Adaptive and Efficient Dynamic Memory Management for Hardware Enclaves

Vijay Dhanraj*, Harpreet Singh Chawla†, Tao Zhang‡, Daniel Manila‡,
Eric Thomas Schneider‡, Erica Fu‡, Mona Vij*, Chia-Che Tsai†, Donald E. Porter‡
* Intel Corporation
‡The University of North Carolina at Chapel Hill
†Texas A&M University

Abstract

The second version of Intel® Software Guard Extensions (Intel SGX), or SGX2, adds dynamic management of enclave memory and threads. The first version required the address space and thread counts to be fixed before execution. The Enclave Dynamic Memory Management (EDMM) feature of SGX2 has the potential to lower launch times and overall execution time. Despite reducing the enclave loading time by 28–93%, straightforward EDMM adoption strategies actually slow execution time down by as much as 58%.

Using the Gramine library OS as a representative enclave runtime environment, this paper shows how to recover EDMM performance. The paper explains how implementing mutual distrust between the OS and enclave increases the cost of modifying page mappings. The paper then describes and evaluates a series of optimizations on application benchmarks, showing that these optimizations effectively eliminate the overheads of EDMM while retaining EDMM's performance and flexibility gains.

1 Introduction

Intel SGX [26] is a powerful building block for protecting application code and data on a remote system, such as cloud. Specifically, an application can create a private region called an *enclave*, wherein the hardware protects sensitive code and data from system software, including the OS and hypervisor(s). The hardware encrypts enclave memory using a key known only to the CPU, thereby protecting the enclave from unauthorized access by system software and, in some configurations, via physical access (§2). SGX is particularly beneficial for securing cloud applications, and multiple cloud providers offer SGX-protected platforms [1, 2, 13, 14, 27, 50].

SGX is an ambitious attempt at hardware supporting mutual distrust between system and application software. However, Intel SGX version 1 (**SGX1**) imposes constraints on runtime management of enclave virtual memory, leading to performance overheads as high as 58% (§3.6). Specifically, the enclave cannot dynamically add, remove, or protect virtual pages during execution. The SGX driver can swap physical pages for an enclave, but the enclave's virtual memory layout must be set *before* execution. The startup time for an enclave is directly correlated with enclave size, creating an unsavory trade-off: a small enclave may run out of heap space, but has a fast startup time; in contrast, a large enclave has high startup times, but may over-allocate heap space. These limitations negatively affect the utility of SGX1 in practical deployments.

Intel SGX version 2 (**SGX2**) addresses these limitations with Enclave Dynamic Memory Management (EDMM) [25]. SGX2 adds new instructions that the SGX driver can use to add, remove, and change permissions of virtual pages in an enclave. In order to maintain the integrity of the enclave, the enclave must explicitly accept any allocation, deallocation, or modification of virtual memory before the change will be accepted into the TLB. In addition, newly allocated pages are zeroed by the hardware, as are deallocated pages.

Why does EDMM matter? Dynamic memory management is a fundamental necessity for any application that cannot statically predict its memory requirements, such as sizing its heap. Similarly, some applications need the ability to remove or change a mapping, such as to read-protect static data after initialization. Without EDMM, an application must overestimate its heap size and over-permission the heap's pages as readable, writable, and executable. Overestimating the heap size increases the enclave loading time in order to map and measure each page. This issue is particularly salient for a compatibility layer, such as a library OS that implements system APIs such as `brk`, `mmap`, and `mprotect`, as well as any userspace runtime environment that dynamically manages memory, such as a language runtime.

Why is designing an efficient EDMM scheme challenging? Intuitively, one would expect EDMM use to be a strict win: eliminate over-provisioned memory resources, and reduce application startup time spent on measurement of heap pages that may never be used. And indeed, using EDMM improves startup time. However, straightforward use of EDMM makes runtime application performance *worse* in many cases. For SGX2, we evaluate two separate strategies: one strategy immediately maps pages into the enclave when the applications makes a call such as `mmap`; the second strategy, proposed but not evaluated by Xing et al. [46], is demand allocation, which allocates upon the first access to a virtual page. The results of an experiment running GCBench, a garbage collection benchmark, shows that both EDMM designs cause significant slow down to the execution time, up to 41% and 58%, compared to just using SGX1. More details of the result are in §3.6.

The key intuition behind this result is that modifying memory mappings is more expensive in an enclave than in a normal process, due to the additional context switching required for the enclave to accept the changes. Adding an enclave mapping requires at least three context switches, and at least five if the mapping is added using demand paging (§2). Removing an enclave mapping is even more expensive, requiring at least *nine* context switches to ensure that the TLB entry is properly flushed. On our test machine, the cost of a demand fault on a newly mmap-ed area in a normal process is about 8 μs, whereas creating an enclave page mapping takes around 30 μs on SGX2.

Primary goals and findings: This paper investigates and addresses these overheads. We evaluate real-world applications on Gramine (formerly Graphene) [40, 41], as a representative compatibility layer for SGX. The **contributions** of this paper are as follows:

- An analysis on how straightforward use of EDMM *increases* application execution time, as does the optimization suggested in previous work [46].
- A series of optimizations for EDMM in Gramine and SGX that *remove* the overheads while retaining the space efficiency gains of EDMM.
- A thorough evaluation of these optimizations using both microbenchmarks and applications. These optimizations significantly enhance application performance, making EDMM in Gramine comparable or better to static allocation, while greatly improving application startup time.

2 Background

This section introduces background on SGX and the Enclave Dynamic Memory Management (EDMM) feature of SGX2, and relates this model to the more recent VM-based trusted-execution model of Intel TDX and AMD SEV.

Hardware trusted execution environments (TEEs), such as Intel SGX [26] and AMD SEV [3], protect security-sensitive computation and data from system-level attackers, including malware, OS rootkits, and sometimes even physical attackers. TEEs isolate the execution of a program, including CPU registers and memory, from the host operating system and other software. The CPU ensures integrity, and can generate attestation reports as proofs of integrity to remote entities. When a program runs inside a TEE, the program's memory is protected by hardware, using encryption [3, 5, 26] or access control [22], preventing a system attacker from accessing program memory. Currently, hardware TEEs have been widely applied for data-intensive computation [18, 20, 21, 34, 35, 37], control-plane software [8, 33], and privacy-preserving systems [9, 10, 19, 28, 33].

Leaf Func.	**Description**
System-tier (`ENCLS`)	
EAUG	Add a zeroed virtual page to an enclave.
EMODT	Modify the page type of a virtual page.
EMODPR	Reduce access permissions of a virtual page.
User-tier (`ENCLU`)	
EACCEPT	Accept addition or changes of a virtual page made by the OS
EACCEPT-COPY	Copy an existing virtual page into an `EAUG`'ed page and accept the `EAUG`'ed page into the enclave.
EMODPE	Extend access permissions of a virtual page.

Table 1. Summary of EDMM leaf functions.

This paper focuses on Intel SGX, one of the earliest hardware TEEs in widespread production. A hardware *enclave* created by SGX is a protected virtual memory range within a process's address space. In SGX version 1, the CPU does not allow changes to the enclave's virtual address space after initialization of the enclave. The only exception is swapping: the untrusted host OS may update the page table to unmap or remap an enclave page, but this involves a more complex check that the contents did not change while the page was unmapped. Swapped pages are encrypted by the hardware. Early versions of SGX limited the physical memory for all enclaves on a machine to 128MB; with the addition of Intel Total Memory Encryption (TME) [15], SGX now supports enclaves as large as 1 TB.

2.1 Enclave Dynamic Memory Management (EDMM)

Intel SGX version 2 (SGX2) introduced Enclave Dynamic Memory Management (EDMM). EDMM includes new *leaf functions* to SGX's `ENCLS` (system-tier) and `ENCLU` (user-tier) instructions, to add, remove, and protect a virtual page within an enclave. In SGX, `ENCLS` is only called inside the kernel, while `ENCLU` is called from userspace, either within or outside an enclave. The functionality of `ENCLS` and `ENCLU` is determined by an extra opcode (given by the `EAX` register), to indicate a leaf function. Table 1 lists the new leaf functions added for EDMM. For simplicity, this paper describes these leaf functions as instructions.

System-tier leaf functions (ENCLS) require agreement from the enclave software, since the host OS is not trusted to always provide the correct parameters. A benign OS is responsible for adding, removing, or protecting a virtual page in an enclave, as well as updating the page table accordingly. For example, to make an initially unmapped virtual page available, the OS needs to call EAUG with a physical page from the **EPC** (Enclave Page Cache, or physical pages reserved for use in enclaves). The OS subsequently maps the virtual page to the physical page in the page table, with the corresponding access permissions. The enclave must then approve the changes using EACCEPT to make the changes take effect. Note that to extend the access permissions of a virtual page, the enclave does not need the OS to intervene and can directly request the change using EMODPE, assuming that the page table is more permissive for this virtual address.

Anecdotally, the introduction of EDMM is useful for libraries or unikernels [6, 7, 40] for porting legacy applications into enclaves: (1) EDMM allows dynamic allocation or deallocation of virtual pages after enclave launch. Without EDMM, most enclaves will have to overpopulate virtual memory, increasing enclave startup time. (2) EDMM allows an enclave to dynamically and securely change page permissions after enclave launch, which is crucial for implementing systems APIs like mprotect, as well as supporting the process of ELF loading. Without EDMM, a library OS needs to *unsafely* make the entire heap readable, writable, and executable, because mprotect does not function after enclave launch. (3) EDMM allows dynamically changing a virtual page to a Thread Control Structure (TCS) page, which is needed for thread creation. Without EDMM, the number of enclave threads must be determined statically, and multithreaded enclaves may have to overestimate the number of threads since TCS pages cannot be added afterwards.

In this paper, we focus on optimization of the system flow for allocation and deallocation of virtual pages and kernel threads. We leave optimization of thread creation and changing page permissions to future work.

2.2 VM-based TEE Memory Management

Another trend of trusted execution environment (TEE) is to place an entire virtual machine (VM) in a hardware-protected domain, in which the sensitive application can be fully supported by a trusted guest OS. For example, AMD SEV-SNP [3, 4] isolates VM state from the hosting hypervisor, with the VM's associated physical memory encrypted by the CPU, and changes to guest-to-host memory mapping being detectable. Intel Trusted Domain Extensions (TDX) [17] adopts a similar strategy, by including the entire DRAM as the EPC (Total Memory Encryption) and protecting the extended page table (EPT) from the host.

For these VM-based TEEs, the virtual memory of the sensitive application(s) in the VM is fully managed by the guest OS, which also manages a pool of physical pages with hardware encryption. Unlike SGX, a VM-based TEE does not require intervention of the host to extend or shrink the virtual memory space of applications. But because both VMs and EDMM-enabled enclaves must dynamically add and remove physical pages from a TEE, they require a similar flow to SGX's EDMM. For example, in TDX, a protected VM needs the host OS to issue the same EAUG instruction to add a physical page, and then the VM needs to approve the change using EACCEPT. This suffers a similar as SGX2's EDMM, although it may occur less frequently since some changes, such as changing page permissions or remapping a page, can be made by the guest OS without involving the host OS.

3 Baseline EDMM Performance

We study the overheads of EDMM in the Gramine library OS [41], as a representative example of a "lift-and-shift" framework for running applications in SGX. This section begins by describing baseline virtual memory management support in Gramine, followed by our straightforward implementation of EDMM support in Gramine. We implemented all operations synchronously.

3.1 Threat Model

This work follows a common threat model for SGX applications. The only trusted components are the CPU(s), remote attestation services, and the code and data running inside the enclave. For remote attestation, we trust a special enclave and its containing software, called aesmd, provided by Intel. The hardware outside of the CPU package, the hypervisor, OS, and other code outside of the enclave are untrusted. The recent *Scalable SGX* foregoes integrity protection against physical attacks, such as a memory interposer, but does ensure confidentiality against physical tampering. Our prototype, based on Gramine, uses the in-kernel SGX driver, but does not trust this driver. Denial-of-service, cache-based side-channels, and controlled channel attacks [47] are out of the scope of this work.

3.2 Experimental Setup

We use a desktop, with a 144-core 2.40 GHz Intel(R) Xeon(R) Platinum 8360Y CPU with a 108 MB L3 cache and hyperthreading enabled, 248 GB RAM, an approximately 4 GB EPC, and a Samsung 970 EVO 500 GB SSD. The host OS is Ubuntu 20.04.6 LTS, with Linux kernel 6.7.0 and the in-kernel SGX driver. Our kernel

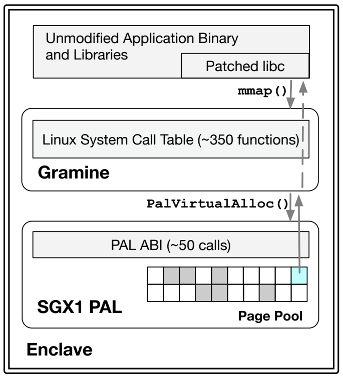


Figure 1. The Gramine architecture on SGX1. The PAL (platform adaptation layer) internally manages a page pool to return statically allocated, free virtual pages (in cyan) to `mmap`.

includes a patch to the SGX driver to support an optimization described later (§4.2); this patch does not affect the behavior of the SGX driver when the optimization is not used, as in these tests.

Unless otherwise noted, results are the average of 10 runs, and error bars represent 95% confidence intervals.

Application Workloads. We use three applications to evaluate EDMM performance: GCBench, RBench, and Redis. GCBench [32] exercises the Python (version 3.8.10) garbage collector by allocating and reclaiming representative data-structures, such as a set of balanced binary trees and a large array of floating point values. We use version 2.5 of RBench [43], which tests a variety of scientific computing tasks, such as matrix computation and calculating Fibonacci numbers, using version 3.6.3 of R. Lastly, Redis [36] uses Yahoo! Cloud Serving Benchmark (YCSB) [48] to test the performance of Redis (version 6.0.5), an in-memory key-value database. For RBench and GCBench, we collect the end-to-end execution time using the `time` command. YCSB on Redis reports throughput across six sub-workloads. We size enclaves for running RBench benchmark at 2 GB, and GCBench benchmark, Redis server at 512 MB, based on their memory usage.

3.3 Baseline 1: Static Allocation (static)

Gramine was originally designed for SGX1, on which the enclave's virtual memory layout is determined statically. Because of this limitation, Gramine on SGX1 cannot implement memory-related system calls, including `brk`, `mmap`, `munmap`, and `mprotect`. Although the virtual memory space is statically configured, Gramine does still dynamically assign allocated pages to purposes such as the heap, using Gramine-internal bookkeeping.

Gramine implements enclave management memory logic in its **Platform Adaptation Layer (PAL)**. Gramine implements one PAL per supported host platform (e.g., Linux on SGX); the PAL abstracts and encapsulates host-specific differences, so that the library OS can run on top of any PAL without modification. The three PAL APIs for memory management include: `PalVirtualMemoryAlloc`, `PalVirtualMemoryFree`, and `PalVirtualMemoryProtect`. Figure 1 shows the architecture of Gramine and its memory management mechanisms.

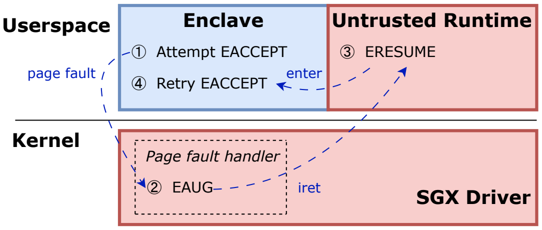


Figure 2. The current system flow for dynamically adding a page to an SGX enclave with EDMM. Trusted software is shaded blue. Compared to allocating a page to a normal process, this requires an additional context switch, back into the enclave, and that the enclave accepts the new mapping.

On SGX1, Gramine implements these APIs using a **page pool** to manage the heap. Gramine pre-allocates a range of virtual pages within the enclave as the page pool. The page pool is used to serve subsequent requests for pages, including mapping files or extending the heap. Because these pages cannot change permission dynamically, they are mapped readable, writable, and executable. Because most dynamic linking happens after enclave launch in Gramine, only the PAL loader binary is mapped read-only; all other binaries are in writable inside the enclave. Gramine implements `PalVirtMemoryProtect` as a no-op on SGX, since SGX1 does not allow changing page permissions during runtime. In the experimental results, we refer to this design as static.

3.4 Baseline 2: Basic EDMM Support (edmm)

We first describe a straightforward approach to adopting EDMM in Gramine. We use the same page pool abstraction as described previously, but rather than map all of these pages at enclave launch, we instead map them in response to the first use by an enclave-level, emulated system call in Gramine. For instance, when an enclave application issues an `mmap` call to Gramine, Gramine will select an unmapped page from the page pool to return to the application. The application could trigger a demand fault by simply reading or writing to the page, which would be serviced by the SGX kernel driver. The SGX kernel driver then creates the mapping (using `EAUG`), and returns to the enclave, which in turn accepts the new mapping, exits the enclave, and then re-enters the

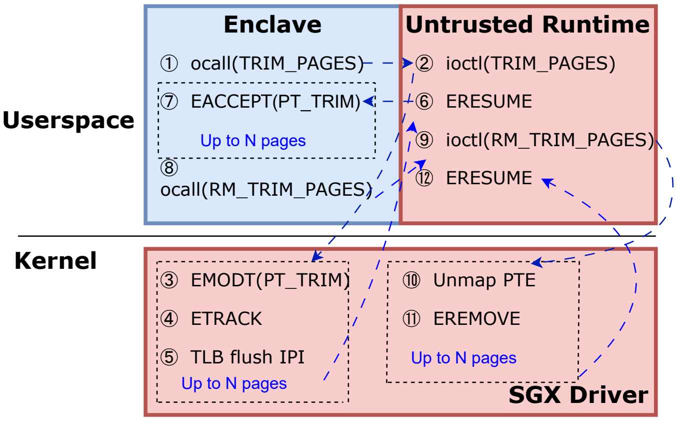


Figure 3. The system flow for removing a virtual page mapping from an enclave.

enclave to resume execution. This additional enclave exit is an SGX hardware requirement.

We adopt an optimization proposed, but not evaluated, by Xing et al. [46] (their §5.2), that avoids the additional exit between accepting the mapping and resuming execution. The trick is to have the EACCEPT be the instruction that triggers the fault, so that the page is accepted while resuming execution. This is illustrated in Figure 2. When an enclave issues an EACCEPT instruction on an unmapped virtual address, this exits the enclave and raises a page fault in the kernel. Then, the SGX driver will map the faulting address using EAUG. When the process resumes execution, it will pass through the untrusted runtime to transition back into the enclave; then, the enclave will sucessfully retry the EACCEPT. Gramine implements this by synchronously calling EACCEPT on each virtual page mapped by PalVirtMemoryAlloc (called by either mmap or brk). In the experiment results, we refer to this design as edmm.

Removing a mapping in EDMM uses a similar, but more complex system flow, in order to avoid race conditions with stale TLB mappings. Figure 3 illustrates the system flow recommended by the Intel manual, which is implemented by the SGX driver. To remove a virtual page, the enclave first issues an ocall to exit the enclave, and then uses ioctl to get the SGX driver to transition each page in the range into the TRIM (or pending removal) state. The driver also blocks creation of any new mapping at the page (using ETRACK) and shoots down any cached TLB mappings on each CPU core. Then, the flow returns to the userspace and re-enters the enclave to call EACCEPT. Next, the enclave issues an ocall again to exit the enclave and then uses ioctl to officially call EREMOVE on each virtual page.

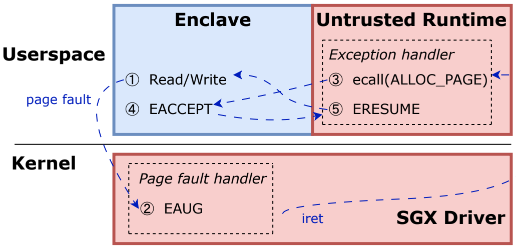


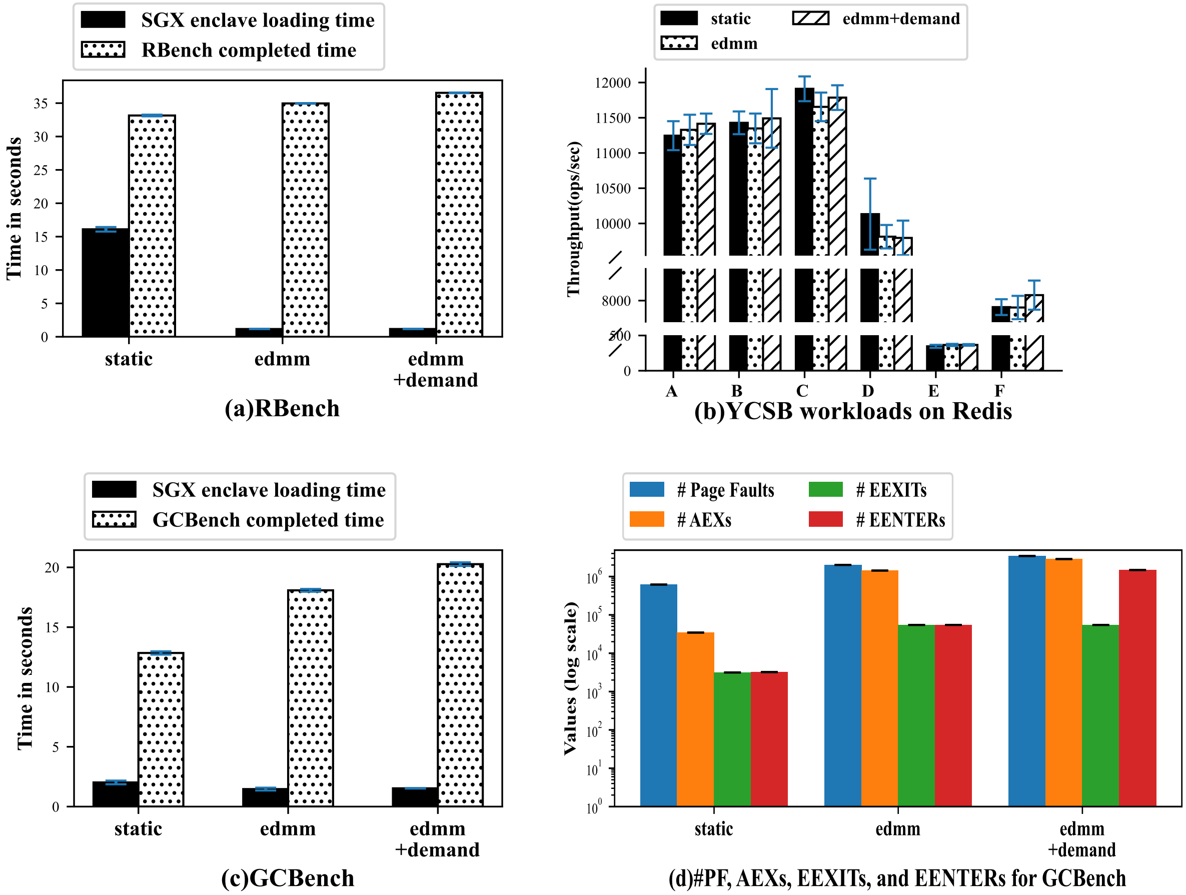
Figure 4. System flow of creating a new enclave virtual memory mapping via demand allocation, involving five context switches among the untrusted kernel, Gramine's untrusted runtime, and the enclave. Trusted components are in blue, untrusted in pink.

3.5 Baseline 3: Demand Allocation (edmm+demand)

Demand allocation is a standard optimization in memory management. In prior work, Xing et al. [46] propose, but do not implement or evaluate, a design for demand allocation in SGX. They propose an *implicit region* as an abstraction within the enclave's address space, wherein a page fault is assumed to be an implicit allocation request to the kernel. The enclave would communicate these regions to the kernel and SGX driver in advance of the page faults.

We adapt this design to Gramine by treating the entire enclave page pool as the implicit region. When the enclave launches, the page pool is initially unmapped. Just as with the baseline EDMM design, as the application or library OS requests mappings, Gramine logically allocates regions of the page pool. The key difference is that each page in the page pool now has a state, mapped or unmapped, which Gramine tracks with an additional bit per page. The first time a page from the page pool is allocated to an application, it will be in the unmapped state. Once the page is faulted in, Gramine transitions it to the mapped state, where the page remains until the mapping is removed.

A key distinction compared to Baseline 2 is that the faulting instruction is no longer EACCEPT; this means that the enclave must provide an exception handler to issue the EACCEPT. Figure 4 illustrates the revised system flow. When the enclave accesses a virtual address that is not yet mapped, a page fault is raised and the kernel handles the fault. The SGX driver maps the page using EAUG and returns to untrusted, user-level runtime, which then re-enters the enclave in order to invoke the in-enclave exception handler. The exception handler consults Gramine's internal bookkeeping to confirm that the faulting address is in a virtual page that the application requested and not yet been accepted. Then the exception handler calls EACCEPT on the virtual page and marks the page as mapped.



(a)RBench

(b)YCSB workloads on Redis

(c)GCBench

(d)#PF, AEXs, EEXITs, and EENTERs for GCBench

Figure 5. Benchmarking of three application workloads: (a) RBench; (b) Redis; (c) GCBench, and (d) the numbers of page faults, AEXs, EEXITs, and EENTERs during the GCBench execution. Each set of results is collected on static allocation (sgx1), basic EDMM support (edmm), and EDMM with demand allocation (edmm+demand). Lower is better.

After accepting the new mapping, the enclave must again context switch back to the untrusted runtime and re-enter the enclave to resume execution. The underlying issue is the hardware needs to decrement a counter that tracks the number of register-saving regions used so far, and this can only happen when resuming an enclave execution using ERESUME. Newer versions of SGX (including upcoming firmware updates for some existing chips) has a feature called AEX-Notify [16] that will allow an enclave exception handler to resume normal execution without the additional context switches. We will adopt and evaluate the impact of AEX-Notify in future work.

3.6 Baseline Performance Evaluation

We start with measuring the performance of three application workloads in Gramine on the three baselines: static allocation, basic EDMM support, and EDMM with demand allocation. Figure 5 shows the enclave loading time and execution time of RBench and GCBench, as well as the throughputs of YCSB benchmarks on Redis, and the numbers of page faults, asynchronous exits (AEXs), enclave exits (EEXITs), and enclave enters (EENTERs) for GCBench. We omit these numbers for RBench and Redis because they show a similar trend as the numbers for GCBench.

The trends in Figure 5 for loading time and execution time are consistent: while basic EDMM support decreases the enclave loading time by 28–93%, it increases the execution time by 5% for RBench and 41% for GCBench versus static configuration. This performance degradation correlates with an increase in costly page faults and other enclave crossings.

Not only does adding basic EDMM support harm performance, demand allocation worsens the situation further. Overall, demand allocation increases the execution time by 10% for RBench and 58% for GCBench; worse, demand allocation does not reduce enclave loading time further compared to basic EDMM support. For Redis, demand allocation results in both gains and losses across different workloads. These variations are relatively small, due to the minimal demand allocation occurring during individual requests. Although demand allocation is an optimization in other contexts, in the case of EDMM, demand allocation requires an additional, costly enclave entrance and exit to accept the new mapping, which is not offset by any gains from delaying creation of the mapping.

> **Insight**: EDMM's enclave loading time gains can be quickly offset by expensive enclave crossings required for dynamic memory management, causing a net slowdown in application performance.

4 Removing EDMM Overheads

This section presents four optimizations that reduce the overheads demonstrated in the previous section. First, because it is cheaper to create static mappings at loading time than dynamic mappings, we show how a modest pre-allocation strategy can reduce overheads without over-provisioning space. Second, one can amortize the cost of dynamic mappings by batching one set of enclave entrace and exits across multiple `mmap` requests. Third, we observe that demand allocation often occurs on contiguous virtual pages, so we can proactively map neighboring pages in response to a demand fault. Finally, because unmapping is more expensive than mapping pages in an enclave, an asynchronous, lazy unmapping strategy can further reduce costs and create opportunities for mapping reuse. In order to understand the impact of each optimization, we evaluate the improvement over baseline Gramine (and the prior optimizations) in each subsection.

4.1 Optimization 1: Pre-allocation (+pre)

Given that it is faster to allocate memory at enclave launch than during runtime, the first optimization is simply to pre-allocate a reasonable starting size for the heap. Although the number of pages used by an application can be input-dependent, one can often reliably predict a minimum amount of expected dynamic memory usage, say based on the smallest input.

Our first optimization, named **pre-allocation** (denoted as +pre), allows the user to specify an initial page pool size to fully allocate during enclave launch. Gramine allocates an initial set of pages at launch, and after the application exhausts this allocation, Gramine switches to dynamic allocation. During the initial stage of the enclave execution, Gramine has a fixed memory footprint, mostly storing internal data structures of Gramine and loading application binaries. Pre-allocation prevents frequent enclave exits or page faults until the application is fully loaded, as well as servicing some memory allocation requests from application itself.

We note that pre-allocating too much memory may have downsides: first, the enclave potentially uses more memory than needed, and, with a sufficiently large initial page pool size, degenerates to the static configuration. Second, as our evaluation shows, pre-allocating memory increases enclave loading time. If the space is going to be used, pre-allocation is more efficient than dynamic allocation and this is a net win; if the space is unused, this is a needless performance cost. Worse, if the system is under memory pressure, needless pre-allocation can impact the performance of other running enclaves or even regular applications. We leave experiments on memory pressure for future work.

Evaluation. Figure 6 shows the impact of pre-allocation on our three application workloads: RBench, Redis, and GCBench. We experiment with different preallocation sizes: 64M, 128M, 256M, and 512M. We also test the optimization on two EDMM baselines: the basic EDMM support, and demand allocation. For RBench, pre-allocation with 512M brings down the cost of using EDMM to be on par with static allocation, yielding only a 4% slowdown in execution time but 3.5× faster enclave load time compared to static allocation. For GCBench, pre-allocation with 128M shows performance on par with static allocation regarding execution time, while also providing better enclave load time. Contrast this with baseline EDMM overheads of 58% with and 41% without demand paging. For Redis, pre-allocation with 256M improves almost all of the workloads over static allocation. Workload A has improvement up to 4%.

Fig. 6(d) presents enclave entrances and exits for GCBench, showing a reduction in enclave crossings that corresponds to the performance gains for pre-allocation.

> **Insight**: Pre-allocating pages trades an increase in loading time for a larger reduction in execution time, provided the pages are used. Pre-allocating unused pages lowers execution time compared to not mapping them.

4.2 Optimization 2: Batch Allocation (+batch)

A major source of runtime overhead in EDMM is the additional, synchronous context switch for the enclave to approve each change to a page mapping. Xing et al. [46] propose, but do not implement or evaluate, a solution that amortizes one single round trip into the kernel over a virtually contiguous set of pages. We call this technique **batch allocation** (denoted as +batch). We note that the

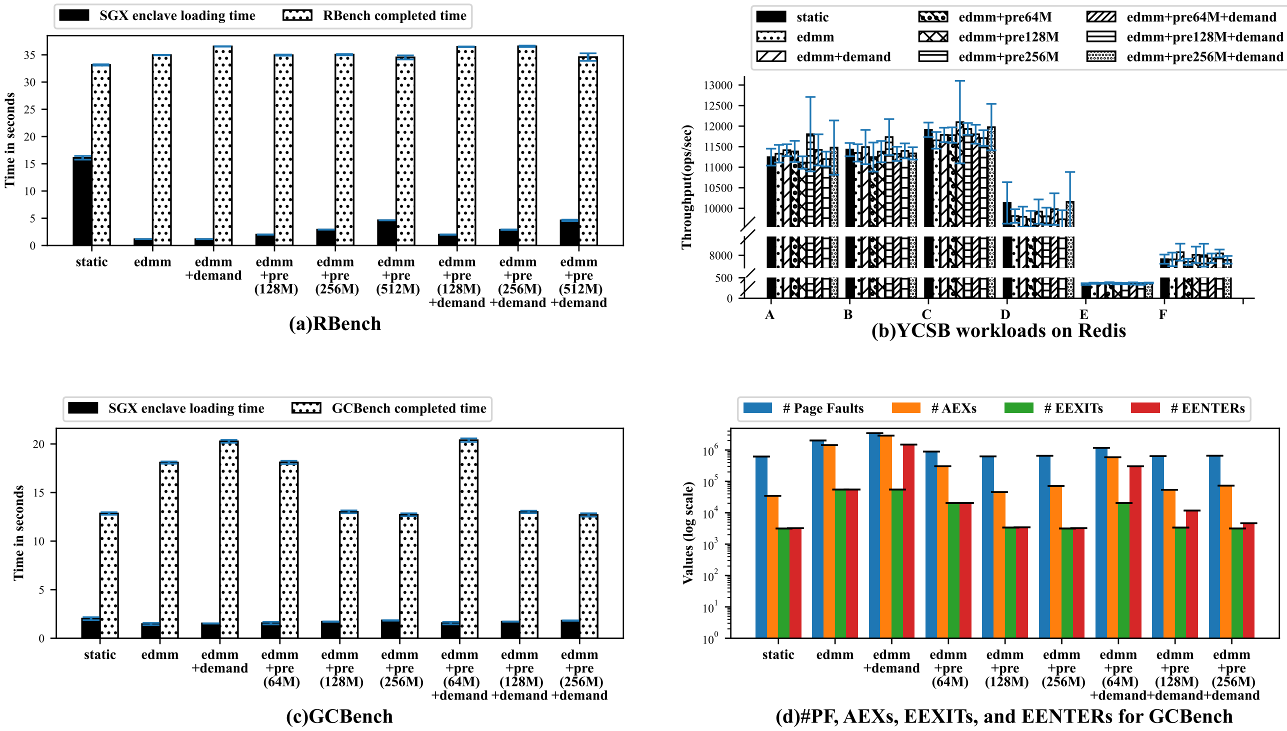


Figure 6. (**Pre-allocation**) Benchmarking of three application workloads: (a) RBench; (b) Redis; (c) GCBench, and (d) the numbers of page faults, AEXs, EEXITs, and EENTERs during the GCBench execution. Each set of results is collected on the three baselines (sgx1, edmm, and edmm+demand), EDMM with different pre-allocation sizes: 64M (+pre(64M)), 128M (+pre(128M)), 256M (+pre(256M)), and 512M (+pre(512M)), either with or without demand allocation (+demand).

technique of batch allocation defined in this paper is specifically an optimization to the basic EDMM support. A similar idea of "batching" can be applied to demand allocation, which we will discuss in §4.3.

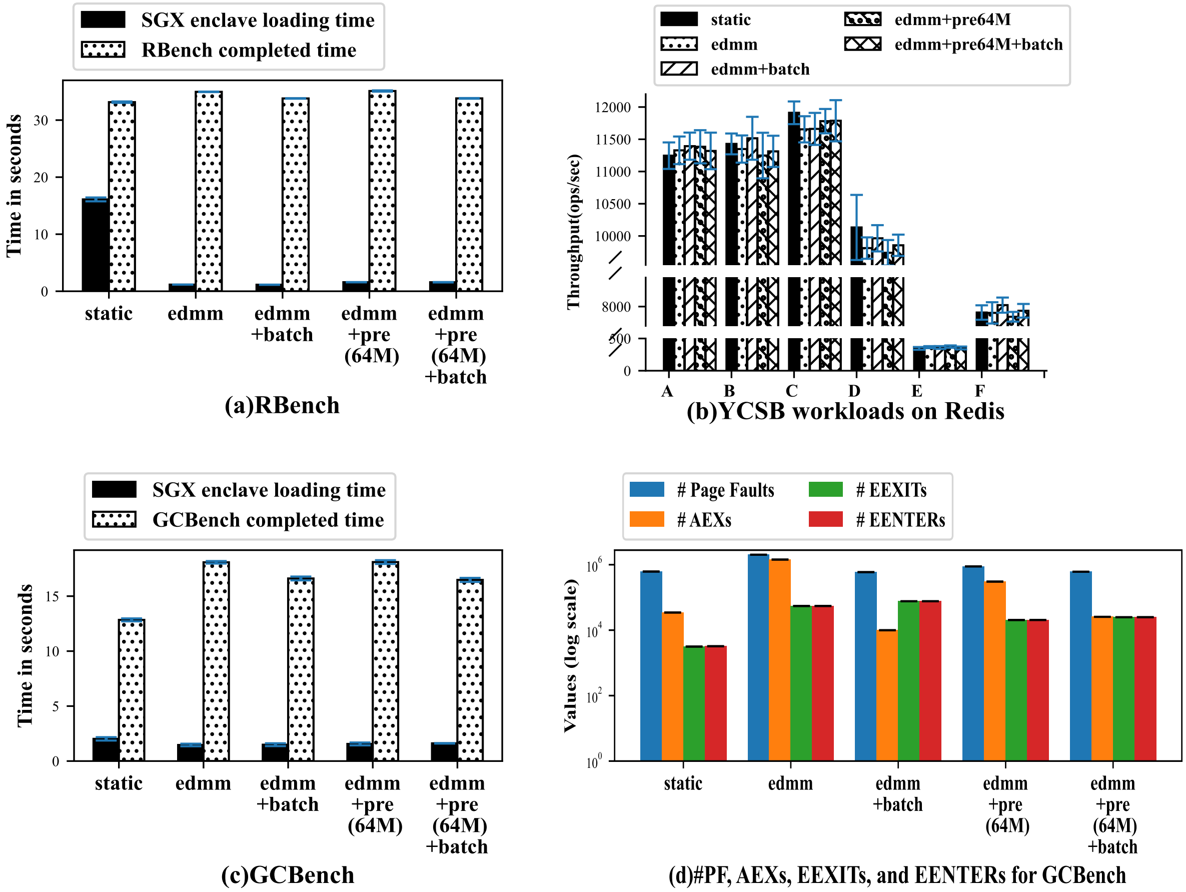
Recall that, in basic EDMM support without demand paging, when an application issues a multiple-page `mmap`, Gramine allocates a region from the page pool. Gramine then issues an `EACCEPT` instruction on each virtual page in the newly mapped range, which triggers a demand fault on each page. This causes the kernel to issue the `EAUG` instruction, which then allows the `EACCEPT` to succeed. Put differently, this strategy incurs one enclave crossing and one system call per page mapped.

Rather than demand fault these pages one at a time, the key intuition of batch allocation is to amortize one enclave and kernel crossing over an `mmap`-ed range. Before Gramine's `mmap` implementation issues an `EACCEPT` instruction on each page, it first issues an `ocall` to the untrusted runtime, which in turn issues a special `madvise` system call to the SGX driver on behalf of the enclave. This `madvise` call tells the kernel the location and size of the mapping change, causing the driver to issue a series of `EAUG` instructions over the requested virtual address range. Upon return to the enclave, the enclave can then issue a series of `EACCEPT` instructions over the same range without further page faults.

This `madvise` feature is not implemented in the Linux SGX driver yet. We are working with the Intel team and plan to upstream a patch to Linux in the future.

In total, the batch optimization lowers the costs of dynamically modifying page mappings from one enclave-kernel round trip per page to one per contiguous memory region. From the application's perspective, there is no demand allocation in either the baseline or with this optimization; after an `mmap`, the returned region of virtual address space is fully mapped and usable.

Evaluation. Fig. 7 shows the impact of batch allocation on our three application workloads over the basic EDMM support with the pre-allocation optimization. For simplicity, we pick the smallest pre-allocation size tested so far (64M), based on the diminishing returns of increasing this size. Adding batch allocation yields throughput improvement on most of the YCSB Redis workloads. It also brings down the execution time of RBench and GCBench. With 64M pre-allocation, batch



(a)RBench

(b)YCSB workloads on Redis

(c)GCBench

(d)#PF, AEXs, EEXITs, and EENTERs for GCBench

Figure 7. (**Batch Allocation**) Benchmarking of three application workloads: (a) RBench; (b) Redis; (c) GCBench, and (d) the numbers of page faults, AEXs, EEXITs, and EENTERs during the GCBench execution. Each set of results is collected on the three baselines (sgx1, edmm, and edmm+demand), EDMM with only batch allocation (+batch), EDMM with 64M pre-allocation (+pre(64M)), either with or without batch allocation (+batch).

allocation further reduces the overhead of EDMM on GCBench from 41% in the baseline to 28%.

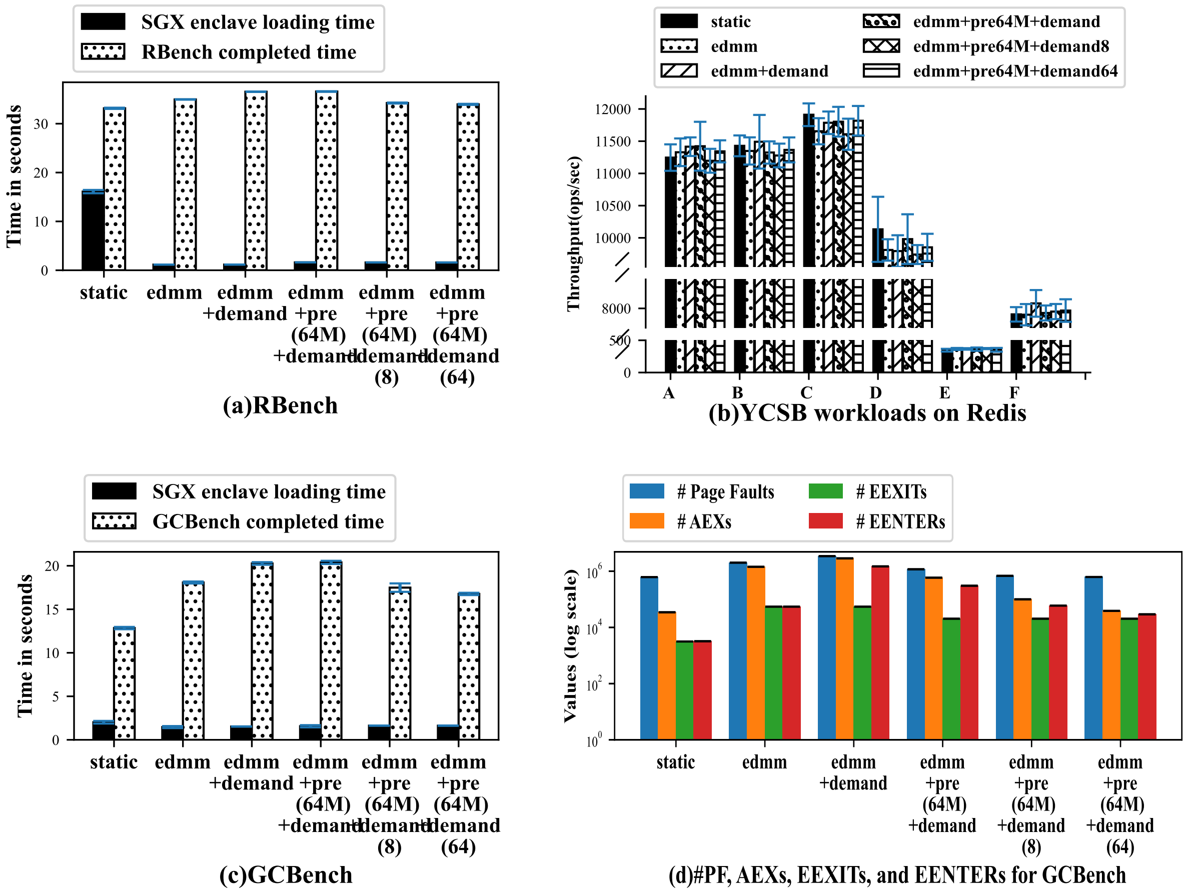
Insight: Batch allocation can further lower the overheads of mapping changes for memory that cannot be pre-allocated.

4.3 Optimization 3: Contiguous Demand Allocation (+demand<N>)

Demand allocation is a common optimization because a typical application commonly `mmaps` more virtual memory than the application accesses. Thus, demand allocation can potentially lower enclave memory footprints. The downside we have already illustrated is that the cost of page faults is very high on SGX. However, a high cost can potentially be amortized over more mappings.

Contiguous Demand Allocation (denoted as +demand<N> amortizes the cost of a demand allocation fault over as many as N neighboring pages. Figure 9 shows the optimized system flow to implement contiguous demand allocation. Similar to demand allocation, the allocation process is triggered by a memory access to a virtual page that is not yet mapped by the kernel (the application has logically `mmap`-ed it in Gramine). As with baseline demand paging, the in-kernel page fault handler will call `EAUG` on the faulting virtual page. When the kernel returns to the exception handler of Gramine's untrusted runtime, it issues the same `madvise` system call as used in batched allocation to map the subsequent $N-1$ virtual pages. Then, the untrusted runtime re-enters the enclave, which will iteratively call `EACCEPT` on N contiguous virtual pages starting with the faulting page. Finally, to restore enclave state, it exits and resumes back to the original execution inside the enclave.

Evaluation. Figure 8 shows the impact of contiguous demand allocation on our three application workloads over



(a)RBench

(b)YCSB workloads on Redis

(c)GCBench

(d)#PF, AEXs, EEXITs, and EENTERs for GCBench

Figure 8. (**Contiguous Demand Allocation**) Benchmarking of three application workloads: (a) RBench; (b) Redis; (c) GCBench, and (d) the numbers of page faults, AEXs, EEXITs, and EENTERs during the GCBench execution. Each set of results is collected on the three baselines (sgx1, edmm, and edmm+demand), EDMM with 64M pre-allocation (+pre(64M)) and **three different demand allocation sizes**:

demand allocation, with 64M pre-allocation. We tested with three demand allocation sizes: 1 page (baseline), 8 pages, and 64 pages. Unsurprisingly, in the absence of memory pressure, increasing contiguous demand allocation size lowers overheads. In the case of RBench, demand allocating 64 pages at once effectively offsets the cost of demand allocation to only 2% compared to baseline EDMM support—gaining the potential space savings of demand allocation without the high runtime cost. In the case of Redis, contiguous demand allocation with 64 pages offsets not only the costs of demand allocation, but EDMM in general—bringing throughput up to match static. For GCBench, contiguous demand allocation with 8 pages and 64 pages further reduces the overhead of demand allocation by 5% and 10%, compared to baseline EDMM. The root cause of this improvement can be seen in Figure 8(d), in which both the number of page faults and enclave entrance and exit are reduced with contiguous demand allocation.

> **Insight**: Demand paging must be at a larger granularity than one page to amortize higher mapping costs on SGX.

4.4 Optimization 4: Lazy Free (+lf)

Based on the observation that unmapping enclave memory is more expensive than allocating it, the lazy free optimization caches some number of freed pages in a pool to serve subsequent allocation requests. We add a manifest configuration option where the user can set a maximum amount of freed memory to hold in reserve for future allocations. We also add bookkeeping for the page pool to track the state of free pages (allocated, unmapped, or cached). When an application issues an `munmap` to Gramine, if the cached page count is below

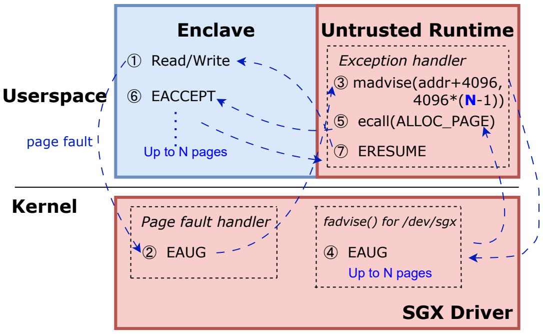


Figure 9. The optimized system flow of allocating N new enclave virtual memory mappings (N is a parameter set in the enclave's configuration) via demand allocation, involving seven context switches among the untrusted kernel, Gramine's untrusted runtime, and the enclave. Trusted components are in blue, untrusted in pink.

the threshold, these pages change state from allocated to cached. When the cached page count goes above the threshold, pages are unmapped accordingly.

When an application issues an `mmap` to Gramine, Gramine first checks for a cached page region of an appropriate size; if the allocation can be satisfied, cached pages are used. Otherwise, new pages are demand faulted in, as in prior subsections.

Lazy free does risk exacerbating internal fragmentation of the virtual address space over time. However, the virtual address space of an enclave can be large and sparse. In the case where there are enough total free physical pages, but an allocation by the OS fails, one can simply free all of the cached pages and try again.

Another important caveat to this optimization is that it is not strictly POSIX compliant. An application that attempts to access an `munmap`-ed region should page fault. We believe that applications deliberately faulting on an unmapped page are rare in practice. For these rare cases, this optimization should be disabled.

Evaluation. Figure 10 shows the impact of lazy free on our three application workloads with the optimizations we explored so far. In particular, we test the strategy of lazy free on both batch allocation and contiguous demand allocation (8 pages at each page fault), and compare with the results in which lazy free is disabled. We also choose two different de-allocation threshold, 5% and 15%, to control the *eagerness* of lazy free. We also tested with 64 pages at each page fault instead of 8, but did not find a significant difference.

For GCBench, with a very modest threshold of 5%, the execution time overhead drops from 28% to a 1.8% gain with batch allocation, and drops from 36% to a 2.8% gain with 8-page contiguous demand allocation. Increasing the threshold to 15% further improves the GCBench execution time performance on par with static allocation. For RBench, increasing the threshold to 15% even yields improved performance over static allocation in both cases with batch allocation and contiguous demand allocation. For Redis, lazy free also yields improved performance on YCSB workload A and F over static allocation.

Figure 10(d) shows the reduction of enclave exits and entrances during the execution of GCBench with lazy free, which contributes to the performance improvement.

Insight: For applications that aggressively allocate and free virtual pages, lazy free further eliminates the remaining overheads of EDMM.

5 Related Work

The memory management of hardware enclaves has been long explored by previous works. As one of the earliest work, Eleos [30] introduces the use of Secure User-Managed Virtual Memory (SUVM) to swap the virtual pages in and out of an enclave, without relying the untrusted SGX driver to swap the pages. Eleos shows significant performance benefits (up to 2.3× throughput), by eliminating the cost of context switching and the subsequent cache pollution. A key requirement for Eleos is that the application be compiled with an indirection mechanism for pointers, wherein the trusted runtime system could intercept dereferencing of pointers to objects that are swapped out. CoSMIX [31] extends the idea of using a compilation pass to indirect pointers with a software-managed oblivious RAM (ORAM), to hide page access patterns from the untrusted OS and defend against controlled-channel attacks [47] and other side-channel attacks [11, 12, 44, 45]. As a follow-up work, Autarky [29] explore self-paging in enclaves as a way to eliminate controlled-channel attacks, as an extension to SGX [47]. They introduce a page cluster abstraction, which must be swapped as a group, and a cooperative page management framework.

VAULT [39] introduces architectural changes to reduce the cost of page swapping in and out of the Enclave Page Cache (EPC). VAULT replaces Intel's SGX Integrity Tree, an in-memory data structure for authenticating the values and versions of enclave memory, with a Variable Arity Unified Tree, significantly reducing the memory accesses necessary to verify a virtual memory block.

Civet [42] explores enclave-aware garbage collection in the OpenJDK runtime. Specifically, Civet introduces a three-generational garbage collection design, which corresponds to the primary performance regimes in an enclave: one generation fits in last-level cache (LLC), another generation fits in EPC, and the even larger, oldest generation.

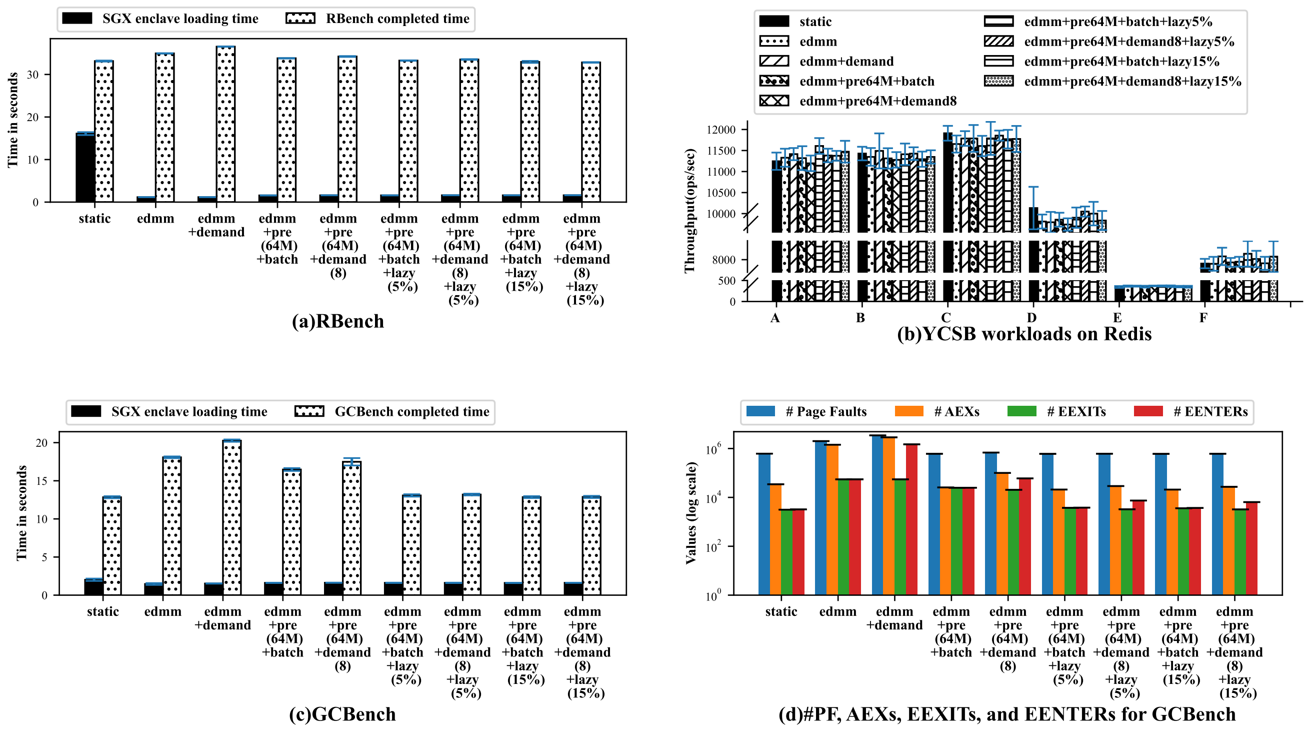


Figure 10. (**Lazy Free**) Benchmarking of three application workloads: (a) RBench; (b) Redis; (c) GCBench, and (d) the numbers of page faults, AEXs, EEXITs, and EENTERs during the GCBench execution. Each set of results is collected on the three baselines (sgx1, edmm, and edmm+demand), EDMM with 64M pre-allocation (+pre(64M)) and either batch (+batch) or demand allocation (+demand), either with or without **lazy free with 15% threshold** (+lf(15%)).

Liu et al. [24] show how swapping overheads in SGX can be alleviated by preloading pages likely to be accessed in the future into the Enclave Page Cache, either based on source analysis or observed behavior in prior execution.

RISC-V Keystone [22] uses the Linux memory allocator to assign memory to its enclaves. Page faults caused during memory management incur high overheads, dampening the performance when stressed. Ashman [23] performs dynamic memory management on RISC-V by moving the mappings of the enclaves in order to avoid fragmentation, such that allocated and free memory are contiguous.

Occlum [38] is another library OS akin to Gramine, albeit written in Rust, which utilizes Intel Memory Protection eXtensions (MPX) to isolate memory regions. Without EDMM, it has a better overall average performance than Gramine without EDMM. As EDMM degrades the performance of SGX-based enclave, we show that our optimizations with Gramine-SGX reduce these overheads over baseline EDMM. Occlum has an up-to-date EDMM implementation, and we leave a comparison between Occlum and Gramine with EDMM as future work.

Elasticlave [49] is an extension to Keystone providing first-class support for sharing pages between enclaves. Unlike SGX, Elasticlave allows enclaves to selectively share memory to other enclaves, reducing overhead, as well as natively guaranteeing atomicity.

6 Conclusion

In this paper, we have examined the performance impact of dynamic memory management strategies on enclave applications, including the effect on garbage collection and server workloads. The results of our experiments show that, despite the reduction of enclave start time, naïve implementations of dynamic memory allocation, either at user-level mapping or at first access, cause significant slowdowns to runtime. We have demonstrated that by optimizing the system flow of demand allocation and lazy freeing, the runtime overhead of dynamic memory management can be significantly reduced to being comparable to workloads under static memory allocation.

Acknowledgments

This work was supported in part by NSF grant CNS-2244937, CNS-2148374, Intel, and the Confidential Computing Consortium (CCC). Porter also has a significant financial interest in Fortanix and Amazon.

References

[1] Alibaba. 2020. Alibaba Cloud Released Industry's First Trusted and Virtualized Instance with Support for SGX 2.0 and TPM. https://www.alibabacloud.com/blog/alibaba-cloud-released-industrys-first-trusted-and-virtualized-instance-with-support-for-sgx-2-0-and-tpm_596821. (October 2020).

[2] Alibaba. 2023. Alibaba Cloud, Elastic Compute Services, Instance Type Families, Overview. https://www.alibabacloud.com/help/doc-detail/60576.htm?spm=a2c63.p38356.b99.95.32ae1160CQKT0I. (August 2023).

[3] AMD. [n. d.]. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. ([n. d.]).

[4] AMD. 2020. White PaperAMD SEV-SNP: Strengthening VM Isolationwith Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. (2020).

[5] ARM. [n. d.]. ARM Secure IP. https://developer.arm.com/ip-products/security-ip. ([n. d.]).

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 689–703. http://dl.acm.org/citation.cfm?id=3026877.3026930

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 267–283. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[8] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/2988336.2988350

[9] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 169–183. https://doi.org/10.14778/3364324.3364331

[10] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. IRON: Functional Encryption Using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 765–782. https://doi.org/10.1145/3133956.3134106

[11] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[12] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 299–312. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

[13] IBM. 2020. IBM Cloud Data Shield Now Generally Available. https://www.ibm.com/blog/announcement/ibm-cloud-data-shield-now-generally-available/. (April 2020).

[14] IBM. 2023. Provisioning a bare metal server with Intel® Software Guard Extension architecture. https://cloud.ibm.com/docs/bare-metal?topic=bare-metal-bm-server-provision-sgx. (January 2023).

[15] Intel. 2021. Intel&Reg; Hardware Shield–Intel&Reg; Total Memory Encryption. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf. (2021).

[16] Intel. 2022. Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function. https://cdrdv2.intel.com/v1/dl/getContent/736463?explicitVersion=true. (2022).

[17] Intel. 2022. Intel Trust Domain Extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419. (2022).

[18] Kyungtae Kim, Chung Hwan Kim, Junghwan "John" Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. 2020. Vessels: Efficient and Scalable Deep Learning Prediction on Trusted Processors. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 462–476. https://doi.org/10.1145/3419111.3421282

[19] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2017. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 145–161. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kim-seongmin

[20] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 15 pages. https://doi.org/10.1145/3302424.3303951

[21] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N. Asokan, Andrew Simpson, and Robin Ankele. 2016. Exploring the Use of Intel SGX for Secure Many-Party Applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3007788.3007793

[22] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages. https://doi.org/10.1145/3342195.3387532

[23] Haonan Li, Weijie Huang, Mingde Ren, Hongyi Lu, Zhenyu Ning, Heming Cui, and Fengwei Zhang. 2022. A Novel Memory Management for RISC-V Enclaves. In *Proceedings of the 10th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '21)*. Association

for Computing Machinery, New York, NY, USA, Article 3, 9 pages. https://doi.org/10.1145/3505253.3505257

[24] Ximing Liu, Wenwen Wang, Lizhi Wang, Xiaoli Gong, Ziyi Zhao, and Pen-Chung Yew. 2020. Regaining Lost Seconds: Efficient Page Preloading for SGX Enclaves. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 326–340. https://doi.org/10.1145/3423211.3425673

[25] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP 2016)*. ACM, New York, NY, USA, Article 10, 9 pages. https://doi.org/10.1145/2948618.2954331

[26] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP 13)*. ACM. http://doi.acm.org/10.1145/2487726.2488368

[27] Microsoft. 2023. DCsv3 and DCdsv3-series. https://learn.microsoft.com/en-us/azure/virtual-machines/dcv3-series. (January 2023).

[28] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Popa. 2018. Oblix: An Efficient Oblivious Search Index. 279–296. https://doi.org/10.1109/SP.2018.00045

[29] Meni Orenbach, Andrew Baumann, and Mark Silberstein. 2020. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 16 pages. https://doi.org/10.1145/3342195.3387541

[30] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys 17)*. 238–253.

[31] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. 2019. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 555–570. https://www.usenix.org/conference/atc19/presentation/orenbach

[32] Samuele Pedroni, Hans Boehm, John Ellis, and Pete Kovac. 2018. GCBench. https://github.com/mozillazg/pypy/blob/40795dcad7e1b0be53d2f95a94f0278086d2d448/rpython/translator/goal/gcbench.py. (2018).

[33] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. 2016. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/2988336.2988346

[34] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. Safebricks: Shielding Network Functions in the Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 201–216.

[35] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB – A Secure Database using SGX. In *Oakland*. IEEE. https://www.microsoft.com/en-us/research/publication/enclavedb-a-secure-database-using-sgx/

[36] Redis. 2023. Redis benchmark. https://redis.io/docs/management/optimization/benchmarks/. (2023).

[37] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy (S&P 15)*. IEEE, 38–54.

[38] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA.

[39] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 665–678. https://doi.org/10.1145/3173162.3177155

[40] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *EuroSys*.

[41] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[42] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *USENIX Security*.

[43] Simon Urbanek and Philippe Grosjean. 2008. R Benchmark. https://mac.r-project.org/benchmarks/. (2008).

[44] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[45] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1041–1056.

[46] Bin (Cedric) Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel® Software Guard Extensions (Intel® SGX) Software Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP 2016)*. ACM, New York, NY, USA.

[47] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*.

[48] Yahoo. 2019. Yahoo! Cloud Serving Benchmark. https://ycsb.site. (2019).

[49] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. 2022. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium*

(USENIX Security 22). 4111–4128.

[50] ZDNet. 2020. Cloud security: Microsoft Azure's SGX VMs hit GA, Google's Shielded VM is now default. https://www.zdnet.com/article/cloud-security-microsoft-azures-sgx-vms-hit-ga-googles-shielded-vm-is-now-default/. (April 2020).