssys/on-demand-fork

```
for (int i = 1; i <= 120; i++) {
  size_t size = i * (1024 * 1024 * 512);
  void *buffer = mmap(NULL, size, ...);
  clock_gettime(..., &ts1);
  pid=fork();
  switch (pid) {
    case 0:    /* child */
      return 0;
    default: /* parent */
      clock_gettime(..., &ts2);
      print_cputime(ts1, ts2, size);
      waitpid(-1, NULL, 0);
  }
  munmap(buffer, size);
}
```

Figure 1. Fork benchmark program.

2 The Case for a Microsecond Fork

This section analyzes the performance of the Fork system call and makes the case for a responsive and efficient fork design. Our results show that, contrary to the early days' expectation, the current Fork design is no longer adequate for a wide-range of use cases given the increasing size of applications.

2.1 The Performance Status Quo of Fork

The Fork system call duplicates the kernel state of the parent process, which requires copying the process page tables – typically the largest process-specific kernel data structure. Thus, we would expect the execution time of the Fork system call invocation to be proportional to the number of pages mapped in the caller's virtual address space.

Using various microbenchmarks, we analyzed the performance of the Linux Fork system call to assess its impact on several use cases. In particular, we focused on analyzing the Fork scalability with increasing size of the parent process's virtual memory and the use of concurrent Fork calls.

Our benchmark program (Figure 1) consists of a loop that allocates private anonymous memory in 512 MB increments, fills it with data, and then forks. The time it takes to fork is measured by the parent using *clock_gettime* just before and after Fork. The child process immediately exits. The experiments ran on the same machine described in §5.1.

Figure 2 shows the results of our benchmark when configured to allocate a memory buffer that ranges from 0.5 GB to 50 GB. The results confirm that when the size of the caller's

virtual memory is sufficiently large, the cost of the fork invocation grows roughly linear with the size of the caller's allocated memory. They also show that the fork latency enters the *millisecond* range (i.e., > 1 ms) even for modestly sized applications with just 176 MB of allocated memory.

Additionally, we observed that the performance of fork degrades when called in parallel, even when the cores are not saturated. When only one instance of the benchmark is running on the testing machine with 16 physical cores, forking a process with 1GB of allocated memory takes on average 6.5 ms and a minimum of 5.4ms; when three concurrent benchmark instances fork a process with 1 GB of allocated memory, it takes on average 22.4 ms and a minimum of 21.3 ms. The results show that even if applications were to exploit concurrency, the standard fork *throughput* would still be very limited and the latency of each fork call would deteriorate. Fork's lack of scalability in multi-core systems is likely caused by atomic instructions that lock the system bus on the code path that reference counts physical pages.

The low performance of Fork is harmful for many applications. Some applications, such as Redis [60], use Fork on the critical path to perform online snapshots but also require very low request handling latency throughout the entire execution. In these cases, reducing the latency of the Fork invocation is particularly important, especially considering that in datacenters, latency can accumulate with long pipelines of services and other latency sources (e.g., interrupts, swap, garbage collectors) [38, 51]. Tail latencies fill up queues and degrade quality of service, thus, it is important to ensure that kernel services, including fork invocations, execute fast. Other applications that leverage Fork, such as testing frameworks, make Fork calls at a high rate and only expect the child process to access a tiny fraction of the memory allocated before exiting. In these cases, setting up page tables on-demand would prevent wasting time (and memory), hence improve the overall system efficiency.

It is worth noting that even 1 GB of memory is by no means large by today's standards. Databases and key-value stores, for instance, often use hundreds of gigabytes of memory [56]. However, our experiments have shown that when applications reach even just 50 GB of allocated memory, the average time to invoke fork reaches 253.9 ms and the minimum time reaches 252.3 ms, which is prohibitive for many use cases. Hence, having a faster and more responsive fork is especially important for current and future memory-intensive and latency-sensitive applications.

2.2 Bottleneck Analysis

To further understand the Fork cost for processes with large memory footprints, we used the perf-events toolkit [2] to analyze the Linux Fork implementation. For this analysis we modified our benchmark application to allocate memory

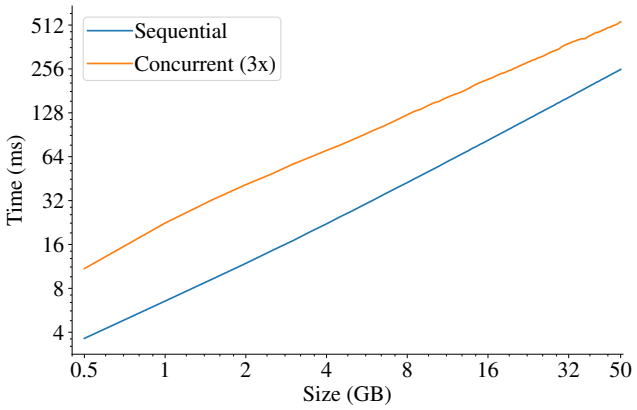

Figure 2. Fork execution time with different memory allocation sizes. Results include measurements with sequential executions and concurrent executions (with 3 concurrent instances of the benchmark). Tests ran on a 16-core machine and were repeated 5 times. Each run allocated a memory buffer that varied from 512 MB to 50 GB in 512 MB increments.

only once and fork repeatedly thereafter. Our profile results are shown in Figure 3. There are two hot spots in the virtual memory setup path of fork. The first is in *compound_head()*, which detects and resolves a *compound page*. A compound page is a grouping of multiple physically contiguous pages that can be treated as a single unit [1]. Compound pages are, among other purposes, used to implement huge pages in Linux. The high cost of *compound_head()* results from cache misses when accessing for the first time the struct page that stores the information. The second hot spot is in *page_ref_inc()*, which increments the reference counter of a physical page. For every physical page referenced in the last-level page tables, the kernel locates its corresponding struct page and increments the page reference counter atomically.

Although none of these functions operate on the same data when multiple instances of our benchmark run concurrently (i.e., each benchmark instance forks its own process), the additional slowdown observed is caused by memory and cache line contention when accessing struct page data structures that describe every physical page.

Both hot spot functions are in the kernel code that processes the *last-level page tables*, given that regular pages (4 kB) were used. The code handling the upper-level page tables merely traverses the page table hierarchies and has significantly lower execution cost. This observation motivates our design choice to copy the last-level page tables on-demand but still copy all upper-level tables at process creation time.

2.3 What About Huge Pages?

A straight-forward approach to mitigate the poor performance of Fork for memory-intensive applications is to switch

```
copy_one_pte():
0.00% |       mov    %r12,%rdx
0.57% |       callq  vm_normal_page
0.22% |       test   %rax,%rax
0.00% |       je     307
__read_once_size():
0.01% |       mov    0x8(%rax),%rcx
compound_head():
63.38%|       lea    -0x1(%rcx),%rdx
0.07% |       and    $0x1,%ecx
0.42% |       cmove  %rax,%rdx
arch_atomic_inc():
0.57% |       lock   incl   0x34(%rdx)
13.88%|       lock   incl   0x30(%rax)
__read_once_size():
15.27%|       mov    0x8(%rax),%rdx
```

Figure 3. Profiling results using perf-events when the benchmark program is running. The left column shows the percentage of time spent on the instruction during *copy_one_pte()*.

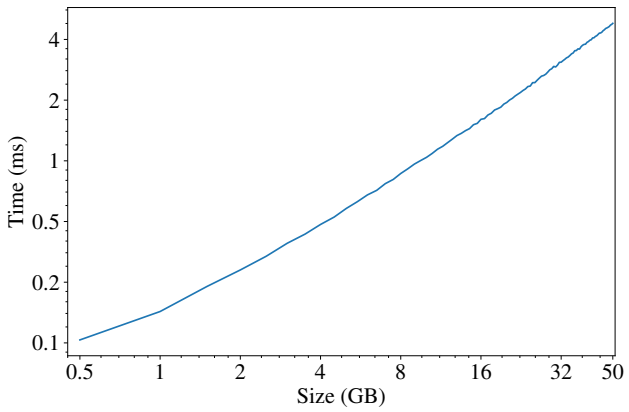

Figure 4. Time to fork vs. size of memory allocated with huge pages. Size is in 512 MB increments. Only 1 instance of the benchmark program ran. Tests ran on a 16-core machine and were repeated 5 times.

to huge pages. Most kernels, including Linux, provide an interface for applications to use different page sizes, depending on architectural support [12]. On x86, most CPUs can support 2 MB pages besides the standard 4 kB pages. Allocating the same amount of memory backed with huge pages needs fewer number of pages, which translates into fewer page table entries that need to be copied during Fork.

We ran the benchmark using huge pages (2 MB pages) and the results are shown in Figure 4. Our results show that forking a process with 1 GB of allocated memory with huge pages takes about 0.17 ms, which is over 50 times better than with regular 4 kB pages.

However, huge pages have serious drawbacks due to their coarser granularity and are not suitable for many applications. First, huge pages increase internal fragmentation, hence waste memory. Second, huge pages reduce the opportunities for parent and child to share memory, hence more memory will need to be duplicated during page faults caused by writes. Third, and perhaps more importantly, the time needed to handle write page faults increases approximately proportionally with the size of the page copied, hence huge page faults can take 512x longer to to handle than regular 4KB pages – this can be very detrimental to application performance [7, 61]. In particular, huge pages increase copy-on-write latency which can increases the application end-to-end tail latency in very significant and unpredictable ways. Hence, a common advise to improve application performance is to disable huge pages [23, 52].

Given these limitations, the application source code often has to be changed significantly to effectively use huge pages [59], creating a burden for developers; huge pages also have to be reserved before any allocation, creating a hassle for sysadmins. Transparent Huge Pages (THP) make huge pages more practical by obviating the requirement to reserve huge pages and modify application code. THP works by scanning memory pages in the background and finds opportunities to promote contiguous 4 kB pages to a 2 MB page [24]. However, THP is known to further increase CPU usage and cause long system-wide pauses [42, 57]. It is also widely acknowledged as especially harmful to database workloads [23, 52], as these workloads tend to have random rather than contiguous memory access patterns. In fact, THP setting defaults to `madvise` opt-in in popular Linux distributions, such as Ubuntu. We compare On-demand-fork with huge pages in §5.

2.4 Applications Benefit from a Fast Fork

Based on the observation that fork is inefficient for memory-intensive applications, this section make the case that a faster *microsecond fork* for regular 4KB pages would be very beneficial to various applications and enable novel system designs.

2.4.1 Snapshotting The fork system call is commonly used as a snapshot mechanism for high-performance in-memory databases [40, 45, 60]; these databases commonly use snapshots for persistence, query execution, and backup and recovery. However, as Fork does not scale well, databases with large memory footprint suffer from long fork invocation latency and could fail to serve user requests during the fork call [61]. A fast fork with high scalability would speed up snapshotting significantly for such memory-intensive applications, and fulfill the low-latency performance requirements of these applications.

2.4.2 Testing In software development, Fork is widely used to increase testing performance. As most test inputs cause short-lived executions, the cost of initializing the target application typically dominates the cost of software testing. To mitigate this problem, state-of-the-art testing frameworks [8, 13, 31, 53], including fuzzers, usually initialize the

target application only once and then leverage Fork to repeatedly duplicate the process at a properly initialized state. Although the traditional Fork is useful, it is still a major performance bottleneck especially when the application uses a large amount of memory or when the testing framework has to call Fork concurrently [5, 20, 69]. A fast Fork with a much shorter invocation time and higher efficiency is expected to significantly improve testing.

2.4.3 Serverless Computing Fork and related techniques are also used in serverless computing to optimize the startup time of lambda functions, which is crucial for serverless applications due to their short-lived execution. A large body of prior work [14, 25, 68] focuses on reducing lambda startup time by exploiting fork or similar techniques to implement lambda function caching, cloning, and de-duplication. A fast and efficient fork would provide a simple and general mechanism to improve significantly the performance of emerging serverless frameworks.

3 On-demand-fork

We address the limitations of fork with On-demand-fork, a redesign of the standard fork implementation that executes within the microsecond range without using huge pages. On-demand-fork reduces the cost of process creation by employing copy-on-write on the kernel virtual memory structures.

The profiling results in Figure 3 show that, when the size of the allocated virtual memory is large, most of the cost of fork comes from reference counting physical pages that back virtual memory. This process occurs when the kernel analyzes and copies the last-level page tables of the parent process during the fork invocation. On-demand-fork ensures that when the child process is created, it shares the *page tables* with its parent process, as shown in Figure 5. Hence, the bulk of the work of copying the virtual memory structures from the parent to the child is deferred to the page fault handler, and is performed *selectively* and *on-demand* according to copy-on-write semantics. Consequently, On-demand-fork significantly improves the kernel *responsiveness* when applications have moderate or large virtual memory spaces. Furthermore, On-demand-fork improves the overall kernel *efficiency* when applications only write to a small fraction of pages after process creation. Figure 6 shows the main events of On-demand-fork during application execution.

3.1 Overview

Paging Structure. Modern computer architectures feature MMUs that support hierarchical paging data structures. Linux supports up to 5 levels of page tables [35], namely Page Global Directory (PGD), P4D (if 5-level paging is enabled), Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entry Table (PTE table).

Last-level Page Table Sharing. On-demand-fork reduces the fork cost by making the parent and the child share *last-level page tables*. On-demand-fork adds reference counters to shared last-level page tables and modifies several kernel functions that touch them, including functions invoked by the page fault handler and the On-demand-fork, `munmap`, and `mremap` system calls. Shared last-level page tables may survive beyond the creating process lifetime to remain valid for other processes in the child-parent lineage tree.

During the system call invocation, On-demand-fork copies the top 4 levels of page tables of the parent and assigns them to the child. The write permission of all pages controlled by the shared tables is disabled by clearing the writable bit in the PMD entry of each shared last-level page table. Disabling the write permission ensures that the kernel page fault handler will capture subsequent write attempts to memory to implement copy-on-write. On-demand-fork then increments the reference counter of the PTE tables of the parent and has the child's PMD entries point to them. We chose not to share page tables at all levels in our current implementation because page tables are structured as a tree (with a 512 branching factor for $4\,\mathrm{kB}$ pages), so there are many more last-level page tables (leaf nodes) than upper-level page tables (non-leaf nodes) and we do not expect significant performance gains for most use cases to justify a more complex design.

Handling Writes On-demand. When a process with shared page tables attempts to write to memory mapped by a shared page table, a page fault is raised. The page fault handler recognizes that the page table is shared by reading its reference counter. On a page fault, the kernel allocates a dedicated PTE table for the faulting process and copies the shared PTE table entries. This operation only happens once per process and per last-level page table, which represents a $2\,\mathrm{MB}$ range (i.e., only the first write access to such a range incurs an increased cost). Although in general, copying page tables all at once may be good for throughput, by coping lazily, On-demand-fork gives applications much shorter fork invocation latency. This improved responsiveness applies even to write-intensive applications. Furthermore, this design increases overall system efficiency when processes do not write to subsets of the memory address space.

3.2 Challenges

Writable regions in the parent have to be write-protected before page tables can be safely shared, but changing each last-level page table *entry* would be slow. Hence, On-demand-fork must employ an efficient method to disable the write permission of all pages mapped by a shared last-level page. On-demand-fork leverages the fact that page tables are hierarchical data structures and that attributes in the upper-level

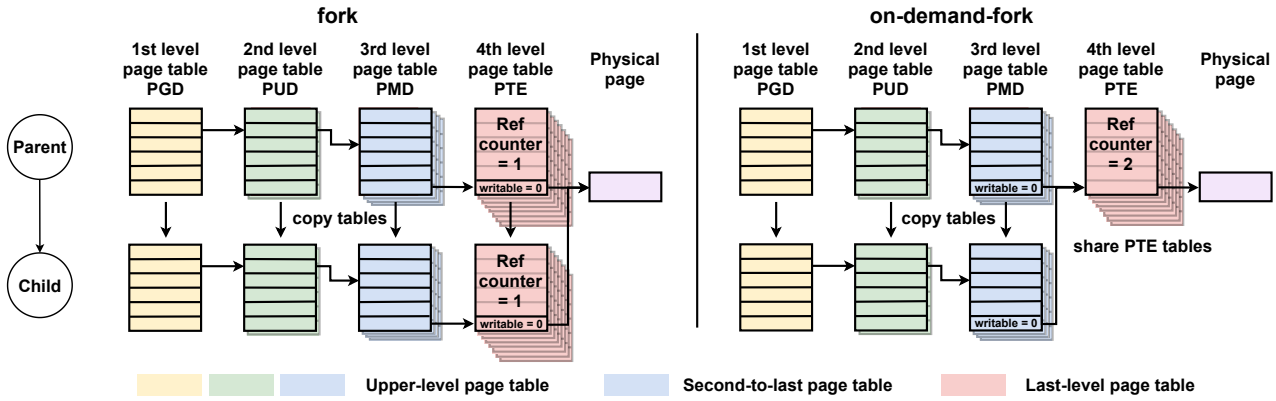

Figure 5. Fork and On-demand-fork page table management comparison. Unlike Fork, On-demand-fork shares the last level page tables across processes, which represent the vast majority of the page table structure.

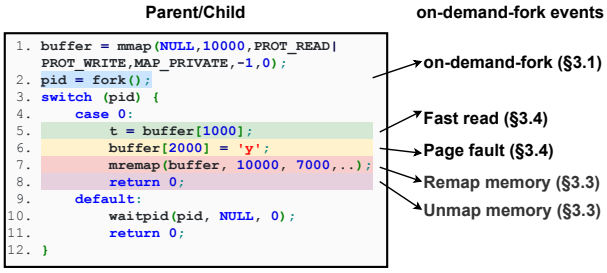

Figure 6. Main events when using On-demand-fork in a sample program fragment.

page table override attributes in the lower levels – "hierarchical attributes" capability [36] are supported by most mainstream architectures. In particular, On-demand-fork uses the capability to control the write permission of the entire region mapped by a last-level page table by modifying a single entry in upper-level page tables.

Page table entries include "accessed" and "dirty" bits that are modified by the CPU. The "accessed" bit is used, most importantly, for picking a page to evict under high-memory pressure; under On-demand-fork the CPU still marks pages mapped by a shared page table as "accessed", as normal. During page faults On-demand-fork duplicates the "accessed" bit value when copying shared page tables. The "dirty" bit will never be set while page tables are shared because the write permission is always disabled for shared pages.

3.3 Unmapping or Remapping VMAs

When unmapping or remapping (i.e., moving) a memory region, the kernel has to clear the corresponding page table entries to prevent the process from accessing the unmapped or remapped memory regions. Hence, On-demand-fork needs to change the system calls that perform these operations. For shared last-level page tables (PTE tables), the kernel can clear the reference to PTE tables in PMD entries and preserve the values of shared page table entries. But if multiple virtual memory areas (VMA) are mapped by the same shared PTE table, the kernel has to copy the shared page table before unmapping or remapping the region since other active VMAs of this process still need their corresponding PTE entries. In essence, On-demand-fork performs copy-on-write on these operations if the PTE tables are shared.

3.4 Subsequent Memory Access

After a call to On-demand-fork, both the parent and the child process share the PTE tables, with write permission disabled. Since there is no distinction between the parent and the child process, this approach allows for unlimited processes sharing a page table due to multiple On-demand-fork invocations. If a process reads from a memory address mapped by a shared page table, virtual memory translation is done normally without triggering page faults (*Fast Read* in Figure 6). However, if a process writes to memory mapped by a shared page table, a page fault is raised. In this case, the page fault handler, in addition to copying the page, will copy the shared last-level page table, i.e., it will perform copy-on-write on the *page table*.

In practice, On-demand-fork first checks whether the PTE table is shared by reading its reference counter. If the page table is shared, the kernel makes a copy of the shared PTE table for the faulting process and then decrements the shared last-level page table's reference counter. The new page table will no longer be shared, hence, subsequent writes to any addresses mapped by the page table, in the 2 MB region, will not trigger additional page table copies. If the counter of the previously shared last-level page table reaches the value one after the decrement operation, then both the previously shared table and the new table become dedicated (i.e. not shared).

To enable the new last-level page table, the corresponding entry in the PMD table of the faulting process will be changed to point to the new dedicated PTE table. In addition, the write permission bit in the PMD entry will be turned on to allow entries in the PTE table to determine the write permission of individual pages. Finally, as in regular Fork, the page

fault handler copies the (data) pages as needed and ensures that the new PTE table uses the new page copy (while still sharing the other pages).

On-demand-fork relies on the observation that *page tables* are data structures read by software and hardware, but only *written* by the kernel (with the exception of the accessed and dirty bits). This is distinct from *pages*, which are written by applications (i.e, the child/parent process). Thus, the kernel is always in control when page tables are modified and can copy page tables as needed before it modifies any. Hence, a copy-on-write approach on user data pages requires hardware support to trigger faults when pages are modified, but copy-on-write on page tables can be implemented by just modifying the kernel.

3.5 Last-Level Page Table Lifecycle

Every last-level page table has its reference counter, which is initialized to one in the last-level page table constructor. The count always matches the number of processes that share the page table. Last-level page tables are copied during page faults when a process that makes a write to pages mapped by shared page tables. The faulting process gets its own copy of the table (with the reference counter set to one) and the old table, which might still be shared by multiple processes, will have its counter decremented.

During the On-demand-fork system call invocation, the reference counter of every last-level page table in the paging structure of the process is incremented, indicating an additional user of the tables and preventing their premature deallocation. Write permission is disabled for all writable memory regions controlled by the table. This process is illustrated in Figure 5. During unmapping and remapping, the reference counters of last-level page tables mapping the respective memory regions are decremented, indicating one fewer user of the tables. If any page table reaches a zero reference count, its destructor is called.

3.6 Physical Pages Accounting

Anonymous mappings are backed by physical pages, which are also reference counted. The kernel uses this reference count to decide whether a *page* can be freed. Because On-demand-fork defers processing PTE tables, which includes incrementing the *page* reference counter, it must prevent pages from being prematurely freed when (a) a page fault gets a new page for copy-on-write and (b) the memory region gets unmapped or remapped.

On-demand-fork does not attempt to accurately track the usage of every page. Rather, it leverages the reference count of the last-level page table (PTE table) to decide whether the page is free or in-use. If the page in question is mapped by a PTE table that has a reference count greater than one, then the page cannot be safely freed.

3.7 File-backed Mappings

File-backed mappings are important and frequently used because the executable files of programs are mapped using file-backed mapping. In addition, many application use file-backed mapping to conveniently perform file I/O. On-demand-fork provides full support for file-backed memory mappings. Every file-backed mapping has an operating function structure pointing to functions that handle the construction, destruction, and faulting of the region, which forward the events to the page cache and filesystem subsystems of the kernel. On-demand-fork identifies file-backed regions and, similarly to Fork, leaves the work of managing physical memory pages to those functions.

4 Implementation

We implemented On-demand-fork on Linux kernel version 5.6.19. We added 1108 lines of code (LoC), modified 23 LoC, and removed 40 LoC of the mainline code. Most of the changes were done to the memory subsystem.

Memory Usage. We deliberately avoided adding new fields to the Linux data structure *struct page*, which describes the physical pages for both general memory and page tables. Because every physical page has a corresponding *struct page*, any increase in its size would be magnified by the amount of RAM installed in the machine and consume a large amount of memory. Hence, our implementation stores the reference counters of shared page tables in a union inside the *struct page* structure that is unused for last-level page tables. In essence, we exploited the free space in existing kernel data structures.

Robustness. PTE tables may need to be allocated in the page fault handler. Under low memory conditions, the faulting process is put to sleep, and the kernel takes appropriate action to free more pages (e.g., flush buffered pages, swap out pages, or invoke the OOM killer). We tested our implementation with an assortment of correctness unit tests that create, change, and destroy mappings under different scenarios. Our implementation passed all functional tests conducted.

Flexibility. Different applications have different performance requirements. For some applications it may be better to pay the cost of process creation upfront and have a lower runtime latency due to page faults; others may write to a substantial portion of the memory after process creation, not conforming to the design assumptions of On-demand-fork. Hence, we implemented On-demand-fork, a new system call, as an opt-in alternative for Fork. On-demand-fork is designed as a drop-in replacement for fork, so the only change needed in the application source code is replacing fork calls with On-demand-fork calls in suitable places. We also implemented a process-specific configuration through

procfs to provide full application transparency while allowing users to choose the fork implementation, hence On-demand-fork requires no changes to the application code.

Thread Safety. On-demand-fork protects the integrity of paging structures under concurrency. Whenever page table entries are changed, locks are acquired in accordance with the kernel locking protocol. Because multiple processes can share page tables, it is important that they are safe from corruption or incorrect reference counting. Since we treat shared tables as read-only, the contents of them are safe. The reference counters of PTE tables are always modified using test-and-set atomic instructions when one is being dereferenced and potentially freed.

Huge Page Support. The On-demand-fork implementation only supports 4 kB pages. However, its design is generalizable to support larger page sizes (e.g., 2 MB and 1 GB on x86). For example, on x86, the Linux kernel creates a 2 MB huge page by describing a page directly in a PMD entry instead of pointing to a PTE table; On-demand-fork can support 2 MB pages by sharing PMD tables describing them. This would offer to applications already using huge pages the advantages of On-demand-fork, although as discussed in §3, the benefits would be limited because of the smaller number of upper-level page tables.

Portability. Although we chose to implement On-demand-fork in x86, our design is general and should be easy to port to other architectures that support hierarchical attributes in page tables (e.g., ARMv8 [15]). The only architecture-specific code in On-demand-fork is for handling hardware-defined page table structures, and we expect porting to ARMv8, for instance, mostly involves adding code for page table entry manipulation. However, there are two features of ARMv8 that might increase porting complexity. Armv8-A has three different translation block sizes ("translation granules" [47]), 4 kB, 16 kB, and 64 kB, that provide more flexibility for the kernel to configure the system with different page sizes and different page table sizes; supporting this function would likely require adding more code to the page table entry manipulation. In addition, ARMv8 provides a contiguous bit in page table entries that Linux uses to provide alternative huge page sizes (e.g., 64 kB and 32 MB huge pages under 4 kB granule) [66]; such huge pages are not created by reducing the depth of page tables and may also require additional page table handling logic.

5 Evaluation

5.1 Setup

We ran all experiments on a machine with a 16-core AMD EPYC 7302P CPU and 256 GB memory. In all our experiments we compare the original fork system call (Fork) in the unmodified 5.6.19 Linux kernel with our fork implementation (On-demand-fork) in the modified kernel. We use the unmodified kernel as the baseline to measure the original fork performance because our page fault handler modifications could affect its performance. For all experiments, swap space is disabled to avoid a detrimental performance impact. Huge pages are disabled except when they are an evaluation target in specific experiments.

5.2 Microbenchmarks

5.2.1 Methodology This section evaluates On-demand-fork through a set of microbenchmarks. In particular, it addresses the following questions:

1. How fast is On-demand-fork compared with Fork (with and without huge pages)?
2. What are the overheads introduced by On-demand-fork when handling page faults?
3. How is memory access performance after On-demand-fork call?

Unless otherwise specified, the following configuration is used by microbenchmark programs: the read and written memory is private anonymous mappings allocated by mmap, backed by 4 kB pages. Huge pages are 2 MB and are always tested with the original fork. The memory is initialized (written to) before any measurement to ensure that every page is backed by a physical page. Time duration is measured using clock_gettime() with a CLOCK_MONOTONIC clock.

The fault handling measurement code makes the parent wait for the child exit before starting the next iteration because tearing down the child virtual memory has non-negligible costs that would add measurement noise.

5.2.2 System Call Latency We measure the time it takes to call Fork and On-demand-fork for a process with different allocated memory sizes. Time is measured by the parent just before calling the system call and immediately after it returns to the parent.

As shown in Figure 7, On-demand-fork has significantly better scalability and lower overhead than Fork, specially when the application allocates a large amount of memory. In particular, On-demand-fork takes 0.10 ms with 1 GB of memory and 0.94 ms with 50 GB of memory. Fork, on the other hand, clearly shows a very significant latency increase over the range of allocated memory considered, taking 6.54 ms with 1 GB of memory and 253.94 ms with 50 GB of memory. The performance of On-demand-fork over Fork at 1 GB of memory is 65× better and grows to 270× better at 50 GB.

On-demand-fork is also slightly faster than huge pages combined with Fork for two reasons: (i) On-demand-fork does not allocate memory for new page tables during system call invocation, and (ii) On-demand-fork does not acquire the spin lock of PMD entries, which is necessary for huge pages to prevent concurrent conversions by THP between normal pages and huge pages.

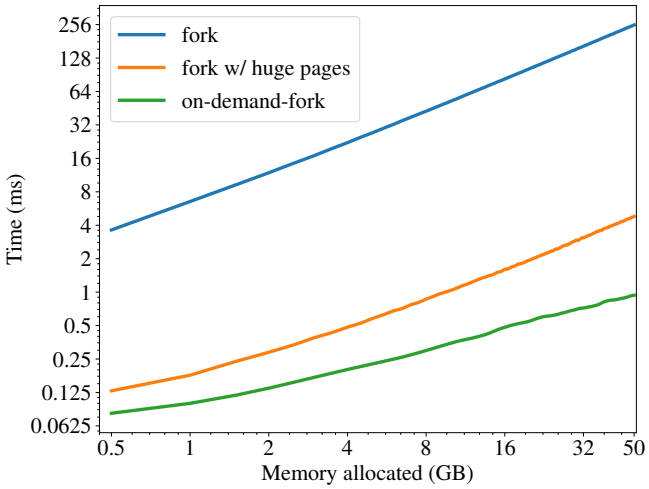

Figure 7. Fork and On-demand-fork invocation latency as the size of allocated memory by the process increases. Experiments were conducted in 512 MB increments and averaged over 5 runs.

Type	Avg. time (ms)
Fork	0.0023
Fork w/ huge pages	0.1984
On-demand-fork	0.0122

Table 1. Worst-case cost to handle a page fault using Fork, with regular and huge pages, and On-demand-fork. The results are the average of 10 runs.

5.2.3 Page Fault Handling This section evaluates the cost of page fault handling caused by the additional operations on the page fault handler, which is expected to be higher when the faulting address is mapped by a shared last-level page table. For this experiment, the benchmark program writes one byte to the middle of a 1 GB memory region using the child process in order to trigger a page fault. Starting at the middle of the memory region ensures On-demand-fork will copy a page table during the page fault, which is the worst case for On-demand-fork. This worst case can only occur once, per process, for every 2 MB memory region mapped by a shared page table (see §3.4). The timestamp measurements are taken just before and after the write.

Table 1 shows the fault handling measurement results. The results show that On-demand-fork takes on average 5.3x longer than Fork with regular pages to handle a page fault in the worst case scenario. This is expected because the page fault handler additionally has to copy a page table in this scenario, which is significantly more work than just performing a copy-on-write of one (data) page. As expected, Fork with huge pages is very slow at handling faults that require a copy-on-write because it needs to copy a 2 MB page, as opposed to a 4 kB page. The results show that the fault handling latency of On-demand-fork is 16× lower than

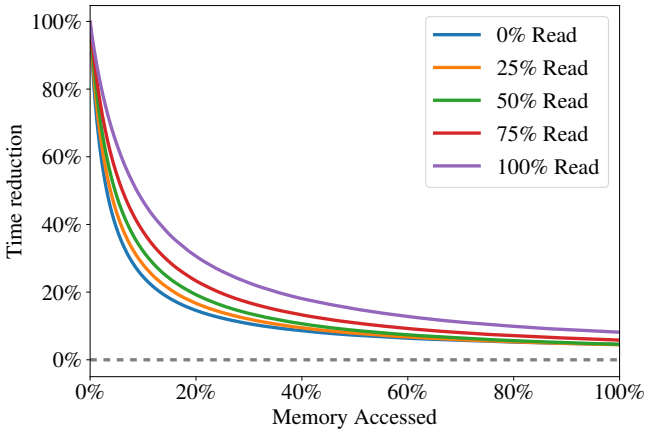

Figure 8. Total cost to create process and access a portion of the allocated memory. The different lines represent different mixtures of read and write accesses. For example, "25% Read" implies 75% write. The values are the average of 10 experiment runs.

Fork with huge pages. This latency difference is particularly important when the application cannot tolerate well high latencies or when the process is not expected to modify a significant fraction of the address space, in which case some of the work copying larger pages would be unnecessary and hence wasteful.

5.2.4 Overall Performance Because with On-demand-fork processes pay the deferred cost of copying page tables when write-accesses occur in the parent or child process, it is important to understand On-demand-fork's overall performance regarding process creation *and* subsequent memory accesses, as compared with Fork.

We measure the percentage difference in the time it takes to fork *and* access a fraction of the memory by the parent process, using different read/write mixes. The benchmark program first allocates a 50 GB memory region. The time is recorded before the parent invokes the fork system call and after memory accesses have completed. The program accesses sequentially the first X percent of memory after the system call, in the following 5 mixes of read versus write accesses: 100%-0%, 75%-25%, 50%-50%, 25%-75%, and 0%-100%. The program performs the memory accesses using `memcpy()` from the target memory region to a 32 MB buffer in the case of reads, or from the buffer to the target memory region in the case of writes.

The result is shown in Figure 8. On-demand-fork reduces the execution time by as much as 99% when no access to the memory region is made after fork. This is because while Fork spends a significant amount of time on the fork invocation (hundreds of milliseconds), On-demand-fork spends far less time on it (hundreds of microseconds). As expected, this advantage shrinks as the fraction of memory accessed, and overall program execution time, increases. This is consistent with the fact that as more memory is accessed, the cost of accessing memory starts to dominate execution time, and the

difference in fork invocation time is increasingly amortized. Among the different access mixtures evaluated, the 100% read mixture consistently has the largest time reduction when using On-demand-fork, while 75% read, 50% read, 25% read, 0% read get progressively diminished reduction benefits, in this order. This is caused by an increasing number of page tables that have to be copied under On-demand-fork when the number of written pages increases.

When all memory is accessed (x-axis at 100%), the 100% read mixture has a time reduction of 8%, and the 0% read mix has a time reduction of 4%. Other mixes have values in-between. Although On-demand-fork eventually has to copy the same number of page tables as Fork for the write-only workload ("0% Read") at 100% accessed, the time reduction of On-demand-fork is still positive because (i) shared last-level page tables are more likely to stay in the complex CPU cache hierarchy, and (ii) fewer page table and page metadata operations during process creation increase the opportunities for user data caching. Hence, an additional benefit of On-demand-fork is that it exploits more effectively caching opportunities provided by modern CPUs [17, 39].

5.3 Real-world Applications

This section discusses several concrete use cases for On-demand-fork and evaluates the cost-benefit of using On-demand-fork as opposed to Fork.

5.3.1 Fuzzing: AFL American Fuzzy Lop (AFL) is a popular and highly-effective coverage-guided fuzzer [8] that has found thousands of security vulnerabilities [32]. AFL uses a "fork server" approach that allows it to efficiently run the same application with different inputs. As a first step, AFL runs an instance of the target program to initialize it by calling `execve()`, dynamically linking it, and initializing libc. After initializing the application once, AFL forks multiple instances of the target program to avoid initializing each instance individually. The performance of fork is crucial to the fuzzing throughput of AFL, especially when considering that the input size is recommended to be kept small and most target applications are not computation-intensive so most executions are very short-lived (and typically test error-paths due to malformed inputs). In fact, AFL documentation explicitly suggests the use of smaller and simpler targets [7], which we presume to be partly due to the high cost of Fork.

We investigated whether the cost of the original fork is hindering the performance of AFL, and whether using On-demand-fork can open up additional fuzzing approaches. To this end, we built on AFL version 2.57b in "LLVM deferred fork server" mode. The deferred fork server mode allows the fork server to start after a customizable point in the code of the target, allowing more costly initialization steps to be finished and shared through fork [30]. Our only modification to AFL is to add an option to AFL's LLVM instrumentation module to switch between fork and On-demand-fork. We take measurements of fuzzing throughput by measuring the target application executions (i.e., tested inputs) per second using the AFL counters.

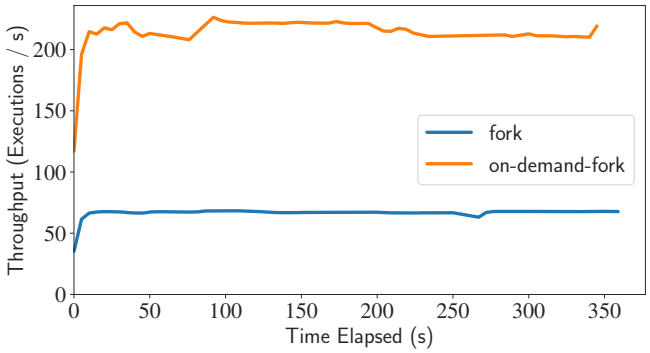

Figure 9. Execution throughput of AFL on SQLite over the duration of a test campaign. Results show the throughput with Fork and On-demand-fork.

Our evaluation used SQLite as the target application. Our testing harness adapted the official fuzzer shell [3] to support loading an initial database with an in-memory size of 1078 MB (1001 MBs on-disk). The database contains integer and string-typed columns and foreign key constraints between columns. A dictionary including the names of the tables and columns in the initial database is passed to AFL. Having a large and complex database loaded allows the fuzzer to reach complex states inside SQLite faster, especially given the complex structures and optimizations of database engines, but would have a prohibitive cost with the original Fork. To the best of our knowledge, this is the first attempt to *systematically fuzz* database engines on large databases. All databases, including the initial one, are in-memory to avoid confounding factors caused by disk I/O.

Figure 9 shows the AFL performance when testing SQLite using Fork and On-demand-fork. The performance under Fork and On-demand-fork is mostly stable. The occasional performance dips are caused by special events (e.g., long execution paths, hangs, and crashes) and are normal during fuzzing. The average throughput while fuzzing SQLite using AFL is 63 executions/s and 206 executions/s with Fork and On-demand-fork respectively. The 2.26x throughput increase over Fork shows that On-demand-fork provides significant efficiency gains when testing applications with large memory footprints.

5.3.2 Unit Testing: SQLite SQLite includes its own test suites, similarly to other systems that are well maintained. To implement fine-grained testing, unit tests generally only test a tiny part of the functionality of the system. Because many systems need to go through an initialization phase (e.g., reading configuration files or loading an initial data set), short-lived test executions are prime targets for On-demand-fork. A fast fork implementation can be used to implement an efficient snapshot-mechanism, which initializes only one

Phase	Avg. time (ms)	Relative
Initialization	24189.36	99.94%
Forking	13.15	0.05%
Testing	0.18	0.01%
Total	24202.69	100%

Table 2. Average and relative time to initialize (i.e., load the initial database) and run a SQLite test case. Tests execute sequentially. Test initialization dominates the total execution time.

Phase	Fork	On-demand-fork
Forking	13.15 (98.6%)	0.12 (36.4%)
Testing	0.18 (1.4%)	0.21 (63.6%)
Total	13.33	0.33

Table 3. The time in milliseconds taken to run SQLite test cases in a child process, using Fork vs. On-demand-fork. The results are the average of 10 runs.

instance and repeatedly calls On-demand-fork to run unit tests on the child process from the post-initialization state. Always starting tests from a clean and identical state leads to better debuggability and reproducibility compared to executing all inputs sequentially.

Our test harness loads the same large initial database as in §5.3.1, which we expect will increase the chances of the unit tests exposing corner case bugs compared with starting from an empty database. The test harness then creates a child process to run each unit test. We use three unit tests that test: (1) SELECT when filtering rows, (2) row deletion that satisfies a condition on record values, and (3) row update that satisfies a condition on record values. The relative sizes of the dataset in comparison with the test operations make initialization take far longer than the operation execution. The test harness measures the time spent on process creation and test operation execution.

Table 2 shows the breakdown of the time spent sequentially running the test cases when running the initialization for each test. In this experiment Fork was called between initialization and the actual test to measure the relative time Fork takes. The results show that the average time spent on initializing the database is far higher (99.94%) than the average time spent on actually running a test (0.01%), and initialization reaches more than $24\,\mathrm{s}$. Conducting unit tests without forking would not amortize the initialization cost and would be too slow to be practical; hence developers often opt for not loading a large, realistic database for testing, which reduces testing coverage.

To assess the benefit of a fork-based unit testing approach, we ran each unit test in a child process and compare the performance when using Fork and On-demand-fork for process creation. Table 3 shows the results of this experiment. When using Fork, it takes on average $13.15\,\mathrm{ms}$ to fork and $0.18\,\mathrm{ms}$ to run the actual tests; when using On-demand-fork, it takes on average $0.12\,\mathrm{ms}$ to fork and $0.21\,\mathrm{ms}$ to run tests. This difference shows that Fork is very inefficient since it takes a high fraction of the total running time (98.6%) and significantly reduces the overall testing performance. The average time to fork under On-demand-fork is 99.1% shorter than that of Fork, enabling the actual tests to take the bulk of the execution time. These results confirm that implementing test suites with On-demand-fork and shared initialization across tests significantly improves performance. Moreover, when multiple instances of the test harness are launched in parallel, a Fork-based approach would suffer from further and significant performance degradation as discussed in §2.1, unlike On-demand-fork.

5.3.3 Snapshot: Redis Redis is a popular in-memory key-value store that is widely used to store critical data [60]. Since Redis is an in-memory key-value store, it is recommended to frequently save snapshots to disk for fault-tolerance. The Redis default setting is to take one snapshot per 60 seconds if at least 10000 keys changed. Redis achieves this by writing a consistent snapshot representation of the entire in-memory data to a file. During the snapshot process, Redis creates a child process so that the child can perform I/O in parallel while the parent continues to handle client requests. However, despite this optimization, *during* the Fork system call invocation, the parent process is unable to serve any requests. This causes latency spikes that may be unacceptable when latency and, in particular, tail-latency are important [67].

We analyzed the latency introduced by the snapshot process when Redis uses Fork and On-demand-fork. We used Redis 6.0.6 and added a configuration option to switch between Fork and On-demand-fork. Redis is configured to take a snapshot when at least 10000 keys have changed; this is to compare Fork and On-demand-fork fairly when an equal amount of data is written between fork calls. To simulate production conditions, we populate Redis with $996\,\mathrm{MB}$ of data before the experiment starts.

We first measure the end-to-end latency experienced by the client. We use the traffic generator `memtier_benchmark` [62], with 3 concurrent connections and a pipeline depth of 2000.

Table 4 shows the comparison of the Redis request-response latency when configured to use Fork and On-demand-fork. As expected, the request latency with On-demand-fork is similar to the latency observed when using Fork for most requests. However, the *tail latency* is significantly lower when using On-demand-fork: 99.9% of the requests take less than $6.335\,\mathrm{ms}$ with Fork and $4.799\,\mathrm{ms}$ with On-demand-fork (24.25% reduction) and the 99.99% percentile request shows 65.95% lower latency with On-demand-fork than Fork ($16.255\,\mathrm{ms}$ vs $5.535\,\mathrm{ms}$). This shows that using On-demand-fork reduces the performance

Percentile	Latency (ms)		Reduction
	Fork	On-demand-fork	
≥50%	4.319	3.871	10.37%
≥90%	5.247	4.159	20.74%
≥95%	5.343	4.255	20.36%
≥99%	5.695	4.575	19.67%
≥99.9%	6.335	4.799	24.25%
≥99.99%	16.255	5.535	65.95%

Table 4. Redis request-response percentile latency when configured to take snapshots with Fork and On-demand-fork. The benchmark ran for 135 seconds, averaging over 1.5 million requests per second. The latency values are the average of 5 repeated runs.

Type	Fork	On-demand-fork	Reduction
Mean (ms)	7.40	0.12	98.38%
Std. Dev. (ms)	0.42	0.007	98.33%

Table 5. The time Redis takes to fork when taking snapshots, using Fork vs. On-demand-fork. The results are the average of 5 issued snapshot commands.

impact of snapshots, lowers the chance that requests hit the unresponsive server while the process is forking, and is beneficial to the overall performance of Redis.

To better understand how On-demand-fork helps Redis take snapshots faster, we examine the time Redis spends on the fork call. The experiment is done under the same setup where 996 MB of data is loaded. The time to fork is measured using the *latest_fork_usec* Redis metric. Table 5 shows the comparison of the time Redis takes to fork when taking snapshots under Fork or On-demand-fork. The results show that On-demand-fork delivers 98.38% reduction in the time to fork and has a much lower standard deviation (0.007 ms vs 0.42 ms).

5.3.4 TriforceAFL: VM Cloning Virtual machines (VMs) usually take up significantly more memory than individual applications and for some use cases they need to be cloned in rapid succession. Therefore, VM cloning systems are good candidates for On-demand-fork. VM cloning is useful in *serverless computing* [14, 25, 68], where lambdas run in virtual machines and a hot start requires cloning VMs, and in operating system kernel testing, which have to run in virtual machines [27, 29, 41]

TriforceAFL [4] is a modified version of AFL that supports fuzzing operating system kernels using QEMU's full system emulation [9]. It relies on a modified version of QEMU to support fork and uses AFL's fork server model.

We extended TriforceAFL to support switching between Fork and On-demand-fork. We measure the fuzzing throughput using the internal statistics provided by TriforceAFL. The fuzzing driver that runs inside the VM is provided by TriforceAFL. It executes as the init process in the guest VM and fuzzes system calls with inputs sent from TriforceAFL. During the fuzzing session, we observed that the QEMU process in this setup typically only takes 188 MB of memory, in part because it runs on a trimmed down VM and the QEMU emulator includes optimizations to allocate memory on-demand.

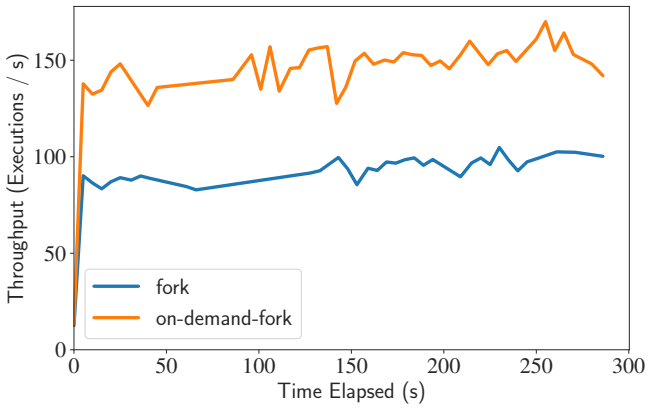

Figure 10. TriforceAFL execution throughput using Fork and On-demand-fork. The dips are due to inputs that cause long system calls.

Figure 10 shows the result of our experiment. The average numbers of executions per second is 91 executions/s and 145 executions/s for Fork and On-demand-fork respectively. Hence, On-demand-fork provides 59.3% higher throughput even with a small sized VM. In other use cases where VMs are much larger – even lambdas can take up to 10 GB of RAM [16] – we expect even more significant improvements when cloning VMs with On-demand-fork.

5.3.5 Apache HTTP Server The Apache HTTP Server is an open-source web server, and has been the most popular web server on the Internet for decades [10]. Apache has many Multi-Processing Modules (MPM), one of which is the `prefork` module [11]. In this operating mode, Apache launches a single control process that reads the configuration, and spawns, using fork, and manages worker processes that listen for HTTP connections. It is considered the best MPM for isolating requests so that a problem with a single request does not affect others [11].

We evaluated Apache using Fork and On-demand-fork, and used `wrk` [28] to measure the end-to-end request latency experienced by clients. After starting Apache, we execute `wrk` to run a benchmark session of 1 second. Because Apache neither maps a large amount of virtual memory (only maps 7 MB of virtual memory before fork), nor creates processes frequently (at most creates 256 worker processes by default), we expect it to get little to no benefit from using On-demand-fork.

	Fork	On-demand-fork	Difference
Mean (μs)	34.3	33.7	-1.75%
Max (μs)	285.2	304.0	+6.59%

Table 6. Response latency of Apache HTTP Server immediately after it is started. The latency values are the average of 5 experiment runs.

Percentile	Latency (μs)		Difference
	Fork	On-demand-fork	
≥50%	35.0	32.4	-7.4%
≥75%	36.5	36.4	-0.3%
≥90%	38.0	39.8	+4.7%
≥99%	51.8	53.6	+3.5%

Table 7. Response latency distribution of Apache HTTP Server immediately after it is started. The latency values are the average of 5 experiment runs.

Table 6 shows that when using On-demand-fork, Apache does not get significant benefits in terms of mean and max latency compared to that under Fork. Differences in mean and max latency between On-demand-fork and Fork are lower than the standard deviation of mean or max for either Fork or On-demand-fork. Similarly, Table 7 shows that there are no meaningful changes in request latency at various percentiles. This demonstrates that not all workloads benefit from On-demand-fork.

6 Related Work

This section discusses related work on improving fork and related mechanisms, applications that leverage fork, and optimizations to the kernel memory subsystem.

6.1 Fork Performance

Fork Alternatives. The Linux kernel has other system calls for creating a process that avoid page table copies but have different semantics that are incompatible with the use cases explored in this work. For instance, `vfork` does not copy page tables but also does not allow processes to run concurrently and has no copy-on-write semantics. `clone` can be called with the `CLONE_VM` flag to prevent page table copying but changes the semantics such that the pages of child and parent become shared (i.e., no copy-on-write). `posix_spawn` provides the combined functionality of `clone` and `exec`, which is incompatible with our use cases evaluated since, internally, `posix_spawn` passes the `CLONE_VM` flag to `clone` to avoid copying page tables. On-demand-fork uses the exact same semantics of the standard fork, which is crucial for our use cases, and drastically improves its performance.

Fork-like OS Primitives. Xu et al. [69] designed a fast fork-like OS snapshot/restore primitive that strips some unnecessary operations done by fork. During the snapshotting phase, it sets up copy-on-write for pages in writable VMAs, records the `brk` value, and saves the status of file descriptors. On recovery, it restores the saved states. The primitive reuses the calling process instead of creating any new process. This work applied the restore approach to fuzzers and showed improved scalability on multi-core machines. However, it is not clear whether it can be safely applied to broader types of workloads because some applications may have kernel states that are not covered by its snapshot/restore operation.

Shared Page Tables. Dave McCracken tried 14 years ago to introduce patches to the Linux kernel to support shared page tables [50]. However, his design only supports page tables of memory areas marked as `MAP_SHARED` (i.e., areas that remain shared across processes even on writes) [49] and only supports file-backed mappings. The design goal was to reduce page table memory consumption when a large number of processes explicitly share memory regions, i.e., without copy-on-write, and speeding up fork is not their main objective. Consequently, his design does not implement the crucial copy-on-write semantics, where each child process gets a conceptually private copy of the pages and page table, and does not support anonymous mappings that all our evaluated use cases require. This approach was not adopted by kernel developers at the time because of its inability to deliver significant performance boost and also a lack of applications that would benefit from it. Fourteen years later, the need for a fast fork has become more pressing; in addition, On-demand-fork has none of these limitations and yields significant performance benefits across a wide-range of realistic scenarios.

The Corey [19] operating system, which increases the kernel scalability on multi-core systems, introduced page table sharing support for *shared* memory regions (i.e., memory regions through which processes can communicate) that are explicitly marked by the caller of `cfork`; such shared regions do not have copy-on-write semantics in Corey. In contrast, On-demand-fork preserves the copy-on-write semantics specified by POSIX while sharing page tables, which is important for all use cases we considered.

Write-protected Page Tables. Similar to EPTI [33] and LVD [54], On-demand-fork leverages the upper-level page table to write-protect lower-level page tables. However, unlike EPTI and LVD, On-demand-fork uses this mechanism to efficiently write-protect userspace pages (not kernel page tables).

6.2 Fork Applications

There is a wide range of applications for the fork system call. This section provides an overview of related work that exploits fork to improve application performance.

Testing and Fuzzing. Testing is computationally intensive and often leads to redundant operations. For example, Pike [26] runs multiple executions of an application to explore different thread interleavings when testing applications for semantic concurrency bugs. Hence, testing frameworks, particularly fuzzers, have numerous optimizations to speed up execution. AFL [7] has a fork server to avoid the cost of loading the target and initializing libc. HonggFuzz [31] and kAFL [64] collect hardware-assisted coverage feedback to increase fuzzing performance, making use of hardware features such as Intel PT [34] and BTS [48]. Other testing tools include algorithmic optimizations [44, 58]; FuZZan [37] includes more efficient metadata structures for sanitizers used in fuzzing and automatically selects the optimal structure. On-demand-fork can be used in conjunction with these optimizations as demonstrated in §5.

Virtual Machine Cloning. SnowFlock [43] implements efficient cloning of VMs in the cloud by designing a new VM fork abstraction. It instantiates a child VM with a *VM Descriptor*, the minimal set of metadata needed to start execution on a remote site; it then provides *Memory-On-Demand*, where accessed memory is fetched from the parent. Because it forks heavyweight virtual machines, it still has startup latency in the order of seconds. SKI [27] uses a a multi-threaded forking mechanism to explore different interleavings and different inputs from a single OS snapshot, amortizing the cost of resuming from a VM snapshot. HyperFork [22] reduces the time to boot a virtual machine by fast cloning a pre-booted VM. It overcomes the high cost of fork from user-space by deregistering the memory regions backing guest physical memory prior to forking and re-registering it in the parent after the clone.

6.3 Kernel Memory Subsystem Efficiency

RadixVM [21] employs a scalable reference counter, Refcache, to lower the cost of reference counting on multi-core systems, and builds a new virtual memory system that supports fully concurrent operations on virtual memory areas. OpLog [20] is a library for maintaining scalable update-heavy data structures. It is shown to increase the performance of various kernel operations, including fork. On-demand-fork addresses the inefficiency of fork for some use cases and is orthogonal to work that makes virtual memory operations more scalable.

7 Conclusion

This paper presents On-demand-fork, a new fork system call design that provides fast process creation by sharing page tables at fork time and copying page tables in small chunks as needed. We implemented On-demand-fork on Linux. Our evaluation shows that On-demand-fork can be applied to a wide range of use cases and can significantly improve the throughput, latency, and efficiency of memory-intensive applications.

Acknowledgment

We are thankful for the insightful feedback provided by the anonymous reviewers and our shepherd Haibo Chen, whose input significantly improved our paper.

References

[1] 2014. *An introduction to compound pages [LWN.net].* https://lwn.net/Articles/619514/
[2] 2015. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial
[3] 2015. *SQLite: fuzzershell.c at [6bf67376].* https://www.sqlite.org/src/file?name=tool/fuzzershell.c&ci=6bf673767b8e5ced
[4] 2016. *Project Triforce: Run AFL on Everything!* https://www.nccgroup.com/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/
[5] 2017. *spawn() is not asynchronous, blocks event loop for 2-3 seconds · Issue #14917 · nodejs/node.* https://github.com/nodejs/node/issues/14917
[6] 2018. *819228 - Consider using posix_spawn() on Linux - chromium.* https://bugs.chromium.org/p/chromium/issues/detail?id=819228
[7] 2019. *AFL - Tips for performance optimization.* https://github.com/google/AFL/blob/master/docs/perf_tips.txt
[8] 2019. *American Fuzzy Lop.* https://github.com/google/AFL
[9] 2020. *QEMU System Emulation User's Guide.* https://www.qemu.org/docs/master/system/index.html
[10] 2021. *The Apache HTTP Server Project.* https://httpd.apache.org/
[11] 2021. *prefork - Apache HTTP Server Version 2.4.* https://httpd.apache.org/docs/2.4/mod/prefork.html
[12] Online. *Huge Pages - The Linux Kernel Archives.* https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[13] Online. libFuzzer: a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html#libfuzzer-a-library-for-coverage-guided-fuzz-testing
[14] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference* (2018). 923–935. https://www.usenix.org/conference/atc18/presentation/akkus
[15] ARM Limited. 2017. *ARMv8-A Address translation.* https://static.docs.arm.com/100940/0100/armv8_a_address%20translation_100940_0100_en.pdf
[16] AWS. 2020. *AWS Lambda quotas.* https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
[17] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: skip, don't walk (the page table). 38, 3 (2010), 48–59. https://doi.org/10.1145/1816038.1815970
[18] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA) *(HotOS '19)*. Association for Computing Machinery, 14–22. https://doi.org/10.1145/3317550.3321435
[19] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (USA, 2008-12-08) *(OSDI'08)*. USENIX Association, 43–57.

[20] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2014. OpLog: a library for scaling update-heavy data structures. (2014). https://dspace.mit.edu/handle/1721.1/89653

[21] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013-04-15) *(EuroSys '13)*. Association for Computing Machinery, 211–224. https://doi.org/10.1145/2465351.2465373

[22] Michael James Colavita. 2020. HyperFork: Improving Serverless Latency and Throughput Through Virtual Machine Flash-Cloning. (2020). https://dash.harvard.edu/handle/1/37364698

[23] Couchbase. 2020. *Disabling Transparent Huge Pages (THP) | Couchbase Docs*. https://docs.couchbase.com/server/current/install/thp-disable.html

[24] Linux Kernel Documents. Online. *Transparent Hugepage Support*. https://www.kernel.org/doc/Documentation/vm/transhuge.txt

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020-03-09) *(ASPLOS '20)*. Association for Computing Machinery, 467–481. https://doi.org/10.1145/3373376.3378512

[26] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the sixth conference on Computer systems* (New York, NY, USA, 2011-04-10) *(EuroSys '11)*. Association for Computing Machinery, 215–228. https://doi.org/10.1145/1966445.1966465

[27] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014-10-06) *(OSDI'14)*. USENIX Association, 415–431.

[28] Will Glozer. 2021. wg/wrk. https://github.com/wg/wrk

[29] Google. 2015. *google/syzkaller*. https://github.com/google/syzkaller

[30] Google. 2019. *Fast LLVM-based instrumentation for afl-fuzz*. https://github.com/google/AFL

[31] Google. 2020. *honggfuzz*. https://github.com/google/honggfuzz

[32] Google. Online. *OSS-Fuzz: Continuous Fuzzing for Open Source Software*.

[33] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. 2018. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs. 255–266. https://www.usenix.org/conference/atc18/presentation/hua

[34] Intel. 2013. *libipt*. https://github.com/intel/libipt

[35] Intel. 2017. *5-Level Paging and 5-Level EPT*. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf

[36] Intel. 2020. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-volume-3a-system-programming-guide-part-1.html

[37] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. 249–263. https://www.usenix.org/conference/atc20/presentation/jeon

[38] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012-10-14) *(SoCC '12)*. Association for Computing Machinery, 1–14. https://doi.org/10.1145/2391229.2391238

[39] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. 2014. Performance analysis of the memory management unit under scale-out workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014-10). 1–12. https://doi.org/10.1109/IISWC.2014.6983034

[40] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering* (2011-04). 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[41] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings 2020 Network and Distributed System Security Symposium* (San Diego, CA, 2020). Internet Society. https://doi.org/10.14722/ndss.2020.24018

[42] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (USA, 2016-11-02) *(OSDI'16)*. USENIX Association, 705–721.

[43] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems* (Nuremberg, Germany, 2009-04-01) *(EuroSys '09)*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/1519065.1519067

[44] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018-09-03) *(ASE 2018)*. Association for Computing Machinery, 475–485. https://doi.org/10.1145/3238147.3238176

[45] Liang Li, Guoren Wang, Gang Wu, and Ye Yuan. 2018. Consistent Snapshot Algorithms for In-Memory Database Systems: Experiments and Analysis. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018-04). 1284–1287. https://doi.org/10.1109/ICDE.2018.00131

[46] Xinyu Li, Lei Liu, Shengjie Yang, Lu Peng, and Jiefan Qiu. 2019. Thinking about A New Mechanism for Huge Page Management. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hangzhou, China) *(APSys '19)*. Association for Computing Machinery, New York, NY, USA, 40–46. https://doi.org/10.1145/3343737.3343745

[47] ARM Limited. 2021. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest/

[48] Linus Torvalds. 2015. *Intel Branch Trace Store*. https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-bts.txt

[49] Linux manual page. Online. *mmap(2) - Linux manual page*. https://man7.org/linux/man-pages/man2/mmap.2.html

[50] Dave McCracken. 2003. *Shared Page Tables Redux*. https://www.kernel.org/doc/ols/2006/ols2006v2-pages-125-130.pdf

[51] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019-03-25) *(EuroSys '19)*. Association for Computing Machinery, 1–15. https://doi.org/10.1145/3302424.3303973

[52] MongoDB. 2020. *Disable Transparent Huge Pages (THP) — MongoDB Manual*. https://docs.mongodb.com/manual/tutorial/transparent-huge-pages

[53] S. Nagy and M. Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[54] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS*

International Conference on Virtual Execution Environments (New York, NY, USA, 2020-03-17) *(VEE '20)*. Association for Computing Machinery, 157–171. https://doi.org/10.1145/3381052.3381328

[55] Linus Nyman and Mikael Laakso. 2016. Notes on the History of Fork and Join. 38, 3 (2016), 84–87. https://doi.org/10.1109/MAHC.2016.34

[56] Oracle. Online. *Managing Memory*. https://docs.oracle.com/database/121/ADMIN/memory.htm

[57] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018-03-19) *(ASPLOS '18)*. Association for Computing Machinery, 679–692. https://doi.org/10.1145/3173162.3173203

[58] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[59] RedHat. 2020. *5.2. Huge Pages and Transparent Huge Pages Red Hat Enterprise Linux 6*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge

[60] RedisLab. Online. Redis. https://redis.io/

[61] RedisLab. Online. Redis latency monitoring framework – Redis. https://redis.io/topics/latency-monitor

[62] RedisLabs. 2013. *RedisLabs/memtier_benchmark*. https://github.com/RedisLabs/memtier_benchmark

[63] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX time-sharing system. 17, 7 (1974), 365–375. https://doi.org/10.1145/361011.361061

[64] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)* (2017). 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[65] Jonathan M Smith and Gerald Q Maguire. 1988. Effects of copy-on-write memory management on the response time of UNIX fork operations. (1988), 10.

[66] The kernel development community. 2021. *HugeTLBpage on ARM64 — The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/arm64/hugetlbpage.html

[67] Trivago technology. 2017. *Learn Redis the hard way (in production)*. https://tech.trivago.com/2017/01/25/learn-redis-the-hard-way-in-production/

[68] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany, 2019). ACM Press, 1–16. https://doi.org/10.1145/3302424.3303978

[69] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA, 2017-10-30) *(CCS '17)*. Association for Computing Machinery, 2313–2328. https://doi.org/10.1145/3133956.3134046