# Regaining Lost Seconds: Efficient Page Preloading for SGX Enclaves

Ximing Liu
liuximing@mail.nankai.edu.cn
Nankai University
Tianjin, China

Wenwen Wang
wenwen@cs.uga.edu
University of Georgia
Athens, GA, USA

Lizhi Wang
nku_liz@mail.nankai.edu.cn
Nankai University
Tianjin, China

Xiaoli Gong*
gongxiaoli@nankai.edu.cn
Nankai University
Tianjin, China

Ziyi Zhao
troppingz@gmail.com
Nankai University
Tianjin, China

Pen-Chung Yew
yew@umn.edu
University of Minnesota
Minneapolis, MN, USA

## Abstract

Intel SGX is already here, with a strong emphasis on security and privacy. However, it is not free. Studies have shown that it incurs a significant performance overhead to take advantage of the security and privacy enhancement offered by SGX. In particular, it only provides limited physical memory for applications to use SGX. As a result, page faults can be frequently triggered during program execution, especially for memory-intensive applications with a large memory footprint. Therefore, it is imperative to look into possible optimization opportunities to enhance the efficiency of SGX.

To this end, this paper proposes to leverage memory page preloading techniques to mitigate such a problem. More specifically, we propose two effective schemes to preload memory pages before they are accessed. This way, the number of page faults can be significantly reduced. To demonstrate the effectiveness of the proposed schemes, we have implemented them in a prototype using LLVM and an untrusted operating system.

Experimental results on benchmarks from SPEC CPU2017 and a micro-benchmark program show that, on average, these two mechanisms can achieve 11.4% and 7.0% performance improvement with a maximum performance improvement of 18.6% and 9.0%, respectively. The two mechanisms are also evaluated when they are deployed together. The combined approach can achieve an improvement of 7.1% on some real-world applications such as SIFT and MSER.

*Corresponding author.

*Middleware '20, December 7–11, 2020, Delft, Netherlands*
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425673

***CCS Concepts:*** • **Software and its engineering → Memory management**; • **Security and privacy** → *Hardware security implementation*.

***Keywords:*** Intel SGX, Memory Preloading, Page Fault, Source Code Instrument

**ACM Reference Format:**
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 *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425673

## 1 Introduction

Protecting a user application against various security attacks is critical, especially for privacy-sensitive applications that store and manage user accounts and passwords. A previous study shows that there can be as many as 50 attack scenarios from various aspects of a software system [16]. Furthermore, an increasingly large number of applications and services are distributed and deployed on cloud computing platforms, which poses a greater challenge in preserving the security and privacy given that cloud environments are shared by a variety of applications and users.

To address this problem, trusted execution environments (TEEs) have been proposed by processor vendors [2, 3, 17, 18]. In general, a TEE is an isolated execution environment that aims to protect the code and data with a strong guarantee of confidence and integrity. Compared to a regular execution environment, a TEE offers a smaller exposed area for attacks. TEEs have been embedded into mainstream processors and are available via hardware extensions. For example, Intel Software Guard eXtensions (SGX) [18] have been integrated into Intel's processors since 2015.

SGX provides a set of security-oriented instructions that allow a user application to create a private memory region, called *enclave*, which is protected and exclusive to the application running inside the enclave. The data in an enclave cannot be accessed from outside of the enclave, including the privileged operating system and hypervisor, i.e., the threat model of SGX assumes only the inside of the enclave is trusted. Given the strong and attractive security guarantees, SGX has been adopted by many applications, e.g., secure Linux containers [4] and cryptography algorithms in wolfSSL [43], and has been widely deployed in commercial cloud platforms, e.g., Microsoft Azure [25] and IBM Cloud [21].

However, it is not free to take advantage of the security guarantees provided by SGX. Existing studies have shown that an application running inside an SGX enclave can be more than 10X slower than the same application running outside of SGX [42]. In particular, we observed a performance degradation of about 46X when porting a simple program with sequential accesses of 1GB data into an SGX enclave. Further investigation showed that the main reason for such a high overhead was due to the limited physical memory provided by SGX.

More specifically, SGX reserves a limited contiguous physical memory region for enclaves, called *Enclave Page Cache* (EPC), and is managed at the page level. In its current implementation it has 128MB. Due to the required enclave metadata, it is further reduced to around 96MB for user applications. A virtual memory interface is provided to users, and a paging mechanism is employed to support applications with larger memory requirements. Hence, an application with a memory footprint larger than the size of EPC can result in a significant number of page faults, and each page fault typically takes 60,000 ~ 64,000 clock cycles.

Therefore, optimizing and adapting applications to the limited EPC resource is crucial to the performance of the applications. To this end, previous research work has proposed user-level page management schemes to reduce the page fault overhead [26, 27]. In those approaches, a user-level software runtime is added to instrument (manually[26] or automatically with the help of the compiler[27]) the memory accesses, and swap the EPC pages in and out of the enclave without the help of OS. Although this approach achieves a substantial performance improvement, it has several fundamental limitations.

First, these approaches cannot maintain the same security guarantees ensured by the enclave. This is because they use a user-level software runtime to bypass the secure SGX instructions in order to reduce their high overhead. Those SGX instructions are typically used by the operating system kernels to update the EPC pages and exchange information without losing confidentiality, atomicity and freshness. Second, the integration of the software runtime will enlarge the trusted computing base (TCB) that needs to be kept in an enclave. This poses additional security risks. Third, keeping the runtime in enclaves all the time further exacerbates the pressure on EPC.

To overcome the above limitations, this paper proposes to leverage *memory page preloading* techniques to mitigate the performance overhead incurred by page faults in enclaves. Specifically, two effective page preloading schemes, targeting different application scenarios, are designed to load memory pages in advance during the execution of an application in enclaves. This way, the number of page faults in enclaves can be reduced.

More specifically, the first scheme works outside of an enclave at runtime with the support of the operating system. The page fault history is collected by the OS to predict the memory access behavior. The predicted pages are then loaded in advance into the EPC without the knowledge of the application. The second scheme assumes the source code of an application is available. An off-line program profiling is done to analyze and predict the application's memory behavior. The source code is then instrumented with page preloading instructions by the compiler based on the profiled data. These two preloading schemes can work individually or collaboratively as deemed appropriate.

The page preloading techniques proposed in this paper try to address the aforementioned limitations in prior approaches. First, the preloading schemes do not change the work flow of the page fault handler in enclaves, i.e., the SGX hardware instructions are still used in the page fault handler. Hence, the security offered by SGX instructions is preserved. Second, the TCB of an application is only increased slightly due to the instrumented code. The binary can remain unchanged using the first scheme if the source code is not available. Meanwhile, other than the small amount of instrumented code added in the second scheme, both schemes introduce minimal pressure on the limited EPC resource, as the major part of the code and data structures used in the framework are kept outside of enclaves.

We have implemented the proposed page preloading schemes in a prototype in an untrusted Linux operating system with the help of the LLVM compiler [22]. To evaluate the effectiveness of the preloading mechanisms, we use micro-benchmark program, the SPEC CPU2017 benchmark suite and two real-world applications. Experimental results show that, on average, the two mechanisms can achieve 11.4% and 9.0% performance improvements, respectively, and 7.1% when combined together. Furthermore, the performance overhead introduced by the preloading mechanisms is negligible.

In summary, this paper makes the following contributions:

- We propose two page preloading schemes to mitigate the performance overhead introduced by adopting Intel SGX.
- We implement a prototype to demonstrate the effectiveness of the proposed schemes. The prototype is implemented based on the SGX driver provided by Intel and the open source LLVM compiler. We also address several critical implementation issues.
- We conduct a comprehensive evaluation on the prototype, using applications from the SPEC CPU2017

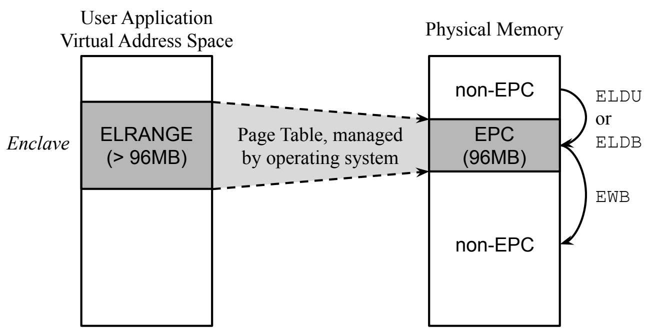


**Figure 1.** An enclave with a large ELRANGE is supported through the EPC paging mechanism in the operating system.

benchmark suite and some real-world applications. Experimental results show that the proposed schemes achieve promising performance improvement with reasonable runtime overhead.

The rest of this paper is organized as follows. Section 2 provides the background of Intel SGX and motivates this work. Section 3 describes the design details of the two page preloading techniques for SGX enclaves. Section 4 presents the implementation details of the prototype. Section 5 shows the evaluation results and our insights. Section 6 discusses related work, and Section 7 concludes the paper.

## 2 Background and Motivation

This section provides the background of Intel SGX necessary to understand the rest of the paper. More details on Intel SGX can be found in [11].

As mentioned earlier, Intel SGX is supported through hardware extensions to existing instruction set architectures. In order to support various user applications, the SGX hardware, including EPC, is managed by the operating system, even if it is not trusted. Thus, the majority of SGX instructions are designed to be executed in the privileged mode. For instance, to create an enclave, an application needs to invoke an `ioctl()` system call, which then executes the privileged SGX instruction `ECREATE` to establish the enclave.

It is worth noting that even though there is only limited *physical* EPC memory space for user applications, a user application is allowed to create an enclave in its *virtual* address space that is as large as necessary, which is called the *enclave linear address range* (ELRANGE). This is supported by the *EPC paging* mechanism in the untrusted operating system as shown in Figure 1.

The EPC paging mechanism is responsible for swapping memory pages between EPC and non-EPC physical memory regionss. Each time the application accesses a memory page that is not present in the EPC physical memory, a page fault is triggered. The page fault forces the application to exit the enclave to preserve the confidentiality of the enclave because the page fault is to be serviced by the untrusted operating system. This kind of exit is typically called an *asynchronous enclave exit* (AEX).

After that, the page fault is processed by the operating system. It tries to find an available EPC page first. If no EPC page is available, the operating system will evict a selected EPC page to non-EPC physical memory. This is realized by executing the `EWB` instruction. The operating system swaps in the faulted memory page from the non-EPC physical memory using the `ELDB/ELDU` instructions, and marks the page as available in the Page Table. Finally, the application continues its execution after the `ERESUME` instruction.

Existing studies show that it takes about 10,000, 44,000, and 10,000 clock cycles to complete the above-mentioned three steps: `AEX`, `ELDB/ELDU`, and `ERESUME`, respectively [42] after the latest security related micro-code update [32]. Hence, it incurs a total of around 60,000 ~ 64,000 cycles to handle an enclave page fault. Compared to handling a page fault outside of an enclave, which takes around 2,000 cycles [12], this poses a significant performance overhead, especially for memory intensive applications with large memory footprints. Therefore, it is imperative to mitigate such high performance overhead for better efficiency using SGX enclaves.

## 3 Preloading Pages for SGX Enclaves

In general, there are three key principles to follow when we design optimization schemes for SGX enclaves:

- **Preserving security.** Given that the SGX hardware is provided for security and privacy purposes, it is obvious that an optimization scheme should not break the security guarantees offered by SGX.
- **Maintaining a small TCB.** The threat model of SGX precludes all potential untrusted sources from entering a secure enclave, which include the native operating system and hypervisor, and thus leaves a sufficiently small TCB. Therefore, an optimization scheme should limit the increase of TCB as much as possible.
- **Providing good usability.** It is impractical to manually apply an optimization scheme as it will be labor intensive and error prone. Therefore, when integrating an optimization scheme with SGX, it should be either done by a compiler [27] or encapsulated in a framework [39].

The page preloading schemes proposed in this paper conform to all of these design principles. More specifically, they are carefully designed to mitigate the performance penalty introduced by the limited EPC resource in different application scenarios of SGX enclaves. The proposed schemes aim to preload EPC pages before they are actually accessed in the enclave. This way, the number of page faults in the enclave can be reduced. Next, we describe the details of the proposed page preloading schemes.

### 3.1 DFP: Dynamic Fault History-Based Preloading

The first preloading scheme is called *dynamic page fault history-based preloading* scheme, or *DFP* for short. Inspired by the observation that the page faults triggered in an enclave need to be handled by the operating system, it is natural to capture the history of the faulted pages at runtime in the operating system. Based on the page accessing history, we can dynamically predict which EPC pages will be accessed in the near future and preload those pages that are not in EPC yet. This prediction and preloading process happens outside of the enclave, i.e., in the untrusted operating system, without modifying any enclave source/binary code. Thus, it preserves the security guarantee of the enclave as required, which also means it has no effect on the TCB size.

Figure 2 illustrates the performance benefit that can be gained by DFP. As shown in the figure, there are three page faults in the original execution, triggered by accesses to Page2, Page3, and Page4, respectively. With the help of DFP, the page faults on Page3 and Page4 can be eliminated by preloading these two pages after the operating system handles the page fault on Page2. Here, the key is to accurately predict that Page3 and Page4 will be accessed after Page2 based on the access history to these pages. However, it is quite challenging to design an effective and accurate prediction scheme due to three major reasons.

Firstly, the information we can leverage is very limited. Compared to traditional data prefetching schemes for cache memories [6, 23, 30], which collect the *entire* memory access history to better extract the access patterns, we only have *partial* information of the access history here, i.e., faulted pages. This is mainly because SGX hides majority of memory accesses operations in enclaves for security and privacy reasons, and only page fault events can be trapped by the OS. Even for a page fault event, the bottom 12 bits of the faulted address are cleared by SGX, which means the history collected is at the page level.

Secondly, the performance penalty of a misprediction is significantly higher. This is because a misprediction will cause an EPC page to be preloaded into the EPC physical memory unnecessarily. The EPC page loading procedure is extremely slow compared to loading a cache line. Based on our measurements, the memory channel of EPC page loading is limited, i.e. it can only load one page at a time, and the page loading operation (`ELDU/ELDB`, which takes about 44,000 cycles) cannot be preempted when in progress, which make the mispredtion penalty even worse.

Lastly, different SGX applications can have completely different EPC page access patterns. It is extremely difficult to design a prediction scheme that is general enough to cover all types of applications [6]. In addition, memory protection mechanisms such as ORAM [37] may have different access patterns in different runs of the same program. Therefore, besides a carefully designed preloading strategy based on the dynamic analysis of the collected page fault history, a mechanism to abort incorrect preloading is also integrated into DFP to minimize the mispredition penalty.

Data prefetching has been well studied for decades, and it is widely deployed in modern processors for cache line prefetching [6]. Generally, the memory access patterns are tracked and then used to predict the data to be accessed. Data prefetching can be implemented in either hardware or software.

To further understand the access patterns at page level, we instrument the source code to gather the page number and time stamp of every memory instruction. A table is used to track the recently accessed pages. The trace data is then analyzed offline with curve fitting. Based on our observation, applications also exhibits some access patterns at the page level. For example, we have analyzed the SPEC CPU2017 benchmark suite, and some of the applications (e.g. *bwaves* and *lbm*) show an evidently sequential access pattern as in Figure 3 (a) and (c).

In the DFP scheme, we collect the history of faulted pages in each thread through the operating system, and analyze the page access patterns to predict future page accesses for preloading. However, there are applications that have highly irregular or near random page access patterns (e.g. *sjeng* in Figure 3(b)). In order to cut down the misprediction rate and its incurred penalty, an abort mechanism is placed in DFP to stop the page preloading if the number of preloaded pages that are not accessed exceeds a certain threshold. This way, DFP can achieve a reasonable performance target with an acceptable penalty due to misprediction.

### 3.2 SIP: Source-Level Instrumentation-Based Preloading

The benefit of DFP will be limited or even negative if the page access patterns of an application are highly unpredictable and the prediction accuracy is poor. In this case, we propose the second page-preloading scheme, called *source-level instruction-based preloading* (SIP) scheme.

In practice, when software developers are porting their software to SGX using Intel SGX software development kit (SDK) [19], it is usually done at the source code level. Hence, we are provided with an opportunity to perform source-level analyses and using instrumentation to implement software-base page preloading. This is opposed to DFP, which is done at the runtime. SIP is done at the compile time to instrument the source code and to initiate page preloading. We take advantage of the approach used in profile-guided optimization (PGO) schemes [7, 24, 28], which is used to improve the performance of an application using offline profiling information to aid the compiler to perform targeted optimizations. Irregular access patterns such as those shown in Figure 3(b) are difficult to capture accurately by DFP at runtime.

Figure 4 uses an example to illustrate the event sequence in the execution with a page fault (marked as *Baseline* in the

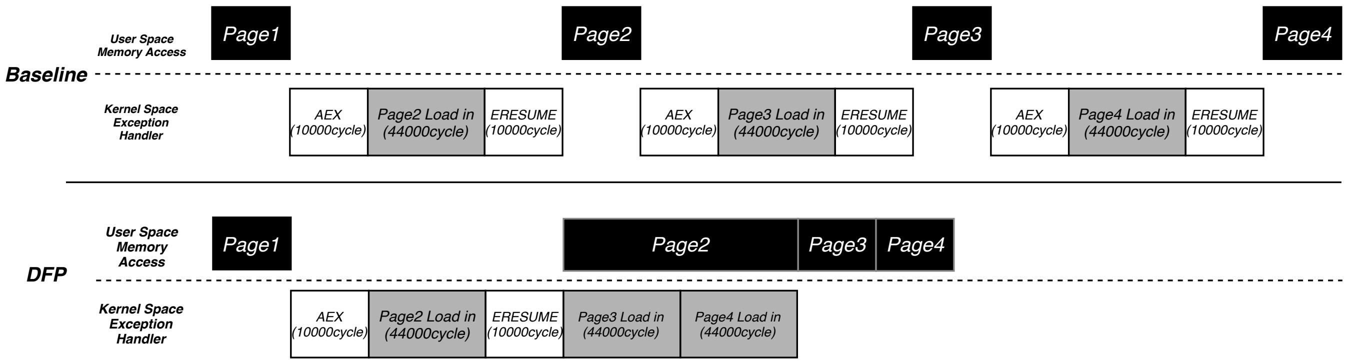

User Space Memory Access
Baseline
Kernel Space Exception Handler
Page1
Page2
Page3
Page4
AEX (10000cycle)
Page2 Load in (44000cycle)
ERESUME (10000cycle)
Page3 Load in (44000cycle)
Page4 Load in (44000cycle)
DFP

**Figure 2.** The time sequence of loading pages to EPC. In the *Baseline*, the time to finish loading all of the pages is $t_{access\_page\_1-4} + 3 * (t_{AEX} + t_{ERESUME}) + t_{load\_page2} + t_{load\_page3} + t_{load\_page4}$. For DFP, the time to finish loading all of the pages is $t_{access\_page\_1-4} + t_{AEX} + t_{load\_page2} + t_{ERESUME}$.

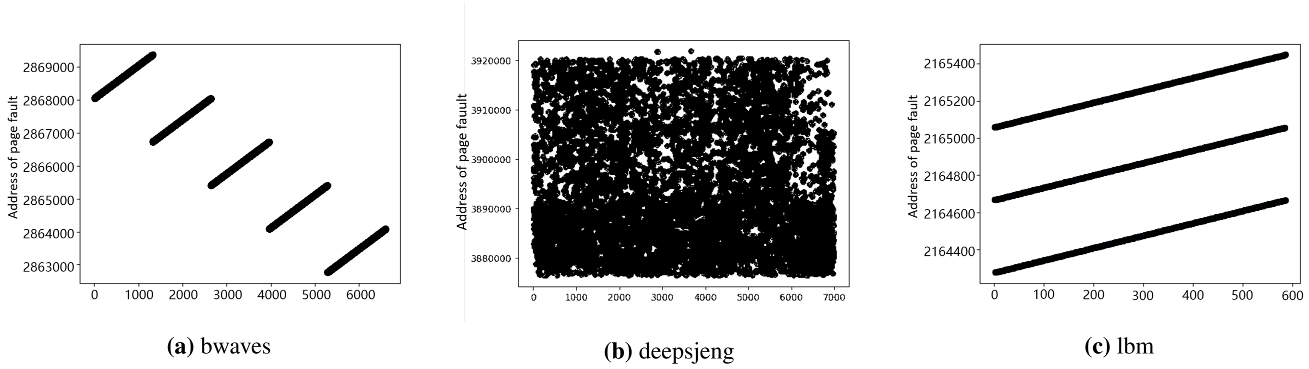


**Figure 3.** Representative memory access patterns of *bwaves, deepsjeng* and *lbm*

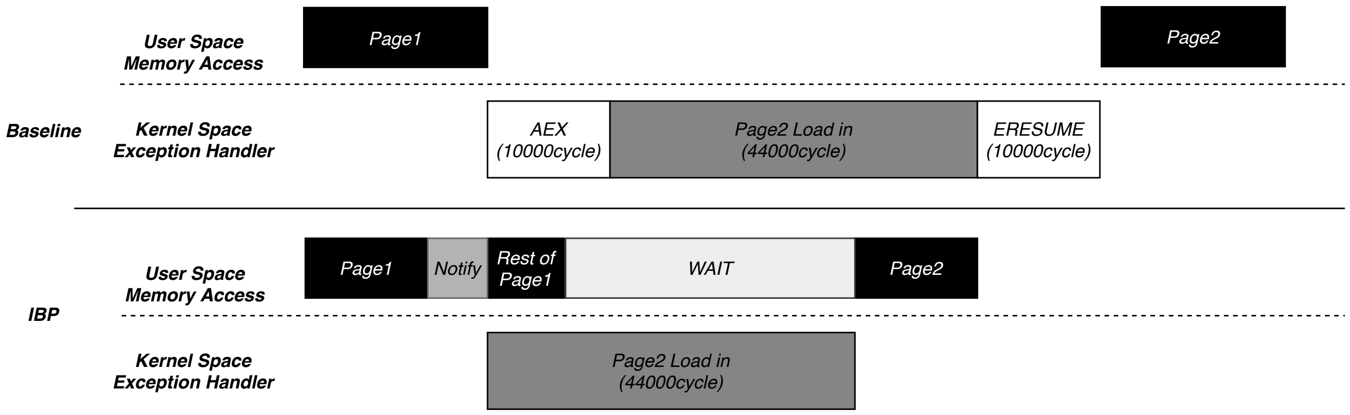


**Figure 4.** Memory access sequences of *Baseline* and instrumentation-based (SIP) scheme. In *Baseline*, the time of loading page2 is $t_{AEX}(10,000\ cycles) + t_{load\_page2}(44,000\ cycles) + t_{ERESUME}(10,000\ cycles)$. In SIP, the time of loading page2 is $t_{load\_page2} + t_{notification}$. The performance benefit is thus around $t_{AEX} + t_{ERESUME} - t_{notification}$.

figure), and the optimized execution sequence using SIP to avoid the page fault. As shown in the figure, there is a *Notify* operation instrumented in the application to inform the kernel to preload the marked page. With the help of the notification mechanism, two parts of the overhead incurred in handling a page fault, i.e., `AEX` and `ERESUME` mentioned in Section 2, can be removed in the optimized execution. In fact, it not only removes the overhead of an EPC page fault, but also allows the user application to stay in the enclave during the page loading. However, if the instrumented memory access does not trigger a page fault, i.e. it issues a preload request for an existed page, the instrumented *Notify* operation will introduce some overhead and offset some of the benefit of page preloading.

SIP can be done automatically by the source code analysis and code instrumentation in the compiler. If a memory access operation is predicted to trigger a page fault with a high probability, the compiler will insert a notification operation to preload the page. The more accurate the instrumentation is, the more performance gain can be obtained. However, in practice, it is very challenging to predict accurately whether a memory access will trigger a page fault or not through static analyses alone. To improve the prediction accuracy, we use a PGO (profile-guided optimization) approach to collect memory traces as mentioned earlier. The compiler can then use such profiled information to more accurately instrument the code.

In PGO, the program is first executed with some profiling input data, and the memory access trace is collected along with the source line numbers and timestamps. By analyzing the collected traces, the memory access patterns can be identified. For the source code with a lot of irregular accesses, which means it potentially can trigger a lot of page faults at runtime, a preload notification shown in Figure 4 is instrumented before a memory access.

Figure 4 also shows that it is possible to hide the latency of page preloading if the nofity operation can be issued early enough to allow the page preloading to overlap with other code execution. However, as mentioned earlier, page loading usually takes a long period of time (around 44,000 cycles). It is extremely difficult to find code regions that are large enough to overlap with such a long page loading time, i.e. it is very difficult to find a notification time far advance enough to offset the 44,000-cycle page loading time. Therefore, our SIP for irregular accesses is set to be *conservative*, and the instrumented code is inserted right before the memory access. The requested page number is sent to the OS kernel. The kernel will load the page, and the application will resume after the page is loaded. In this way, as shown in Figure 4, the overhead of *AEX* and *ERESUME* can be eliminated.

Figure 5 shows an example of how an application is statically instrumented using SIP. In this example, the access to *array[st]* and *result_map[key]* are found to trigger a lot of page faults through profiling. They are instrumented with preloading notifications. In a preloading notification, the status of the target page is checked with *BIT_MAP_CHECK*. If the requested page is not in EPC, a page loading request will be issued in *page_loadin_function*, and the function will return only after the EPC page loading is completed. When the application resumes its execution, the required page will have been loaded into the EPC, and the overhead of context switching that could have been triggered by a page fault is eliminated.

```
int solution(vector<int>& array, vector<int>& case){
    int s_left=0;
    int s_right=0;
    int flags[26];
    unorder_map<int, int> result_map;
    for(int i=0; i<26;i++)
        flags[i] = 0;
    for(int i=0; i<n; i++)
    {
        int tempsum = s_left+s_right*i;
        int st = tempsum + case[i];
        address = &array[st];
        if(BIT_MAP_CHECK == true)
            page_loadin_function(address);
        int key = array[st];
        address = &result_map[key];
        if(BIT_MAP_CHECK == true);
            page_loadin_function(address);
        result_map[key]++;
        s_left++;
        s_right++;
    }
}
```

**Figure 5.** Example code instrumented by SIP

It is still challenging to determine where to insert the preloading notification at compile time even with the help of profiling information. For example, the access patterns of `result_map` in Figure 5 are determined by the content of `array`. Therefore, the profiled memory access trace of a single source line can be mixed with sequential and irregular behaviors at different periods of the execution. In addition, the likelihood of page faults is also determined by the status of EPC, which could be affected by the page management thread and other enclaves at runtime. Nevertheless, the instrumentation points should be carefully selected, so that the overhead of unnecessary *BIT_MAP_CHECK* will not offset the benefit of page preloading. We will discuss this in more details in Section 4.4.

## 4 Implementation

To demonstrate the effectiveness of the proposed page preloading schemes, we have implemented a prototype. For the DFP preloading scheme, we implement it in the Linux operating system kernel as part of the SGX driver provided by Intel [20], and a simple linear predictor is also implemented to demonstrate our idea. For the SIP preloading scheme, we use LLVM [22] to instrument the source code and to obtain the profiling data. To further reduce the engineering effort, we execute the compiled binary code in SGX enclaves using Graphene-SGX [39], which can run unmodified binaries in SGX enclaves.

### 4.1 Page Access Predictors in DFP

There exists a large body of research on data prefetching techniques. But, it is still a challenging problem to find specific prefetchers that are suitable for a particular application [6].

**Algorithm 1:** Multiple Stream Predictor Algorithm

```
Input: The page number npn, on which a page access
       fault event is triggered;
The ID of the process which triggers the page fault
event.
Output: list_to_load. The list of pages should be
        loaded based on the prediction.

1  set_empty(list_to_load);
2  stream_list = find_stream_list(ID);
3  foreach stream entry n in stream_list do
4      if npn is sequential to n->stpn then
5          n->stpn = npn;
6          direction = get_direction(npn, n->stpn);
7          move_to_head(n, stream_list);
8          add_to_list(list_to_load, npn,
             LOADLENGTH, direction);
9  if n is the last enty of stream_list then
10     n->stpn = npn;
11     move_to_head(n, stream_list);
12 return list_to_load;
```

The data prefetchers integrated in the modern processors, such as Intel Xeon, use more conservative schemes such as *next-line* and *stride* prefetchers [6, 41].

Similarly, based on the mechanism of DFP, many complex strategies can be implemented that include heuristic schemes or even machine learning based schemes [15]. Without losing generality and simplicity, a multiple-stream predictor is implemented in DFP. It is similar to the read-ahead mechanism in Linux virtual file system [14]. With the help this predictor, the sequential streams of the recently triggered page faults can be recognized to predict the following group of pages to be accessed.

As shown in Algorithm 1, the stream of page faults is maintained through a linked list with a fixed length, called *stream_list*. It is managed using a *least recently used* (LRU) scheme. In each entry of *stream_list*, the page number of the most recently triggered page fault in the stream, called *stpn* (i.e. *stream tail page number*), is recorded.

Anytime a new page fault is triggered, its page number, called *npn* (i.e. *new page number*), is extracted by the OS. Our algorithm will go over *stream_list* to check if *npn* is the page next to any *stpn* on the list. If there is such an entry in *stream_list*, the field of *stpn* is updated to *npn*, and the entry is moved to the head of the list. At the same time, the following *LOADLENGTH* pages are added to the *list_to_load*, which will be loaded into EPC asynchronously after the prediction procedure. The *LOADLENGTH* is called *preload distance*, and is the number of pages to be preloaded into EPC. Otherwise, we will replace the last *stpn* on *stream_list* (i.e. the least recently used entry on the list) with *npn*, and move it to the head of the list.

For example, assume *page(1)* is the *stpn* of an entry on the list, and *page(2)* is the page number of the newly-triggered page fault that follows *page(1)*, i.e. *npn*. This multiple-stream predictor will detect the pattern and, in addition to loading *page(2)*, it will preload *page(3)*, *page(4)*... and *page(npn+ LOADLENGTH-1)* into EPC asynchronously. Let us assume *LOADLENGTH* is 8, i.e. it will preload 8 pages into EPC at a time. If a page fault on *page(5)* occurs during such a preloading process, the page fault handler will check if *page(5)* has already been preloaded into EPC. If it is not in EPC yet, and assume the preloading only reaches *page(3)*. All the remaining pages yet to be preloaded, i.e. *page(4)* through *page(8)* will be aborted, and *page(5)* will be considered as the start of a new stream.

There are two design parameters in the current predictor of DFP, which may affect the effectiveness of DFP and can be tuned during the implementation. One is the length of *stream_list*. The other is the number of pages to be preloaded each time, i.e. *LOADLENGTH*. Our current implementation uses empirical values for these two parameters. More details are presented in Section 5.

## 4.2 Implementation of Abort Mechanism in DFP

The DFP preloading process needs to be aborted if too many preloaded pages are not accessed by the application, i.e. there is a mis-prediction. To implement this, a list called `PreloadedPageList` is maintained to track all preloaded pages. There is an access bit for each page on the list. The bit is set to 0 when a new page is loaded, and is set to 1 if the page is accessed.

In the current implementation of the SGX driver, similar to the *CLOCK* replacement algorithm [8], there is a service thread periodically checking and cleaning the access bits in the page table entries to select the best candidate page to evict, if necessary. We piggyback the service thread to update our `PreloadedPageList` during the scan. We collect the number of preloaded pages that are marked as accessed in the page table. This number, called *AccPreloadCounter*, shows the number of pages that are correctly preloaded. Additionally, there is another counter, called *PreloadCounter*, to record the total number of pages preloaded that includes both correctly and incorrectly preloaded pages.

The page preloading thread compares *AccPreloadCounter* and *PreloadCounter* periodically. If the *AccPreloadCounter* is smaller than *PreloadCounter* by a preset threshold value, which means it is preloading too many unused pages, the preloading thread stops itself to avoid excessive overhead. The threshold is an empirical value that can be tuned. In the current implementation, the preloading thread stops when $AccPreloadCounter + 200000 < PreloadCounter/2$. This empirical formula is obtained via curve fitting and manual tuning.

### 4.3 Page Preloading Mechanism in SIP

As mentioned earlier, it is the instrumented code in SIP that sends EPC page preloading requests to the operating system at runtime. The loading of the EPC page is performed by the operating system. More specifically, a Linux kernel thread is created to perform the page preloading. We use a set of memory locations shared between the application in the enclave and the operating system to pass these messages.

It is worth noting that, although the data in an SGX enclave cannot be accessed from the outside of the enclave, the code in the enclave can access non-EPC memory regions, as this is necessary to pass the arguments to the applications running in the enclave. In our implementation, the shared memory locations are allocated in the user address space of the application, and the addresses of the locations are passed to the operating system through a system call we have implemented.

To mitigate the overhead introduced by imprecise static analysis of which memory access will trigger a page fault, we import the presence status of each enclave page from the operating system to the enclave. Each time, before the instrumented code sends a preloading request, the presence status of the corresponding page is first checked to see whether it is already in the EPC or not. If it is already in the EPC, the enclave continues the execution without preloading it.

The presence status is implemented as a bitmap array, i.e., each enclave virtual page corresponds to one bit in the array, and there is one bitmap for each enclave shared between the enclave and the operating system. The bitmap array is created when the enclave is established, and initialized when EPC pages are claimed by the enclave. During the execution of the application in the enclave, the bitmap array is updated by the operating system only when an EPC page is evicted or loaded back from the non-EPC physical memory. Note that, the presence status of the pages is not considered as part of the protected information in the enclave since it is always available in the untrusted operating system.

### 4.4 Instrumentation for Page Preloading in SIP

To mitigate the performance overhead incurred by SIP instrumentation, we selectively instrument memory access instructions that are very likely to trigger EPC page faults during the *profiling runs*. To this end, we also utilize Algorithm 1 to analyze the information gathered during the *profiling runs*, and classify each memory access based on the characteristics of the page it accessed.

- **Class 1.** The page is on *stream_list*, i.e. the page can be found in EPC with a high probability.
- **Class 2.** The page is not on *stream_list*, but follows one of the entry in *stream_list*. This is a situation similar to those described in DFP (Section 4.1).
- **Class 3.** The page is neither on *stream_list* nor following any entry on *stream_list*. This implies that it is an irregular access, which may trigger an EPC page fault.

| Category | Benchmark |
|---|---|
| Small Working Set | cactuBSSN, imagick, leela, nab, exchange2 |
| Large Working Set with irregular access | roms, mcf, deepsjeng, omnetpp, xz |
| Large Working Set with regular access | bwaves, lbm, wrf, microbenchmark |

**Table 1.** Classification of Benchmarks

Obviously, it is unnecessary to instrument instructions in **Class 1** as it is already in EPC. As the multistream predictor in DFP may capture sequential accesses more effectively than SIP with the help of runtime information, we can leave instructions in **Class 2** to DFP, if both SIP and DFP schemes are used. As a result, we only need to perform the SIP instrumentation for instructions in **Class 3**. To handle the case in which memory accesses by an instruction can be classified into more than one class, we make the instrumentation decision based on the majority of the memory accesses issued by the instruction, which is determined by an empirical ratio among its different classes (see Section 5 for more details).

## 5 Experimental Results

In this section, we evaluate the performance improvement that can be achieved by the proposed page-preloading schemes. Our evaluation includes a selection of benchmarks from the SPEC CPU2017 suite [10]. We also use a microbenchmark that sequentially accesses an 1GB memory region through a loop to verify the correctness, and to provide some baseline performance for our implementation. Table 1 shows some detailed characteristics of the benchmarks. We run the applications with Graphene-SGX and vanilla SGX drivers, and profile their memory access behaviors. We mainly focus on the performance improvement of benchmarks with large working sets that exceed the size of EPC, i.e. the benchmarks that spend a significant amount of time in page fault handling.

In addition, we also evaluate two real-world applications, *Scale Invariant Feature Transform* (SIFT) and *Maximally Stable Extremal Regions* (MSER), both of which are image processing applications from the San Diego Vision Benchmark Suite [40]. SIFT extracts features from images that are robust to scaling, rotation and noise. It is an important tool in diagnosis and subsequent treatment of eye diseases, and has been widely used in medical imaging analysis [33, 36]. MSER is an application used to detect blobs in images. It is widely used in medical image segmentation, visual surveillance, traffic analysis and vehicular tracking [35, 45].

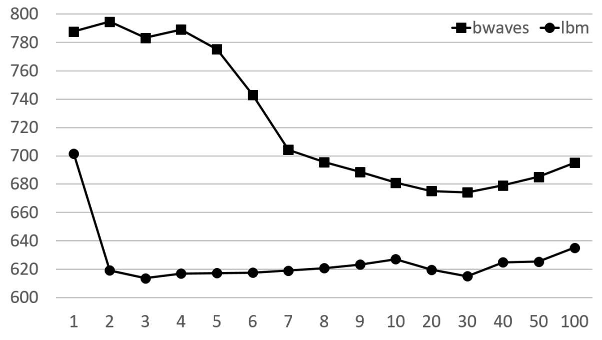


**Figure 6.** Execution time of two benchmarks, i.e., lbm and bwaves, using DFP with different lengths of the linked list used to save the addresses of faulted pages.

The experimental platform is an HP-Z238 workstation, equipped with an Octa-Core Intel Xeon E3-1240v5 CPU running at 3.5GHz with 8MB last-level cache, with the latest CPU firmware of Release-V20191115. The main memory is 32GB and the operating system is Ubuntu 18.04 with the Linux-4.4.0-141. The version of the SGX driver is 2.6, which was the latest version when we conducted these experiments. The source code is compiled with the LLVM compiler, and the binary code is executed in SGX enclaves using Graphene-SGX [39] V1.1. To more accurately measure the performance of an application, we use the execution time of an empty binary running on Graphene-SGX as the baseline, and subtract it from the total execution time in the final analysis. The experimental platform is used exclusively for the evaluation. Also, to reduce the influence of random factors on performance, each application is executed 5 times and their arithmetic means are used.

## 5.1 Performance Study of DFP

As mentioned earlier, there are two design parameters in DFP. One is the length of the linked list used to record the streams of faulted pages. Figure 6 shows the performance results with different lengths for two benchmarks, *lbm* and *bwaves*, which have large memory footprints with mostly regular page access patterns. As shown in the figure, although the two benchmarks achieve the shortest execution time at different lengths, the combined execution time of them is shortest when the length is around 30. Therefore, our current implementation of DFP uses 30 as the default length for the linked list.

Another parameter is the number of pages preloaded at each preloading. Figure 7 shows the normalized execution time of the 7 benchmarks that have large memory footprints with both regular and irregular access patterns. The figure shows the performance when different numbers of pages are preloaded at each page preloading. As shown in the figure, when the number exceeds 4, a substantial performance loss can be observed for some benchmarks, such as *mcf* and *deepsjeng*. As a result, we set the preloading pages to 4 EPC pages each time in DFP.

Figure 8 shows the performance improvements achieved by DFP. The execution time of each benchmark is normalized to its original execution time without using DFP. As shown in the figure, DFP achieves performance improvements for every large memory-footprint benchmark with regular access patterns that includes *bwaves*, *lbm*, and *wrf*. Especially, the performance improvement of the microbenchmark can be as high as 18.6%, while *lbm* is improved by 13.3%. However, for some other benchmarks, such as *mcf*, *deepsjeng*, *roms* and *omnetpp*, DFP can introduce additional performance overhead. Further investigation shows that this is mainly caused by the mispredictions. This again demonstrates that it is quite challenging to design a general preloading scheme to cover different types of applications. On average, DFP can achieve 11.4% performance improvement for the evaluated regular benchmarks.

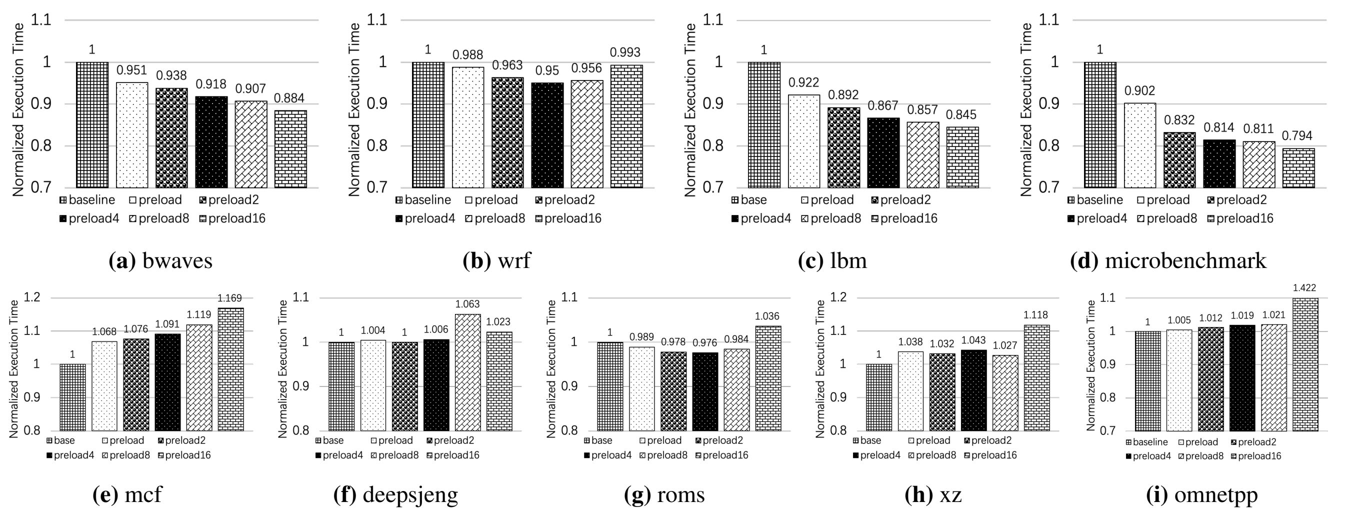
To reduce the performance overhead incurred by mispredictions in DFP, we further introduce an abort mechanism for DFP, denoted as DFP-stop. In particular, we abort a page preloading when the number of its mispredictions exceeds a threshold. The performance improvement can be seen in Figure 8. As shown in the figure, DFP-stop helps to mitigate the performance penalty introduced by mispredictions even though the performance improvement is not as significant as expected. There is some noticeable improvement. For example, the overhead of *deepsjeng* and *roms* decrease from 34% and 42% to 0% and 0.1%, respectively. On average, the performance overhead decreases from 38.52% to 2.82%.

Note that, as described in Section 4.1, there is already an abort mechanism within a stream of page preloading, i.e. within the range of *LOADLENGTH*. The abort mechanism described here is only used as a "safety valve" to prevent misprediction from causing too much penalty in some extreme cases. The overhead it incurs is negligible. The abort mechanism is integrated into the DFP and enabled by default.

## 5.2 Performance Study of SIP

In Section 4.4, we mention that each memory instruction can issue many memory accesses of different classes. The percentage of *irregular* memory accesses (i.e. the class 3 accesses) is a useful parameter for SIP. In our implementation, we set a threshold to determine if the instruction should be instrumented for SIP or not. Every memory instruction with a percentage of irregular accesses above the set threshold will be instrumented with a preloading notification.

If this threshold is set too low, which means we are instrumenting in an aggressive manner, the overhead incurred by the instrumented code may exceed the expected performance gain. On the other hand, if this threshold is set too high, i.e. we only select instructions with a large number of irregular accesses, we may miss some potential opportunities that can



**(a)** bwaves **(b)** wrf **(c)** lbm **(d)** microbenchmark

**(e)** mcf **(f)** deepsjeng **(g)** roms **(h)** xz **(i)** omnetpp

**Figure 7.** Normalized execution time when preloading different numbers of EPC pages each time in DFP. The baseline is the original execution without preloading.

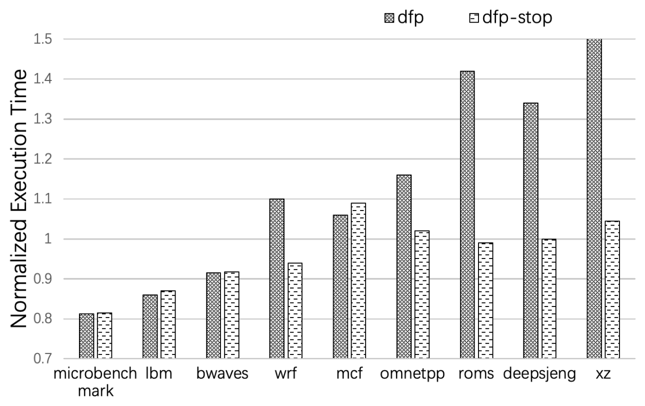


**Figure 8.** Performance improvement achieved by DFP. The execution time is normalized to the original execution time without DFP.

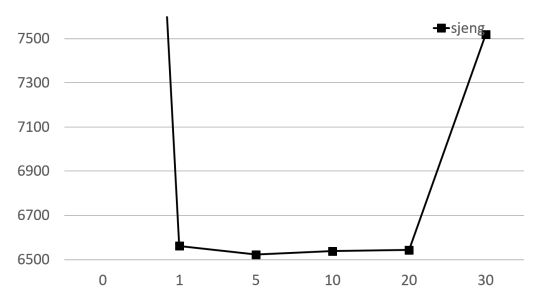


**Figure 9.** Running time of *deepsjeng* with different irregular access ratio. The y-axis is the execution time of instrumented sjeng with train input set, while the x-axis is the instrumentation threshold of the irregular access ratio.

improve overall performance. In order to find a sweet spot, we conduct a series of experiments on *sjeng* using different threshold values, and the results are shown in Figure 9. It can be seen that the application has the best performance at around 5%. We also test it on *mcf* and get a very similar result. So, the misprediction threshold is set at 5% in all of our experiments.

Due to the limitation of our current implementation, only C/C++ applications are supported. Therefore, *bwaves*, *roms* and *wrf* (written in Fortran) are not included here. The benchmark *omnetpp* is also omitted because our instrument tool cannot fully support it. We also included *mcf* from SPEC CPU 2006 to expand the benchmark applications (denoted as mcf.2006). We present the performance results in Figure 10.

To evaluate a more realistic scenario in practice, we use different input data sets for profiling and performance-collecting runs. For benchmarks from SPEC CPU 2017, the training data is used for profiling and the reference data is used for performance-collecting runs. For SIFT and Another vision based app, we use one sample image for profiling and other images for performance-collecting runs.

As shown in Figure 10, SIP achieves 9.0% performance improvement for *deepsjeng* and 4.9% for *mcf.2006*. From profiling, we learn that these two applications contain a large number of irregular accesses, and the page faults triggered by the applications can be reduced by more than 70% after using SIP. However, the profiled data of *lbm* and *microbenchmark* shows that they have very few irregular accesses, and SIP cannot find proper locations to do the instrumentation. Therefore, the performance of those three applications cannot be improved by SIP, as reflected in the experimental results.

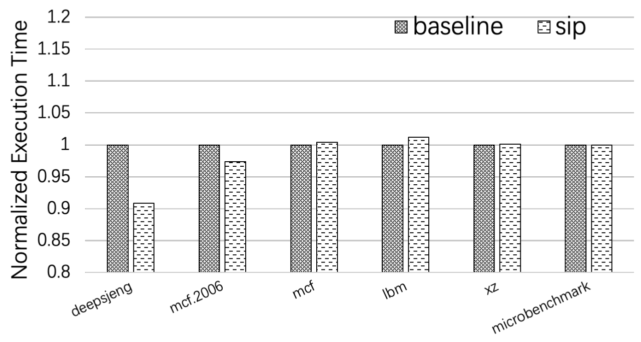


**Figure 10.** Performance improvement achieved by SIP. The execution time is normalized to the original execution time without SIP.

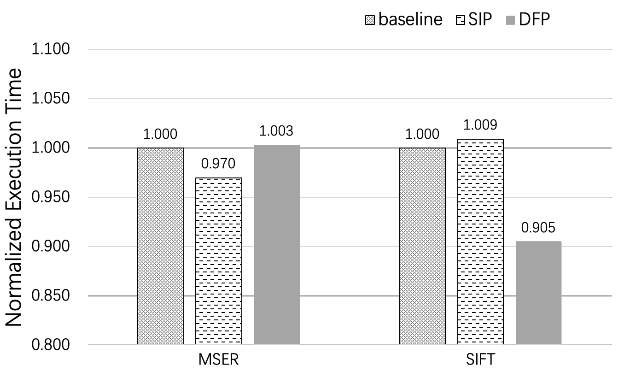


**Figure 11.** Normalized execution time of SIFT and MSER using DFP and SIP, respectively. The performance baseline is the original execution of MSER and SIFT without using preloading.

It is also worth noting that we find that even if there is a lot of irregular accesses during the profiling runs, there could still be no performance gain using SIP, such as *mcf*. After a more thorough study on the profiled data, we find that it contains many memory-intensive instructions whose memory behavior exhibits a large number of both **Class 1** accesses (i.e. EPC page hits) and **Class 3** accesses (i.e. irregular accesses), but with very few **Class 2** accesses (i.e. sequential stream accesses), as defined in Section 4.4.

If such instructions are instrumented with preloading notifications, a large number of page faults can be avoided for those Class 3 accesses, but it will incur significant overheads for those Class 1 accesses due to the instrumented code. If those instructions are ignored by the SIP, the page faults triggered will cause heavy runtime overheads because they lack Class 2 accesses, i.e. DFP cannot help those instructions because of the lack of Class 2 accesses.

We find *mcf* happens to be these kind of applications. The benefit gained from the instrumentation via SIP for Class 3 accesses is offset by the overheads incurred on Class 1 accesses. They offset each other, and the end result is a wash in the performance gain. This presents an interesting dilemma on the use of SIP and DFP to mitigate the performance overhead incurred by page faults in SGX enclaves.

It is interesting to note that even the functionality of *mcf* and *mcf.2006* is supposed to be the same, their preloading schemes are quite different because of different ratios in the types of memory accesses in their implementation. It can be further optimized with more careful instrumentation and fine tuning. It is another proof that coming up with a general preloading scheme is quite challenging.

## 5.3 Results of Two Real-World Applications

To evaluate the two real-world image processing applications, i.e., SIFT and MSER, we use the images from the MIT-Adobe FiveK Dataset [1] as the input. We obtain the profiling data of these two applications using one sample image. It shows that both of these two applications have a large memory footprint. The difference between them is that SIFT has more sequential accesses, while MSER has more irregular accesses. This makes SIFT a good candidate for the DFP preloading scheme and MSER a good candidate for the SIP preloading scheme.

Figure 11 shows the normalized execution time of these two applications using the original execution time without preloading as the baseline. As shown in the figure, with the help of our page preloading techniques, SIFT and MSER can obtain 9.5% and 3.0% performance improvements, respectively. This demonstrates the proposed preloading schemes are beneficial to real-world applications as well.

## 5.4 Combining SIP and DFP

Next, we study the performance improvement that can be achieved by combining SIP and DFP, i.e. a hybrid scheme. It is clear that SIP and DFP can improve different parts of an application with different memory behaviors, so that the performance can be further improved if they can collaborate with minimal interference.

Figure 12 shows the normalized execution time of SIP, DFP, and the hybrid scheme. We have omitted the applications that are not written in C/C++ due to the limitation of our implementation as mentioned earlier. It is worth noting that, based on our profiling and analysis, our benchmark programs exhibits memory behavior either mostly sequential accesses (i.e. Class 2 accesses that are amenable to DFP) or irregular accesses (i.e. Class 3 accesses that are amenable to SIP), but very few benchmarks with a substantial number of accesses in both classes. This means that they can only be improved either by SIP or DFP, but not much by the hybrid scheme. Our experimental results seem to bear this out.

As shown in Figure 12, by combining these two page preloading schemes, the performance improvement achieved

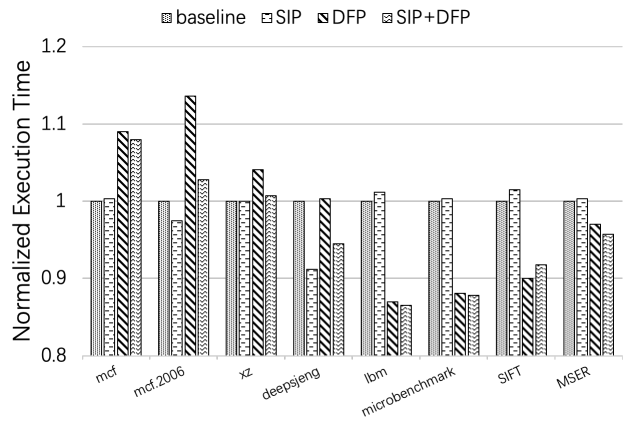


**Figure 12.** Normalized execution time of SIP, DFP, and the combination of them. The baseline is the original execution time without using preloading.

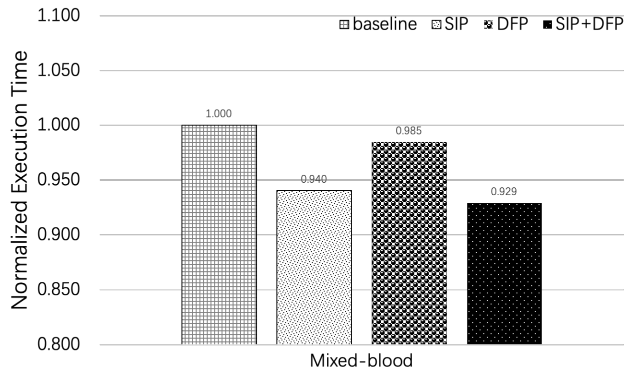


**Figure 13.** Normalized execution time of mixed-blood

is mostly close to the better of the two schemes, but not substantially more. Nonetheless, it shows that it is feasible to combine these two preloading schemes without hurting each other. It is also worth noting that in the worst case, such as *mcf* mentioned earlier, the average overhead is about 4.2%.

To validate our arguments, we synthesize an application with a similar number of sequential accesses (i.e. Class 2 accesses) and irregular accesses (i.e. Class 3 accesses). In this synthesized program, we sequentially scan an image and then invoke MSER for blobs detection. This application is called *mixed-blood* in Figure 13. We did the same experiments on *mixed-blood* and the results are shown in Figure 13. From the figure, we can see that SIP alone gets 1.6% improvement and DFP alone gets 6.0% improvement, while the performance is further improved by the hybrid scheme (marked as "SIP+DFP") with 7.1% improvement.

## 5.5 TCB Size Study

Finally, we quantitatively evaluate the increased size of enclave TCB in SIP. Note that the TCB does not increase in DFP. The size increase of TCB for SIP can be contributed to two factors: (1) the code in the preloading notification; and (2) the number of the invoked points for notification. The preloading notification function is implemented in C with 23 lines of code. We measure the number of invoked points inserted after the SIP instrumentation and show the data in Table 2. This demonstrates that SIP is able to maintain a small TCB size.

| Benchmark | Instrumentation Points |
|---|---|
| mcf.2006 | 114 |
| mcf | 99 |
| xz | 46 |
| deepsjeng | 35 |
| lbm | 0 |
| MSER | 54 |
| SIFT | 0 |
| microbenchmark | 0 |

**Table 2.** The number of instrumentation points in different benchmarks

## 5.6 Discussion

Compared to the memory footprint of the modern software, the EPC size is relatively small. Even though an enclave can be configured to fit the memory requirement of an application, EPC over-subscription will lead to thrashing and significant performance degradation. Sharing EPC among multiple processes or multiple virtual machines[13] is supported on Intel processors, but the total EPC size remains the same and each enclave will receive a smaller portion. As each enclave can handle its preloading independently, our proposed schemes (i.e. SIP and DFP) will work for each enclave. However, EPC contention becomes a serious issue. Such an issue is similar to the sharing of the last-level cache (LLC) or the main memory. Much work has been dedicated to address the "fairness" and the "performance" concerns of such resource sharing [9, 31, 44], which is beyond the scope of this paper.

Even though page preloading can increase the bandwidth utilization between the EPC and the untrusted memory, the memory latency can be hidden if a page is correctly preloaded before its access. However, the performance improvement is limited by the software characteristics. For memory-bound applications, i.e. applications spending less time doing computing, it is difficult to improve the performance by preloading because there is not enough computation time to hide (i.e. overlap with) its memory latency.

The limitation on the hardware also makes the preloading quite challenging. Because of the hardware constraints and the scalability limitation of the Linux page management [29], the page load-in process is exclusive and non-preemptible. In addition, the page load-in time is very long (i.e. 44,000 cycles). Therefore, any page load-in operation issued between two page faults with an interval less than the loading time will delay regular memory accesses. In this situation, even if the prediction is correct, the performance benefit is still limited.

Some recent work has been trying to improve the page-loading process, e.g. Eleos[26] and CoSMIX[27]. They show that the performance can be improved with effective software-based memory management at runtime. Our proposed page-preloading schemes can be integrated with those page-loading processes to deliver an even higher performance.

## 6 Related Work

A large amount of research has been conducted focusing on the limited resource and heavy overhead of world switch in SGX, which include enlarging the size of EPC and reducing the overhead to handle EPC page faults. For example, Morphable Counters[34] and VAULT [38] have been proposed to provide new data structures and algorithms for integrity verification of EPC, so that the EPC size can be enlarged theoretically. It can be seen that the application performance can be significantly improved with a larger EPC. However, it may take a long time for the processor vendors to integrate these new technologies into the product. The overhead of EPC page swap is still very heavy when the footprint of applications exceed the limited EPC size. The preloading scheme we proposed can be considered as a latency hiding scheme for the memory hierarchy between EPC and untrusted memory space.

To mitigate the overhead of world switch, SCONE [5] and Hotcalls [42] have been proposed with new SGX interface to reduce the overhead of specific operations in SGX such as entry-calls, out-calls. Similar to the method we used in SIP, delegating mechanism in untrusted world is employed to avoid the world switch. However, these techniques are not aimed at reducing the performance overhead introduced by page faults in enclaves, which are the major issues targeted in this paper.

Eleos [26] and CoSMIX[27] have addressed the similar problem to our research. To mitigate the heavy overhead of page fault, a software page management framework has been implemented to manage the presence status of the pages in EPC, and exchange the page content with untrusted memory space. Even though the same encryption algorithm have been employed, it is difficult to maintain the same security guarantee with the hardware implemented instructions, i.e. `EWB` and `ELDU/ELDB`, especially at the micro-architecture level. On the contrary, in our approach, we exactly follow the page management procedure to take advantage the security feature offered by SGX. Another concern when implementing user-level page management is that every memory access in the enclave should be instrumented to check the status of the corresponding page. A cache oriented optimization and software TLB have been implemented to minimize the runtime cost. However, it is unnecessary in our solution since only the sensitive memory access is instrumented in SIP while the application is not affected by DFP. Despite the code space consumed by the runtime software, maintaining the software implemented page table in the enclave still cause additional pressure on limited EPC size, since the address space of an enclave can be as large as the conventional 64-bit application software. Unlike them, our solution cause negligible overhead on EPC since only some checking and notification code has been instrumented to selected positions in SIP.

## 7 Conclusion

Intel SGX offers a strong guarantee for security and privacy. However, user applications have to pay a significant performance overhead to take advantage of SGX. One major reason is that the secure physical memory provided by SGX is limited. This can lead to a large number of page faults, especially for memory-intensive applications with large memory footprints. To address this issue, this paper proposes to leverage page preloading techniques to mitigate the performance overhead introduced by the page faults. With the help of these two mechanisms, the number of page faults can be reduced significantly. To demonstrate the effectiveness of the proposed preloading mechanisms, we have implemented them on a prototype. We then evaluate this prototype using benchmarks from the SPEC CPU2017 benchmark suite, some real-world applications, and a micro-benchmark program. Experimental results show that, on average, the proposed two preloading mechanisms can achieve on average of 11.4% and 7.0% performance improvement, respectively. The two mechanisms can also be combined together to achieve an improvement of 7.1% on real applications.

The source code of our work is publicly available at `https://github.com/NKU-EmbeddedSystem/preloading-for-sgx`

## Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (61702286), the National Key Research and Development Program of China (2018YFB1003405), the Natural Science Foundation of Tianjin, China (18JCYBJC15600), the CERNET Innovation Project (NGII20190514), and a faculty startup funding of the University of Georgia.

## References

[1] 2017. MIT-Adobe fivek datasett. (2017). http://data.csail.mit.edu/graphics/fivek/.

[2] AMD. 2020. AMD GuardMI Technology. https://www.amd.com/en/technologies/guardmi.

[3] ARM. 2020. Introducing Arm TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.

[4] 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* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 689–703.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703.

[6] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.

[7] Thomas Ball and James R Larus. 1994. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 4 (1994), 1319–1360.

[8] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[9] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing*. IEEE, 749–758.

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[12] Damon. 2019. Cost of a page fault trap. (2019). Accessed 19 Apirl 2020. https://stackoverflow.com/questions/1022 3690/cost-of-a-page-fault-trap.

[13] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. 2019. Everything You Should Know About Intel SGX Performance on Virtualized Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 5 (March 2019), 21 pages.

[14] WU Fengguang, XI Hongsheng, and XU Chenfeng. 2008. On the design of a new linux readahead framework. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 75–84.

[15] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *arXiv preprint arXiv:1803.02329* (2018).

[16] Greg Hoglund and Gary McGraw. 2004. *Exploiting Software: How to Break Code*. Pearson Higher Education.

[17] IBM. 2020. IBM Secure Service Container. https://www.ibm.com/us-en/marketplace/secure-service-container.

[18] Intel. 2020. Intel Software Guard Extensions. https://software.intel.com/en-us/sgx.

[19] Intel. 2020. Intel Software Guard Extensions SDK. https://software.intel.com/en-us/sgx/sdk.

[20] Intel. 2020. Intel(R) Software Guard Extensions for Linux* OS. https://github.com/intel/linux-sgx-driver.

[21] Pratheek Karnati. 2020. Data-in-use protection on IBM Cloud using Intel SGX. https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–.

[23] Hui Lei and Dan Duchamp. 1997. An analytical approach to file prefetching. In *USENIX Annual Technical Conference*. 275–288.

[24] Roy Levin, Ilan Newman, and Gadi Haber. 2008. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 291–304.

[25] Microsoft. 2020. Protection by design: Intel SGX and Azure Confidential Computing. https://azure.microsoft.com/en-us/resources/videos/ignite-2018-protection-by-design-intel-sgx-and-azure-confidential-computing.

[26] 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*. ACM, 238–253.

[27] 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.

[28] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 151–165.

[29] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 813–827.

[30] David K Poulsen and Pen-Chung Yew. 1994. Data prefetching and data forwarding in shared memory multiprocessors. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, Vol. 2. IEEE, 280–280.

[31] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.

[32] RedHat. 2019. CVE-2019-0117. (2019). Accessed 10 Sept 2020. https://access.redhat.com/security/cve/CVE-2019-0117.

[33] Yang Sa. 2015. Medical Image Registration Algorithm Based on Compressive Sensing and Scale-Invariant Feature Transform. In *2015 8th International Conference on Intelligent Computation Technology and Automation (ICICTA)*. IEEE, 547–551.

[34] Gururaj Saileshwar, Prashant Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose Joao, and Moinuddin Qureshi. 2018. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 416–427.

[35] Ehab Salahat, Hani Saleh, Andrzej S Sluzek, Baker Mohammad, Mahmoud Al-Qutayri, and Mohammad Ismail. 2015. Novel MSER-guided street extraction from satellite images. In *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. IEEE, 1032–1035.

[36] Luis A Salazar-Licea, C Mendoza, MA Aceves, JC Pedraza, and Alberto Pastrana-Palma. 2014. Automatic segmentation of mammograms using a Scale-Invariant Feature Transform and K-means clustering algorithm. In *2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*. IEEE, 1–6.

[37] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. [n.d.]. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[38] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 665–678.

[39] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. 645–658.

[40] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 55–64.

[41] Krishnaswamy Viswanathan. 2014. Disclosure of Hardware Prefetcher Control on Some Intel® Processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html.

[42] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 81–93.

[43] wolfSSL. 2020. wolfSSL with Intel® SGX. https://www.wolfssl.com/wolfssl-with-intel-sgx.

[44] Yuejian Xie and Gabriel H Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 174–183.

[45] Haijiang Zhu, Junhui Sheng, Fan Zhang, Jinglin Zhou, and Jing Wang. 2016. Improved maximally stable extremal regions based method for the segmentation of ultrasonic liver images. *Multimedia Tools and Applications* 75, 18 (2016), 10979–10997.