Reducing Minor Page Fault Overheads through Enhanced Page Walker

CHANDRAHAS TIRUMALASETTY, Department of Electrical & Computer Engineering, TAMU, USA
CHIH CHIEH CHOU, Amazon Lab 126, USA
NARASIMHA REDDY, Department of Electrical & Computer Engineering, TAMU, USA
PAUL GRATZ, Department of Electrical & Computer Engineering, TAMU, USA and Department of Computer Science & Engineering, TAMU, USA
AYMAN ABOUELWafa, Hewlett Packard Enterprise, USA

Application virtual memory footprints are growing rapidly in all systems from servers down to smartphones. To address this growing demand, system integrators are incorporating ever larger amounts of main memory, warranting rethinking of memory management. In current systems, applications produce page fault exceptions whenever they access virtual memory regions that are not backed by a physical page. As application memory footprints grow, they induce more and more minor page faults. Handling of each minor page fault can take a few thousands of CPU cycles and blocks the application till the OS kernel finds a free physical frame. These page faults can be detrimental to the performance when their frequency of occurrence is high and spread across application runtime. Specifically, lazy allocation-induced minor page faults are increasingly impacting application performance. Our evaluation of several workloads indicates an overhead due to minor page faults as high as 29% of execution time.

In this article, we propose to mitigate this problem through a hardware, software co-design approach. Specifically, we first propose to parallelize portions of the kernel page allocation to run ahead of fault time in a separate thread. Then we propose the Minor Fault Offload Engine (MFOE), a per-core hardware accelerator for minor fault handling. MFOE is equipped with a pre-allocated page frame table that it uses to service a page fault. On a page fault, MFOE quickly picks a pre-allocated page frame from this table, makes an entry for it in the TLB, and updates the page table entry to satisfy the page fault. The pre-allocation frame tables are periodically refreshed by a background kernel thread, which also updates the data structures in the kernel to account for the handled page faults. We evaluate this system in the gem5 architectural simulator with a modified Linux kernel running on top of simulated hardware containing the MFOE accelerator. Our results show that MFOE improves the average critical path fault handling latency by 33× and tail critical path latency by 51×. Among the evaluated applications, we observed an improvement of runtime by an average of 6.6%.

CCS Concepts: • Software and its engineering → Virtual memory; Main memory; • Hardware → Hardware-software codesign;

This work was funded in part by an NSF IUCRC grant I/UCRC-1439722 and a grant from Hewlett Packard Enterprise.
Authors’ addresses: C. Tirumalasetty and N. Reddy, Department of Electrical & Computer Engineering, TAMU, 188 Bizzell St, College Station, 77843, TX; emails: [chandrhas996, reddy]@tamu.edu; C. C. Chou, Amazon Lab 126, 1100 Enterprise Way, Sunnyvale, 94089, CA; email: ccchou2003@tamu.edu; P. Gratz, Department of Electrical & Computer Engineering, TAMU, 188 Bizzell St, USA and Department of Computer Science & Engineering, TAMU, USA; email: pgratz@gratz1.com; A. Abouelwafa, Hewlett Packard Enterprise, 6280 America Center Dr, San Jose, 95002, CA; email: ayman.abouelwafa@hpe.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1544-3566/2022/09-ART57 $15.00
https://doi.org/10.1145/3547142

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 4, Article 57. Publication date: September 2022.
1 INTRODUCTION

Virtual memory is a useful and critical mechanism used in many modern operating systems that helps in managing memory resources while giving applications the illusion of owning the entire (or sometimes more than the) available physical memory. To do so, virtual memory basically relies on two fundamental techniques: (1) Lazy allocation and (2) Swapping.

Lazy allocation is a policy of physical memory allocation wherein memory pages are actually allocated only when accessed. By doing so, the OS memory subsystem does not have to allocate applications any memory at \texttt{malloc} time; instead, memory is allocated when memory is indeed needed, upon its first use. Lazy allocation can work because it is very rare for an application to touch/access all pages it requests immediately after the application requests it. Typically, program working sets are much smaller than their whole memory footprints. Based on this characteristic of programs, the memory subsystem can let multiple applications concurrently execute and therefore improve the overall system performance without creating a scarcity of memory. Even as lazy allocation reduces the actual allocation of memory to applications, it is still possible that all memory becomes allocated when many applications are executing concurrently in the system. If the memory runs out, the swapping mechanism is adopted to store the content of some memory pages in the non-volatile storage devices, and those memory pages, after their contents are wiped out, are reused and re-allocated to other applications.

These two mechanisms usually work as part of the kernel page fault exception handling. When the memory is “allocated” (via \texttt{mmap} or \texttt{malloc}), actually only a region of the virtual address space is created by the kernel for the calling program; physical pages are not allocated to pair with those virtual pages until they are touched. A memory access (touch) within this newly created region triggers a page fault exception by the hardware page walker, and then the kernel (software) exception handler will check and confirm this access as legal and in turn allocate a physical page for it [5].

The page fault exception handler must carry out a number of operations to handle a page fault. Some of these operations include checking the validity of the faulting address, acquiring a page from the available free memory page pool, filling the page with the corresponding data content from the storage device (major page fault) or zeroing the page (minor page fault), and creating some data structures for memory management. These operations result in considerable overhead for handling page faults. As we will show later, page fault handling can result in overheads of thousands of processor cycles.

This overhead is not a big concern when pages are accessed from slow devices. With faster SSDs and Non-Volatile Memory (NVM) storage devices, this overhead has recently received attention [23]. Our work here focuses on the impact of this overhead on minor page faults. We show that even when sufficient memory is available, overheads from handling minor page faults can contribute to considerable performance loss. Table 1 shows that the overhead from handling minor page faults can range up to 29% of the application runtime. The overhead calculated is just the measure of time spent in the kernel to handle minor page faults. But actual overhead due to minor faults also includes page table walks and context switching, and is expected to be slightly higher than the fraction shown.
Table 1. Benchmarks with Anonymous Page Fault Overhead

| Benchmark            | Avg. Fault Rate (K.faults/sec) | Fault Overhead (%) |
|----------------------|-------------------------------|--------------------|
| GCC compiler         | 365.31                        | 29.09              |
| FaaS                 | 108.92                        | 9.05               |
| Parsec-dedup         | 165.88                        | 8.26               |
| YCSB-memcached       | 120.06                        | 6.47               |
| Splash2X-radix       | 259.25                        | 16.17              |
| Splash2x-fft         | 222.62                        | 10.06              |
| XSBench              | 89.09                         | 4.90               |
| GAP-RC               | 77.74                         | 3.08               |
| Integer Sort         | 104.91                        | 4.20               |

We highlight the impact of minor page faults on one of the popular applications, Function-as-a-Service (FaaS). The latency due to minor page fault handling is a non-trivial contributor to invocation latency in FaaS systems. More specifically, we observed that minor fault frequency increases with number of cold starts. We use the FaaS-profiler framework of [31] to measure the program invocation rate and the minor fault rate. For this experiment, we have chosen the OCR-img application, which needs a container runtime. As shown in Figure 1, for the given application, the minor fault rate reaches the peak at around 110 Kfaults/sec for 20 invocations/sec. As we increase the invocation rate beyond 20 invocations/sec, the minor fault rate drops off. This happens because we have exceeded the system capacity for invocations at 20 invocations/sec for this particular application, and each invocation’s latency at this stage is more dependent on the wait time of the invocation inside the queue. FaaS providers charge their users based on the latency of invocations. The system spending more time in the kernel handling minor faults results in users paying the bill for non-application runtime.

More discussion on benchmarks and the measurement methodology is provided in Section 5.1.

Here we propose a new approach to memory management, with the goal of minimizing overheads for minor page faults in such applications. Our proposed approach rethinks how the memory management function can be organized and realigns the work to be done within the OS and hardware. We move the majority of the work in handling a minor page fault to the background through pre-allocation of pages, instead of the current model of on-demand inline allocation of pages to faulting processes. Our approach carries out minor page fault handling within the context of the faulting process through hardware, thus obviating the context switch of traditional approaches in current OSs. This approach moves most of the OS work from the critical path of the application’s execution to a separate, ahead-of-time thread, with the remainder being accelerated in hardware upon minor fault demand. This requires rearranging the memory management functions in the kernel space of the OS and redesigning the hardware page walker to carry out the remaining minimal processing in hardware at the time of a page fault, within the context of the faulting process.

The contributions of this article are as follows:

- Proposes a new page fault handling architecture, composed of a coordinated kernel thread and hardware page walker, to reduce the inline page fault overheads
- Proposes a redesign of memory management functions in the OS and hardware to enable the realization of the proposed architecture
• Implements the proposed changes to processor architecture on the gem5 simulator and modifies the existing Linux kernel to accomplish our new minor page fault handling mechanism
• Evaluates the proposed minor page fault handling mechanism using microbenchmarks to show significant critical path latency improvements
• Shows that the proposed mechanisms can improve full application running times by 1% to 25.4%

The remainder of this article is organized as follows. Section 2 describes the background and related work. Section 3 presents the motivations and key design concepts of software and hardware modifications of our optimized page fault handling. Section 4 explains the implementation of our new page fault handling approach by modifying the Linux kernel and x86-64 CPUs on the gem5 simulator in more detail. Section 5 presents our results of evaluation with benchmarks. Finally, Section 6 provides a conclusion.

2 BACKGROUND

2.1 Page Fault

User programs request memory using malloc system call or mmap. However, both of these calls do not allocate any memory for applications when called. Instead, they only create and validate a region of virtual address space for the calling process and return the start address of this region. Later, when applications access some address within this newly created region (if no page has been allocated for this address yet), an exception (page fault) would be triggered by hardware. The execution context is changed from userspace to the kernel, and the page fault handler of the kernel starts running on behalf of the faulting application. The handler allocates a page for the faulting address, updates the page-table entries, and updates the kernel book-keeping data structures. This kind of fault is called a minor page fault.

After the page fault exception handler completes its job, the faulting load/store instruction is re-executed. This mode changing (from user mode to kernel mode and then back to user mode) requires some “state,” such as local variables, hardware registers, program counter, etc.) of the user program to be stored (or pushed) on the stack. The last few operations of the exception handler are to pop that state from the stack back to the original place, so that the faulting instruction can continue to execute. In addition to the overhead of pushing onto and popping from the stack, context switch results in pollution of architectural resources like caches, TLB, branch predictors, and so forth.

In systems with NVM devices, memory can be accessed by applications through the file system read/write interface. This method does not incur page faults, but it needs to rely on system calls, which can also result in context switching. A large body of recent work has observed that systems software overhead incurred by file system APIs is high relative to NVM access times and tries to reduce this overhead through various methods [12, 21, 25, 32, 36].

2.2 Hardware Page Walker

Hardware page walker [8] is popularly employed in modern CPUs. On a TLB miss in x86_64 architecture, the hardware page walker assumes the job of reading the faulting address from the CR3 register and “walking” the multiple levels of the page table to find the appropriate page table entry (PTE). If the page walker can reach the lowest PTE of the faulting address and finds the present bit of PTE is set, meaning that a page has been allocated for this virtual address and its physical frame number is also stored at the PTE, then the page walker would simply update the corresponding TLB entry and re-execute the faulting instruction again. Otherwise, the page walker will trigger a page fault exception and let the kernel handle the page fault.
Enhancements to page walkers have been proposed [4, 16] previously to achieve different goals. Our work here can be viewed as an enhancement to the hardware page walker to carry out the entirety of page fault handling’s critical path.

2.3 Related Work

2.3.1 Hardware Paging. Recent work [23] advocated hardware-based demand paging for major page faults. This work has similar motivation as our work, i.e., reducing the cost of page faults, but focuses on major page faults. Our work here shows the importance of handling minor page faults even when the datasets can be entirely memory resident. Nevertheless, our approach could be extended to support major faults. In contrast to a single pre-allocation queue proposed in earlier work, we have proposed pre-allocation tables for each core, thus providing flexibility to implement NUMA policies on pre-allocated pages. Major page fault handling latency was observed to be an overhead when paging was used as a mechanism to fetch pages from a disaggregated memory node [29]. Their approach is similar to ours (i.e., they have offloaded fault handling to HW accompanied by kernel changes) but targets different systems.

Adding page fault support to network controllers to avoid page pinning for DMA requests is proposed in [24]. Our page fault handling mechanism is orthogonal to this proposal and can be integrated with this work, to enhance the fault handling latency. The hardware-based page fault handler is used for virtual memory implementation at disaggregated memory nodes in [18]. This proposal focuses on implementing lightweight virtual memory for remote memory accesses with offloading of fast path to ASIC and FPGA and slow path to an ARM processor, in a co-designed memory node. Albeit there are design similarities, this proposal is unrelated to our approach in this article. Our approach focuses on offloading generic anonymous fault handling to an enhanced page table walker.

2.3.2 TLB and Paging Optimizations. Large pages have been proposed to reduce the costs of the number of page faults to be handled [17, 22, 26, 27] and sometimes to increase the reach of the TLBs [30]. Large pages are beneficial when addresses are allocated over contiguous ranges, but many new applications allocate small objects [13] and have required additional support for efficiencies [28]. From a memory management perspective, the kernel has the job of promoting and demoting huge pages based on their usage. Also, the kernel has to perform memory compaction of base (4K) pages to ensure that it has free 2MB pages for future allocation requests. Our approach here works across different workloads and moves much of the memory management overhead away from inline processes. Mitosis [3] replicates page table entries in multi-node systems to avoid costly remote accesses during page table walks by the HW page table walker. Our approach makes use of the hardware page table walker but doesn’t put any restrictions on page table entry placement. Hence, the Minor Fault Offload Engine (MFOE) can be made to work with Mitosis.

Recent work [19] uses intermediate address space-Midgard to manage the virtual memory of applications. Midgard uses virtual memory areas (VMAs) instead of physical pages for memory management. Since an application has a small number of VMAs to track, it achieves low address translation overhead. Although Midgard increases the efficiency of TLB, it doesn’t remove the cost of page faults altogether. So, the overhead due to page faults exists even in systems that use Midgard.

2.3.3 Miscellaneous. Tracking PTE access bits to avoid unnecessary TLB shoot-downs has been proposed in [6]. Our enhanced hardware page table walker sets the PTE access bit during both page walk and page fault handling in hardware. Hence, no disruption to the page access tracking mechanism is caused by our enhanced page table walker. Prefetching has also been proposed to
reduce the cost of misses on page accesses [9, 15]. Prefetching is complementary and can work with our approach. Alam et al. [5] adopt a hardware helper thread to reduce the number of context switches incurred by page faults. Their work, however, requires a pair of registers to indicate a single region of virtual address space (one for the start address and the other for the end address). Their approach would be better for virtual machine workloads because a guest OS will allocate a huge amount of contiguous virtual memory region from the hypervisor as its physical memory but might not be suitable for general workloads.

3 DESIGN OVERVIEW

In this section, we discuss the design choices that form the basis for our enhanced page fault handling, through MFOE. While the discussion is in the context of Intel X86_4 and Linux OS, our modified fault handler could be implemented in any other architecture and any other operating system.

We have broken down the page fault handling in software into three parts: tasks that can be done before the page fault, tasks to be done for handling the page fault (critical path), and tasks that can be done after the page fault. Design of MFOE is primarily motivated by two main observations pertaining to these sub-tasks: (1) tasks that can be done before and after the page fault occurrence can be removed from the critical path to reduce the fault handling latency, and (2) the critical path tasks are hardware amenable and hence would benefit if the hardware (MFOE) executes the critical path instead of software.

Figure 2 contrasts the page fault handling flow of MFOE with typical page fault handling in software. We categorize physical page allocation and constructing the page table as pre-fault tasks. Post-fault tasks include updating the kernel metadata structures and establishing the reverse mappings for the handled page fault. For the critical path, MFOE has to check the validity of the faulting address, update the page table entry by consuming the physical page made available by the pre-fault task, and lastly make information available for the post-fault maintenance routine. Pre-fault and post-fault functions are combined into a background thread, so their latencies are removed from the fault handling function in the context of the faulting program.

This section is organized as follows: we discuss the page pre-allocation and legal virtual address space indication in Sections 3.1 and 3.2, respectively. Both of these are pre-fault tasks that are moved into the background. In Section 3.3 we discuss the design of MFOE and how it handles page fault by working in conjunction with the background thread. In Section 3.4 we discuss post-fault handling tasks.
3.1 Page Pre-allocation

The existing page fault handler obtains a physical page from the buddy system\(^1\); this action may result in blocking the faulting program if no page is available and the kernel must try hard to reclaim pages either from the page cache or by writing some dirty pages to swap space as a last resort. The latter action would require issuing I/O requests to the backing storage. So it’s reasonable not to attempt to offload such operations to hardware. Nevertheless, physical page allocation can be done without the knowledge of the faulting virtual address, and hence can run beforehand in the background, i.e., pre-allocation. We consider a mechanism to provide pre-allocated pages; these pages are used by hardware when page fault handling is completely done by hardware or by critical path software when MFOE is employed only as a software solution.

When a page is allocated, usually a pointer of struct page (or the virtual address in the kernel of that page) is returned by the buddy system in the kernel. This pointer (of struct page) can be translated into the physical page frame number of that page easily by page_to_pfn macro. If those frame numbers of pre-allocated pages can be saved beforehand in a certain format and the hardware can access them easily when a page fault occurs, we will be able to use a free page frame to service the fault as well as remove page allocation out of the critical path.

This design choice has the following benefits:

- Since allocating pages might block the faulting programs, page pre-allocation by a kernel background thread only blocks this kernel thread and will not hurt (slow down) the user application.
- In the existing Linux kernel, to allocate pages for malloc or anonymous private mmap, the kernel always “zeros” the entire page (4KB) or entire huge page (2MB) in the page fault critical path due to security concerns. Our pre-allocation mechanism also moves this time-consuming operation out of the page fault critical path.
- It reduces the inline work to be carried out at the time of a page fault and keeps the required hardware enhancements simple.

3.2 Legal Virtual Address Space Indication

The first and most important operation of the existing page fault handling routine is to determine if the faulting virtual address is legal. This check operation is implausible to be implemented in hardware. Also, the faulting virtual address can only be known at the moment of page fault occurrence, and there is no way the hardware can know if the faulting address is legal in advance. So, instead of trying to validate the faulting address in hardware, we let the software tell the hardware if the faulting address is a legal address or not.

To this end, we narrow down the scope of our page fault offloading mechanism. We limit ourselves to handling page faults happening in the user space; in particular, our new page fault mechanism currently only deals with minor page faults caused by accessing the virtual address space regions created by malloc and anonymous private mmap functions within user programs. The kernel keeps track of all the valid virtual memory areas of an application. The only missing piece is a mechanism to inform the hardware of this information. The correct avenue to enable this communication would be at the hardware page walker. The HW page walker will be the first to witness the faulting address in the HW. So, if the HW page walker is aware of the validity of the faulting address, it could revert to software handling of the fault quickly. To establish this understanding,

\(^1\)The buddy system is an organizational data structure used by the Linux kernel to manage available physical memory frames/pages. The available memory is broken into groups of blocks, each of size power of two of base page size (4K). Based on the incoming request for free pages, the blocks are broken down into two blocks half the original size, with each broken-down block being “buddies” to each other [5].

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 4, Article 57. Publication date: September 2022.
we put an MFOEable (Minor Fault Offload Enable) bit in the page table walking path to tell the hardware whether the currently accessed address is legal. The lowest Page Table Entry (PTE) seems to be a reasonable candidate for accommodating this indicator. Hence, we re-purpose a bit in un-used (invalid) PTE to be used as a valid bit.

Therefore, our proposed flow is as follows. When applications call malloc or mmap, just before the system call is about to return, a short-lived background thread is spawned by the kernel. This background thread walks the page table path for all pages within this newly created virtual address region in the background, until all the lowest PTEs belonging to this newly created area have been reached. If the paths to PTE of some addresses have not been constructed yet, the kernel background thread will construct these paths as the exception handler does. This path construction process is also moved out of the page fault critical path. Meanwhile, this kernel background thread sets an MFOEable bit as a legal address indication for PTEs of all pages of this virtual address region. If the page table walker does not find this bit in PTE, then it does not offload the minor fault handling to MFOE; it raises an exception and defaults to fault handling in the kernel.

3.3 MFOE-Minor Fault Offload Engine

Our central idea is to move most of the minor page fault operations out of the page fault critical path and to execute them before and after a page fault by software; only the mandatory operations are tackled by hardware during a page fault. Thus, we propose to keep MFOE as simple as possible by implementing it as an addition to the hardware page walker, the TLB sub-system.

Realizing the design considerations discussed in Sections 3.1 and 3.2, MFOE is equipped with a pre-allocation page table whose entries contain page frame numbers of pre-allocated pages and MFOEable bits set at the leaf entries of the page table across its legal address space.

These two components make the design of MFOE straightforward: When a faulting address is accessed, there will always be a TLB miss because TLB only caches legal address translations. The hardware page walker starts to walk the multi-level page table until it reaches the lowest PTE. If the page table walker finds out that the MFOEable bit is set (meaning the faulting address is legal) but the present bit is not set yet (meaning that this address does not have an allocated page), then the page table walker offloads the fault handling to MFOE. MFOE picks an entry from the pre-allocation page table, updates the PTE with the page frame number of this entry and sets some flag bits into the PTE (as the software exception handler does), creates a TLB entry, and continues to execute the faulting instruction without the need for kernel software. For all the other cases, different from the above-mentioned scenario, the typical kernel page fault exception is triggered by the hardware page table walker to handle the fault.

3.4 Post-Page Fault Handling

Besides the page allocation, the address validation, and updating PTE entries, there are still some operations left to be executed by the page fault handling routine, such as increasing the counter of the mm object (by calling inc_mm_counter function), establishing the reverse map for physical page (through the page_add_new_anon_rmap function), putting the page into the LRU list for swapping later (through the lru_cache_add_active_or_unevictable function), and accounting for cgroup-based memory usage metering (by calling the mem_cgroup_commit_charge function). These operations can be delayed and executed later in the background if the program is not terminated. Additionally, due to pre-allocation of physical pages for future faults, in the event of program termination, there is a necessity to clean up the unused physical pages. This clean-up might not be needed in a real-world implementation if MFOE was used by default for page fault handling for all the programs.

Figure 3 details the design of MFOE and its key components.
4 IMPLEMENTATION

In this section, we explain the implementation details of the MFOE prototype that we built. The implementation consists of both hardware (MFOE) and software (modified Linux kernel) components. In Section 4.1 we discuss the system calls we implemented to enable/disable MFOE. In Section 4.2 we present the organization of the pre-allocation table. The details about pre-fault and post-fault components of the background thread are presented in Sections 4.3 and 4.5, respectively. We present the details of page fault handling by hardware in Section 4.4. In Section 4.6 we talk about how our implementation accounts for errors, and Section 4.7 has a discussion on disabling MFOE. In Section 4.8, we discuss software emulation of MFOE. Finally, we analyze limitations and issues involved with the current implementation in Section 4.9.

4.1 MFOE Enable/Disable System Calls

To enable optimized page fault handling through MFOE, applications should use the MFOE_enable system call at the beginning of their source code. MFOE is automatically disabled by the MFOE_disable() function called inside __mmput() by kernel when applications are about to be terminated. We have also implemented the MFOE_disable system call, which would directly call the MFOE_disable() function; this allows applications to disable MFOE in the middle of their runtime if they decide to do so. If MFOE is disabled in the middle of the program runtime, the PTEs that have been populated with the MFOEable bit set will stay intact, but the faults on such pages will be defaulted to kernel page fault handling.

MFOE-related software can be implemented as a kernel module, and operations of MFOE_enable and MFOE_disable system calls can be executed when the MFOE module is loaded and unloaded, respectively. By doing so, all user applications will automatically enable MFOE by default. However, we use a system call to simplify our evaluation of MFOE.

---

This is the only application-level change that is required to enable MFOE.
4.1.1 MFOE_enable System Call. MFOE_enable system call expects an input "preallocation_size," which indicates the number of physical frames that are to be pre-allocated per core. With "preallocation_size," the kernel calculates the physical frames needed to house the per-core pre-allocation table itself. The pre-allocation table contains the frame numbers of the pre-allocated pages that can be consumed by the faults occurring on the particular core. Since our implementation chooses to have a pre-allocation table per core, we have to construct a separate table for each core. We should mention that the construction of the pre-allocation table is not needed for every MFOE_enable system call by the application, but only for the very first MFOE_enable system call since the system boot. In our design, the size of each entry in the pre-allocation table is 16 bytes. Suppose for "preallocation_size" 256, we need to allocate a 4K page per core for the pre-allocation table. This can be viewed as memory overhead for using MFOE.

After the pre-allocation tables are constructed, the start address of the pre-allocation table is conveyed to the hardware. The schedule_on_each_cpu() kernel function executes a small code on each core in the system. This code writes the page frame number (34 bits) of the first page of the corresponding pre-allocation table, the number of entries of the pre-allocation table (16 bits), and MFOE_ENABLE bit (1 bit) to a newly added 64-bit register. We used the architecturally hidden CR9 control register of X86 for this purpose.

We have made an implementation choice to construct only the pages that constitute the pre-allocation table, but not the the pre-allocated physical frames that form the entries of this table. We have chosen to do so to minimize the runtime of the system call as well as the blocking time of the user-space process. A background kernel thread that is spawned at the end of the system call will be responsible for populating the pre-allocation tables of all the cores. Hence, the time taken to complete the pre-allocation for all the cores increases with pre-allocation width and number of cores in the system. This implementation choice is justified because we minimize the time in kernel context due to system call and avoid non-deterministic blocking time taken due to pre-allocation.

It is possible that MFOE hardware looks up the pre-allocation table to satisfy a page fault even before this aforementioned background kernel thread completes its task of filling each core’s pre-allocation table. In this case, MFOE hardware fails to get a valid page frame number from the table and then will subsequently trigger a normal page fault to be handled in the kernel.

To indicate to the kernel whether a process has enabled MFOE, we have added a Boolean type member called “mfoe” to the memory descriptor mm_struct. When the program calls the MFOE_enable system call, this value is set to true. The memory allocation routine (mmap or malloc) checks "mfoe_enable" to see if the page table entries have to be allocated differently, i.e., with MFOEable bits.

4.2 Pre-Allocation Table

Our pre-allocation table adopts a lockless ring buffer architecture [2] with one producer (the kernel background thread, which produces/pre-allocates pages) and one consumer (the MFOE hardware, which consumes pages). Each entry of the pre-allocation table has 16 bytes, and the first entry (entry number 0) stands for the table header, which contains the head index (ranging from one to the number of entries), tail index (also ranging from one to the number of entries), number of table entries, and lock bits. The size of each field in the header is 4 bytes. Except for the table header, the entry’s format is delineated in Figure 4. As shown below, the fields of each pre-allocation entry are faulting virtual address, thread group id (TGID), page frame number, used bit, and valid bit.

The producer only looks at the head index, and the consumer only checks the tail index. When the kernel (producer) pre-allocates a page for the pre-allocation table, first it looks up the corresponding entry of the head index from the table header. If this entry is not valid (the valid bit is not set), meaning that this entry does not contain a valid pre-allocated page, then the kernel
allocates a page and puts its page frame number into the corresponding field of the pre-allocation entry, increments the head index by one, and checks the next entry. The kernel would continue this page pre-allocation process until it reaches an entry whose valid bit has already been set or till it traverses the entire pre-allocation table for a particular core.

Since our implementation uses a per-core pre-allocation table, the kernel, when choosing pages for pre-allocation, can follow an allocation policy. For example, if a system has multiple non-uniform memory access (NUMA) nodes, then pages can be pre-allocated from the near memory node first. After the pages from the near memory node are exhausted, we can decide either to pre-allocate pages from the far memory node or not to pre-allocate pages at all. Effectively, with this policy, MFOE will trigger a typical page fault exception after it runs out of near memory node pages. In the current implementation we enforce a static policy where we prefer local node (to the given core) physical pages for all the pre-allocations. The kernel API used for pre-allocation (buddy allocator) automatically falls back to remote node pages if the local node pages are unavailable. Hence, after we run out of local node pages, we will use remote node physical pages for pre-allocation.

4.3 Pre-Page Fault Software Handling

Every time a user-space application calls malloc or anonymous private mmap to allocate memory, the kernel will check if the mfoe member of the memory descriptor object (mm_struct) is set as true. If that’s the case, then the kernel will flag the virtual memory area as belonging to an MFOE-enabled process. To achieve this we add a new flag VM_MFOE to struct vm_area_struct. This flag will be set for the virtual memory areas belonging to the process, which are created after the process has enabled MFOE through the MFOE_enable system call.

At the end of the malloc or anonymous private mmap function, we account for the kernel’s responsibility to craft the leaf entry of the page table with MFOEable bit. Since we know the start and end address of the newly created virtual memory area, we walk through the page table of the calling process until it reaches all the corresponding lowest level page table entries and set the MFOEable bit (bit-position 2) to 1. If the walking path of the multi-level page table (non-leaf

\[\text{struct vm_area_struct is a data structure used by Linux kernel to manage contiguous virtual memory areas. The memory descriptor-mm_struct of each process maintains a list of vm_area_struct to represent all of a process's virtual memory.}\]

\[\text{Fig. 4. The pre-allocation table and its fields after pages are pre-allocated. Header not shown.}\]
entries) has not been constructed yet, the kernel will construct it the same way as the page fault handler does. This is the realization of the design motivation provided in Section 3.2.

In addition to setting the MFOEable bit, the kernel also fills the RW bit (bit position 1, if the region is writable), PFN field of PTE with the TGID of the current process. Here we are re-purposing bits of an otherwise empty/invalid PTE to convey the MFOE hardware about the legality of the address, the RW permissions, and the TGID of the process that enabled MFOE. This use of PTE is safe and justified because these bits are not used for swap computation. In an MFOE-disabled process, these bits would have been all zeros; hence, their use is safe. Figure 5 shows the status of the lowest-level page table entry after the execution of pre-page fault operations.

The page table construction doesn’t add overhead to the program performance, as it has to be done sometime during its runtime. If not done now, the table will be constructed as part of the fault handling routine. We should also highlight that this page table entry creation could be done by the kernel background thread spawned through the kthread_run() function. In this implementation we have made a choice of using a background thread to minimize blocking time in user-space. Plus, using a background thread could be advantageous for large programs with high virtual memory footprint. With the background thread it is possible that MFOE reaches a PTE of a page that is MFOEable but its MFOEable bit has not been set by the asynchronous kernel thread. In this case, the MFOE will simply treat this page as “non-MFOEable” and trigger a typical page fault.

### 4.4 Hardware Page Fault Handling

The MFOE is only activated after a TLB miss and after the hardware page table walker reaches the last level of PTE. If the hardware page table walker recognizes that the present bit is absent in PTE, it activates the MFOE to handle the page fault.

The operation of MFOE is as follows:

1. If MFOE sees that both the present bit and the MFOEable bit of PTE are not set, this means that this page cannot be pre-allocated or this faulting address might be illegal. MFOE triggers the page fault exception in this case.
2. If MFOE sees that the present bit of PTE is not set but the MFOEable bit of PTE is set, this means that this PTE belongs to a legal virtual address. MFOE can now proceed to the next step of handling the minor fault.
3. MFOE then retrieves the starting (physical) address of the pre-allocation table by reading the register CR9 from its context. The retrieved address is used to read the header of the
pre-allocation table. The header read from the pre-allocation table reveals the tail index into the pre-allocation table.

4) MFOE computes an entry into the pre-allocation table, with the address to the pre-allocation header and tail index retrieved from the header. With this address, MFOE reads an entry of the pre-allocation table.

5) The entry read returns a 64-bit word, which consists of a valid bit of the pre-allocated entry. If the valid bit is set, then MFOE can proceed to the next step of page fault handling. If not, this means that all the entries in the pre-allocation table have been consumed and MFOE has to raise the page fault exception for the fault to be handled by the kernel.

6) In the next step, MFOE writes back the pre-allocation entry with updated TGID, VA fields, and used bit set and valid bit unset. MFOE used the CitationTGID (written by kernel at Section 4.3). It also modifies the PFN field of PTE with PFN obtained from the pre-allocated entry. MFOE also increments the tail index of the pre-allocation table and writes back the updated tail index to the pre-allocation table header.

7) MFOE also updates the PTE here itself, without needing to do the page walk again. It also makes an entry in the TLB to prevent a page walk when the access is retried by the core after fault handling.

Figure 6 shows the operations of MFOE during a page fault.

4.5 Post–Page Fault Software Handling

We used the delayed work queue of the Linux kernel to periodically execute the post–page fault processing function. After the pages of all pre-allocation tables are filled by background thread (triggered by MFOE_enable system call), the post–page fault processing function is inserted into the work queue with a programmed delay. This programmed delay dictates how frequently the post–page fault processing should be done. For our implementation we have set it to 2 milli-seconds.

The following tasks are executed in order in the post–page fault processing function, for each per-core pre-allocation table.

1) First, we use the contents of the CR9 register to access the pre-allocation table header. Then we use the head field of the header to index into the corresponding head entry of the
pre-allocation table. If the head entry’s used bit is set, we proceed to the next step. If the used bit is not set, it means that the MFOE hasn’t used the pre-allocation table, possibly indicating that there are no faults since the last pre-allocation table refresh.

(2) From the used pre-allocation table entry, we obtain the TGID, faulting virtual address, and page frame number. Using these values we sequentially call `anon_vma_prepare`, `inc_mm_counter`, `page_add_new_anon_rmap`, and `lru_cache_add_active_or_unevictable` functions. After all the functions are executed, all the fields of the pre-allocation table entry are cleared.

(3) We now allocate a physical page to replenish this head entry. The frame number of the new page is written to the PFN field, along with the valid and used bits set to 1 and 0, respectively. The head index is incremented by 1 and all the tasks are repeated for the next entry.

Figure 7 illustrates the operations of the post–page fault processing function and how it updates the entries of the pre-allocation table.

Since we are performing post–page fault processing once for a programmed delay time (2 ms), it is possible that during this interval, the program might have exhausted all the allocated pages. In this case, the MFOE would raise an interrupt to the kernel and fall back to typical page fault handling. Plus, in our current implementation, the post–page fault processing is single threaded, but implementing a multi-threaded version is plausible.

The current implementation of the post–page fault processing does leave room for optimization. For example, instead of periodically triggering the post–page fault processing function, we could dynamically awaken the function based on pre-allocation pages available for the application. Also, based on the usage, one could dynamically increase/decrease the number of the pre-allocation pages for each core. As the post–page fault processing function is working to replenish the entries of the pre-allocation table of one core, the other core might be starving for pre-allocated pages, resulting in the latter core suffering longer latency in its minor faults. This effect can potentially lead to fairness issues in a multi-core system. To mitigate this to some extent in the current implementation, we use a round-robin policy to select the cores for pre-allocation.

4.6 Error Handling

Since the post–page fault processing function executes synchronously once every programmed delay period, there can be a situation where an application is terminated while some of its
pages (allocated through MFOE handled page faults) are yet to be processed. This scenario becomes worse if some of the pre-allocated pages are still unused and need to be reclaimed by the kernel.

To handle such scenarios, we implement an error handling function that will be called when the application is getting terminated. This function will scan pre-allocation tables of all the cores to find the used entries (with used bit set) whose TGID is the same as the terminated application. If such entries exist, then the error handling function executes the same set of functions as the post-page fault processing function. Also, since the error handling function and the post-page fault processing function might execute at the same time, we need a lock bit to avoid race conditions between them. The bit 0 of the locks field in the pre-allocation table header is employed as a test_and_set lock bit here.

Because the error handling function needs to scan all pre-allocation tables, it should only scan when necessary. Therefore, the error handling function is only called by the function zap_pte_range() function, when the page's page_mapcount() returns zero and when the vma’s VM_MFOE flag is set. It is also worth mentioning that if the parent thread (the thread whose PID is also the TGID) of a thread group is killed before its child threads, then the parent thread must be responsible for handling the unprocessed pages of the pre-allocation tables before the kernel selects a new thread group leader.

4.7 Disabling MFOE

The MFOE_disable function sets the mfoe member of the calling process’s memory descriptor (mm_struct) object to false. Like MFOE_enable, the last caller disables MFOE hardware of each core by calling the schedule_on_each_cpu() function to clear the MFOE_ENABLE bit in the CR9 register for all cores, and it also frees/releases all valid pages (whose valid bit is set) from all the pre-allocation tables.

As discussed in Section 4.1, MFOE is implemented to be automatically disabled by the MFOE_disable() function from the __mmput() kernel function, which is called when the applications are terminated. So disabling MFOE within the source code of applications is unnecessary. Nevertheless, we implemented the MFOE_disable system call to give the applications the flexibility to disable MFOE when needed. Also, we implemented a protection code to ensure that calling the MFOE_disable() function twice accidentally will not be harmful. When called twice, the second MFOE_disable system call would directly return.

4.8 Software Emulation of MFOE

The critical path of minor page fault handling can also be implemented in software. Hence, we implement the software emulation of MFOE in the form of mfoe_se() system call. mfoe_se is called with the faulting virtual address as input in the microbenchmark used in the evaluation. Although this implementation needs the programmer to have explicit knowledge of the faulting address beforehand, the functionality of this system call could be easily integrated with a typical page fault handling routine (inside do_anonymous_page()) in future implementations. The software emulation of MFOE is made possible because of the kernel’s capability to track both the page tables and pre-allocation tables.

The implementation of mfoe_se() is as follows:

1. First we obtain the memory descriptor of the process. Then we hold the non-exclusive access to mmap_sem using down_read. We get the virtual memory area descriptor (struct vm_area_struct) by calling the find_vma() function with the current memory descriptor and faulting address as inputs.
(2) After obtaining the virtual memory descriptor, we check if it’s flagged with VM_MFOE. If it is not, then we release the mmap_sem and return with error. If it is flagged as required, we proceed to next steps.

(3) Now we start the multi-level page walk through a series of kernel function calls pgd_offset, pud_offset, pmd_offset. We read the leaf entry in the page table-pte by obtaining the spinlock to the entire last-level page table, by using pte_offset_map_lock. At this stage we hold exclusive access to the last-level page table.

(4) Now we read the pte to check if it has the MFOE’able bit enabled, no valid translation present, and with TGID filled in the PFN field. If either of these fields are not present, we release the spinlock and mmap_sem, and return with error. Otherwise we proceed to the next steps.

(5) After checking the validity, we access the pre-allocation table of the faulting core (core number is obtained by smp_processor_id() kernel function). We access the entry in the pre-allocation table that’s pointed out by the tail pointer. If the entry is available (used bit as 0 and valid bit as 1), then we proceed to the next step. Otherwise we release all the locks and return with error.

(6) Now we take the TGID present in the pte and fill it in the tail entry of the pre-allocation table. We also update the pre-allocation table tail entrys virtual address, used, and valid fields. Subsequently, we update the pte with PFN from pre-allocation table. set the used and valid bits to 1 in the PTE.

(7) We release both the page table spinlock and mmap_sem and return with success to the faulting process.

At this stage, when the virtual address is accessed through load/store, it won’t generate a page fault but will result in a TLB miss. The hardware page table walker performs the walking to the get the translation we create through the mfoe_se system call and caches it in TLB.

4.9 Other Issues

**Resource Limitation:** Each process in Linux can only allocate a limited amount of resources, such as CPU time, memory, and files. Although it is rare, it is still possible that the MFOE-enabled process would allocate pages more than its limitation because MFOE hardware cannot check during a page fault. To prevent this case, the periodic post–page fault background thread needs to check the allocated pages and the quota of pages for this process. If the total allocated pages of a process are over some threshold, say 80% of main memory, this background thread will call the MFOE_disable() function for this process and trigger another background thread to clear the MFOEable bit of PTEs of all unprocessed pages of this process.

**Huge Page Support:** MFOE can also support huge pages easily. The current architecture can be re-purposed to use huge (2MB) pages instead of base (4KB) pages easily. If the intention was to use both 4KB and 2MB pages for handling the faults, another pre-allocation table per core is needed to contain all pre-allocated huge pages for a core. The kernel background thread in Section 4.3 will set the MFOEable bit at PMD if a huge page can be allocated for certain virtual address regions. When MFOE reaches PMD and finds the MFOEable bit is set but the present bit is not set, MFOE will obtain a huge page from the pre-allocated table.

Our approach in this article provides some advantages similar to the use of huge (2MB) pages, while managing the memory with base (4KB) pages. For example, using huge pages would reduce the number of faults that the application has to endure, but the kernel would have to deal with promoting/demoting huge pages and also perform memory compaction to make sure that it doesn’t run out of huge pages. On the other hand, with MFOE, the application would have to endure more faults, but the latency of each page fault now would be reduced and subsequently the kernel has to do less amount of compaction.
Virtual Machine Support: Prior work shows that the memory translation in the virtual machine environment becomes two-dimensional [5, 8]; i.e., each physical address in the guest will be a virtual address in the host. TLB in a virtual system caches the guest virtual address to host physical address translations. A TLB miss would trigger the 2D hardware page table walker, which walks both the guest and host page tables. MFOE in a virtual system environment will be implemented as optimizations to the 2D page table walker.

We outline one possible way of implementing MFOE in hardware-assisted virtualization. The guest OS will have its software changes in the Linux kernel, as if it’s run on a real machine. The guest OS would construct the page table walking path and set the MFOEable bit in the leaf entry of the page table, as explained in Section 4.3. Since the virtual machine is like any process for the host, the host OS (hypervisor) also constructs the page table walking path and sets the MFOEable bit for the lowest PTE. Both guest OS and hypervisor would maintain their own pre-allocation tables, with MFOE being able to access both of them.

In the event of a TLB miss, the page table walker starts to walk the 2D path till it reaches the lowest PTE in the guest page table first. If the MFOEable bit is set, MFOE will be consulted to obtain a page from the guest pre-allocation table. But the walk won’t stop here; it continues till the walker reaches the lowest page table entry of the host. Again here, if the MFOEable bit is set, MFOE is consulted now to get the physical page number from the host pre-allocation table. Finally, the MFOE updates the corresponding TLB entry and re-executes the faulting instruction. We should mention that we should avoid breaking the translation path from pages in the guest pre-allocation tables to its physical address in the hypervisor. The ballooning driver and the hypervisor swapping mechanisms [34] adopted in current hypervisors might have to be disabled for the physical pages in hosts that support pre-allocated pages in guest OSs. We leave this virtual machine support as a future work.

Multi-thread Support: With multi-threaded applications, two or more threads might access the same page and therefore encounter the page fault (of the same page) at the same time. Furthermore, we should consider a more complicated race condition case between MFOE hardware and the kernel software page fault exception handler. In Section 4.3, a kernel background thread is used to set the MFOEable bit for all PTEs. Consider the following rare case: the MFOE hardware reaches a PTE before the background thread sets its MFOEable bit. Therefore, MFOE hardware will treat this page as “non-MFOEable” and trigger a page fault exception. Before the page fault exception handler executes, our kernel background thread could be scheduled and sets the MFOEable bit of this PTE. MFOE hardware from another core could reach this PTE (since its present bit is not set but the MFOEable bit is set) and execute the MFOE page fault handling. Meanwhile, the page fault exception handler triggered from the first core executes and starts to handle the page fault for the same faulting page. This hardware and software race condition might be a serious problem, so we have to avoid it.

To solve this problem, we employ a bit lock at the lowest PTE. That is, when the MFOE hardware and kernel page fault exception handler want to service a page fault, they both need to obtain a lock at the lowest PTE first. We re-purpose the avl bit of the empty PTE as the lock bit. In software, the test_and_set operation is used by the fault handling method to atomically set the lock. If the kernel sees that the lock bit is already set, this means that MFOE from some hardware context

---

4 The kernel can check the VM_MFOE flag from VMA to decide if it needs to get the lock or not. If VM_MFOE is not enabled, this locking step is skipped.
5 This bit is not used in an invalid PTE for a typical system and is not used for detecting if the page is swapped. Hence, this use is justifiable.
is handling the fault. It is justifiable for the kernel to busy wait on a bit lock, as the MFOE fault handling takes a very small amount of time.

The page fault handling flow mentioned in Section 4.4 becomes as follows. Once the MFOE is able to obtain a free page from the pre-allocation table, MFOE must do a read_modify_write atomic operation on the PTE to acquire the lock. This is accomplished in hardware using locked memory read and write operations. MFOE does a locked memory read on PTE, resulting in blocking of any other core trying to read the same PTE. If MFOE sees the lock bit is 0, it does a locked write to PTE, setting the lock bit set to 1. The other core that has been blocked trying to read the same PTE will now read the PTE with lock bit as 1. If the the read_modify_write operation is successful, MFOE proceeds to the next step of page fault handling. After the page fault has been handled, MFOE will clear the lock bit to unlock the PTE. However, if the initial locked read returns 1 on the lock bit, this means that MFOE from another core or kernel page fault exception handler is handling a page fault for the same page, and then MFOE would need to busy wait here until the lock bit is clear and the present bit is set (this means that the page fault has been serviced), and then update TLB as well as re-execute the faulting instruction.

This architecture is more scalable compared to the existing kernel exception handler approach. MFOE only busy waits at the page level, and this busy waiting happens only when multiple MFOEs or the kernel exception handler tries to access the same page simultaneously. However, the existing kernel page fault exception handler uses a global page table lock to synchronize all page faults of the threads belonging to the same process. Therefore, in a typical system, all faulting threads of the same process must busy wait no matter if their faulting pages are the same or not.

5 EVALUATION

Here we first discuss our evaluation methodology, followed by a discussion of results on microbenchmarks and full applications.

5.1 Methodology

We evaluate the full hardware/software MFOE technique against traditional minor fault handling in OS software baseline. To explore the benefits of the proposed MFOE-enhanced hardware page walker, we also examine a software emulation of MFOE, without the enhanced hardware page walker, implemented as a system call. These systems are evaluated under a combination of large memory footprint applications as well as a set of microbenchmarks used to aid in the analysis of the system. For full application workloads we select applications taken from several benchmark suites like GAPBS [7], Memcached as a database layer of YCSB [14], SPLASH, and PARSEC [37]. The reason we have picked only a few applications from their respective benchmark suites is that these applications have considerable minor fault overhead for evaluating MFOE. We also wanted the evaluated benchmarks to be diverse. More information about each benchmark and their configuration can be found in Table 2. All the applications are run in multi-threaded mode (if possible), and parallel fault events are taken into consideration when calculating minor fault overhead.

Because MFOE proposes both hardware and software changes to the system, we must simulate the full proposed system to accurately model the impact of the proposed changes. The primary benefit of MFOE, however, will be seen on workloads with large memory utilization, which makes simulation challenging. Thus, to evaluate the performance of our proposed MFOE system, we take a combined approach of detailed microarchitectural simulation together with real system tracing and analytical modeling.

For detailed microarchitectural simulation, we implemented MFOE on gem5 [11] in full-system emulation mode as it allows prototyping with both hardware and software. The simulated gem5 system configuration is as shown in Table 3(a). The gem5 page walker-TLB sub-system
Table 2. Benchmarks Description

| Benchmark          | Description                                                                 |
|--------------------|-----------------------------------------------------------------------------|
| GCC compiler       | Representative of Compiler workloads.                                        |
|                    | Linux kernel 4.19 compilation using gcc(multi-threaded mode)                  |
| Parsec-dedup       | Multi-threaded data de-duplication benchmark taken from PARSEC [10] benchmark suite |
| Function-as-a-service | Faas-profiler [31] generated synthetic traffic-20 invocations/sec          |
|                    | No pre-warm containers. Container memory limit = 128MB                       |
| YCSB-memcached     | Memcached used as database layer for YSCB [14].                              |
|                    | Workload used is 100% insert.                                                |
|                    | Memcached server is running on 4 threads                                     |
| Splash2X-radix     | Multi-threaded radix sort benchmark taken from SPLASH2X [35].                |
|                    | Input size is chosen to be “native”                                         |
| Splash2X-fft       | Multi-threaded fourier transform benchmark taken from SPLASH2X [35].         |
|                    | Input size is chosen to be “native”.                                         |
| XSBench            | Mini-app representing key computational kernel of the Monte Carlo neutron transport algorithm [33]. |
|                    | Used input size “large”                                                      |
| Integer Sort       | Integer sort benchmark taken from NPB [20].                                  |

Table 3. System Configuration Used for Evaluation

(a) Gem5 Simulated System Configuration

| Architecture          | x86_64          |
|-----------------------|-----------------|
| CPU model             | TimingSimpleCPU |
| No. of cores          | 8               |
| Cache levels          | 2               |
| Clock speed           | 3GHz            |
| L1 I-Cache            | 32KB            |
|                      | 4-way set-associative access-latency: 2 cycles |
| L1 D-Cache            | 32KB            |
|                      | 4-way set-associative access-latency: 2 cycles |
| L2 Cache              | 256KB           |
|                      | 8-way set-associative access-latency: 8 cycles |
| Cache coherency       | MOESI Hammer    |
| Main memory           | 16GB DDR4 2400 MT/s |
| Kernel                | Linux 4.9.182   |

(b) Real System Configuration Used for Evaluation

| Architecture          | x86_64          |
|-----------------------|-----------------|
| CPU model             | Intel Xeon E-2136 |
| Clock speed (max)     | 3.3GHz          |
| No. of cores          | 6               |
| No. of threads        | 12              |
| Cache levels          | 3               |
| L1 I-Cache            | 32KB            |
|                      | 8-way set-associative                                      |
| L1 D-Cache            | 32KB            |
|                      | 8-way set-associative                                      |
| L2 Cache              | 256KB           |
|                      | 4-way set-associative                                      |
| L3 Cache              | 12MB            |
|                      | 16-way set-associative                                     |
| Main memory           | 32GB DDR4       |
| Kernel                | Linux 4.9.182   |

was enhanced to implement MFOE. This system was used to perform microbenchmarking and establish the latency of typical minor page faults as well as MFOE-enhanced minor page faults. The microbenchmark analysis is presented in Section 5.2.1.

We also implemented a software-only version of MFOE in the form of a system-call (mfoe_se) whose implementation was explained in Section 4.8. We evaluated mfoe_se on a real system whose configuration is detailed in Table 3(b).
Since gem5 is too slow for memory-intensive applications to complete in a tenable amount of time, we estimate the gain for benchmarks as if they are run on the proposed hardware with a background thread refreshing the pre-allocation tables. This was done via analysis with the performance model of a system equipped with MFOE. The performance model takes the workload traces generated from real systems running these benchmarks as an input and gives the estimated performance gain and MFOE hit rate.

We incorporated the MFOE hit latency and MFOE miss penalty measurements taken from gem5 simulation in this performance model. Plus, we have measured background thread throughput and initialization throughput from the software-only implementation of MFOE. Background thread throughput is the rate at which the kernel background thread will process the MFOE consumed pages and re-allocate frames for future consumption. Initialization throughput is the rate at which the pre-allocation tables are initialized with free pages at the start of the application. Initialization throughput is expected to be larger than background thread throughput because the quantum of work for initialization (frame allocation) is smaller than the background thread (frame allocation plus kernel meta-data updates). We believe that this performance model will give us the best approximation of performance gains due to MFOE.

The working mechanism of our performance model is event based, the event being minor page fault. Each entry in trace we captured from the real system has the core number on which the fault has occurred, and the latency of the page fault. The performance model keeps track of pre-allocation tables, and their refresh timings have been known due to measurements we incorporated from microbenchmarks. So, given the occurrence of a fault on a core, we check the pre-allocation table for free physical page availability. If it’s available, we declare the fault as MFOE hit and all the future faults that occur on the faulting core will be moved back in time (page fault latency - MFOE hit time). If the fault is MFOE miss, we move all the future faults occurring on the same core into the future by MFOE miss penalty. Also, to simulate the producer-consumer nature of software and hardware, we check for faults between each scheduled pre-allocation table update.

Through the explained mechanism, the performance model generates a new trace (timeline) of minor fault events with modified latency. Comparing the fault overheads in the original trace, we can estimate the performance gain through MFOE. Overall performance model methodology is shown in Figure 8. The values of parameters chosen for modeling are shown in Table 4.

5.2 Microbenchmarks

5.2.1 Latency Benchmarks. We first examine the critical-path latency of MFOE-handled anonymous faults, which is the in-line time taken by MFOE to service the page fault. We call this latency MFOE hit latency. Not all minor faults that occur when MFOE is enabled will result in a hit. MFOE
misses occur when the given thread uses up all the pre-allocated pages before the background post-page fault processing thread replenishes them. In this case, fault handling reverts back to the default Linux kernel. Because MFOE is enabled, however, there is some wasted time as MFOE first attempts to service the fault. We call this extra time the MFOE miss penalty.

We devised a microbenchmark that creates and strides through a memory region. Between each stride access, the benchmark does a programmable amount of redundant data accesses (non-faulting) to simulate the behavior of a real program. We do the redundant accesses to simulate different levels of cache availability for the MFOE memory accesses. We made this program multi-threaded; i.e., each core will perform faulting accesses and redundant accesses. Each thread generates a total of 32K faults. The fault latency is measured as the difference between the clock ticks of start and end events in hardware. The start event is the MFOE invocation by the hardware page table walker, when the walker sees there is no translation for an MFOE-able page. The end event can be either of two possibilities: one possibility is when MFOE indicates to the faulting core to retry the faulting instruction (MFOE hit) and the other is when MFOE reverts back to fault handling in software due to lack of pages in the pre-allocation table (MFOE miss).

We observe that the MFOE hit latency on average is 36 CPU cycles, with standard deviation of 42 CPU cycles. Such a large standard deviation is expected as handling page fault with MFOE involves multiple memory accesses, which might or might not hit in CPU caches. The tail latency (95th percentile) of the MFOE hit is observed to be 125 CPU cycles. Since the standard deviation is higher than the mean, we take 78 cycles (mean + SD) as a tighter estimate for MFOE hit latency. On the other hand, MFOE miss penalty has an average of 14 CPU cycles, with standard deviation of 5 cycles. The tail latency of MFOE miss is found to be 14 cycles. The MFOE miss penalty has little standard deviation because it performs less number of memory accesses compared to MFOE hit.

We ran the same program on the real system and analyzed the latency of minor page fault handling in an unmodified Linux kernel. We measured the wall clock time spent in the kernel function `do_anonymous_page()`, which implements the core functionality of anonymous fault handling. The measurement is done using ftrace utility of the Linux kernel. ftrace captures entry and exit instants of the `do_anonymous_page` function. The average page fault handling latency in an unmodified Linux kernel was observed to be 1.27 $\mu$s (2,552 CPU cycles); the tail latency is found to be 3.20 $\mu$s (6,432 CPU cycles). Comparing these measurements, we find that MFOE improves the average minor page fault handling latency by a factor of 33× compared to the current systems. The tail latency of a minor page fault is improved by 51× compared to the baseline. Given that the default Linux minor page fault handling latency on average takes around 3,000 cycles, miss penalty incurred by MFOE is negligible.

It might seem that we are estimating the gain in critical path latency by comparing two different setups. Our evaluation choice is justified due to following reasons. Baseline measurements have to be done on real systems as the simulation system doesn’t reflect the configuration of the processors to date. While it would have been possible to measure the typical fault handling latency on the simulated system, based on our experience, the software timing measurements are more accurate in real systems than in full system simulation. The hardware timing measurements, on the other hand, are accurate in simulations, so we used those to measure how much time the MFOE takes for handling the page fault.

### Table 4. Parameters Used for Modeling

| Parameter                        | Value               |
|----------------------------------|---------------------|
| MFOE Hit Latency                 | 78 cycles           |
| MFOE Miss Penalty                | 14 cycles           |
| Background Thread Throughput     | 580.169 KPages/sec  |
| Pre-allocation Initialization Throughput | 1,093.075 KPages/sec |

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 4, Article 57. Publication date: September 2022.
5.2.2 Software-only Implementation of MFOE. For measuring the latency, we ran the same benchmark as in Section 5.2.1, but with mfoe_se() system call before every faulting access. We measured the time it takes for the mfoe_se to execute by measuring the entry and exit points of the system call through ktime_get_ns(). Our experiments indicate that minor faults handled through software emulation of MFOE have an average latency of 795 ns; the tail latency was observed to be 1,757 ns. Comparing with the baseline, software-emulated MFOE can speed up average fault handling latency by 1.59× and improves the minor fault handling tail latency by a factor of 1.82×.

These measurements confirm two things for us: (1) Pre-allocation and lazy kernel data structure updates are indeed helpful in reducing the critical page fault handling latency. Even when the simplified critical path is implemented in software (mfoe_se()), we are able to achieve latency improvement. (2) The hardware/software MFOE approach is superior in reducing minor fault handling critical path time. Hardware-implemented MFOE resulted in 33× reduction in critical path latency, compared to 1.59× improvement provided by software-only implementation.

We should highlight the performance contrast between hardware offload (MFOE) and software emulation. MFOE poses substantial improvement in fault handling latency because it can handle the page fault in less than five memory accesses, which translates to very few cycles depending on cache hits/misses. All the memory accesses that MFOE makes are with physical addresses, so memory virtualization overhead is removed. On the other hand, software emulation implementation has to call multiple kernel functions for fault handling, which, on aggregate, results in many memory reads/writes. Also, with software-emulated implementation of MFOE, there will be a second TLB miss when the access is retried, which will result in a redundant page table walk. But with MFOE, there will be a TLB hit when the access is retried. It is noted that software-only implementation provides substantial 59% improvement over the baseline Linux kernel, even though it cannot match the gains from a hardware implementation of MFOE.

Additionally, we have devised microbenchmarks to measure background thread throughput and initialization thread throughput for the software-only version of MFOE. These throughput measurements give us an approximation of how fast the pre-allocation tables on this system could be updated, if this system were to implement MFOE. These measurements are shown in Table 4 and are used in our performance model to estimate the gains due to MFOE. The comparison of mean and tail latency of MFOE and SW-MFOE with that of unmodified Linux kernel (baseline) is shown in Figure 9(a).

5.2.3 Throughput Benchmarks. We also ran a multi-threaded microbenchmark to stress MFOE of given number of CPUs by generating minor faults at an identical rate in parallel fashion. Here we examine what fraction of minor page faults can be handled under different rates of minor fault
Table 5. Estimated Performance Improvement

| App.              | Hit Rate (%) | Fault Overhead with MFOE (%) | Estimated Speedup |
|-------------------|--------------|-----------------------------|-------------------|
| GCC compiler      | 68.61        | 12.09                       | 1.254             |
| Function-as-a-service | 56.99    | 4.06                        | 1.057             |
| YCSB-memcached    | 97.25        | 0.66                        | 1.062             |
| Parsec-dedup      | 64.10        | 3.87                        | 1.049             |
| Splash2X-radix    | 72.92        | 7.46                        | 1.109             |
| Splash2X-fft      | 42.19        | 7.13                        | 1.035             |
| XSBench           | 22.91        | 3.95                        | 1.01              |
| GAP-BC            | 38.15        | 2.41                        | 1.010             |
| Integer Sort      | 56.18        | 1.80                        | 1.029             |
| Geo Mean          |              |                             | 1.066             |

occurrence. We perform this experiment on our simulated gem5 system. The results of this experiment are shown in Figure 9(b). In the figure we observe that MFOE reaches a hit rate of 90% at a very high maximum page faulting rate of 1 page fault every 21,000 cycles. We also observe that the eight-thread workload has a lower MFOE hit rate compared to lower multi-thread workloads in this experiment; this means that the evaluated MFOE configuration (pre-allocation table of 256 pages and post-page fault processing thread sleep duration of 2 ms) is unable to keep up with the increased page fault rate of eight threads. Should a given workload need to service page faults at a higher rate, it is straightforward to dynamically schedule this thread more often based on observed thresholds in the pre-allocation table or to deploy more background threads that process pre-allocation tables. Increasing the size of the pre-allocation table might also help in tackling a higher page fault rate.

5.3 Memory-intensive Full Applications

Table 5 shows the MFOE hit rate (i.e., the likelihood that a given minor page fault would be covered by MFOE) as well as the overhead of minor page faults remaining after MFOE (due to MFOE misses as well as other MFOE overheads) and the total performance gain provided by MFOE on these applications. As the table shows, MFOE provides an average gain of 6.6% across all the workloads examined.

Of particular note, memcached has a high hit rate of 97%, thus achieving runtime improvement of 6.2%, nearly completely removing the overhead due to minor faults. gcc has a high 68% hit rate and thus achieves a nearly 25% performance gain, effectively capturing most of the ideal runtime improvement of 1.40. By contrast, splash2x-fft is only able to achieve a 42% MFOE hit rate and thus its overheads are higher and only a 3.7% gain is realized out of a possible 11%. The same is the case with XSBench which is able to get a meagre gain of 1.01 due to 22.91% MFOE hit rate.

To characterize the effect of number of pages allocated during pre-allocation and the background thread refresh interval, we calculate the performance improvement and MFOE hit rate for four benchmarks across different MFOE configurations. The results are shown in Figure 10(b) and 10(a), respectively. As expected, these results show that the MFOE hit rate, and performance gain due to MFOE, decreases as the refresh interval of background thread increases. Also, the MFOE hit rate increases with increase in per-core pre-allocation width. The benefits from bigger pre-allocation width and/or faster background refresh depend on the nature of the application. When the pre-allocation width changes from 512 to 1,024, with a background thread refresh interval of 2 milliseconds, the FaaS application had a better MFOE hit-rate increment compared to gcc, dedup, and radix applications. This implies that the FaaS application benefits from
bigger pre-allocation widths, a possible indication of occurrence of burst faults. On the other hand, for 2-millisecond background refresh, splash2x-radix showed only 21% improvement in hit rate when the pre-allocation width changed from 128 to 1,024. This indicates that radix benefits pre-dominantly from a faster background refresh. Additionally, under low MFOE hit rates (for some of the refresh interval = 32-millisecond configurations), the applications show lower performance improvement but no performance loss, implying that MFOE miss penalty is not substantial compared to performance gains due to MFOE.

6 CONCLUSION
Application virtual memory footprints are growing rapidly, and to address this growing demand, system integrators are incorporating ever larger amounts of main memory, warranting rethinking of memory management. These trends are leading to growing overheads due to minor page faults in large memory workloads. In this article we proposed the Minor Fault Offload Engine (MFOE) to mitigate this problem through a hardware-software co-design approach. MFOE parallelizes portions of the kernel page allocation to run ahead of fault time in a separate thread. This is combined with a hardware accelerator for minor fault handling. MFOE moves much of page fault handling work to background kernel threads and carries out inline work in hardware at the time of the fault. We show that MFOE can nearly eliminate minor page fault handling costs for most applications and provides performance improvements of up to 25.4%.

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 4, Article 57. Publication date: September 2022.
[26] J. Navarro, S. Iyer, P. Druschel, and A. Cox. 2003. Practical, transparent operating system support for superpages. *ACM SIGOPS Oper. Syst. Rev.* Vol. 36, Association for Computing Machinery, New York, NY. [https://doi.org/10.1145/844128.844138](https://doi.org/10.1145/844128.844138)

[27] A. Panwar, S. Bansal, and K. Gopinath. 2019. HawkEye: Efficient finegrained OS support for huge pages. In *ACM ASPLOS*.

[28] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh. 2020. Perforated page: Supporting fragmented memory allocation for large pages. In *ACM SIGARCH Conference*.

[29] Nathan Pemberton, John D. Kubiatowicz, and Randy H. Katz. [n. d.]. *Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters*. Ph.D. Dissertation.

[30] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *IEEE HPCC Conf*.

[31] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’52)*. Association for Computing Machinery, New York, NY, 1063–1075. [https://doi.org/10.1145/3352460.3358296](https://doi.org/10.1145/3352460.3358296)

[32] S. Haria, M. Hill, and M. Swift. 2020. MOD: Minimally ordered durable datastructures for persistent memory. In *Proceedings of ACM ASPLOS*.

[33] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSbench - The development and verification of a performance abstraction for Monte Carlo reactor analysis. In *The Role of Reactor Physics toward a Sustainable Future (PHYSOR'14)*. [https://www.mcs.anl.gov/papers/P5064-0114.pdf](https://www.mcs.anl.gov/papers/P5064-0114.pdf).

[34] Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI’02)*.

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA’95*. ACM, 24–36.

[36] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC’11)*. ACM.

[37] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X. *SIGARCH Comput. Archit. News* 44, 5 (Feb. 2017), 1–16. [https://doi.org/10.1145/3053277.3053279](https://doi.org/10.1145/3053277.3053279)

Received December 2021; revised May 2022; accepted June 2022