Karlsruher Institut für Technologie

On the Applicability of More Accurate Page Access Information

Masterarbeit
von

Julian Faude

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl.-Inform. Marc Rittinghaus

Bearbeitungszeit: 1. Februar 2014 – 31. Juli 2014

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 31. Juli 2014

Contents

Chapter 1

Introduction

Paged virtual memory management introduces a classical allocation problem. The computer system is equipped with x fast physical page frames and runs an operating system and a number of processes. The memory requirements of the operating system and the processes amount to $y > x$ virtual memory pages. Pages that do not fit into physical memory at a point in time, must be buffered on slow external storage. The operating system is tasked to constantly decide which pages should reside in fast physical memory and which pages should be pushed to slow external storage until they are needed again. The operating system's objective is to reduce page traffic, that is the number of times a page must be brought from external storage into physical memory, without knowledge of the future memory needs of the operating system and processes. To this end, it applies a paging algorithm. Paging algorithms use page access information describing past memory usage provided through the system's page tables, acting as an interface between the hardware and software. Paging algorithms employ assumptions about memory usage in order to extrapolate future memory usage from past memory usage. However, the IA-32e architecture page table format reserves only a single bit of access information for each page.

The objective of this work is to assess the nature of potential inaccuracies and whether this 1 bit capacity provides sufficiently accurate page access information to the paging algorithm. In case 1 bit proves in fact insufficient, thereby introducing inaccuracies into the operating system's perception of past memory usage, it yet remains unclear whether or not this inaccuracies worsen the paging algorithm's allocation decisions preventing it from performing at its best. A second objective of this work is thus to assess whether or not the paging algorithm may potentially benefit from more accurate page access information.

To achieve this objective, as a first step, we trace the memory usage of an entire computer system, running an instrumented guest operating system and a number of processes, including a set of benchmarks. Additionally, we record the interactions between the operating system's paging algorithm and the hardware page tables. In order to learn to which extend the limited storage capacity of the page tables introduces inaccuracies into the operating system's perception of memory usage, we replay the recorded memory usage and interaction between the paging algorithm and page tables. This allows us to reconstruct and compare the operating system's different perceptions of the benchmark's memory usage, with respect to both the limitations of IA-32e page tables as well as hypothetical, less limited pages tables. To this end, we define the perception induced by IA-32e page tables as distorted in case it differs from the hypothetical perception based on higher resolution access information.

As a second step, we conduct a paged virtual memory management simulation of the recorded memory usage of a number of benchmarks and record the number of page faults it generates. The simulation employs paging algorithms like least frequently as well as least recently used page replacement. It provides them with either low resolution but authentic 1 bit page table page access information or higher resolution page access information based on hypothetical, higher capacity page tables. The simulation results for different paging algorithms operating on page access information with different resolutions allow us to assess whether or not they benefit from higher capacity page tables. In case they do, we take this as an indication that the paging algorithms suffer from inaccurate page access information provided by low capacity page tables.

We employ five benchmark programs: a custom program performing a series of binary searches, a custom program sorting an array of integers using the quicksort algorithm, a database undergoing a series of complex transactions, physically-based rendering, and a custom program compressing an archive using the zlib library. We assess the interaction between the Linux paging algorithm and the IA-32e page tables for the binary search as well as the quicksort benchmark very closely. The evaluation confirms that the operating system's perception of memory usage induced by 1 bit page access information provided by IA-32e page tables is in fact distorted as it significantly differs from the hypothetical perception based on completely accurate access information. The evaluation also finds that the induced perception not only does not improve but potentially loses accuracy with moderately increased page table capacity. We find that only increasing the page table's capacity to at least 16 bit for every page actually improves the operating system's perception of memory usage significantly, to the point where it does not further suffer from distortion.

The evaluation of the paging simulation, however, reveals that paging algorithms do not benefit from more accurate page access information. For four out of five benchmarks the simulation shows that more accurate page access information does not effect the paging algorithms performance at all, regardless of the extend to which we increase the page table's capacity. For the fifth benchmark, the quicksort benchmark, the simulation shows that the number of page faults drops by less than 4% when comparing the results of 1 bit page access information for each page to inaccuracy free 16, bit page table page access information.

1.1 Outline

Following the introduction, Chapter 2 provides background of paged virtual memory management, common page replacement approaches and the IA-32e architecture. Furthermore, it gives an overview of the Mattson stack algorithm [21] as an approach to paging simulation. The chapter is supplemented with an introduction to full system simulation and tracing using Simutrace [11,25]. Chapter 3 analyzes what inaccuracies the limitations of IA-32e architecture page tables introduce into the paging algorithm's perception of memory usage. Furthermore, it suggests extended page table formats providing more accurate page access information and discusses key aspects of paging algorithm performance assessment. Subsequently, Chapter 4 presents our proposed analysis method for analyzing the distortions that inaccurate page access information introduce into the operating system's perception of memory usage. Additionally, it outlines our approach to paging simulation. Chapter 5 discusses the setup of the simulator as well as the simulation guest, the trace data postprocessing and the implementation of the memory perception reconstruction as well as the paging simulation. An evaluation is performed in Chapter 6. Finally, we close with a conclusion and prospect of future work in Chapter 7.

Chapter 2

Background

The purpose of this chapter is to give short explanations of the technical concepts and terminology used throughout this work as well as to discuss related work. The chapter begins with a introduction of the concept of paged virtual memory, i.a., paging, in Section 2.1. Section 2.2 discusses the purpose of paging algorithms and outlines a number of prominent algorithms and their workings. Section 2.3 discusses the Mattson stack algorithm as a mean to simulate paging algorithms. The workings of memory management in the IA-32e architecture is presented in Section 2.4. The chapter closes with a discussion of Simutrace, a toolchain for operating system introspection and memory tracing using full system simulation in Section 2.5.

2.1 Paging

Paging is a memory management mechanism introduced in order to achieve *a*) isolation between processes and *b*) high utilization of memory resources. The Atlas computer [10] was the first computer to support paging. Paging is based on breaking virtual memory as well as physical memory into fixed-sized blocks called *page* and *pages frames* respectively. Paging also introduces a level of indirection between the logical memory addresses, referred to as *virtual addresses*, and *physical addresses*. Each virtual address falls into exactly one page whereas each physical address falls into exactly one page frame. In order to reference a page, a page frame must be allocated to hold the contents of said page. For the purpose of translating a virtual to its corresponding physical address paging employs a *page map*. Aho et al. [3] formalize this concept as follows: $N = \{1, \ldots, n\}$ denotes the set of virtual pages. The set $M = \{1, \ldots, m\}$ denotes the physical page frames.

Then the page map is a function $f : N \to M$ given by

$$f(i) = \begin{cases} j & \text{if page } i \text{ is mapped onto page frame } j \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

for page to page frame translation.

Through this level of indirection paging achieves the desired level of *isolation*. Every isolation domain (e.g., a process) operates in its own virtual memory *address space* [1]. Thus, effectively every address space a is linked to an exclusive set of pages N_a translated by an exclusive page map f_a. However, all address spaces compete for the same set M of page frames.

For the purpose of high utilization paging also allows for *overcommitment*. Virtual memory may comprise more pages than the system supports page frames. In general it holds $1 \leq m \leq n$. In that case only a subset of the pages may be assigned a page frame. The other pages must temporarily be held on external storage and the page map indicates $f(i) = \mathit{undefined}$ for any page i not present in memory. In the event of an reference to such a page a *page fault* is issued. The system then finds a suitable page frame for the page for allocation and potentially *pulls* [10] the page into memory. Before the pull is performed it may be necessary to *push* a page that is currently residing in memory to external storage, effectively freeing the page frame it occupies.

Another of the advantages of paging is *page sharing*. Several identical pages may get allocated a common page frame. For contentwise identical pages i and i' sharing a page frame holds $f(i) = f(i')$. i and i' share a page frame.

Figure 2.1 illustrates those aspects of paging. On the left there are processes represented as logical memory split into several pages. The physical memory (RAM) is shown on the right and likewise split into page frames. Arrows between pages and page frames depict the allocation of page frames as indicated by the page map. As discussed before, some pages share a common page frame, while others do not reside in memory at all.

2.1.1 Address Translation and Page Faults

As a program runs on the CPU it generates memory accesses in the form of references to virtual addresses. In order to retrieve the data identified by a virtual address, first, the address must be translated to the corresponding physical address

[1] In the remainder *address space* and *process* are used interchangeable.

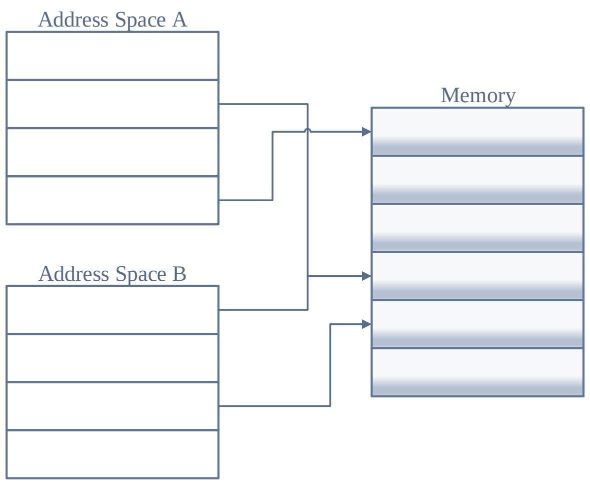

Figure 2.1: Example of pages of two processes A and B being mapped on physical page frames.

using the page map. It is a physical address that identifies the location in memory where requested data is currently stored. This process of translating virtual addresses to physical addresses is referred to as *address translation*.

Aho et al. [3] define address translation as follows: Given a page size of z, the set of pages N and the set of page frames M, address translation finds for any valid virtual address the page i the address falls into and the offset w within said page with $i \in N$ and $0 \leq w < z$. Afterwards it queries the page map for $f(i) \in M$ and hence finds the corresponding physical address as $(f(i) - 1) * z + w$[2].

A page fault occurs in case of a reference to page i with $f(i) = undefined$, that is i is not allocated a page frame. A page fault causes the execution of the process to be interrupted while the paging algorithm handles the page fault. The time the execution remains interrupted is called the *page wait time* [3]. It includes the time it takes to select and potentially free a page frame to hold the page, to potentially load the page contents from external storage into the page frame, as well as the time it takes to adjust the page tables accordingly. Hence, the page wait time depends greatly on two factors: *a*) whether or not a push is necessary to free a page frame for allocation, and *b*) whether or not a pull is necessary to bring page contents from external storage into memory.

However, virtual memory is comprised of pages of different types. The differences stem from differences in the semantic context of the pages and greatly af-

[2]Page i includes addresses $[(i-1) * z, \ldots, i * z)$.

fect the necessary effort of moving pages out of or into memory. Pages whose contents is exclusively defined by the execution of a program at runtime are called *anonymous*. The contents of anonymous pages is initially undefined and only generated at runtime. Since in general the contents of anonymous pages is independent of data from external storage, at runtime such pages must be considered unique. Pages whose contents is determined by data from external storage are called *named*. Such pages include first of all the text sections of processes but also memory mapped files.

Whether or not a push is necessary to remove a page from memory is determined by the page type. On the one hand anonymous pages have to be pushed to external storage in order to safe their unique contents for future use since their content is only present in memory. On the other hand named pages do not have to be pushed when removed from memory because their content is determined by data on external storage and therefore, already present on external storage, ready for subsequent pulls.

Whether or not a pull is necessary to bring a page into memory also depends on the page type. Since the initial contents of anonymous pages is initially undefined, no pull is necessary for the initial fault. Instead, the page is filled with zeros so any prior page frame content is lost for the sake of process isolation. However, subsequent page faults on anonymous pages do require a pull in order to bring the contents from external storage into memory. Named pages also require a pull when brought into memory. Since the inital content of named pages is not undefined but determined by data from external storage, this even holds for the initial page fault.

2.1.2 Paging Policy

According to Aho et al. [3] a *paging policy* specifies how to use the paging mechanism. A paging policy includes three subpolicies.

The *fetch policy* is used to decide when to map a page to a page frame, thereby effectively bringing it into memory. In case the policy states that a page should not be brought into memory before it is referenced for the first time the policy is referred to as a *demand fetch policy*. Otherwise it is called a *nondemand fetch policy*.

Once the fetch policy causes the paging mechanism to bring a page into memory, the *placement policy* finds it a suitable page frame among the set of readily available page frames[3]. Such a policy might for example have all free page frames organized into a list and said page assigned to the first page frame in it.

The third policy is called the *replacement policy* and determines which pages should be removed from memory, effectively freeing pages frames for new allocation. A page selected by the replacement policy is called a *victim*. According to Section 2.1.1 the page wait time partially depends on the page type of the victim page.

An implementation of the paging policy and hence the three subpolicies is called a *paging algorithm* [3]. A special class of paging algorithms are *demand paging algorithms*. As the name suggests demand paging algorithms employ a demand fetch policy, that is never allocate a page frame for a page until it is actually referenced. Furthermore, such algorithms never free a page frame unless absolutely necessary. Put differently, a demand paging algorithm only invokes its replacement policy in case its placement policy finds that it has no free page frame available. Finally, the placement policy places pages in arbitrary page frames as long as there is a number to choose from or accepts any page frame the replacement policy frees.

2.2 Paging Algorithms

Aho et al. [3] also give a formal definition of a paging algorithm. It boils down to a state machine processing the *reference string*. The authors define the reference string as a sequence $\omega = r_1, \ldots, r_t$ of page references[4] with $r_x \in N$ and some t. The states correspond to the current page allocation and a *control state* which is used to encode information about past state transitions. For each page reference r_x in the reference string a state transition occurs. The transition potentially adjusts the page map and updates the control state as specified by the page replacement policy. In case no unused page frame is available on a page fault, it applies its page replacement algorithm on its current control state to select a victim page.

Additionally, Aho et al. [3] present two important findings. The authors state that in general, when studying paging algorithms, the objective is to minimize the total time required to execute a process. However, they find that replacing this criterion for optimality with a slightly different one benefits their studies without significantly limiting the generality of their following conclusions. As a new criterion

[3]In this case readily available page frames refers to a frame which is not allocated to any page.

[4]Page replacement algorithms relocate memory on granularity of pages, not virtual addresses. However, a sequence of accesses to virtual memory address implies a sequence of page references.

Aho et al. suggest the *minimization of aggregate page wait time* instead of execution time and argue that both criteria are effectively equivalent for a demand paging algorithm [3].

Their second finding follows as extension of the first one. A page wait interval always corresponds to a page fault that caused the interruption. Hence, the authors conclude that under a demand paging algorithm minimization of page wait time is only achieved through *minimization of the number of page faults* and suggest to direct the study of paging algorithms towards the study of replacement policies. [3]. However, this correlation only holds in case page wait times are fixed. Yet, Section 2.1.1 argues that page wait times greatly vary on the page types of both faulted page and victim. This finding suggests that the suitability of a page as victim not only depends on the likeliness of future references but also on the necessary effort to move it out of memory. Yet, this work adopts minimization of the number as page faults as criterion for optimal page replacement for the sake of simplicity.

2.2.1 Page Replacement Algorithms

Let $S = \{n \in N \mid f(n) \neq \mathit{undefined}\} \subseteq N$ be the subset of pages residing in memory. The page replacement algorithm selects a victim $n \in S$ to be freed for use by a different page[5].

In general, there are two classes of page replacement algorithms. The first class of algorithms are *offline algorithms*. In this context an algorithm qualifies as offline algorithm in case it has a priori knowledge of the entire reference string. The reference string may for example be obtained from a pre-run of the program in question [4]. The remainder of this section includes a discussion on how knowledge about future references lead to an optimal replacement algorithm.

The second class of replacement algorithms are *online algorithms*. Such algorithms are given no information about future reference. They base replacement decisions on the past history of references. Online algorithms resort to making assumptions about page reference behavior. Furthermore, they employ heuristics to estimate the probability of future references using the past history. In other words, they extrapolate future reference behavior from past reference behavior.

[5]In order to free a page frame the replacement algorithm must move every page the frame is allocated to out of memory. In case the algorithm selects a shared page frame $m \in M$, it effectively turns all pages $n \in N$ with $f(n) = m$ into victims. Hence, all potential victims implied by a certain page frame must be taken into account during selection.

Examples of online algorithms include random page replacement, least recently and least frequently used page replacement.

Optimal and Worst Page Replacement

The *Optimal page replacement* (OPT) algorithm is discussed in great detail in [4]. It is a offline algorithm as it requires a priori knowledge about the reference string and operates as follows: In case one of the pages in S will not be referenced any further it is an obvious victim. However, if there is no such page OPT selects page $n \in S$ which is referenced only after any other page $n' \in S$ with $n \neq n'$ has already been referenced before.

Analogous to OPT the *Worst page replacement* (WORST) algorithm selects page $n \in S$ which is referenced *before* any other page $n' \in S$.

Random Page Replacement

The *Random page replacement* (RAND) algorithm capitalizes on the assumption that page references are evenly distributed among all pages $n \in N$ [4]. In other words, at any point in the reference string a reference to a page $n \in N$ is as likely as a reference to any other page $n' \in N$ with $n' \neq n$. Because of $S \subset N$ this also holds for any pages of S. In that case information for a good candidate can neither be learned from the past nor the future of references. Hence RAND selects a random victim among the pages in S.

Working Set and Least Recently Used Page Replacement

Denning et al. [8] present another assumption about page reference behaviour: the working set model. The authors define the working set of pages as the minimum collection of pages that must reside in memory for a process to operate efficiently without unnecessary page faults and argue that it is a subset of the set of least recently used pages. Their assumption about page reference behavior is that once a page is referenced, further references to the same page are to be expected and hence the working set is a good predictor for the immediate future of references [8]. Thus, the working set model suggest to replace pages which are not member of the working set whenever possible.

Memory is not necessarily larger than the working set and hence all pages in S may be in fact part of the working set. However, under the assumption that there is at least one page in S that is not part of the working set, according to Denning et al. [8] that page must be the least recently used page of S.

The *least recently used page replacement* algorithm (LRU) always replaces the page of which did not receive a reference for the longest time of all pages in S. For that purpose it maintains as control state an ordered sequence of S [3]. It is ordered according to the last reference to the pages with the most recently referenced page at the front and the least recently referenced pages at the end. An example is $S = \{y_1, \ldots, y_k\}$. In case of an reference to page $x = y_i \in S$ the sequence is updated to $\{x, y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_s\}$. In case of a page fault $x \notin S$ the least recently referenced page y_k is replaced and the sequence updated to $\{x, y_1, \ldots, y_{k-1}\}$. However, Sleator et al. [27] find evidence that LRU performs poorly compared to OPT.

Belady et al. [4] find that the complete reordering may be costly but also non-essential. The authors argue that instead of an ordered sequence a partition of S into a set S_1 of recently referenced pages and a set $S_0 = S \setminus S_1$ of pages that have not been referenced recently is of interest. Initially all pages are part of the set of unreferenced pages ($S_0 = S$, $S_1 = \emptyset$). Any time a page $x \in S_0$ is referenced it is moved to set S_1. In case a page needs to be pushed a victim is selected among the pages in S_0. In the event that S_0 becomes empty ($S_0 = \emptyset$, $S_1 = S$) the sets are exchanged and the process starts again.

LRU-K Page and Least Frequently Used Page Replacement

O'Neil et al. [23] present another assumption about page reference behavior[6] that may help to extrapolate the immediate future of references from the past history. The authors assume that the reference string of a program may be partitioned into relatively long disjoint phases with the following property: Within a phase the probability b_n that the at position t referenced page is $n \in N$, is independent of position t. Put differently, at a point $t - 1$ in the reference string the probability that page n is referenced next (at position t) is constant over the course of a phase [23].

This assumption suggests the following heuristic: Always replace the page $n \in S$ which has the lowest reference probability b_n of all pages in S since for any upcoming reference in the current phase it is the least likely target. However, this leaves the question on how to learn the reference probability of the pages in S. O'Neil et al. [23] propose to estimate the expected reference interarrival time

[6]The authors limit their attention to database programs.

$I_n = b_n^{-1}$ instead of estimating n_p directly. The authors present the *backward K-distance*, the backward distance to the K^{th} most recent reference to a given page. The page $n \in S$ with the maximum backward K-distance is assumed to also have the maximum interarrival time and is therefore replaced[7].

The *least frequently used page replacement* algorithm (LFU) operates under the same assumption about reference behavior. However, it employs a different heuristic to estimate the reference probability. Instead of metering the backward K-distance, LFU counts the number of references to the individual pages. In case the duration of the reference string subsequence in which a certain number of references to a pages n occurred is known, it is possible to estimate the reference probability of n. On the other hand, in case the reference counts for every page in S are compiled from the same last K references, the page with the fewest references also appears to have the minimum reference probability and is therefore replaced.

2.2.2 Page Replacement and Multitasking

Multitasking is the concept of executing multiple tasks (processes) concurrently within a single computer system. Since multitasking does not necessarily imply the ability to run processes in parallel[8] it usually employs time sharing where each process executes only for short time slices one at a time. At the end of a time slice a process looses control of the CPU and another process is dispatched to start one of its slices. The entire set of processes running concurrently is called the *workload*. Paging contributes memory isolation to multitasking. Each process executes in its own exclusive virtual address space specified by its private set of page tables. Another important aspect of multitasking is the method by which processes are allocated time slices, referred to as *scheduling*.

In a scheduled environment memory accesses are generated on behalf of possibly many processes. References to an individual address space are clustered in time according to the allocation of time slices to corresponding processes. Hence, the definition of the reference string is extended so it does not only include bare references to pages but also name the address space a reference belongs to, e.g., $(a_1, r_1), \ldots, (a_1, r_x), (a_2, r_1), \ldots, (a_2, r_y)$ with address spaces a_i and references $r_i \in N_{a_i}$.

Under the assumption that scheduling operates independently of paging it produces an extended reference string which serves as input to the paging algorithm.

[7]Maximum I_n is equal to minimum b_n.

[8]The ability of a computer system to run processes in parallel is referred to as *multiprocessing*.

However, in case of a page fault, the page replacement algorithm has two options. It either considers all loaded pages as potential victims or only loaded pages belonging to the faulting address space. The former option is referred to as *global replacement*, the latter as *local replacement*. As discussed in Section 2.2.1 the optimal replacement policy pushes the loaded page which is referenced only after any other loaded page has already been referenced before. Hence, the optimal candidate does not necessarily belong to the faulting address space.

Replacement algorithms employ heuristics based on assumptions about page reference behavior in order to overcome the lack of knowledge about future references. However, those assumptions only apply to behavior (reference string) of individual processes, not page reference behavior of an entire workload running concurrently under scheduling. In other words, time sharing obfuscates page reference behavior. Thus, the aforementioned replacement algorithms are at best suited to select a local victim. This finding suggests a two step process to handle page faults in a multitasking system: First, select a victim address space, possibly with additional information about future scheduling. Second, rely on a suitable replacement algorithm to select a victim locally.

2.3 Dynamic Miss Ratio Curve

Zhou et al. [30] offer another point of view to memory management. The authors stress that in an multitasking environment, virtual memory management must address in fact two core issues. On the one hand, it is the purpose of replacement algorithms to select a victim page for replacement, that is, to to determine *what* pages should be evicted. On the other hand, it is also necessary to determine how to allocate page frames to processes so the overall system performance meets some criterion of optimality. The authors refer to this aspect of efficient memory management as *memory allocation* [30]. In fact, this point of view matches the two step approach to paging in a multitasking environment (Section 2.2.2).

Zhou et al. [30] focus on the memory allocation problem. They argue that additional memory should be allocated only to processes whose performance significantly benefits from extra memory[9]. To that end, the authors suggest to employ the *dynamic miss-ratio curve* (MRC) to encode the dynamic memory needs of processes. The MRC plots the process's cache miss-ratio of a process against varying amounts of physical memory, e.g., varying number of page frames. In this context, the cache miss-ratio refers to paged memory as a fully associative cache.

[9]By implication, the paging algorithm should withdraw memory only from processes that do not perform significantly worse with fewer memory.

Furthermore, the authors suggest to reassign page frames to the process with the highest gradient in the MRC at its current number of allocated page frames.

2.3.1 Mattson Stack Algorithm

The authors further elaborate on the *Mattson stack algorithm* [21] as a method for dynamically tracking MRC at runtime. The Mattson stack algorithm simulates the processing of a page reference string by fully associative caches (e.g., paged memory) of various sizes simultaneously and outputs the number of misses for every size. It is based on the inclusion property[10] for cache replacement algorithms which applies to both LRU and LFU.

The Mattson stack algorithm uses a stack to store the pages referenced so far. The algorithm orders elements in the stack according to the expected future reference behavior as indicated by the underlying assumption and heuristic. In case of LRU replacement the most recently used page is on the top of stack while the least recently used one is at the bottom. For each reference in the reference string the Mattson stack algorithm performs the following three steps:

1. Search the referenced page within the stack. If the page is the i^{th} element from the top its stack distance is i. If it is not in the stack its stack distance is ∞.
2. Increment a hit counter for stack distance i (respectively ∞).
3. Add the referenced page to the stack if necessary and reorder the elements in the stack to reflect the changed "priorities". In case of LRU replacement the newly referenced pages goes to the top of the stack thereby moving all pages formerly on top of it down by one position.

According to the inclusion property, at any time, the first m pages from the top of the stack embody the set of pages the simulated replacement algorithm allocated a page frame in case there are exactly m page frames. Hence, the number of misses (page faults) for a specific number of page frames m is given by the sum of all hit counters for stack distances $d > m$. However, the ordering of the pages in the stack does not depend on the size of the stack. The Mattson stack algorithm, therefore, simulates paging behavior for any given number of pages frames simultaneously.

[10]The inclusion property states that a replacement algorithms given memory of size $k + 1$ never selects a page as victim that the same algorithm given memory of size k or less rules out. In other words, given more memory an algorithm always includes at least the set of pages that it includes given less.

2.4 IA-32e Architecture Memory Management

The Intel IA-32e architecture is a widely used 64 bit architecture also known as Intel64 or AMD64. The IA-32e architecture's memory management facilities implement demand paging virtual memory. The Intel Developer Manual [14, Vol. 3A 3] refers to virtual addresses as discussed in Section 2.1 as *linear addresses*. In the IA-32e architecture the page size is usually 4 KiB. Hence a page comprises $4096 = 2^{12}$ linear addresses. A linear address is 64 bit wide and subject to the *canonical form requirement*. The canonical form requirement limits usable linear address space to 48 bit since at most 256 TiB of linear-address space may be accessed at any given time. It states that for a linear address to be in canonical form bits 48 through 63 must be a copy of bit 47. Hence, it partitions the linear-address space into a 128 TiB bottom half and a 128 TiB top half. The bottom half contains addresses with the 47th bit cleared, starts at address `0x0000000000000000` and ends at `0x00007fffffffffff`. The top half contains addresses with the 47th bit set, starts at `0xffff800000000000` and ends at `0xffffffffffffffff`. All other addresses are uncanonical and therefore illegal.

IA-32e implements page maps as hierarchical data structures residing in physical memory, referred to as *page tables* [14, Vol. 3A 4]. Pages tables function as interface between the OS and the hardware. On the other hand the hardware Memory Management Unit (*MMU*) translates linear addresses transparently using the page tables. The MMU also enriches said page tables with access information and, in case of a page fault, interrupts normal execution and invokes the appropriate OS fault exception handler with details about the nature of the fault. On the one hand the OS implements the fetch, placement and replacement policies, that is manages pages as well as page frames and populates the page tables accordingly. Although the OS is not generally involved in performing memory accesses and therefore cannot directly learn what pages are in use, it polls the hardware supplied access information within the page tables.

The IA-32e architecture uses the 52 most significant bits[11] of a linear address to identify the corresponding page table entry (*PTE*). The entry contains some flags for allocation information and potentially the physical address of the page frame that the corresponding page occupies. The format of page table entries is further discussed later on in this section. The 12 least significant bits identify the specific address within the page and are called *page offset*.

[11]The canonical form requirement effectively reduces this to the 36 lower bits of the 52-bit chunk.

Page Table Entry Format

Figure 2.2 outlines the format of an IA-32e page table entry that maps a 4 KiB page. The bit at position 0 is called the *P bit*. It encodes whether a page is present in physical memory or not. In case it is set the page currently resides in memory and bits at position 12 to 47 hold the 36 bit address of the page frame allocated to it. In case its cleared the page is not present resulting in a page fault.

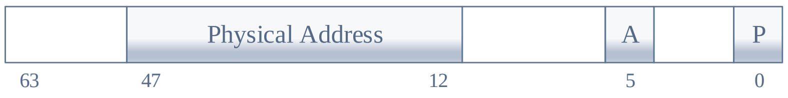

Figure 2.2: Scheme of the IA-32e architecture page table entry format for 4 KiB pages.

Also of interest is the bit at position 5. Is is called the *accessed flag*. The IA-32e architecture provides the flag for use by memory management software such as paging algorithms to manage the transfer of pages into and out of physical memory [14, Vol. 3A 4-31]. The accessed flag is set every time the given page table entry is used to translate a linear address to a physical address. It is also referred to as a *sticky* bit since it is set by the hardware but never cleared. Figure 2.3 illustrates the usage of the accessed flag. Every time the CPU generates a linear address the MMU sets the accessed bit of the PTE it used for translation. Afterwards the requested data is retrieved from the resulting physical address, effectively causing an access to the page frame that currently holds the linear address' page. Hence, a PTE indicates whether or not its corresponding page and therefore also corresponding page frame has been accessed.

Figure 2.3 also includes a shared page frame. A shared page frame is allocated to at least two pages. Each of those pages have individual PTEs referring to said page frame and also record references individually. Hence, in order to learn a certain page frame has been accessed, as necessary for page replacement in a page sharing environment, the OS must consult all PTEs referring to it since every single PTE may be used to access the page frame.

The operating system may read from as well as write to the accessed flag. Analogous the concept of a test-and-set operation, we refer to atomically reading and resetting the accessed flag as a *test-and-clear operation*.

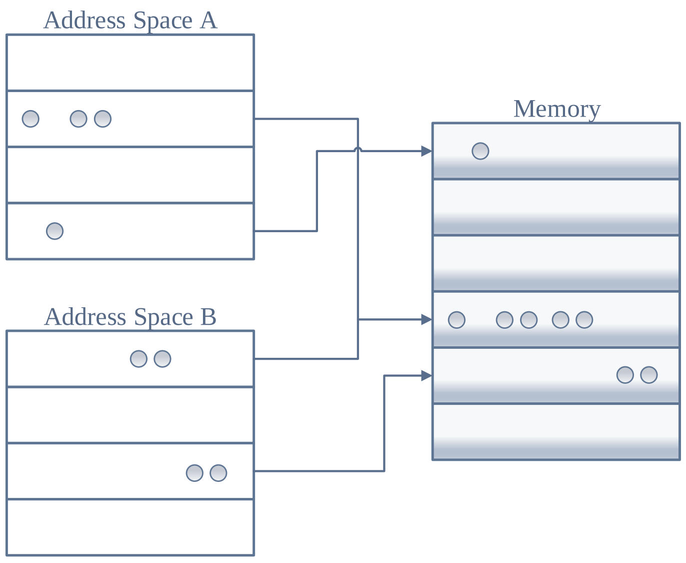

Figure 2.3: Example of pages of two processes A and B being mapped on physical page frames. One of the page frames is shared among two pages. Memory accesses going to the shared page frame therefore cause either the accessed flag of the page belonging to A or the page belonging to B to be set. Which flag is set depends on the linear addresses of the memory accesses.

2.5 Memory Tracing using Full System Simulation

Gröninger [11] investigate the possibility to make memory deduplication scanners more efficient. To this end, the author employs a toolchain called Simutrace for recording memory semantics and ongoing memory modifications of a live system previously presented in [25]. The toolchain records *memory writes and written data*, *memory semantics* and *system state* on a non-interfering level. This ultimately allows the author to analyze sharing opportunities, memory access frequencies, and access patterns [11]. Hence, the author actually presents a method to record a partial reference string, namely all the write accesses since they are one of reasons for memory content changing over time. Thus, the findings about data acquisition prove valuable for the study of page replacement algorithms.

The author evaluates a number of approaches and finds that full system simulation enables the analysis of system memory operations without interrupting the data flow and hence without changing memory content or timing. The author further compares a number of full system simulators and conclude that QEMU [5] is fast, functional correct and allows for memory inspection as well as OS introspection through further modifications. Section 2.5.1 discusses the findings concerning data acquisition in more detail.

The general design of the analysis framework the author presents follows a three step process, 1) system simulation and trace data collection, 2) independent data storage and 3) offline data analysis. The second step is accomplished using a *full data store* which provides a simple storage interface for producers and consumers but also hides storage details such as compression, pre- and post-processing. Section 2.5.2 examines the full data store in more detail.

2.5.1 Full System Simulation

Gröninger [11] identifies memory inspection and OS introspection functionalities, an accurate timing source to correlate system events with memory accesses and acceptable simulation speed as the three main requirements a simulator must meet.

First of all they discuss the advantages and disadvantages of different levels of simulation detail ranging from system call emulation over functional simulation to micro-architectural simulation. While *micro-architectural simulation* simulates architecture internals like for instance CPU pipelines and memory buses *functional simulation* exclusively simulates results of operations and hence is much faster.

Further, the author finds that functional simulation based on *dynamic binary translation* allows memory inspection and OS introspection. The first step of dynamic binary translation has guest code disassembled and translated into an intermediate representation (IR). It is later used to produce a semantically identical instruction sequence of the host's instruction set architecture. Since an IR typically include explicit load and store instructions the authors add interception points (*hooks*) for store instructions in order to track memory content.

In order to collect semantic information for guest OS introspection Gröninger present a *hypercall* interface. A hypercall refers to a magic instruction which calls right into the simulator without altering the system state. As magic instruction the authors employ get and set instructions for model specific registers included in the IA32 architecture and intercept them during the translation process. The hook also receives a pointer to a data buffer and a hypercall ID via general purpose registers. For example annotating a memory mapping operation with a hypercall allows to identify the initial memory content of named pages which is not generated through explicit store instructions[12].

Additionally the authors introduce a new *virtual instruction counter* into simulation. It counts only the guest instructions without considering the overhead caused by binary translation. However, every operation is counted exactly once and hence such a counter does not provide micro-operation accuracy.

Gröninger [11] chooses QEMU [5] as basis for the modifications since it is a functional correct, fast, open-source full system simulator based on dynamic binary translation.

2.5.2 Independent Data Store

Gröninger [11] decouples the simulation and data collection from data analysis using a full data store. It provides a storage interface for the data producing simulation and the data consuming analysis. The interface structures trace data logically and semantically.

The smallest unit of trace data is called a *trace entry*. A single trace entry corresponds to, for example, a single memory access or a single system event. However, every kind of system event uses a dedicated fixed-sized trace entry type to hold associated data. In order to correlate events and accesses, every type contains timing information.

Trace entries of the same type of event are grouped into fixed-size *segments*. Since the type of a trace entry implies a fixed-sized storage layout, a segment allows random access in $\mathcal{O}(1)$ to the trace entries it contains by index. Segments serve as smallest allocation, management and compression unit. They are presented to each client through a middle-ware API as a contiguous memory buffer.

On external storage, segments, in turn, are grouped into contiguous *streams*. There is a single stream for every type of trace event. Since segments have a fixed maximum size, they hold up to a fixed number of entries. Given a stream of n segments, by convention, the first $n - 1$ segments are fully populated, that is hold the maximum number of entries. Depending on the total number of entries in the stream, only the last segment is not fully populated. Analogous to segments and trace entries, streams allow random access to the segments they hold. However, retrieving a segment is an expensive operation since it involves disc I/O, in order to bring the segment into memory, and decompression.

[12]DMA is another example for memory content not detectable via explicit store operations.

The data store interface effectively allows random access to trace entries by type and index. The trace entry type determines the stream which stores the entry. The entry's index, together with the maximum capacity of segments, translates into the target segment to retrieve and an index into it.

Chapter 3

Analysis

Section 2.2 discusses the definition of a paging algorithm as state machine with states comprised of the page map and a control state. In this definition each page reference causes a state transition including control state and potentially page map updates. Thus, it implies that the state machine implementing the paging algorithms processes the entire page reference string as input.

As discussed in Section 2.4, in the IA-32e architecture performs memory accesses using linear addresses transparently, i.a., without operating system (OS) interference. The hardware memory management unit (MMU) translates linear addresses using the page tables. Only in case the MMU fails to translate a linear address, e.g., because the correspond page is not present in memory, it issues a page fault exception for the OS to handle. However, it is the OS that implements the paging algorithm. Therefore, according to the definition of paging algorithms as a state machine, the operating system needs to acquire the entire page reference string as input to the algorithm, not only a sequence of faulted page reference.

As a result, the original definition of a paging algorithm as a state machine processing the reference string does not apply to the IA-32e architecture. In IA-32e, the OS directly observes only page faults, not the entire reference string. Hence, state transitions and especially control state updates cannot occur on every memory access, but at least on every page fault. As an attempt to overcome this limitation, the IA-32e MMU caches page access information in the corresponding page table entry *accessed flag* and makes it accessible to the operating system and the paging algorithm.

With respect to the workings of the IA-32e architecture, we model paging algorithms as follows: Instead of processing the page reference string as input, a paging algorithm processes the sequence of page fault exceptions the MMU generates on the current state of the page table and the page reference string. However, in order to account for the accessed flags, each page fault exception does not only include the faulted page but also grants the algorithm access to the cached page access information in the form of optional test-and-clear operations.

This slightly adapted definition limits the information the paging algorithm acquires about reference behavior to the page access information cached by the page table. Therefore, the paging algorithm relies on the page table entries' accessed flags to record and reflect page access information accurately in order to make informed replacement decisions. This introduces the page table entry format and its semantics as a influencing factor into the study of paging algorithm performance. Hence, the performance of paging algorithms is mainly determined by two factors: *a*) how well its underlying assumptions about page reference behavior apply to a given program's reference string and *b*) the accuracy of the page access information the paging algorithm acquires through the page table entries' accessed flags.

However, the IA-32e architecture's page table entry format reserves only a single bit for caching page access information. It remains unclear whether or not this single bit constitutes enough cache capacity to record access information sufficiently accurate for the replacement algorithm to employ its heuristic. Hence, the scope of this thesis is to examine the influence of the IA-32e page table entry format on the performance of given paging algorithms. To this end, this thesis as a first step studies how the page table entry format affects the operating system's perception of page reference behavior. In other words, we try to learn whether or not the operating system's perception suffers from operating system's inability to observe the reference string directly.

An entirely different question is whether or not potentially inaccurate page access information in fact exerts negative influence on the paging algorithm's performance. In other words, this thesis as a second step explores whether or not given paging algorithms show potential to benefit from more accurate page access information.

To this end, this chapter proceeds as follows: Section 3.1 analyzes the page table entry's limited capabilities to cache page access information. It identifies two different kinds of inaccuracies: temporal inaccuracies, the inability to accurately reflect page reference timings, as well as counter inaccuracies, the inability to accurately count the number of references a page receives. Furthermore, Section 3.2 analyzes paging algorithm performance characteristics and explores means to explore the potential performance improvements due to more accurate page access information.

3.1 Page Table Entry Accessed Flag Inaccuracies

As discussed in 2.4 the IA-32e page table entry (PTE) format includes single bit for recording access information, referred to as the accessed flag. It is set in case the entry is used for translation and never cleared by the hardware. The accessed flag thus records whether or not the page for which it stores address translation information for has been accessed since its last test-and-clear operation.

This section models the usage of the accessed flag as a sequence of accesses to the PTE's page, that is set operations, intermixed with test-and-clear operations. A *test-and-clear interval* is the period of time between two consecutive test-and-clear operations on a given PTE. We illustrate the intermixed sequences as figures which resemble timelines of different pages. The figures depict memory accesses as circles, test-and-clear operations as squares. Throughout this section we assume that every page is allocated an exclusive page frame. Section 3.1.1 examines how the limited storage capacity of the accessed flags affects the operating system's perception of reference timings. Section 3.1.2 discusses to which extend the accessed flag allows to estimate the real number of page references.

3.1.1 Accessed Flag Temporal Inaccuracies

Denning et al. [8] find that memory accesses tend to cluster in time. This finding suggests to estimate the probability of a future reference to a page based on the recency of the page, i.a., the time passed since its most recent reference. For example, LRU evicts the page which has not been accessed for the longest time. However, without access to the reference string, LRU relies on the accessed flag ability to accurately reflect the recency of a page. Section 2.2.1 discusses an implementation of LRU which partitions loaded pages into two sets: S_0, the set of not recently referenced pages, and S_1, the set of recently referenced pages. Pages whose accessed flag is set are members of S_1, pages whose accessed flag is cleared belong to S_0.

However, the temporal accuracy of the accessed flag suffers from strong cohesion to test-and-clear timings and interleaving with page references. The accessed flag encodes whether or not the page is referenced within a test-and-clear interval. Hence, comparing pages on the basis of test-and-clear intervals of different durations proves ineffective. Figure 3.1 shows a timeline for two pages n_1 and n_2. n_1 is actually referenced more recently than n_2. Still, its most recent reference occurs before the relevant test-and-clear interval and so the relevant test-and-clear operation returns a cleared flag. In case the paging algorithm's control state does

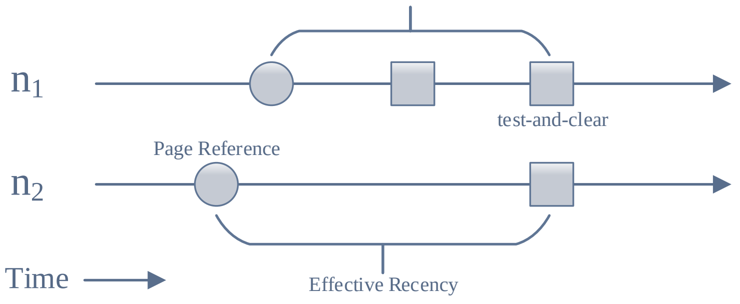

Figure 3.1: Timeline of two pages n_1 and n_2. The diagram depicts page references as circles and test-and-clear operations as squares. The effective recency is the period of time between a current test-and-clear operation and the last reference to the target page. In this example, the two most recent test-and-clear operations fail to reflect that n_1 is in fact more recently referenced than n_2

not reflect that n_1 has been successfully test-and-cleared lately, there is no way to determine that actually both pages have been referenced recently.

Unfortunately, test-and-clear operations even experience inaccuracies in case the paging algorithm test-and-clears all PTEs simultaneously. A test-and-clear operation returns a set flag for every page which was referenced at least once within the past test-and-clear interval. However, the actual recency of the last reference still strongly varies for different pages since this last references may occur at any point in the interval without changing the result of the test-and-clear operation. The last reference within an interval is henceforth referred to as *canonical reference*, the lapse of time between the canonical reference and the end of the interval as *effective recency*. Figure 3.2 illustrates this concept. Pages n_3 and n_4 are both referenced within the last interval but n_3's canonical reference occurs earlier than n_4's canonical reference. Belady's [4] definition of the sets S_0 and S_1 holds but the effective recency may vary among their members. Hence, the accessed flag cannot distinguish between the recency of two recently referenced pages. Information on the timing of the most recent reference to a page is lost. However, since the effective recency cannot grow beyond the duration of a test-and-clear interval, shorter intervals narrow down the effective recency of pages more precisely.

Both issues stem from the relative definition of recency associated with the accessed flag. The first issue is not severe with respect to our model of paging algorithms because it allows to paging algorithm to test-and-clear pages simultaneously. Furthermore, the paging algorithm may encode the timings and results of earlier test-and-clear operations in a more advanced control state. However, the

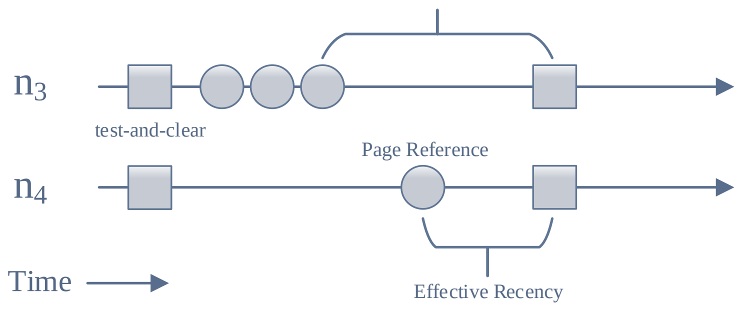

Figure 3.2: Timeline of two pages n_3 and n_4. The diagram depicts page references as circles and test-and-clear operations as squares. The effective recency is the period of time between a current test-and-clear operation and the last reference to the target page. In this example, the two most recent test-and-clear operations fail to reflect that n_4 is in fact more recently referenced than n_3, although no redundant test-and-clear operation are issued.

accessed flag's inability to reflect the effective recency of the pages cannot be mitigated as easily. A single bit accessed flag only allows the paging algorithm to determine a page's recency with a maximum accuracy of an interval duration.

3.1.2 Accessed Flag Reference Counting Inaccuracies

Other findings suggest to replace pages based on the frequency of reference to them [23]. As discussed in Section 2.2.1 LFU identifies the page with the lowest reference frequency as the page with the fewest number of accesses within an interval of past history. Hence, LFU requires the ability to count references to pages individually. To this end, LFU employs the accessed flag as a reference counter. It is effectively used to learn how many references occurred since the last test-and-clear operation.

However, the accessed flag implements only a single bit and therefore saturates after the first reference. Further references go unnoticed. Again, the information the accessed flag captures is effectively reduced to the presence or absence of a reference within an interval. Hence, LFU implemented with an one bit accessed flag effectively resembles LRU with an one bit accessed flag. Figure 3.3 depicts the timeline of two pages n_5 and n_6. n_6 is referenced three times in total within the interval while page n_5 is referenced only once. However, the single access bit fails to reflect the different reference counts and LFU is unable to distinguish the two pages in terms of reference frequency.

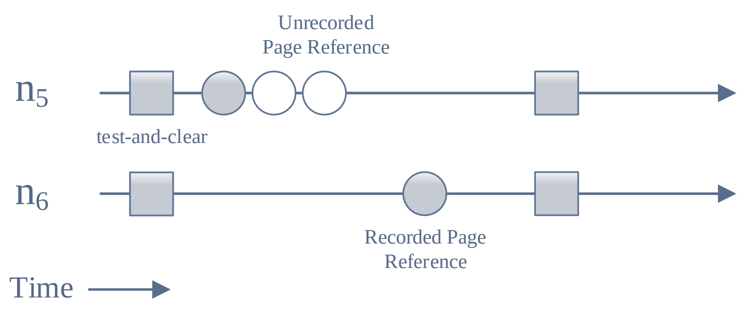

Figure 3.3: Timeline of two pages n_5 and n_6. The diagram depicts unrecorded references as empty circles. n_5's accessed flag fails to record the last two references because it already saturates after the first one. Therefore, the test-and-clear operations cannot reveal that n_5 is more frequently referenced than n_6.

LFU is also vulnerable to test-and-clear intervals of different duration. Comparing pages based on reference counts bears limited relevance in case the counters captured access information for different subsequences of the reference string. However, an approach to overcome this vulnerability is to test-and-clear pages simultaneously or to extend the control state to store test-and-clear timings so the paging algorithm may learn individual interval durations.

The counter inaccuracies stem from the limited counting capacity of the accessed flag. This shortcoming cannot be mitigated by fixed test-and-clear intervals nor by extending control state. Still, analogous to the temporal inaccuracies, the operating system's perception of page reference behavior benefits from smaller test-and-clear intervals since they reduce the likelihood of saturated counters.

3.2 Potential Improvements in Paging Algorithm Performance

Section 3.1 outlines how the limited storage capacity of the IA-32e accessed flags deteriorates the accuracy of page access information they provide to the paging algorithm. However, the actual impact on the performance of a given paging algorithm remains unclear. In other words, another question is whether or not more accurate page access information allows to improve page replacement decisions. One way to answer this question is to extend the page table entry format to provide more accurate page access information, to provide the paging algorithms with accurate page access information, and to examine the resulting performance for improvements.

To this end, Section 3.2.1 analyzes means to assess the performance of paging algorithms and to identify potential for improvement. Section 3.2.2 outlines possible extensions to the IA-32e page table entry format so it provides more accurate page access information.

3.2.1 Paging Algorithm Performance Assessment

Section 2.2 defines the objective of a paging algorithm as the minimization of page faults. This suggest the number of page fault a paging algorithm generates on a given input as indicator for performance. However, when assessing the applicability of more accurate memory access information for page replacement, two further aspects are crucial: First of all, online page replacement algorithms employ assumptions about page reference behavior in order to extrapolate the probability of future references from the past history. However, each program, that is a process to be, shows individual page reference behavior based on the algorithms and data structures it uses. Thus, page replacement algorithms perform differently well on different programs in execution, i.a., produce more or less page faults. Hence, the performance of a page replacement algorithm cannot be assessed based on its applicability to the reference behavior of a single program.

Second, we cannot assess a paging algorithm's performance based on the absolute number of page faults it generates on various inputs. Thus, an evaluation requires well defined baselines for relative comparison. Section 2.2.1 establishes OPT as the optimal page replacement algorithm. Furthermore, it introduces WORST, the worst possible replacement algorithm, and RAND, a trivial algorithm. The ultimate objective of an online replacement algorithm is to match the performance of OPT. In case LRU or LFU already performs close to OPT on a given input, there is few room for improvement. Hence such input does not allow to explore possible improvements based on more accurate page access information. In case a paging algorithm performs considerably worse than OPT for a given program, there is much room for improvement. Such programs are ideal for our performance improvement assessment.

Even if there is much room for improvement, it still remains unclear whether the algorithm performs worse because the lack of accurate information or because its underlying assumption does not apply to the program's page reference string. However, the random replacement algorithm RAND may help to estimate the extend to which an assumption about page reference behavior applies to a given program's reference string. The fact that RAND performs better than a given algorithm is evidence that the underlying assumption does not apply properly. Since more accurate page access information removes more randomness from the paging algorithm's decisions, it seams unlikely that the algorithm benefits from more accuracy. Another reason for comparing paging algorithms to RAND is efficiency. Assuming there is an efficient implementation of RAND, the additional effort the operating system invests to run a more complicated algorithm is in vain, in case the complicated algorithm does not perform considerably better than RAND. Hence, a suitable program for evaluating the applicability of more accurate information causes the algorithm under assessment to perform considerably worse than OPT, but at the same time better than RAND.

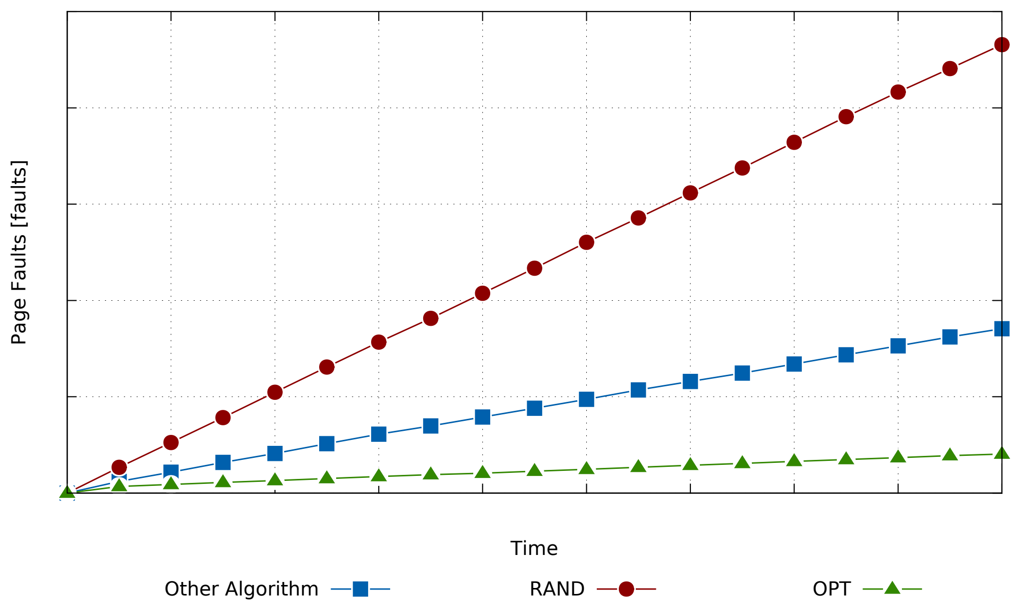

Figure 3.4: Example performance assessment. The x-axis shows the execution time of a selected process. The y-axis show the accumulated number of page fault the paging algorithm generates for a fixed number of page frames allocated to the process.

Figure 3.4 illustrates an example performance assessment of three paging algorithms. It shows some arbitrary paging algorithm under assessment, OPT and RAND. The diagram shows the execution time of a selected process on the x-axis. The y-axis shows the accumulated number of page fault the paging algorithm generates for a fixed number of page frames allocated to the process. It shows the number of accumulated page faults instead of the page fault rate because this way, it is easier to determine which algorithm performs best in the end. The curves of the algorithm under assessment should run below the curve of RAND, but above the curve for OPT. Therefore, the example program renders a suitable program for our performance assessment.

3.2.2 Extended Page Table Entry Format

Section 3.1 discusses two different types of inaccuracies page table entry accessed flags suffer from: temporal inaccuracy and counter inaccuracy. Since LRU is primarily based on learning the recency of pages and LFU relies on the ability to count page references, the two types of inaccuracies affect these algorithms to a special degree. Therefore, we focus our efforts on assessing these two algorithm in the light of the accuracy of page access information. To this end, this section outlines two extensions to the IA-32e page table entry format which are design to provide more accurate page access information in order to study whether or not this improves the performance of either LRU or LFU.

Least Frequently Used Replacement

LFU uses accessed flags as reference counters in order to learn the reference frequencies of the pages. As discussed in Section 3.1.2, the capacity of an one bit counter is very limited and references go unnoticed because it saturates after the first reference. An extended PTE format increases the number of bits reserved for the accessed flag in order to enable it to increase the counter's capacity. However, it remains unclear how many additional bits the extended PTE format should feature exactly. This effectively boils down to the question of up to which number of bits LFU actually benefits from the additional capacity.

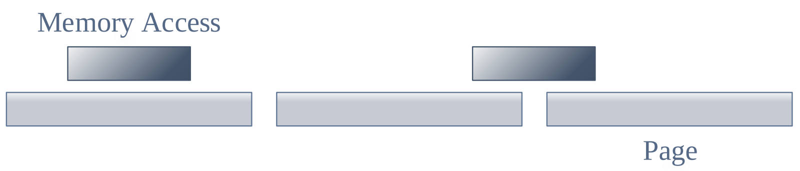

Figure 3.5: Two example memory accesses operating on three adjacent pages. The diagram depicts memory accesses as dark rectangles, pages as light rectangles. The with of the rectangles denotes the hypothetical size of both accesses and pages. The left memory access does not causes only a single page references as all data is located on a single page. The second memory access causes references to two adjacent pages since the data is spread on both pages.

However, in order to answer this question, we must first elaborate on how the MMU is supposed to count references exactly. According to Section 2.4, the IA-32e architecture MMU sets the accessed flag every time the corresponding PTE is used for address translation, i.a., every time a memory access stretches over a linear address belonging to the page in question. Furthermore, the Intel Developer Manual states that in IA-32e the fundamental data types are bytes, words, doublewords, quadwords, and double quadwords ranging in size from one to 16 Byte [13, Vol. 1 4-1]. Although alignment improves performance, data types do not have to be aligned in memory. Therefore, it is possible for memory accesses to affect two adjacent pages in case the datum is not aligned in memory and is located to the last linear address of a page. Figure 3.5 gives an example. We consider such a memory access as two page references, one reference to each page.

Least Recently Used Replacement

LRU uses accessed flags in order to learn the recency of a page, i.a., the time passed since its last reference. As stated in Section 3.1.1 a single bit accessed flag reduces the recency resolution to the duration of a test-and-clear interval. One approach to increase the accuracy is to overcome the correlation between test-and-clear operation and reference timings.

To this end, an extended PTE format enables the page table entry to store exact reference times instead of only indicating whether or not a reference occurred. To be exact, said extended PTE format extends the accessed flag so it is capable of storing a full timestamp. Furthermore, we assume the MMU to update the timestamp every time it uses the PTE to translate a linear address. This allows the paging algorithm to learn the effective recency of a page completely accurate.

3.3 Conclusion

This chapter analyzes the workings of paging algorithms relating to the limitations imposed by the IA-32e architecture. We find that the definition of a paging algorithm as a state machine processing the page reference string of a process does not apply to the architecture because it does not allow the paging algorithm to observe the reference string. Consequently, we outline a slightly adapted model. Instead of processing the reference string, our model has the paging algorithm process the sequence of page faults the paging hardware generates while performing memory accesses. In order to overcome the implied lack of page access information, the model grants the algorithm access to the cached page access information provided by the page table entries. However, the IA-32e page table entry format reserves only a single bit of access information, rending the page table entries essentially unable to accurately reflect page recency as well as to serve as accurate page reference counters.

Furthermore, we explore means to evaluate the potential of more accurate page access information to improve the performance of paging algorithms. We argue that running paging algorithms of interest on more accurate page access information and to examine the resulting performance for improvements is a viable option to do so. To this end, we conclude that the performance of a paging algorithm must be assessed based on the number of page faults it generates on a number of executions of different processes. Finally, we outline two extensions to the page table entry format that allow the page table entries to provide either accurate page recency information or accurate page reference counts, potentially benefiting LRU and LFU, respectively.

Chapter 4

Design

Section 3.1 finds that the IA-32 architecture's page table entry format eventually introduces inaccuracies into the page access information the page tables provide to the paging algorithm. The accessed flag fails both to reflect the effective recency of a page as well as to function as a page reference counter. The purpose of this chapter is to design a method of analysis which allows to quantify the inaccuracies as they arise in a real system. In this context, we choose to focus our efforts on page reference counter inaccuracies and leave the study of the page table's inability to reflect reference timings accurately to future work. To this end, Section 4.1 outlines the design of an analysis which allows us to observe both the actual reference counts of a real system as well as the number of references the IA-32e page tables in fact record.

Furthermore, we find that we cannot assess the applicability of more accurate page access information to improve paging decisions by quantifying the reference count inaccuracies only. Section 3.2 suggests to assess the applicability experimentally by running a number of paging algorithms on different workloads and comparing the results. To this end, Section 4.2 outlines the design of a paging simulation which allows us to simulate a number of paging algorithms running on page tables of varying capacities.

4.1 Detecting Reference Counter Inaccuracies

Section 3.1 discusses two inaccuracies caused by the limited capacity of the accessed flag. However, it remains unclear to which extend those inaccuracies distort the operating system's (OS) perception of page reference behavior of a certain process. We consider the following scenario: A process's address space consists of a number of pages. On the one hand, each page corresponds to a single page table entry (PTE) which is test-and-cleared zero or more times. On the other hand, each test-and-clear operation operates on a single PTE. The timings and target PTEs of test-and-clear operations are specified by the paging algorithm implementation, i.a., the operating system[1]. We consider them implementation details and therefore, a fixed, external effect. In this case, we define the operating system's perception of page reference behavior as the accumulated page access information it gathered through past test-and-clear operations. In order to assess the significance of the inaccuracies in page access information, we employ a three step approach.

First, we need an indicator to quantify the degree of inaccuracy of an individual test-and-clear operation. This indicator also allows us to compare the degree of inaccuracy of several test-and-clear operations. The second step is to accumulate the inaccuracies that the pages experience over the course of the entire process lifetime for every page individually. To do so, it is necessary to examine all PTE test-and-clear operations that operate on the PTE of a given page.

The third step is to quantify the distortion in the perceived page reference behavior caused by the accumulated inaccuracies. However, the operating system's perception of page reference behavior depends both on the timings and target PTEs of the test-and-clear operations as well as the degree of inaccuracy these test-and-clear operations experience. Since the timings and targets of test-and-clear operations are considered an external effect, this factor must be excluded from the assessment of the distortion. To this end, we define the baseline for the perception of page reference behavior as the perception the OS arrives at in case none of the test-and-clear operations suffers from any inaccuracy. Consequently, we refer to every perception that differs from this baseline because of test-and-clear operation inaccuracies as *distorted*. However, whether or not this distortion causes changes in the replacement algorithm's extrapolation of future page reference behavior depends not only on the degree of distortion, but also on the underlying assumption about page reference behavior.

Section 3.1.2 outlines how test-and-clear operations suffer from *counter inaccuracies* as the accessed flag fails to function as a reference counter. In the IA-32e architecture the accessed flag comprises only a single bit. Hence, it saturates after the first reference and it is unable to record any further references. The express the counter inaccuracy, the number of unrecorded references is a suitable candidate for an indicator of the degree of inaccuracy an individual test-and-clear operation

[1]Even in case the replacement algorithm is only allowed to issue test-and-clear operations on page faults, yet it not necessarily test-and-clears all PTE on every page fault.

experiences. The reasoning is the following: The operating system tries to learn how often a page is referenced. The inaccuracy increases as the number of unrecorded references increases. Then, in order to accumulate the inaccuracies, we sum the number of unrecorded references to the pages individually.

In order to count the number of unrecorded references for a given test-and-clear operation, we need to learn the number of references that occur between the test-and-clear operation of interest and the preceding test-and-clear operation to the same PTE, i.a., the number of page references falling into the test-and-clear interval of interest. Hence, it is sufficient to learn the reference string intermixed with the test-and-clear operations and their targets. Another option is to learn the reference string and the test-and-clear operations separately, but to include timing information for both events.

4.1.1 Data Acquisition

We define the operating system's perception of page reference counts as distorted, in case the perceived reference counts of individual pages differ from the genuine number of references the pages receive. In order to assess the accuracy of the operating system's perception, we opt to observe the actual page reference counts of the pages of a single *target process* and to compare them to the OS's perception of these reference counts. To this end, we employ the following scenario: An operating system runs on an IA-32e architecture system. The OS runs the single target process. However, the target process is assigned too few page frames to have every page allocated a page frame to reside in. Hence, the target process produces page faults and pages have to be pushed to external storage. The operating system handles those page faults using a local replacement policy. In order to select victim pages, the OS tries to observe the page reference behavior of the target process. To this end, it test-and-clears the pages of the target process's address space. A data acquisition method suitable for a thorough analysis as described above must extract the following information from the outlined scenario:

a) the reference string of the target process' virtual pages including both reads and writes,

b) all test-and-clear operations issued on pages belonging to the target process and

c) a reliable and consistent timing facility in order to correlate the recorded events.

Gröninger [11] discusses two methods to collect memory accesses[2] of single processes. The first method is step-by-step execution. This method has any page of a process mapped as inaccessible and the page fault handler modified to trace accesses. This causes every instruction accessing memory to fault. After the page fault handler traced the access(es), it marks the requested page(s) accessible and restarts the instruction in single-step execution mode. After the instruction completes the debug exception handler marks the page(s) inaccessible again so any future instruction also faults. The IA-32e architecture also causes page fault exceptions for instruction fetches [14, 4-30 Vol. 3A]. Therefore, this method might be able to capture not only memory accesses triggered by the instruction flow but also memory accesses caused by instruction fetch operations. The second method is to utilize precise event based sampling (PEBS) [15, 18-16 Vol. 3B]. PEBS allows to call a handler routine after every memory access. This routine might be able to trace and record the preceding memory access. However, both methods suffer from significant overhead. Furthermore, they neither provide an out of the box timing facility nor a solution to efficiently record the tremendous amount of collected data. Additionally, both methods cannot provide means to capture test-and-clear operations.

Marathe et al. [20] present another method to record memory access traces of single processes. The authors employ dynamic binary rewriting in order to instrument single instructions or entire blocks of instructions, thereby realizing hooks within the instruction flow. However, Marathe et al. argue that software tracing incurs high overhead, rendering reasonable datasets infeasible. Thus, they resort to partial traces, tracing only a subset of the memory accesses. Additionally, Marathe et al. [20] outline how to compress and store trace data on stable storage. Yet, their approach also neither includes any timing facility nor allows to record test-and-clear operations. Altogether, their approach proves unsuitable for our scenario.

All approaches presented so far suffer from two recurring shortcomings: 1) They acquire only a subset of the required information, e.g. they fail to record test-and–clear operations. 2) They show significant overhead. Full system simulation offers the means to overcome at least the first shortcoming, insufficient data acquisition, as it easily allows monitoring one or more entire systems [25]. Magnusson et al. [19] and Yourst et al. [29] present micro-architectural simulators. Micro-architectural simulation simulates the inner workings of parts or the even the entire architecture. They allow to record memory accesses at a hardware level, i.a., within the simulated memory access facilities. They also provide a timing facility

[2]Collecting memory access is also sufficient since the starting address and the size of a memory access indicates the page references it causes.

operating at micro-operation accuracy. However, micro-architectural simulation fails to overcome the second shortcoming, significant overhead. The overhead produced by micro-architectural simulation renders the simulation of an entire system for the lifetime of the target process infeasible.

Simutrace

However, functional simulation is known to perform several orders of magnitude faster than micro-architectural simulation [28]. Section 2.5 outlines how Rittinghaus [25] and Gröninger [11] employ full system simulation based on function simulation to trace memory writes and internal system state of a live system on a non-interfering level. Their approach adds hooks to the binary translation step of the full system simulator QEMU [5] to collect hardware events, such as memory writes. Hypercalls allow the simulation guest to trap into the host and pass on internally collected system state information. Additionally, the authors also implemented an instruction counter in QEMU to annotate every traced event, externally or internally collected, with consistent timing information. Finally, all collected data is recorded in the form of trace entries to a data store to enable offline analysis.

This approach proves promising because it out of the box provides means to collect test-and-clear operations and a consistent timing facility. Furthermore, the existing memory hooks already include a previously unused read hook[3], thus allow recording the full reference string as required in this work.

In addition, hypercalls provide the means to record test-and-clear operations. On the guest's side hypercalls appear as extensions added to the guest OS kernel at certain points of interest. The purpose of those extensions is to collect required information, e.g., the page a test-and-clear operation is issued on, and to trap into the simulation host to pass on the collected information. The simulator then receives the information, adds timing information, records it to the data store and finally resumes the simulation guest.

While this approach can deliver all required information for our analysis, it also comes with a number of drawbacks. As discussed in Section 2.5.1 the virtual instruction counter does not allow to track time with micro-operation accuracy. It counts executed instructions in the instruction flow. Hence, every instruction is modeled to take the same time, that is the same number of cycles, to complete. In fact, the virtual instruction counter only provides only an indirect timing facility

[3]The authors intended to record memory content and therefore did not pay any attention to non-modifying operations, such as memory reads.

causing distortions in the temporal distance between events. Yet, this does not break the ordering of events. Hence, this drawback does not prevent us from reconstruction the intermixed sequence of memory accesses and test-and-clear operations.

QEMU, the full system simulator used in this approach, employs dynamic binary translation in order to achieve functional simulation [5]. It translates guest instructions to an intermediate representation (IR), mapping any load and store operations triggered by the instruction flow to explicit single IR instructions. As a consequence, this excludes any memory operations that do not appear in the instruction flow, e.g, instruction fetch operations. However, instruction fetches are effectively read operations on text segments. Therefore, the approach fails to provide the entire reference string.

In most cases[4], instruction fetches and operand fetch/write operations rarely target the same chunks of the address space. In other words, the distinction between instruction fetch and other memory operations usually also implies a segmentation of the address space into text segments, targeted only by instruction fetches, and data segments, used only for operands. Consequently, Simutrace records all accesses to data segments and no accesses to text segments, essentially blocking out any activity on one part of the address space without interfering with activity on the other part. This is an acceptable restriction.

Altogether, only the approach presented by Rittinghaus [25] and Gröninger [11] constitutes a ready solution capturing the required information, enriching recorded events with consistent timing information, and processing the recordings for offline analysis. All other approaches provide only partial or infeasible solutions. Therefore, we adopt Simutrace as our method for data acquisition.

4.1.2 Data Refinement

As discussed in Section 2.5.2, Simutrace stores every recorded event as a trace entry. Every kind of event, e.g., a memory access, corresponds to a dedicated trace entry format. Entries of the same format and therefore, events of the same kind, are grouped into streams. Entries of different format go into different streams. Hence, a stream basically forms the ordered sequence of events of a certain kind. This results in separate streams for memory access trace entries and test-and-clear operation trace entries. Since every trace entry is annotated with an instruction count, it is possible to reconstruct the intermixed sequence of memory accesses and test-and-clear operations.

[4]QEMU's binary translator is a prominent example of applications mixing text and data.

Simutrace employs QEMU as a full system simulator that supports the IA-32e architecture. QEMU runs an entire, potentially unmodified guest OS. In turn, the guest OS runs numerous processes, only one of. However, traces are captured in the simulation at the hardware-level, leading to the following four consequences:

1. Whether the guest OS employs local or global replacement is up to the guest OS.

2. The guest OS does not necessarily issue test-and-clear operations on pages belonging to the target process only. Hence, Simutrace records all test-and-clear operations, not only ones affecting pages of interest.

3. Simutrace captures all memory accesses the CPU emulator detects while processing the instruction flow. Therefore, it effectively records a sequence of intermixed memory accesses of not only the target process but all processes.

4. For the same reason, Simutrace also records all memory accesses of the guest operation system itself. In fact, it records a sequence of intermixed memory accesses of both all processes and the guest operating system itself.

Consequence 1 is OS specific and calls for an OS specific solution. Either the OS allows to configure which replacement policy to use or we run a guest OS that only supports local replacement.

According to consequence 2, Simutrace records all test-and-clear operations. However, we are only interested in test-and-clear operations for PTEs belonging to the target process's address space. We identify pages by their base virtual addresses. Since virtual addresses are not unique, they cannot identify the target address space. Therefore, beside target page and timing, Simutrace must also record an address space identifier so the offline analysis is able to identify remove operations to other address spaces. However, this address space identifier is also OS specific.

Filtering the Recorded Memory Accesses

According to consequence 3 and 4, the recorded sequence of memory accesses contains all accesses QEMU's dynamic binary translation step detects in the instruction flow. However, the instruction flow includes both memory accesses generated by processes as well as the guest OS itself. The questions remains how to remove the memory accesses of the other processes and the operating system[5].

The guest OS partly controls the instruction flow as it schedules processes to run on the CPU. QEMU emulates a single CPU [5]. Thus, only a single process executes at all times. As discussed in Section 2.2.2, this causes memory accesses to the same address space to cluster in time. In order to identify the page references issued by the target process we must identify clusters and the processes they belong to. However, instead of clustering randomly, the memory accesses cluster as the guest OS schedules processes to run on the CPU. Therefore, it is sufficient to record dispatch[6] events via hypercalls in order to identify clusters. The recorded dispatch events enable the offline analysis to reconstruct which process ran on the CPU at a given point in the simulation and hence, to remove accesses made by other processes from the recorded sequence of memory accesses.

This leaves the memory accesses the operating system performs. OS memory accesses also cluster. Interrupts and exceptions arrive in any context but must be handled by the OS. Therefore, clusters of OS memory accesses are either contained within a single cluster of process accesses or separate two distinct process clusters. Figure 4.1 gives an example of the entire filtering process. However, OS access clusters are not necessarily encompassed by dispatch events. Therefore, the offline analysis cannot identify those accesses using recorded dispatch events. This is no problem as long as the OS memory accesses do not fall into the timeslices of the target process. The offline analysis removes all accesses falling into the timeslices of other processes, even OS memory accesses. However, potential means to identify clusters of OS memory accesses depend on the implementations details of the guest OS.

4.1.3 Conclusion

In order to quantify the page reference counter inaccuracies as they arise in a IA-32e architecture computer system, we choose to reconstruct the interaction between the memory management hardware and the paging software. We indent the offline analysis to replay a intermixed sequence of memory accesses of a single target process and the operating system's test-and-clear operations targeting the page table entries of said pages. This allows us to learn both the actual but

[5]OS memory access potentially happen on behalf of a process, e.g., in case the OS copies file contents into a process's address space. However, it remains unclear how to identify whether an OS memory access is issued on behalf of a process or which process in fact benefits.

[6]The dispatcher puts scheduling decisions into effect as it prepares the CPU to run the selected process.

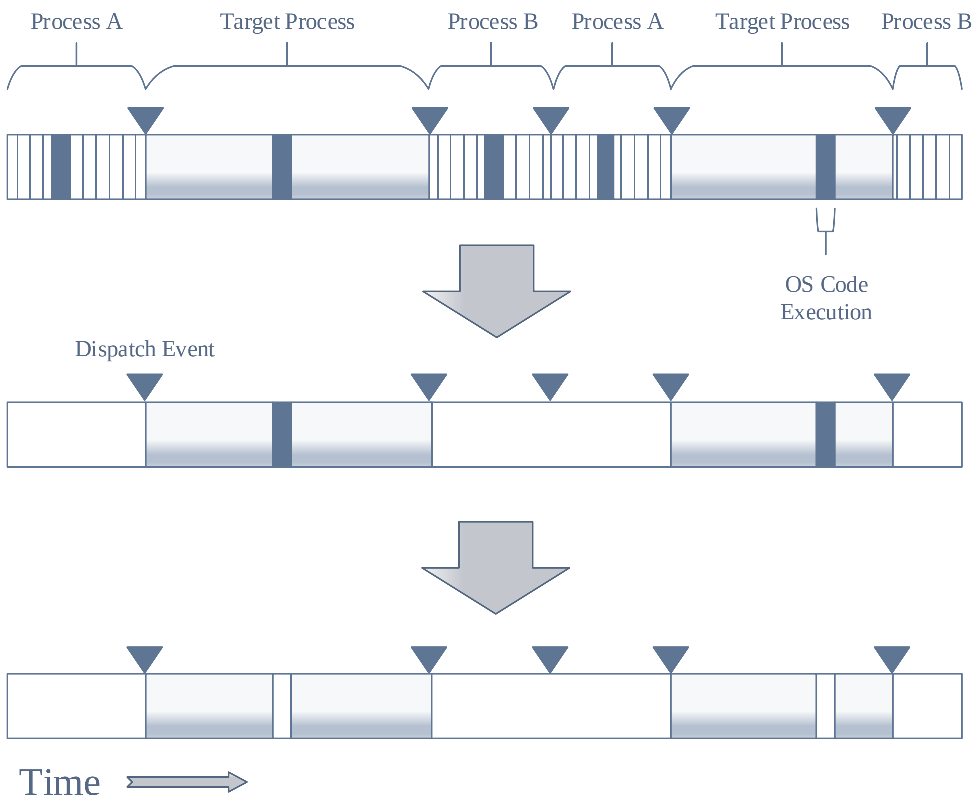

Figure 4.1: Example of the entire filtering process. The diagram shows an hypothetical memory access stream at the top. It shows periods of process memory activity instead of individual accesses. The recorded raw memory access stream is shown at the top. The stream contains periods of target process memory activity (lightly colored periods), periods of other processes' memory activity (striped periods) and periods of OS memory activity (dark periods). The diagram also illustrates recorded dispatch events as triangles. A first step removes all memory accesses not falling into a target process timeslice, including OS memory accesses. The second step removes the leftover OS memory accesses based on a OS specific solution.

potentially inaccurate results of the test-and-clear operations as well as perfectly accurate hypothetical results.

For the purpose of data acquisition we employ full system simulation and Simutrace for tracing memory accesses and test-and-clear operations as the simulation guest issues them. The scenario is the following: We run a yet to be selected operating system as guest in QEMU, execute a benchmark as target process and after it finishes, shut the guest down. Yet, Simutrace traces at a hardware level and therefore records all memory accesses, regardless of which process or even the guest operating system itself issues them. In addition, we add a hypercall to the guest operating system which records test-and-clear operations.

We need to refine the recorded data in order to extract the intermixed sequence of target process memory accesses and corresponding test-and-clear operations. To this end, we use dispatch events, which Simutrace already records, to identify the timeslices of processes running on the CPU. This allows us to remove all memory accesses not falling into the timeslices of the target process. Yet, we leave other consequences of full system simulation, such as causing memory pressure on the target process, to Chapter 5.

4.2 Simulating Paging Algorithms on Extended PTE Format

According to Section 3, the performance of paging algorithms is mainly determined by two factors: *a*) how well its underlying assumptions about page reference behavior apply to a given program and *b*) how accurately the paging algorithm perceives the actual page reference behavior. The first is expressed through the paging algorithm (e.g., LFU). However, the limited capacity of the IA-32e accessed flag introduces inaccuracies into the operating system's perception of page reference behavior. Section 3.2 suggests to simulate page replacement using LRU as well as LFU on reference string acquired from several programs given an extended PTE format in order to access the applicability of more accurate page access information.

However, several issues must be addressed before simulation. First, Section 4.2.1 discusses why memory accesses we acquire for the analysis of reference counter inaccuracies are also sufficient as input for our simulation. Second, Section 4.2.2 outlines a memory management model to be used as basis for our simulation. Section 4.2.3 finally presents the design of our paging algorithm simulation.

4.2.1 Data Acquisition

Before it is possible to simulate a paging algorithm, it is necessary to acquire page reference strings generated by a number of programs. Although, the least recently used page replacement algorithm (LRU) additionally requires timing information of memory accesses, it does not depend on exact page reference timings. LRU aims at replacing the least recently referenced page. In order to identify this page, a definite order of the page references is already sufficient. The reference string itself already provides this order.

A memory access given as start virtual address and size implies the page references it invokes. Therefore, the sequence of memory accesses a program produces is equally useful as the reference string. The sequence of memory accesses is also sufficient as input to the simulation. Section 4.1.1 already describes how to acquire the sequence of memory accesses of a target process using full system simulation and Simutrace. In addition, Section 4.1.2 outlines how to filter the recorded memory accesses so the implied reference string does not include any references performed by other processes or the operating system itself.

4.2.2 Memory Management Model

According to Section 2.1, paging requires a set of page frames, at least one virtual address space, i.a., a process, and a page map. The objective is to simulate local page replacement for a single process. Hence, there is only a single address space and a single page map.

Paging usually allows for page sharing. As discussed in Section 2.2.1, page sharing changes the paging algorithm's approach to replacement. In order to free a shared page frame, every page it is allocated to must be moved out of memory. This further complicates replacement decisions but does not offer any more insight on the page table entries capability to reflect page reference behavior. Additionally, the purpose of the simulation is not to find the minimum number of page frames necessary to run a given process efficiently. It is to assess whether or not more accurate page access information benefits the replacement decisions. Therefore, the paging simulation does not simulate page sharing. Every page is considered unique and allocated an exclusive page frame. Hence, moving a page out of memory effectively frees its former page frame for reassignment.

Page Fault Model

As discussed in Section 2.1.1, page faults occur in case address translation fails. Besides access permissions, address translation fails because a requested page is not present in memory, that is it has no page frame allocated to it. A page fault causes a delay in process execution referred to as page wait time. The page wait time depends both on the state of the selected page frame, as it determines whether or not a push is necessary, as well as the faulting page itself, as it might not require a pull from external storage. However, the paging algorithm influences the page wait time only through the victim it selects. The time it takes to prepare the selected page frame for the new mapping is constant for all page frames.

As discussed in Section 2.2, the paging algorithm's general objective is to minimize aggregate page wait time. Therefore, differences in page wait time suggest to select victims based on both expected future page reference behavior as well as the necessary effort to evict a page from memory. Only assuming fixed page wait times, leads to the conclusion that demand paging algorithms achieve minimal page wait time through minimization of the number of total page faults.

Yet, this thesis studies the applicability of more accurate page access information to improve the prediction of future page reference behavior. The quality of the prediction, in this context, is determined purely by the number of page faults the replacement algorithm produces given a reference string as input. Therefore, the simulation does not need to model the page wait time at all and can ignore whether a victim page must be pushed or not.

4.2.3 Simulation Design

A paging algorithm is the implementation of the three subpolicies: the fetch policy, the placement policy and the replacement policy (see Section 2.1.2). A simple demand paging algorithm implements a demand paging policy. It does not bring any page into memory until it is accessed for the first time and never moves a page out of memory unless it is inevitable. Furthermore, its placement policy places pages in arbitrary, free page frame or queries the replacement policy for one. Its implementation of the replacement policy, the replacement algorithm, is the only non-trivial component.

Section 2.3.1 introduces the Mattson stack algorithm as a method to simulate the behavior of an online demand paging algorithm. It performs the following three steps on every page reference: First, it determines the stack distance of the referenced page. Second, it increments a hit counter. As the third and final step, it reorders the elements of the stack according to the replacement algorithm.

The order of the pages in the stack effectively encodes an *allocation priority*. For an arbitrary number of page frames m, the Mattson stack algorithm allocates these m page frames to the top m pages in the stack. In case it reorders the stack according to the replacement algorithm, it changes the allocation priorities and potentially reallocates a page frame. Figure 4.2 given an example for $m = 4$. Moving a page with stack distance $i > m$ to the top of the stack, in fact allocates it a page frame. However, every page with stack distance $i' < i$ is pushed down

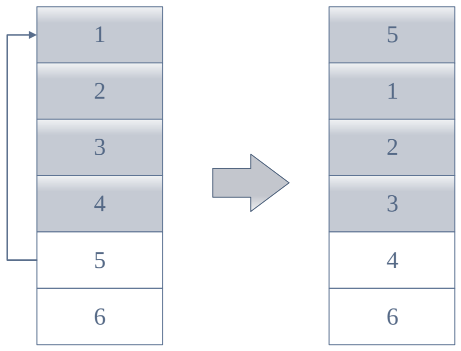

Figure 4.2: Processing of a page reference on a mattson stack for $m = 4$. Page 5 is being referenced. Its stack distance is $5 > 4 = m$. After it is moved to the top of the stack, page priorities shifted so that page 4 looses its page frame.

by one position. Hence, the page with stack distance $j = m$ looses its page frame and is moved out of memory. This newly freed page frame is the one allocated to the new page at the top.

Adapted Stack Simulation

The Mattson stack algorithm operates on the reference string. It uses every memory access to reorders the stack and therefore, to update the allocation priority it assigns to the individual pages. In case a page fault occurs, the stack already incorporates preprocessed information of past page reference behavior so the paging best decision is to replace the loaded page with the lowest priority.

However, in the IA-32e architecture the paging algorithm cannot learn the reference string directly. The paging algorithm may only query the page table entries for recorded page access information. Our simulation is essentially designed to answer the question whether or not a certain page table entry format is able to cache sufficiently accurate page access information for the paging algorithm to restore the correct allocation priorities.

This simulation employs an adapted stack algorithm. With respect to the hardware architecture it differs from the Mattson stack algorithm in the following three ways:

a) The simulation also includes the accessed flags of the pages. It updates the accessed flag of the referenced page as described by the PTE format.

b) It is given a definite number of page frames m. This is also the maximum size of the stack. Hence, a page fault occurs in case a page is not part of the stack.

c) It is only allowed to update the stack on a page fault[7]. In return, it is granted access to the accessed flags of the loaded pages.

The adapted stack algorithm still operates on the references string. However, only faulting references can be used to reorder the stack and therefore, update the allocation priorities. The simulation performs the following steps on every page reference in the string:

1. Determines whether or not the reference issues a page fault. In case it does, increments the page fault counter and proceeds to step 2. In case it does not, skips step 2 and proceeds to step 3 immediately.

2. Sorts the stack according to the state of the accessed flags and the replacement policy. To that end, test-and-clears every page in the stack and caches the results for sorting. Then replaces the page at the bottom of the stack for the requested page.

3. Finally, updates the accessed flag of the referenced page.

Figure 4.3 shows an example of an single page references. The adapted algorithm is given a fixed number of page frames in order to render the decision whether or not a page reference issues a page fault explicit. Furthermore, according to the IA-32e hardware limitations, the adapted algorithm may only update its stack based on the page access information provided by the (extended) page table entries. The page table entries effectively cache the page access information which the Mattson stack algorithm immediately incorporates into its stack.

However, depending on the evaluation baseline, there is another point of view towards the simulation. In case the baseline is set to the number of page faults the simulation reports for the IA-32e single bit accessed flag, replacing the accessed flag with an extended accessed flag allows to evaluate whether or not more accurate page information in fact reduces the number of page faults. In other words, given an individual page reference string, a replacement heuristic, and an extended page table entry format, the simulation allows to evaluate whether the heuristic in fact benefits from more accurate page access information or not.

[7]As a result, it looses its ability to simulate multiple memory sizes simultaneously.

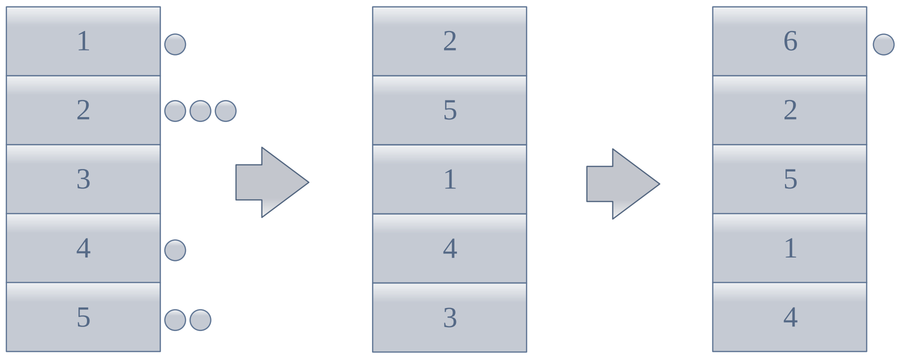

Figure 4.3: Example of the processing of a single page references to page 6 for $m = 5$. On the left the diagram depicts the stack right before it processes the upcoming references. The circles illustrate the state of the page table. The number of circles encode the number of page references the page tables counted since the last page fault. Step 1 determines that page 6 is not present in the stack and, since the stack is fully utilized, sorts the stack according to the LFU paging algorithm. Therefore, it test-and-clears all loaded pages. The new order is shown in the middle. It removes page 3, the page at the bottom of the stack, and in return loads page 6. On the right side the diagram shows the third step as it records the page reference to the page table.

4.2.4 Summary

In order to assess the potential of more accurate page access information to improve paging decisions, we employ a paging simulation similar to the Mattson stack algorithm for a single process. We find that the Mattson stack algorithm models the paging algorithm in accordance with the definition of demand paging algorithms presented in [3] as it reorders the stack at every page reference. Yet, this does not fit our model of paging algorithms which does not allow the paging algorithm to observe every memory access.

Our adapted stack simulation restricts the paging algorithm to reorder the stack only on page faults, however, grants it access to simulated page table entries to acquire cached page access information. This allows us to assess whether or not a certain page table entry format is able to cache sufficiently accurate page access information for the paging algorithm to restore the correct stack order before making a replacement decision. Since the simulation is designed to use the sequence of memory accesses of the target process as input, we are able to recycle the trace data the inaccuracy analysis produces.

Chapter 5

Implementation

In order to study the applicability of more accurate page access information, we have to consider two separate questions (Chapter 3): 1) to which extend the outlined page access information inaccuracies disrupt the operating system's ability to perceive page reference behavior accurately and 2) whether or not paging algorithms show improved performance, in case they are provided more accurate information. Chapter 4 presents two methods of analysis. To address the first question, we choose to reconstruct the interaction between IA-32e memory management hardware and the operating system (OS) paging software in the context of a benchmark process in order to detect page reference count inaccuracies using full system simulation and Simutrace [11, 25]. In order to address the second question, we employ a paging simulation similar to the Mattson stack algorithm [21] using a memory access trace also acquired from a benchmark process. This chapter presents implementation details for both designs. Later on, we select a number of benchmark algorithms and perform both analyses for each of them for evaluation. As of now, we keep referring to the benchmark process to be, as *target process*.

Since the data requirements for reconstructing the interaction between hardware and software subsumes the requirements of the paging simulation, we discuss the details of data acquisition and refinement only in the context of the former analysis. Section 4.1.1 and Section 4.1.2 leave a number of issues concerning data acquisition and refinement unaddressed. First of all, we choose Linux as the simulation guest running in QEMU because it is a widely used, free open source operating system which allows us to incorporate the necessary hypercall extensions without much effort. Another open question is how to cause the simulation guest to perform local page replacement on the target process, forcing it to issue test-and-clear operations on the target process's address space in the first place. To this end, Section 5.1 outlines how we use the Linux control group mechanism in order to confine the target process to a fixed but insufficient number of exclusive page frames. Afterwards, Section 5.2 discusses the data refinement steps necessary to extract the reference string intermixed with test-and-clear operations of interest from the trace date.

Finally, Sections 5.3 and 5.4 present the implementation details of the actual analyses, both the reconstruction of interaction between memory management hardware and paging algorithm as well as the paging simulation.

5.1 Enforcing Local Page Replacement in Linux

The scenario outlined in Section 4.1 aims at the question whether or not the IA-32e architecture's page table entry format enables the OS to reconstruct page reference behavior for the purpose of page replacement without witnessing the generated memory accesses directly. Page replacement becomes necessary in case there are not enough page frames available in order load all virtual pages simultaneously. However, Linux generally performs global page replacement. On a page fault, Linux does not only look within the running process's address space for potential victims, but also considers pages belonging to other processes. To this end, Linux aims at learning the reference behavior of all processes, not only our dedicated target process. Therefore, in order to conduct our analysis, we must both limit the target process's page frame allocation without causing other processes or Linux itself to produce page faults on their own.

The ideal solution for both issues is to cause Linux to voluntary limit only the number of page frames it allocates to the target process. This way, the target process produces non-demand page faults[1] but at the same time plenty of free page frames remain for the rest of the system so Linux does not need to reclaim one of the target process's frames to service a page fault of another process. Furthermore, this also enforces local page replacement. After the target process hits its maximum number of page frames, Linux cannot allocate it another one and therefore needs to free one already allocated to it. Linux already provides the means to do so in the form of control groups.

[1]Such page faults are also referred to as capacity page faults.

5.1.1 Cgroups

Linux *control groups* (cgroups) provide a mechanism for partitioning sets of processes into groups [22]. A cgroup contains zero or more processes and comes with a number of parameters to control the resource allocation of its processes. Cgroups allow to limit the allocation of a number of resources, in particular the memory allocation.

For our purposes we employ the *memory resource controller* [9] subsystem. This cgroup subsystem allows to limit the combined memory usage of the processes in the cgroup in granularity of bytes. In addition, the documentation [9] guarantees local replacement within memory control groups. Listing 5.1 illustrates how to set up a memory control group called "test" and to run a program as a member of the group using cgcreate, cgset and cgexec utilities.

```
cgcreate -g memory:test
cgset -r memory.limit_in_bytes=512K test
cgexec -g memory:test benchmark arg1 arg2
```

Listing 5.1: Creating, configuring and populating a memory cgroup.

For our purposes, we confine the target process to its own cgroup and restrict its memory allocation appropriately. Thereby, the exact allocation limit obviously depends on the selected target process's memory demands and page reference behavior.

5.2 Data Refinement for Linux Guests

In order to reconstruct the interaction between IA-32e memory management hardware and the OS' paging algorithm we need both the sequence of memory accesses the target process generates as well as the sequence of test-and-clear operations performed on its pages. However, Simutrace traces memory access on a hardware level and therefore records all memory accesses of all processes and the operating guest system itself. It also records via the hypercall interface all test-and-clear operations, regardless of the process the target page belongs to. Hence, we must filter both streams before we can run our analysis. To this end, Section 5.2.1 outlines how to identify and remove redundant memory accesses. In addition, Section 5.2.2 discusses how to filter the recorded test-and-clear operations target process pages.

5.2.1 Filtering the Memory Access Stream

Section 4.1.2 suggests to record dispatcher activity in order to reconstruct the allocation of timeslices to the processes. This allows to match memory accesses to processes since only the active process generates memory accesses at the time.

In general, the terms address space and process refer to the same manifestation of ongoing computational activity They appear in a 1:1 relation. Linux, however, knows no distinction between multitasking, the concurrent execution of multiple processes by rapidly switching between timeslices, and multithreading, a mechanism for processes to have multiple independent execution flows operating in the same address space. Linux implements kernel-level threads as independent processes sharing parts of their execution context, e.g., execute in a shared address space. Therefore, in Linux, processes and addresses spaces actually appear in a m:1 relation.

Linux identifies each process by an unique *process ID* (PID). The various threads of a multithreaded program appear as multiple processes, each identified by an unique PID. They perform memory accesses independently to the same address space. Although, it is sufficient to record the PID of dispatched processes in order to identify the active process, it is however not sufficient to identify which address space receives the memory accesses. The IA-32e memory management hardware identifies the active address space using the `CR3` register. The `CR3` register stores the physical address of the first table in the currently active page table hierarchy, referred to as the page global directory (PGD). More importantly, the `CR3` register together with a generated linear address identifies the page table entry (PTE) to use for address translation, therefore also the accessed flag to set. Since the dispatcher needs to set the `CR3` register to prepare a process to run, Linux also keeps track of the PGDs. For this reason, for every dispatch event, instead of the PID, we record the PGD .

In order to filter memory accesses or test-and-clear operations based on the active address space, first of all, it is necessary to determine the PGD of the target address space. In Linux, the `clone` system call creates new processes[2]. The new process is basically a clone of the caller and either executes in the very same address space or receives a shallow copy of the caller's address space. Furthermore, the exec system call allows a process to load a new binary for execution, thereby replacing the old address space. Hence, the execution of a new, potentially multithreaded program takes the following steps:

1) The process designated to start the program creates a clone process using

[2]Therefore, it is also used to create kernel-level threads.

the `clone` system call.

2) The clone loads the program's binary using the `exec` system call.
3) The now executing binary potentially spawns additional threads using the `clone` system call on its own.

In the above steps it is the second step that creates the target address space. Simutrace already records calls to the `clone` and `exec` system calls via hypercalls. For every invocation of `exec` it records both the name of the binary loaded as well as the PGD of the newly created address space. Given the name of the target process's binary, this allows the offline analysis to easily learn the target PGD to filter for.

In case the binary of the target process is known in beforehand, it is possible to apply filters based on the PGD already while recording. Since dispatch events and memory accesses occur in order, the last dispatch event seen indicates the active address space. In other words, the PGD set by the last seen dispatch event for example determines whether or not the following memory accesses belong to the target address space. Thus, it is possible to ignore unnecessary memory accesses already while recording. The same argument applies also to test-and-clear operations.

Operating System Memory References in Linux

This leaves memory accesses generated by the Linux kernel itself. However, the PGD filter leaves only the OS memory accesses that coincide with target process activity since OS memory accesses falling into the timeslices of other processes get already filtered. In general, there are at least two methods to determine whether the OS or a user process generated a memory access.

The IA-32e architecture distinguishes between *supervisor-mode accesses* and *user-mode accesses* [14, Vol. 3A 4-29]. Memory accesses performed while the current privilege level (CPL) is less then 3 are supervisor-mode accesses, every accesses with CPL 3 are user-mode accesses. In Linux, the kernel operates with CPL 0, referred to as *kernel mode*. Regular processes operate with privilege level 3, referred to as *user mode*. Thus, every memory accesses performed by the OS is a supervisor-mode access, every accesses performed by a process an user-mode access. The CPL is determined by the CS and SS segment CPU registers [14, 5-6 Vol. 3A ff.]. However, the operating system itself still access process address space, e.g., in order to read the parameter of a system call after control has already been transferred. This accesses also increment the reference counter. Therefore, CPL fails as a filter criterion as we cannot ignore these memory accesses.

Another option is to detect memory accesses performed by the OS based on the linear address. The IA-32e architecture partitions the linear address space into a top and a bottom half (Section 2.4). Linux uses the top half for kernel data and the bottom half for user process data [17]. User mode processes cannot access kernel data, that is linear addresses above `0x00007fffffffffff`. The Linux kernel itself accesses both kernel data as well as user data. Hence, the linear address potentially excludes a user process as the source of a memory access. However, in case the kernel access a bottom half address, the address does not determine the kernel as the source. In fact, the linear address is a better filter criterion since it does not matter whether or not a access to a bottom half address is performed by the OS or a user process. In both cases refers the address to user process data and therefore also contributes to the memory reference behavior of the user process.

Additionally, this criterion allows to ignore redundant memory accesses already while recording. For our purposes, Simutrace does not record any accesses to linear addresses above `0x00007fffffffffff`.

5.2.2 Filtering Test-and-clear Operations

In our design outlines in Section 4.1.2 Simutrace records all test-and-clear operations, regardless of the address space that the target page table entry (PTE) belongs to. Therefore, beside the target page's virtual base address, Simutrace needs to record some address space identifier so the offline analysis is able to identify the test-and-clear operations of interest.

As discussed in Section 2.4 defines the IA-32e architecture an address space as a hierarchical set of page tables. Each address space is identified by the physical address of the first page table in the hierarchy, referred to as the page global directory or PGD. The dispatcher loads the PGD is into the `CR3` register of the CPU in order to identify the currently executing address space and to specify the page map to use for address translation [14, Vol. 3A 4-19].

Linux associates every address space with a data structure called memory descriptor. The memory descriptor also stores the PGD. Therefore, on the one hand, we modify Simutrace so it records for every test-and-clear operation both the base virtual address of the target page as well as the PGD identifying the address space it belongs to. The offline analysis, on the other hand, processes only test-and-clear operations with the appropriate PGD.

5.3 Reconstructing Page Reference Count Inaccuracies

Section 4.1 defines the baseline for the operating system's perception of memory reference behavior as the perception it arrives at in case none of the test-and-clear operations suffer from any inaccuracy. Consequently, a perception of memory reference behavior is distorted, in case it differs from the baseline. In order to access to which extend the accessed flags capacity limits cause said distortions, it is necessary to learn the baseline, that is the undistorted perception, as well as the perceptions the OS arrives at given certain accessed flag capacities.

We limit our studies to a page reference count based perception of page reference behavior. To this end, we must learn both the actual number of references to the individual pages as well as the number of effectively recorded references. To this end, algorithm 1 computes the number of recorded accesses the OS arrives at, given an intermixed sequence of memory accesses and test-and-clear operations as well as a hypothetical accessed flags bit capacity.

Algorithm 1: Count Recorded Accesses

Input: Filtered memory accesses, filtered test-and-clear operations, accessed flag bit capacity c

Output: Number of recorded accesses for the virtual pages of the target process

```
foreach in chronological order trace entry e do
    if e is memory access then
        foreach page hit p do
            hit p
        end
    else if e is test-and-clear operation then
        p ← target page of e
        hits ← test-and-clear p
        if hits > $2^c - 1$ then
            hits ← $2^c - 1$
        end
        record hits references for p
    end
end
```

The general idea is the following: The algorithm computes the hypothetical results of perfectly accurate test-and-clear operations by counting the actual number of page references in between two consecutive test-and-clear operations. However, it caps the number of recorded references according to the simulated accessed flag's bit capacity. The reconstruction also discards any recorded references recorded after the last test-and-clear operation to the corresponding page table entry, since the operating system as a consequence never acquires these access information.

Line 3 to 5 process the memory accesses. Line 3 translates a memory access into necessary page references. Line 4 updates the reference counters. Line 7 to 12 process the test-and-clear operations. Line 8 test-and-clears the unlimited accessed flag, line 9 to 11 cap the reference count according to the simulated accessed flag capacity. Finally, line 12 records the hypothetically recorded page references.

5.4 The Stack Simulation

Algorithm 2 illustrates the stack algorithm used to simulate demand paging under page replacement algorithm $\mathcal{R}$. As input it requires the filtered sequence of memory accesses, the number of page frames available, and the page replacement algorithm. It outputs the number of page faults.

To do so, the algorithm in line 3 iterates the sequence of memory accesses in chronological order. According to Section 2.5.2, this is a supported operation. Line 4 translates a memory access into one or two page references, depending on the size of the requested data and the linear address. Line 5 to 12 handle the stack. In case the page is not in the stack, the page reference constitutes a page fault. In order to add it to the stack, that is, bring it into memory, it is potentially necessary to remove a victim page from the stack, pushing the victim to external storage. To this end, line 8 test-and-clears the pages in order to sort the stack according to the recorded page access information and $\mathcal{R}$'s heuristic. After the stack is sorted, line 9 removes the page at the bottom of the stack, the page $\mathcal{R}$'s heuristic indicates as most suitable victim. Finally, line 11 adds the faulted page to the stack and line 13 updates the accessed flag.

The most important step, however, is the sorting of the stack in line 8. In contrast to the original Mattson stack algorithm discussed in Section 2.3.1, this simulation does not update the stack on every page reference. It uses the page table entry's accessed flags in order to cache page access information, as real IA-32e architecture

Algorithm 2: Stack Simulation

```
Input: Filtered memory accesses, number of page frames $m$, replacement
       algorithm $\mathcal{R}$
Output: Number of page faults
faults ← $0$
s ← empty stack
foreach in chronological order memory access do
    foreach page hit p do
        if p not in s then
            faults ← faults $+ 1$
            if $|s| = m$ then
                sort s according to $\mathcal{R}$
                remove bottom page from s
            end
            add p to s
        end
        hit p
    end
end
```

hardware does. In case of a page fault, the stack simulation retrieves the recorded page access information and tries to restore the original order, as indicated by $\mathcal{R}$'s heuristic. In case it fails to do so because of inaccurate page access information, the algorithm potentially selects a different victim than the original algorithm. Therefore, the algorithm basically evaluates whether or not the recorded page access information is sufficient to restore the original order. However, in case it does not, the inaccurate page access information may both improve as well as worsen the replacement decisions. This, in fact, depends on whether or not the heuristic applies well to the reference string.

We are interested in the number of page faults of the least recently used page replacement algorithm (LRU), the least frequently used page replacement algorithm (LFU), the random page replacement algorithm (RAND), and the optimal page replacement algorithm (OPT). Overall, those algorithms only differ in the implementation of the accessed flag and the sorting step.

LFU implements the accessed flag as an access counter with limited capacity and saturation addition. The sorting step sorts the stack by number of recorded page accesses in ascending order. To do so, the implementation uses a stable sorting algorithm so pages only swap positions in case of actually different access counts.

As stated in Section 3.1.2, LFU based on a single bit counter is actually identical to an implementation of *LRU* based on a single bit access marker.

LRU comes in two implementations. The basic implementation uses a single bit accessed flag in order to mark whether pages have been accessed or not. It sorts the stack using a stable sorting algorithm based on whether a page has been marked or not. The advanced implementation has the accessed flag function as an actual timestamp and sorts the stack by last access in ascending order.

RAND is not based on any page access information at all. RAND's implementation of the sorting step does not sort the stack, but shuffles it. Hence, after shuffling, the bottom page is a random victim as expected.

5.4.1 The Optimal Page Replacement Algorithm

The optimal page replacement algorithm, referred to as OPT, is an offline replacement algorithm as it requires a priori knowledge of the entire reference string (Section 2.2.1). Whenever a page fault is detected and it is asked to select a suitable victim page to evict from memory, OPT selects the currently loaded page whose next reference lies farthest in the future or, if possible, a page without further reference. This way, it ensures that, in case this push in fact causes a future fault, the future fault occurs as late as possible.

The most prominent operation for OPT is the following: Given a set of pages S and an index into the reference string i, find page $n \in S$ whose next reference with $i' > i$ is farthest away of all the next references to the pages in S. One approach to find the page, is to learn the next references for all the pages in S and to select the page with the next reference with the highest index. In order to find the index of the next reference to a page, starting from the given index, we scan the following memory accesses in order, until at least one reference to every page in S has been discovered or the memory access stream ends.

The stack simulation has the entire reference string at hand. That is, it has access to the stream of memory access entries via Simutrace's storage interface. According to Section 2.5.2, a stream effectively allows random access to its entries by index. However, before it is possible to access the target entry, first it is necessary to retrieve the segment holding it, a potentially costly operation due to the storage I/O and decompression involved. This strongly suggests an one-pass algorithm.

However, a simple implementation eventually scans the same portion of the reference string repeatedly in case it passes another page reference, which is going to fault later, while servicing an earlier page fault. An obvious example is a reference to a page which has not been referenced before and therefore causes a demand fault. In case the algorithm already scanned past this reference in order to find the next reference to some other page, it must rescan the portion of the reference string between the demand fault and its current position.

A more advanced implementation caches portions of the reference string in order to avoid multiple passes. The purpose of the cache is, given a page and an index i, to efficiently find the index $i' > i$ of the next reference to said page. To do so, the cached portion of the reference string forms a sliding window. Figure 5.1 illustrates the concept.

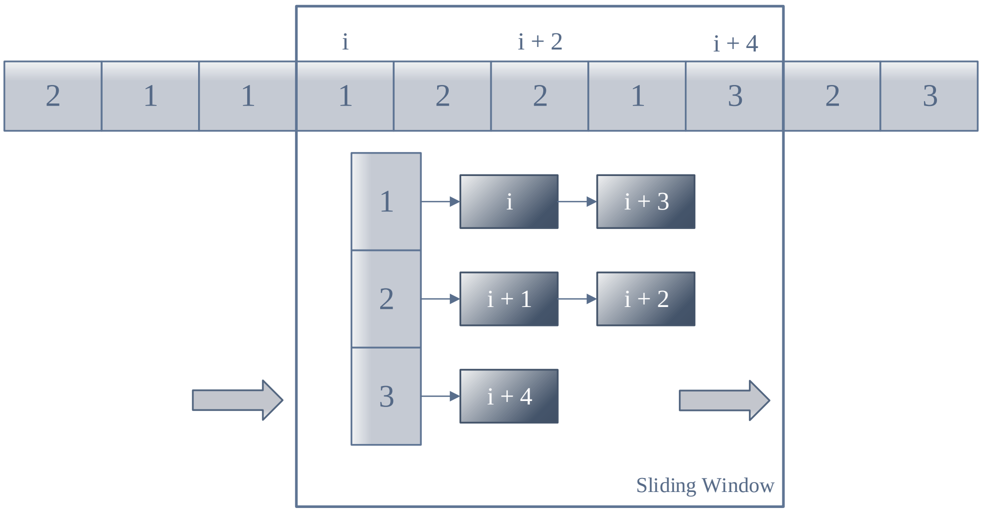

Figure 5.1: Example sliding window data structure on an example sequence of page references. The windows ranges from the reference at index i to the reference at index $i + 4$. The range contains references to pages 1 to 3. The scenario is the following: Page reference i to page 1 causes a page fault. Pages 2 and 3 are currently loaded. Therefore, the windows must be extended to cover the next references to these two pages and is therefore extended to index $i+4$. The cached page references are sorted according to page and index. Since the indexes increase linearly, the page lists in fact form sorted lists, allowing the algorithm to easily find the next reference to a page as long as the window covers it by looking up the first references in the page's list. Extending the windows past a references causes the references to be appended to the appropriate page's list. In case the windows shrinks, the all page lists are scanned for references at indexes the window already went past.

The cache is organized as follows: For every page it maintains a sorted list of cached references to the page. Hence, given a page and an index, the cache easily finds the earliest next reference by searching the page's sorted list. The window only grows at the end, towards the end of the reference string. It grows in case it not yet covers the next reference to a page of interest. Every reference scanned is appended to the appropriate list. The window shrinks only in the front. It shrinks with every memory access that the stack algorithm successfully performed as those accesses effectively lie in the past.

Selecting a victim in case of a page fault works as follows: The algorithm scans for every currently loaded page the sorted list of upcoming references. The head of the list is the next reference. In case a list is empty, the algorithm first checks[3] whether or not there is a further reference to the concerned page at all and potentially grows the sliding windows until it covers to the next references to the concerned page. When the next reference for every loaded page has been found, the algorithm finds the highest index reference and selects the corresponding page as victim.

5.5 Conclusion

In this chapter we present the implementation details of both the inaccuracy analysis as well as the paging simulation. We choose Linux as IA-32e simulation guest since it is a widely used, free open source operating system which allows us to easily add the necessary hypercall extensions. In addition, Linux provides control groups as a mechanism to enforce local replacement to take place on the target process. However, in order to acquire suitable input for our analyses, we need to implement three filters for the recorded trace data.

We employ the PGD as filter criterion for removing all memory accesses that do not fall into timeslices of the target process. Furthermore, we filter the leftover accesses based on the linear address. We discard every memory access with an linear address above `0x00007fffffffffff` in order to free the sequence of all operating system activity. The third filter applies to the recorded test-and-clear operations. In this case, we also use the PGD to identify the address space a test-and-clear operation belongs to.

Consequently, we use a slightly extended Simutrace in order to record the following events annotated with instruction counts: 1) `clone` and `exec` system calls in order to learn the PGD of the target process, 2) dispatch events including the PGD of the dispatched process to identify target process timeslices, 3) all mem-

[3]To this end, the algorithm requires a priori knowledge of the indexes of the very last reference to every page. However, this information can be acquired with a single run of a one-pass algorithm and later fed to every stack simulation.

ory accesses with linear address not above `0x00007fffffffffff`, and 4) all test-and-clear operations including the PGD of the target address space. Since Simutrace records all entries in order[4], it is possible to apply the filters simultaneously with the simulation, preventing unnecessary data from being written to the data store in the first place.

[4]Simutrace acquires and processes trace data in the same order as the simulation produces it. However, it then stores it to different streams, which is why we need the instruction counts in order to restore the global sequence of events.

Chapter 6

Evaluation

This chapter supplements the presented approaches to access the applicability of more accurate page access information for paging algorithms with an evaluation. It begins with an introduction of out evaluation setups in Section 6.1. The evaluation proceeds in two parts: In Section 6.2 we assess the reference counter inaccuracies introduced into the operating system's perception of page reference behavior on the basis of Algorithm 1 (Section 5.3) and our benchmark programs. In Section 6.3 we evaluate the paging simulation of local replacement based on a number of paging algorithm configuration according to Algorithm 2 (Section 5.4).

6.1 Evaluation Setup

Our evaluation setup consists of two systems and five benchmark programs. Section 6.1.1 discusses the systems. Section 6.1.2 introduces our benchmark programs.

6.1.1 System Configuration

We use QEMU [5], a full system simulator, for data acquisition. The evaluation setup includes two systems: The simulation host executing QEMU and Simutrace's full data store as well as the simulation guest executing the benchmarks.

The host system is equipped with a single Intel Core i7-920 processor with four physical and eight logical cores as well as 24 GiB RAM. It runs runs Ubuntu Server [18] 13.04 with Linux kernel 3.11.0-19-generic. The host system runs QEMU 1.3.1 with extensions for memory hooks, the hypercall interface and the instruction counter. Table 6.1 summarizes the host system configuration.

Component	Model / Specification
CPU	1x Intel Core i7-920 processor
Available Cores	4 (8 logical)
Memory	24 GiB DDR3-1333 (6x4 GiB)
Operating System	Ubuntu Server 13.10
Kernel	Ubuntu 3.11.0-19-generic
Architecture	64 Bit (IA-32e)
Simulator	QEMU Version 1.3.1
Simulator Extension	Memory hooks, hypercall interface, instruction counter

Table 6.1: The host system configuration running QEMU and Simutrace's full data store.

The simulated guest system contains a single simulated CPU and 256 MiB of RAM. The guest system also runs Ubuntu Server 13.04 with a Linux vanilla kernel 3.9.2 with activated OS introspection. Table 6.2 summarizes the simulated guest system configuration.

Component	Model / Specification
CPU	1x simulated
Available Cores	1
Memory	256 MiB
Operating System	Ubuntu Server 13.04
Kernel	Vanilla Linux kernel 3.9.2 with OS introspection
Architecture	64 Bit (IA-32e)
Simulated Architecture	x86-64
Simulation Mode	softMMU, TCG, icounter, optimized build
Libraries	glibc v2.17, zlib v1.2.7
Build Tools	gcc v4.7.3

Table 6.2: The guest system configuration running the benchmarks.

6.1.2 Benchmarks

Both parts of the evaluation evolve around the page reference behavior of a number of programs. For the first part, those programs run under memory pressure as we try to access how well the IA-32e page table entry accessed flag is suited to reflect the program's page reference behavior. For the second part, we employ said programs in order to collect page reference strings for offline analysis. This section introduces the benchmark programs.

Binary Search The binary search benchmark is based on glibc's binary search implementation. The benchmark allocates an array b of integers and initializes it to $[0, \ldots, len - 1]$. Furthermore, it memory maps a file of random data as array of integers and performs a binary search for $x \equiv random\ int \pmod{len}$ on b for every integer in the array. We refer to x as the search *key*. The file of random data is obtained once from the /dev/urandom pseudorandom number generator. The benchmark is build using gcc v4.7.3, `-O0` flag, and glibc v2.17.

Quicksort The quicksort benchmark is based on glibc's [1] implementation of quicksort [12]. The benchmark memory maps a file of random data as array of integers and sorts it using glibc's quick sort implementation. The memory mapping uses the `MAP_PRIVATE` flag, so no changes are written back to the file. The random data is obtained once from the /dev/urandom pseudorandom number generator. The benchmark is build using gcc v4.7.3 with `-O0` flag and glibc v2.17.

MySQL MySQL [7] is an open-source relational database management system (RDBMS). Sysbench [2] is a modular, multithreaded benchmark tool. Sysbench includes multiple test modes, one of them `oltp` for benchmarking database performance. `oltp`'s `prepare` stage creates and populates a specific database table and performs a series of SQL statements on it. In our case this table holds 10000 rows on top of the MyISAM storage engine. The `run` stage performs 128 complex transactions using `LOCK TABLES` and `UNLOCK TABLES` statements in two threads. We record the MySQL's reference string for the entire process lifetime, including startup, benchmarking and shutdown. The guest system runs MySQL v5.5.34 and sysbench v0.4.12.

Compression with zlib This benchmark is also a custom program. It memory maps an archive of the Linux kernel version 1.3.100 source files and uses zlib's [26] `compress` utility function to compress the archive at default compression level. This function both reads the input data as well as writes the output data to memory buffers. The benchmark is build using gcc v4.7.3, `-O0` flag, and zlib v1.2.7.

Physically-based Rendering with pbrt pbrt [24] is an open-source physically-based renderer. The guest system runs pbrt v2.0.0. We collect pbrt's reference string with `-ncores 1` flag when rendering the prt-teapot scene from the example scenes included with pbrt. However, we use a modified prt-teapot scene so rendering completes faster[1] Figure 6.1 shows the resulting image. pbrt is build using gcc v4.7.3.

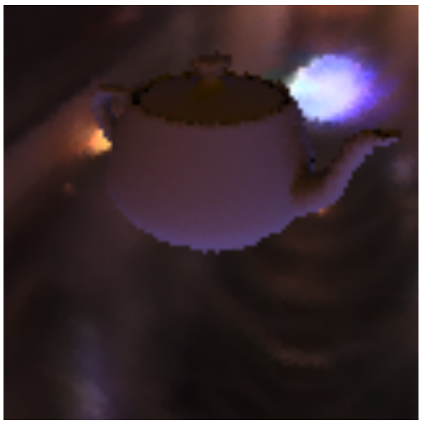
Figure 6.1: Rendered image of pbrt example scene prt-teapot.

6.2 Page Reference Counter Inaccuracies

The purpose of this first part of the evaluation is to assess the degree of inaccuracy that the IA-32e's page table entry accessed flag's limited storage capacity introduces into the operating system's perception of page reference behavior. Section 4.1 describes the evaluation methodology. The inaccuracies in the operating system's perception of page reference behavior stem from two factors: 1) the timings and targets of the test-and-clear operations and 2) the degree of inaccuracy the test-and-clear operations experience. However, this thesis studies the impacts of the hardware's limited capability to cache page access information upon the inaccuracies that the test-and-clear operations experience. Therefore, we consider paging algorithm implementation details such as test-and-clear timings external effects. To this end, Section 4.1 defines the operating system's perception of page

[1]We apply the following modification: 1) The image resolution is reduced to 100 x 100 pixel. 2) The surface integrator's parameters `lmax` and `nsamples` are set to 5 and 128, respectively.

reference behavior distorted, in case any of the test-and-clear operations experiences inaccuracy. Thereby, the operating system's perception of page reference behavior is the number of references it notices for the individual pages of the address space.

In this case, the terminology "noticing page references" refers to the process of the page table entries counting the number of page references by using the accessed flag as a counter as described in Section 3.1.2 and the OS test-and-clearing the counters. As algorithm 1 outlines, there are two reasons why a page reference goes *unnoticed*: 1) because a accessed flag counter has limited capacity and cannot increment any further or 2) because the OS does not issue another test-and-clear operation on an accessed flag and hence does not read the counter. To any other references we refer as *noticed* references.

6.2.1 Visualizing Reference Counter Inaccuracies

This section discusses the operating system's perception of page reference counts of the binary search as well as the quicksort benchmark (Section 6.1.2). It visualizes the OS's perception of page reference counts and presents a number of artifacts introduced by the accessed flag's limited storage capacity. To this end, we calculate the number of noticed page references to the pages holding the two arrays as indicated by the recorded intermixed sequence of memory accesses and test-and-clear operations. Then, we plot the noticed reference counts against different accessed flag counter capacities as discussed in Section 3.2.2. We use the same recorded intermixed sequence of a benchmark for every assessed counter capacity. This way, we exclude varying test-and-clear timings, e.g., the timing of the last test-and-clear operation to a page, as sources of distortion. Since this only leaves the influence of the accessed flags' counter capacity, we use the terminology of accessed flags noticing page references interchangeably to the OS noticing page references via test-and-clear operations.

We obtain an intermixed sequence of memory accesses as well as test-and-clear operations as described in Section 4.1.1. It is then refined as discussed in Section 5.2. Furthermore, Section 5.3 outlines the algorithm for counting noticed page references given the intermixed sequence and a hypothetical accessed flag counter capacity.

Binary Search

The binary search benchmark performs a number of binary searches on an array of sorted integers. In this case, it performs 65536 binary searches using glibc's `bsearch` function for random integers on the array $[0, \ldots, 2^{18} - 1 = 262143]$ of 262144 integers. Since the search keys randomly fall into the range of the integers in the sorted array, we expect them to distribute uniformly among 262144 integers in the array.

The array of sorted integers takes up 1 MiB of memory, that is 256 4 KiB pages. In order to cause memory pressure on the benchmark and force the operating system to perform local page replacement, we run the benchmark in a cgroup with limited memory allocation (Section 5.1.1). We limit the allocation to 1 MiB, the number of pages it takes to hold the sorted array alone. In addition to the sorted array, the address space also includes the stack, the heap, the text, and global variables. Hence, page replacement becomes necessary as the benchmark executes.

Figure 6.2 shows the number of noticed page references into the sorted array given virtual accessed flags of 16 bit, 8 bit, and 4 bit capacity. The x-axis shows the 256 pages used to hold the sorted array. The y-axis shows the number of noticed references. The figure depicts the noticed references for 16 bit accessed flags as blue bars, for the 8 bit accessed flags as orange bars and for 4 bit accessed flags as green bars. All bars are generated using algorithm 1 with the same intermixed sequence but different accessed flag capacities as input. Hence, an accessed flag with larger capacity counts at least as many references without saturating as an accessed flag with smaller capacity. Therefore, we plot the bars for smaller capacities on top of bars for larger capacities.

For 16 bit capacity accessed flags the figure clearly shows the access pattern of the interval bisection method for uniformly distributes search keys. Every search references the page holding the middle array element first. Then, it proceeds either to the left or the right interval depending on the search key and references the page in the middle of that interval and so on. At some point in that process the interval shrinks to the size of a single page. From this point on, every further iteration references the same page. Hence, at least for the first $\log 256 = 8$ iterations, the number of noticed references basically form a geometric pattern since every search references the middle page, every other search references the middle page of the left interval, every fourth search references the middle page of the left half of the left interval and so on.

The 8 bit bars differ significantly from the 16 bit bars only for the middle page. This page's 8 bit accessed flag caches only about half of the references as a po-

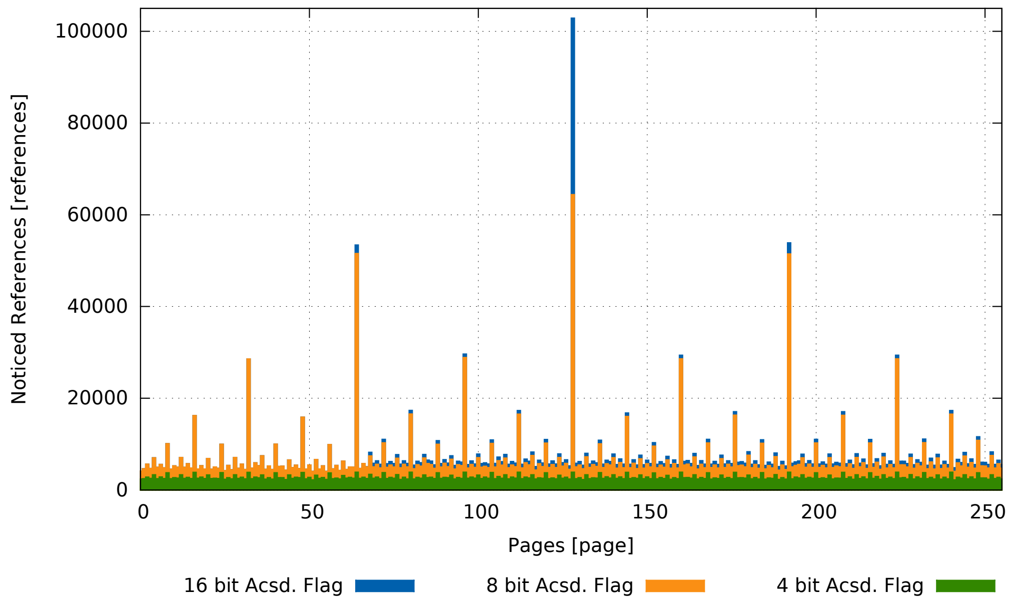

Figure 6.2: Number of noticed page references for binary search benchmark (1 MiB array, 65536 binary searches): page references into sorted array given virtual accessed flags of 16, 8, and 4 bit capacity. Bars for smaller capacities overlay bars for larger capacities.

tential 16 bit accessed flag does. Still, the 8 bit capacity accessed flags show a difference between the middle page and the two second most referenced pages. In general, they also allow the OS to recognize the reference pattern of the interval bisection method. The 4 bit accessed flags, however, do not show the pattern at this scale.

An interesting detail is that for the first about 60 pages the diagram does not show differences between the 16 bit and the 8 bit accessed flags. In fact, these pages show in general less noticed references than their counterparts in other parts of the array. We suspect this to be an artifact introduced by the implementation details of the Linux paging algorithm.

Figure 6.3 depicts the same kind of chart, only for accessed flag capacities of 4, 2 and 1 bit. Again, the x-axis shows the 256 pages, the y-axis shows the number of noticed references. This figure shows the number of noticed references for 4 bit accessed flags as blue bars, for 2 bit accessed flags as orange bars and for 1 bit, the capacity that the IA-32e architecture currently implements, as green bars.

The 1 bit accessed flag bars shows the page reference counts that the operating system actually collects till the end of the benchmark when running on IA-32e. The

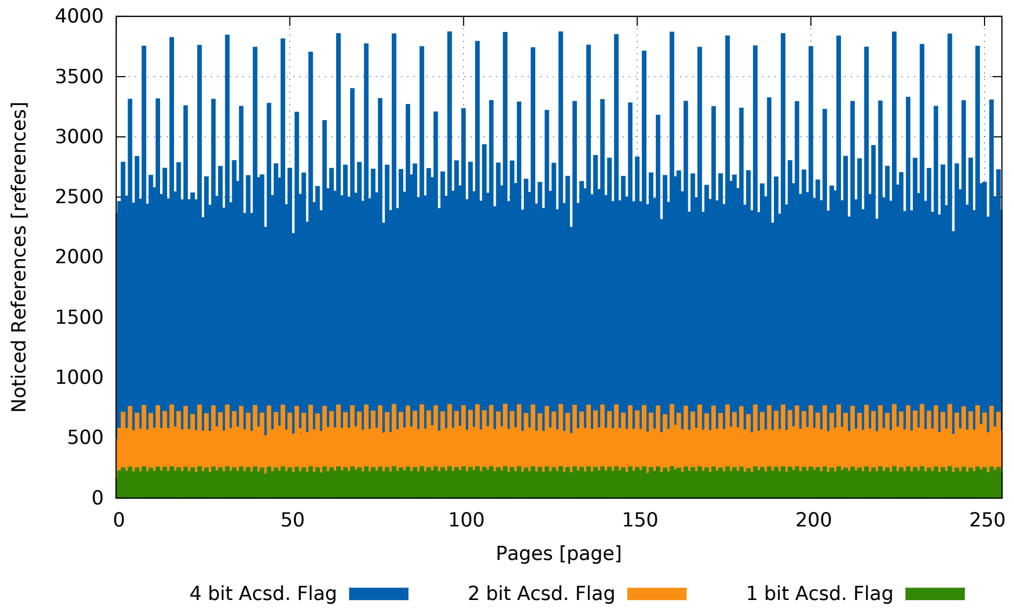

Figure 6.3: Number of noticed page references for binary search benchmark (1 MiB array, 65536 binary searches): page references into sorted array given virtual accessed flags of 4, 2, and 1 bit capacity. Bars for smaller capacities overlay bars for larger capacities.

green bars do not present the geometric reference pattern of the interval bisection method. The 1 bit accessed flags basically identify two categories of pages: less referenced pages and more referenced pages. Less frequently referenced pages and more frequently referenced pages occur alternately. The 2 bit accessed flags in orange already identify three categories of pages. Every second page belongs to the least frequently referenced pages. Between those pages alternate medium and most frequently referenced pages. The later two categories coincide with the more frequently referenced pages of the 1 bit accessed flags, forming two subcategories.

The 4 bit accessed flag already appears in Figure 6.2 as green bars. At this scale, the 4 bit accessed flag bars show four categories of pages, forming smaller peaks. This pattern suggests the geometric page reference pattern of the interval bisection method, but yet does not recognize it. Basically the four categories of pages are the same three categories as the 2 bit accessed flags, plus a fourth category, comprised of every other page of the 2 bit accessed flags' most frequently referenced pages category.

Quick Sort

The quick sort benchmark memory maps an array of random integers and uses glibc's [1] `qsort` function to sort it with quicksort [12]. In this case, the array has a size of 1 MiB, that is 262144 integers or 256 4 KiB pages. In order to cause memory pressure and to force the operating system to perform local replacement, we run the benchmark in a cgroup with memory allocation limited to 512 KiB, that is half the pages required to hold the array of integers alone.

Figure 6.4 shows a bar chart of the number of noticed page references into the array of integers given virtual accessed flag capacities of 32, 16, and 8 bit in blue, orange and green, respectively. Again, the x-axis shows the pages, the y-axis shows the number of noticed references and bars for lower capacity accessed flags overlay the bars for higher capacity accessed flags. All bars are generated using algorithm 1 with the same intermixed sequence but different accessed flag capacities as input.

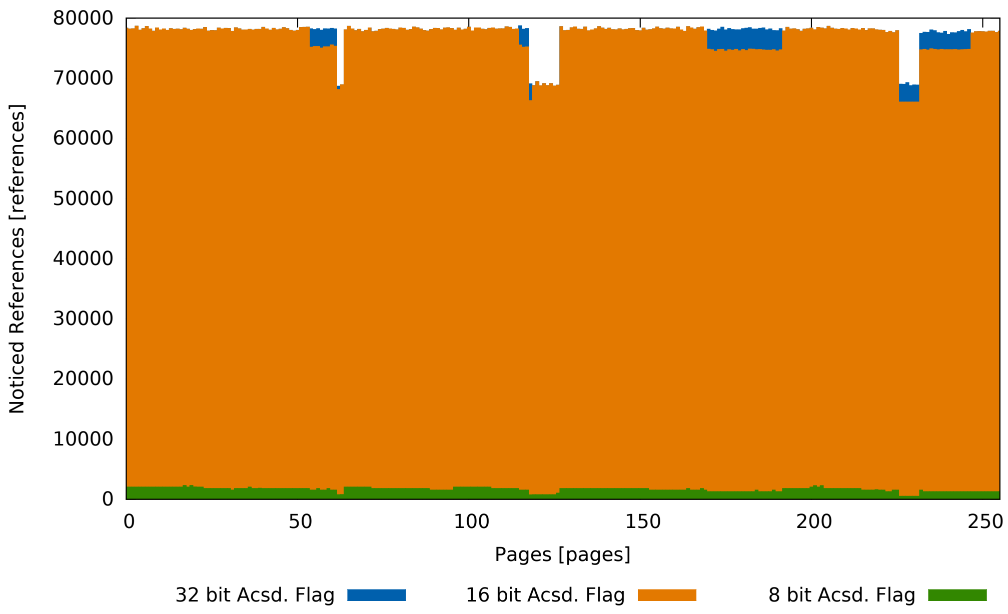

Figure 6.4: Number of noticed page references for quicksort benchmark (1 MiB array): page references into array given virtual accessed flags of 32, 16, and 8 bit capacity. Bars for smaller capacities overlay bars for larger capacities.

The 32 bit accessed flags notice approximately the same number of references for all pages. The only exceptions are four smaller intervals of pages that show slightly less noticed references, potentially because the missing references occur after the last test-and-clear operation on said pages. The 16 bit accessed flags notice almost as much references as the 32 bit accessed flags. However, they show a fourth interval of pages, between pages 150 and 200, receiving less references. Additionally, they show even less references for the pages in the previously mentioned page intervals.

In green, the 8 bit accessed flag bars show way less noticed references over the entire array. Compared to the 16 and 32 bit accessed flags, the noticed references are less evenly distributed among the 256 pages. Some intervals, e.g., between 200 and 250, show disproportional variations in relation to the overall number of noticed references.

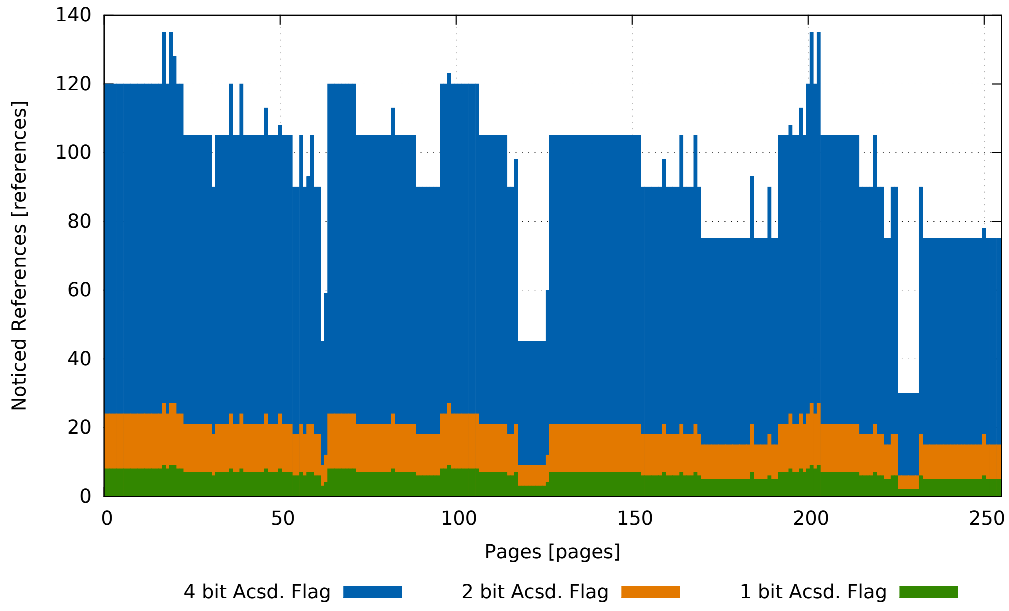

Figure 6.5: Number of noticed page references for quicksort benchmark (1 MiB array): page references into array given virtual accessed flags of 4, 2, and 1 bit capacity. Bars for smaller capacities overlay bars for larger capacities.

Figure 6.5 also depicts the number of noticed page references, for accessed flag capacities of 4, 2 and 1 bit. Although the accessed flags with higher counter capacities indicate that the pages approximately receive the same number of references, the 4, 2 and 1 bit accessed flags altogether show considerable variations in reference counts. However, the strongest downward variations at about page 60, pages 110 to 120 and 225 to 235 coincide with the intervals of less frequently referenced pages from Figure 6.4. Other pages show upward variations, e.g., pages 15 to 20 and 190 to 210. Overall, the accessed flags with less counter capacity show very similar patterns in reference counts.

Yet, the chart shows minor differences. An example are the leftmost pages 0 to 50. The first few pages show an equal reference count for all counter capacities. At about page 15, the figure shows three peaks. Given 4 bit accessed flags, the first two peaks show approximately 135 noticed references, the third peak only approximately 130. The 2 and 1 bit accessed flags also show the peaks, however no difference between the first two and the third one. Other intervals, e.g., pages 155 to 165, show similar patterns.

6.2.2 Quantifying Reference Counter Inaccuracies

Earlier sections capitalize on visual assessment of how well the perception with lower capacity accessed flags, potentially suffering from higher individual inaccuracy, resembles the hypothetical perception with higher capacity accessed flags, suffering from lower individual inaccuracy. However, a more generic approach that also applies to benchmarks with unknown references behavior is to quantify the resemblance between two perceptions, the perception gained via lower capacity accessed flags and high, preferably unlimited, capacity accessed flags.

The selection of an appropriate quantifier for the resemblance strongly depends on the page replacement algorithm implemented on top of the perception of page reference behavior. Least frequently used (LFU) page replacement (Section 2.2.1) replaces pages based on reference frequencies. It prioritizes more frequently referenced pages over less frequently referenced pages. In other words, LFU ranks pages according to reference counts. Hence, in case of LFU, different perceptions of page references behavior show strong resemblance, if the implied page reference count rankings correlate well.

Kendall [16] introduces a measure of rank correlation referred to as *Kendall tau rank correlation coefficient*. It takes two rankings of individuals, in this case, pages ranked according to noticed reference counts, from two different observations, in this case page table entry accessed flags with different counting capacity. It then assesses whether or not the two rankings are sufficiently alike to indicate similarities between the observers [16].

For our purposes, for one of the two rankings we use the ranking indicated by the accessed flags with unlimited counter capacity, our baseline for page reference behavior perception using references counters. As the second ranking we use a ranking indicated by limited capacity accessed flags, e.g., the reference counts as perceived by 1 bit capacity accessed flags. The resulting correlation coefficient then quantifies the resemblance between the two rankings within the interval $[0, 1]$, 0 indicating no resemblance and 1 indicating perfect resemblance. In other words, the kendall correlation provides us the means to assess whether or not limited capacity accessed flags cause the operating system to arrive at a inherently different perception of page reference behavior, namely different page priorities based on reference counts.

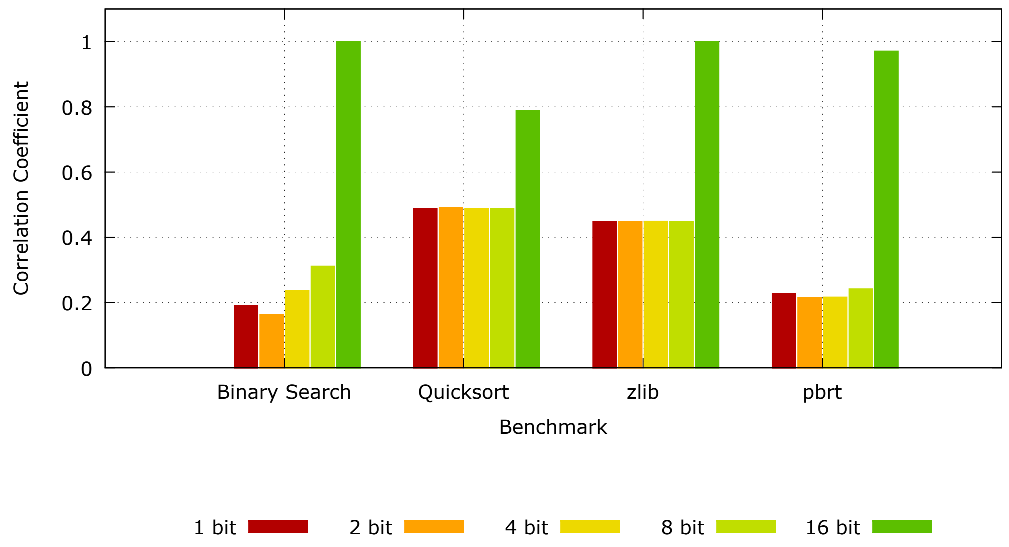

Figure 6.6: Kendall rank correlation between page rankings as indicated by accessed flags with limited counter capacity and unlimited capacity. Every bar gives the correlation coefficient between two rankings for the same benchmark, one acquired using accessed flags of capacity as given by the color, the other using accessed flags of unlimited counter capacity. The x-axis gives the benchmark, the y-axis the correlation coefficient. 0 states no resemblance, 1 perfect resemblance. High resemblance indicates similar page priorities.

Figure 6.6 shows the kendall correlation coefficient between the page rankings as indicated by accessed flags with given capacities and the page ranking as indicated by accessed flags with unlimited counter capacity. The x-axis gives the accessed flags counter capacity in bits. The y-axis gives the correlation coefficient between the page ranking obtained by ranking pages according to the number of noticed references given accessed flags with x bit and the page ranking according to accessed flags with unlimited counter capacity. A coefficient close to 1 indicates high resemblance, a coefficient close to 0 indicates poor resemblance. In this case, high resemblance stands for similar page priorities in LFU. Hence, the coefficient does not meter whether or not limited counter capacities introduce inaccuracies into the perception of page reference behavior but whether or not potential inaccuracies distort the perception in a way that changes page priorities, potentially leading to different page replacement decisions.

The first cluster of bars shows the correlation coefficients for the binary search benchmark with the same parameters as described in the last section. However, in this case, we do not limit the consideration to the pages holding the sorted array but include all pages in the rankings. The 1 bit accessed flags produce poor resemblance. Interestingly, increasing the accessed flags counter capacity to 2 bit results in even worse resemblance in page rankings. 4 bit and 8 bit accessed flags result in only barely better resemblance. Only increasing the counter capacity to 16 bit accessed flags improves the result significantly. In fact, 16 bit produce the same page ranking as accessed flags with unlimited capacity.

The second cluster of bars gives the correlation coefficient for the quicksort benchmark given the same parameters as described in the last section. Again, this time we include all pages of the address space in the consideration, not only pages holding the array to be sorted. For 1 bit accessed flags show medium correlation. However, the correlation coefficients shrink slightly with increasing accessed flag counter capacities. Only 16 bit accessed flags improve the correlation between the rankings. Yet, not even 16 bit produce a page ranking that perfectly resembles the baseline ranking.

The third cluster gives the correlation coefficients for the zlib benchmark. In order to force the operating system to perform local page replacement, we limit the cgroup to 2048 KiB, that is 512 4 KiB pages, which is not even sufficient to hold the approximately 22 MiB Linux kernel v1.3.100 archive. The fourth cluster gives the correlation coefficients for the pbrt benchmark. In this case, we limit the cgroup also to 2048 KiB. Both benchmarks show results similar to the quicksort benchmark. Increasing the accessed flag counter capacities shows almost no effect on the coefficient. Only 16 bit accessed flags improve the correlation significantly.

6.2.3 Discussion

From the content of the array and the distribution of search key in the binary search benchmark we know that pages receive references with constant, well known frequencies depending on their location in the array. Effectively, the array pages break up into sets of pages with very similar reference frequencies. The benchmark shows that accessed flag counter capacity and test-and-clear interval durations impose an upper limit on detectable page references frequencies. It is not possible to distinguish the reference frequencies of two pages with above detection limit, but different frequencies. The 1 bit accessed flag suffer from this effect the most. As a consequence, they are unable to reflect the expected page reference behavior. However, the results illustrate that increasing the accessed flags' counter capacity, for this benchmark, improves the operating system's ability to match pages to sets. In this case, the findings suggest that higher capacity accessed flags, even 4 or 2 bit, improve the operating system perception of memory reference behavior.

In case of the quicksort benchmark, high capacity accessed flags indicate very few and barley relevant variations in reference counts. The 1 bit accessed flags, as currently implemented in the IA-32e architecture, cannot reproduce this characteristic. Instead of few variations, they show considerable and frequent variations in reference counts. However, increasing the counter capacities to 2 or 4 bit does not yield any improvements. The accessed flags still fail to reproduce the stable reference counts of high capacity accessed flags. If anything, they worsen the variations relative to the absolute reference counts. This suggests that low capacity accessed flags are more exposed to ill timed test-and-clear operation as they cannot compensate for reference bursts nor long periods of low reference frequency without being test-and-cleared. Only accessed flags with high counter capacity, such as 16 or 32 bit, allow the operating system to arrive at a better perception of page reference behavior.

In contrast to assessing the perception of page reference behavior by absolute reference counts, the kendall correlation coefficient constitutes a weaker criterion for resemblance, more relevant for least frequently used page replacement. However, it seemingly disproves the positive results of the visual assessment of the binary search algorithm. With increasing capacity, it shows less improvement in correlation as expected. A potential reason for this is the following: The visual assessment suggests better separation of the sets of pages with similar reference frequencies. Assigning pages to sets does not imply a total order of pages. However, the kendall correlation evaluates the similarities between two total orders. Hence, the disappointing results are potentially caused by a wrong page order within the sets.

Furthermore, the assessment of the kendall correlation coefficients endorses the negative findings of the quicksort benchmark. Increasing the accessed flags' counter capacity only slightly, effectively does not improve the operating system perception of page reference behavior, e.g., reference counts. Only increasing the capacities to at least 16 bit results in significant improvements as, in case of ranking based on reference counting, 16 bit accessed flags produce rankings closely resembling the rankings produced by unlimited accessed flags.

6.2.4 Conclusion

In order to assess the degree of inaccuracy in page access information found in IA-32e accessed flags we replay the interaction between a page replacement implementation, embodies by Linux, and the IA-32e accessed flags in a process' page table. As a first step, we assess the inaccuracies visually by comparing the operating system's perception of page reference behavior at the end of the benchmarks for accessed flags with different reference counter capacities. We find that single bit accessed flags provide significantly inaccurate page access information causing the operating system to arrive at a distorted perception of reference behavior. As a second step, we quantify the inaccuracies based on the kendall rank correlation coefficients between the references counts limited capacity accessed flags provide and reference counts unlimited capacity accessed flags provide. We find that reference counts accessed flags with limited capacity, especially 1, 2 and 4 bit, show little resemblance to the accurate reference count. We conclude that the findings endorse the negative conclusions from the visual assessment.

6.3 Paging Simulation

The purpose of the first part of the evaluation is to assess whether or not limited capacity accessed flags cause page access information inaccuracies and changing perception of page reference behavior, in this case, reference counts. It finds that this, in fact, is the case as, without extensively increasing the accessed flags' capacities, the operating system is unable to rank the pages according to reference counts. However, it remains unclear whether or not the inaccuracies in fact worsen performance of the paging algorithm, that is increase the number of page faults in generates. The purpose of the second part of the evaluation is to learn whether or not increasing the accessed flags' counter capacities yields less page faults.

Section 4.2 outlines our approach to a paging simulation which receives both a reference string and a number of available page frames as input and outputs the number of generated page faults. The simulation translates the memory accesses into page references and makes sure that all required page are loaded. In case a page fault occurs and no free page frame is available, the simulation selects a victim based on the simulated page replacement algorithm, which only receives page access information cached in the simulated page table. It synchronously replaces the pages and continues with the next reference. Section 4.2.3 designs the algorithm that drives the simulation, Section 5.4 gives an implementation.

We refer to the combination of paging algorithm and underlying accessed flag format as *configuration*. There are six candidate configurations: 1) least frequently/recently[2] used page replacement (LFU/LRU) simulated on 1 bit accessed flags as currently implemented in the IA-32e architecture, 2) to 5) LFU on extended page table entry format with 2, 4, 8, and 16 bit accessed flags (LFU x) and, 6) LRU on accessed flags storing a timestamp of the last access (timest. LRU).

However, as discussed in Section 3.2.1, we cannot assess a paging algorithm's performance based on absolute number of page faults. Therefore, we assess it relative to two baselines, the optimal page replacement algorithm (OPT, Section 2.2.1) and the random page replacement algorithm (RAND, Section 2.2.1) for all of our benchmarks.

Our figures show the number of accumulated page faults sampled at equidistant points in the simulated execution of the benchmarks. To this end, we use the number of memory accesses a benchmark performs as indicator for execution progress and sample the number of page faults in intervals of a fixed number of memory accesses. The exact step size, however, depends on the individual benchmark. We connect the sampling points by straight lines in order to clarify on the trend. Furthermore, we cut the curves after the last complete sampling interval. Since we use for each benchmark the same recorded reference string as input to the simulation, the position of a memory access in the reference string constitutes an appropriate indicator for benchmark progress. In case of RAND, we run the same simulation ten times and give for every sample point the mean number of page faults.

6.3.1 Binary Search

Figure 6.7 shows the simulation results for the binary search benchmark given 256 4 KiB pages, that is 1 MiB of memory. In this case, the benchmark performs 262144 binary searches on an array of 262144 integers[3]. As the figure shows a constant increase in page faults, all configurations experience a constant page fault frequency over the entire simulations. This is the expected result, as the search keys are uniformly distributed among the integers in the array. Hence, every iteration descends into the left or right interval with 50% probability. As expected, OPT performs best with less than 2500 page faults in total. RAND performs worst with in total almost 25000 page faults, 10 times the number of OPT. LRU/LFU, the original 1 bit replacement algorithm implementation, performs at about 8000 total page faults significantly better than RAND, yet a lot worse than

[2]LFU and LRU degrade to the same algorithm in case of 1 bit accessed flags (see Section 3.1.2).
[3]Again, every search key is taken randomly out of the array.

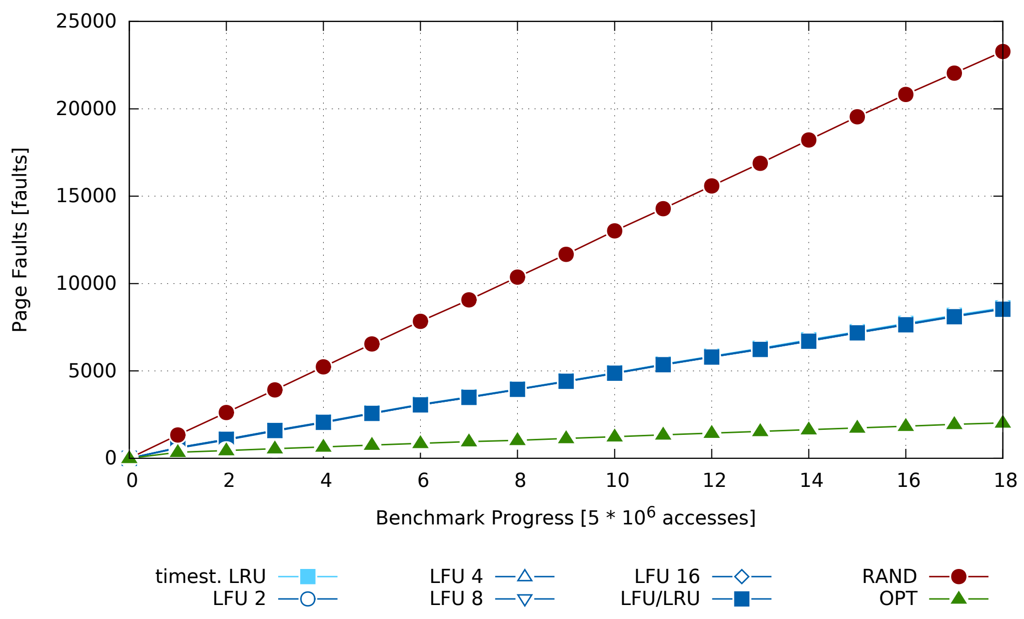

Figure 6.7: Number of page faults the binary search benchmark (1 MiB array, 262144 searches) simulation generates on 256 4 KiB page frames. X-axis shows benchmark progress as the number of performed memory accesses, y-axis the number of page faults. LRU/LFU overlaps other simulation candidates.

OPT. However, the simulation shows almost no improvement for the rest of the configurations. Increasing the capacity of the accessed flags to 2, 4, 8 or even 16 bit does not noticeably reduce or increase the number of page faults.

6.3.2 Quicksort

Figure 6.8 shows the simulation results for the quicksort benchmark, sorting 4 MiB of integers on 256 4 KiB page frames (1 MiB). All configurations show similar results. The accumulated page fault numbers jump sharply at four dedicated points in the execution of the benchmark. This is no surprise as the 1 MiB physical memory cannot hold more than one fourth of the array to be sorted. Hence, those four jumps most probably coincide with quicksort moving from one of the four second level partitions to another one, causing almost all pages to be replaced.

Once again, OPT performs best. RAND performs worse by approximately 30% compared to OPT. Still, LFU/LRU performs a little worse. However, in this case, the extended page table entry formats seemingly improve the paging algorithms' performances. LFU 16, LFU based on 16 bit accessed flags, halves the distance

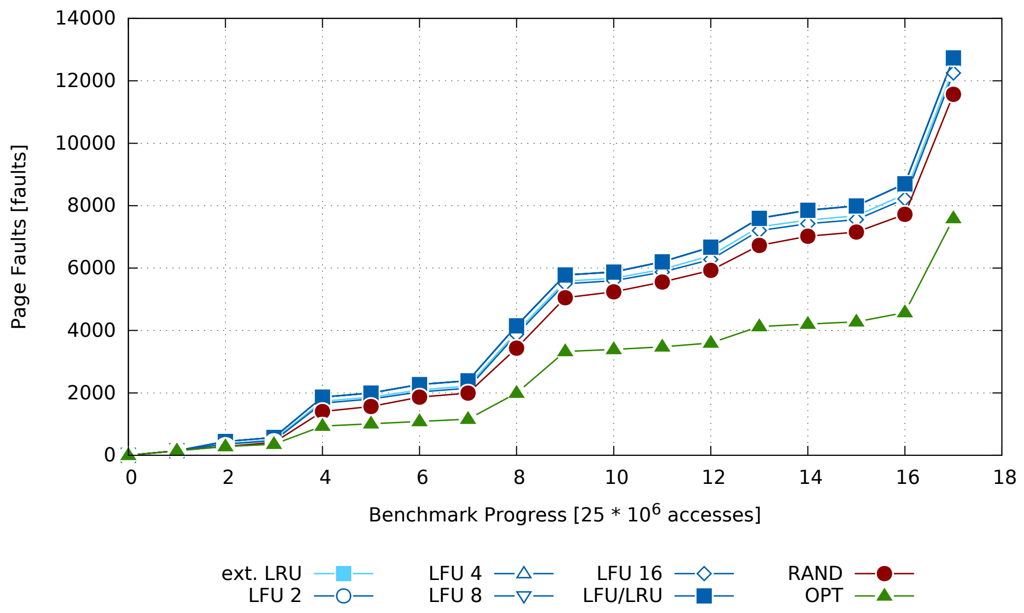

Figure 6.8: Number of page faults the quicksort benchmark (4 MiB array) simulation generates on 256 4 KiB page frames. X-axis shows benchmark progress as the number of performed memory accesses, y-axis the number of page faults. The curves of LFU 2, 4 and 8 run exactly below LFU/LRU.

to RAND. Timest. LRU, LRU based on accessed flags storing a timestamp of the last reference to the corresponding page, shows similar improvements. LFU 2, 4 and 8 perform a little worse than LFU/LRU.

6.3.3 MySQL

Figure 6.9 illustrates the results of the MySQL benchmark simulation for 1024 4 KiB page frames. This benchmark performs 128 complex transactions in two threads on a 10000 row table driven by the MyISAM storage engine. The recorded reference string includes the startup of MySQL, the described transactions sysbench performs as database benchmark as well as the shut down of MySQL. The small jump in page faults after about $80 * 10^6$ memory accesses supposedly marks the start of the benchmark transactions.

As expected, OPT performs best at approximately 25000 page faults. RAND performs worst at approximately 42000 faults, almost twice as many faults as OPT. LFU/LRU finishes in between at approximately 34000 page faults, about 20% worse than OPT. However, for the increased capacity configurations, we see

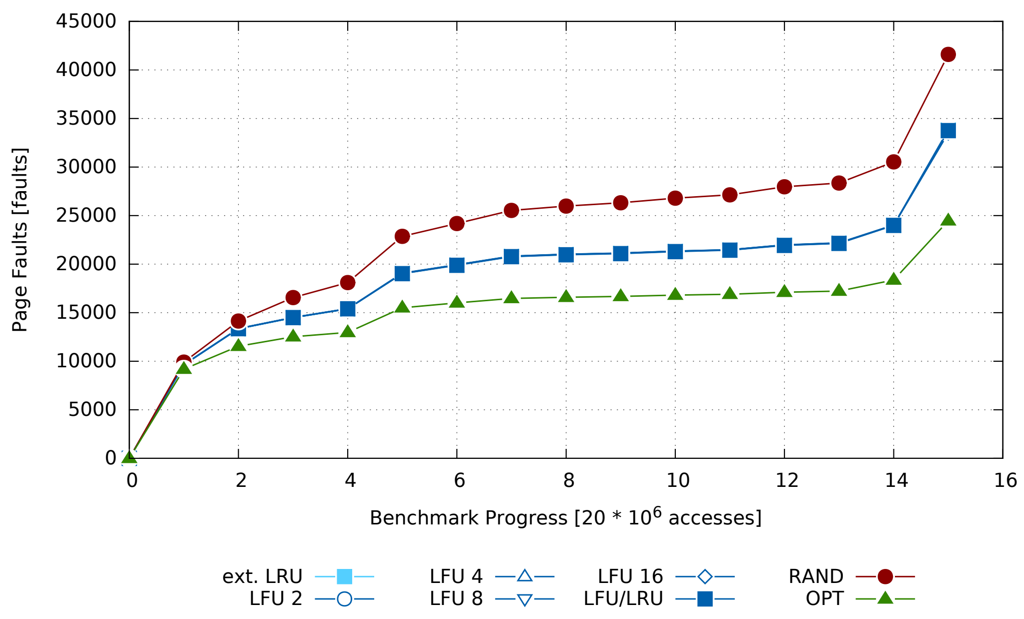

Figure 6.9: Number of page faults the MySQL sysbench benchmark (10000 rows, 128 complex transactions) simulation generates on 1024 4 KiB page frames. X-axis shows benchmark progress as the number of performed memory accesses, y-axis the number of page faults. LRU/LFU overlaps other simulation candidates.

no improvement whatsoever. They perform almost exactly as 1 bit accessed flag LFU/LRU.

6.3.4 pbrt

Figure 6.10 depicts the results of the pbrt benchmark simulation for 1024 4 KiB page frames. The benchmark renders the 100 x 100 pixel image of a teapot depicted in Figure 6.1. All configurations present similar behavior. At the beginning of the benchmark, the simulation produces pages faults at a fast rate. The page fault rate drops after about $150 * 10^6$ memory accesses and finally levels out at a low rate after about $550 * 10^6$ memory accesses.

OPT performs best at approximately 15000 page faults. RAND performs about 25% worse, producing approximately 20000 page faults in total. LFU/LRU performs worst at approximately 22000 page faults. Yet, once again, the higher capacity configurations show no improvements compared to LFU/LRU whatsoever.

Figure 6.11 depicts the results of the simulation of the pbrt benchmark again, how-

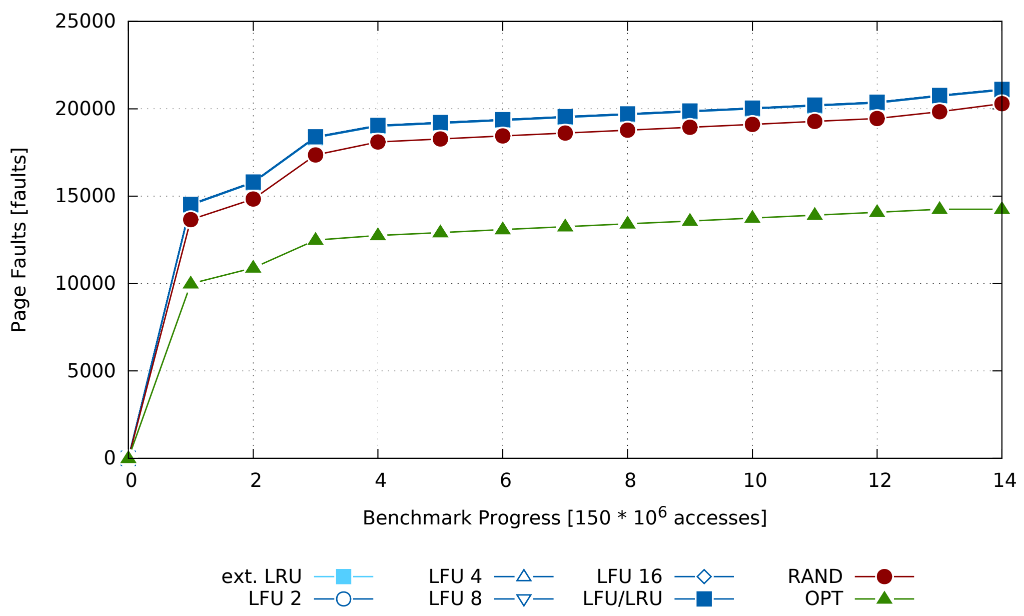

Figure 6.10: Number of page faults pbrt benchmark (prt-teapot) simulation generates on 1024 4 KiB page frames. x-axis shows benchmark process as number of performed memory accesses, y-axis number of page faults. LRU/LFU overlaps other simulation candidates.

ever this time for 512 instead of 1024 4 KiB page frames. This simulation produces very similar results, yet this time, LFU/LRU performs better than RAND. This indicates, that LFU's assumptions about page reference behavior at least to some extend apply to the recorded pbrt's reference string. Nevertheless, increasing the accessed flags' storage capacities does not improve the overall paging performance.

6.3.5 zlib

Figure 6.12 depicts the results of the 256 4 KiB page frame simulation of the zlib benchmark. This benchmark compresses an 22 MiB archive of the Linux kernel 1.3.100 source in memory. All configurations show a constant rate of page faults. RAND performs worst at approximately 9000 page faults. All other configurations perform almost identically. Both OPT, LFU/LRU and all other configurations produce approximately 7000 page faults. Since LFU/LRU already performs as well as OPT, there is no room for improvement. No configuration can produce less page faults than OPT. Therefore, this benchmark does not provide any insight into whether or not more accurate page access information improve paging

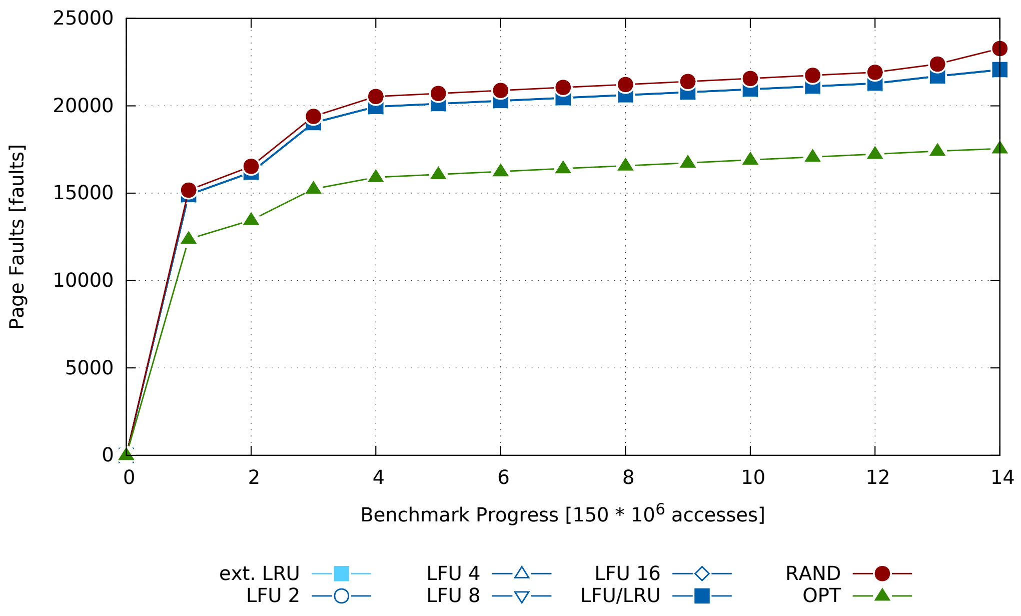

Figure 6.11: Number of page faults the pbrt benchmark (prt-teapot) simulation generates on 512 4 KiB page frames. X-axis shows benchmark progress as the number of performed memory accesses, y-axis the number of page faults. LRU/LFU overlaps other simulation candidates.

performance.

6.3.6 Discussion

Table 6.3 summarizes the simulation results described in the last section. The discussion of the results is oriented towards the following two aspects: whether or not the results of the assessment of the kendall correlation promise improvements and whether or not the base case LFU/LRU performs better or worse than RAND.

The binary search benchmark is a suitable candidate for the paging simulation. According to the kendall assessment, 4 bit and 8 bit accessed flags show improvements in correlation towards the 1 bit accessed flags. 16 bit accessed flags show perfect correlation, that is, provide completely accurate page access information. Furthermore, LFU/LRU, the configuration based on 1 bit accessed flags, performs much better than RAND, which according to Section 3.2.1 indicates that the underlying assumptions of the replacement algorithm apply well to the observed page reference behavior. Still, it performs considerably worse than OPT, which

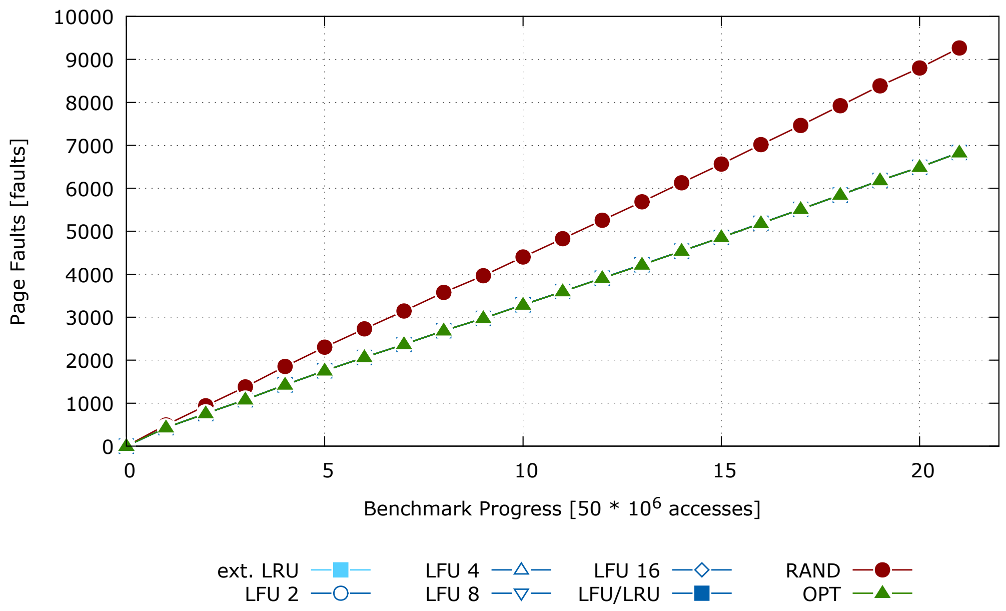

Figure 6.12: Number of page faults the zlib benchmark (22 MiB archive) simulation generates on 256 4 KiB page frames. X-axis shows benchmark progress as the number of performed memory accesses, y-axis the number of page faults.

Configuration	Binary S.	Quicksort	MySQL	pbrt 512	zlib
LRU/LFU	8594	15420	34233	24058	7127
LFU 2	8613	15422	34339	24046	7127
LFU 4	8610	15425	34039	24034	7127
LFU 8	8630	15425	34060	24026	7127
LFU 16	8616	14934	33999	24033	7127
Timest. LRU	8674	15077	34290	24028	7127
RAND	23402	14268	42074	25652	9665
OPT	2060	10015	24931	18985	7097

Table 6.3: Summary of the simulation results presented in Section 6.3. pbrt column gives simulation results of the pbrt benchmark for 512 4 KiB page frames. The results include all page faults instead of being cut at the end of the last full interval of memory accesses.

leaves plenty of room for improvement. However, the binary search benchmark simulation shows no improvement for increased accessed flag capacity whatsoever.

The MySQL as well as the zlib benchmark paging simulations are other examples where 1 bit LFU/LRU performs already better than RAND. Yet, both simulations show also no improvements at all. However, in this case of the zlib benchmark simulation, LFU/LRU already perform optimal, which leaves no room for improvement and renders the benchmark futile.

In Section 6.3 we present two simulations of the pbrt benchmark. One simulation on 1024 4 KiB page frames and one on 512 4 KiB page frames. In case of the former, RAND performs better than 1 bit LFU/LRU. In case of the latter, it performs worse. Yet, both cases show no improvements with increasing capacity accessed flags. This suggests that comparing LFU/LRU to RAND does not allow any conclusion towards the applicability of more accurate page access information.

The only benchmark showing improvements in simulation results is the quicksort benchmark. It shows insignificant improvements for 16 bit accessed flags LFU as well as timestamp LRU. However, LFU 16 outperforms LFU/LRU by not even 4%.

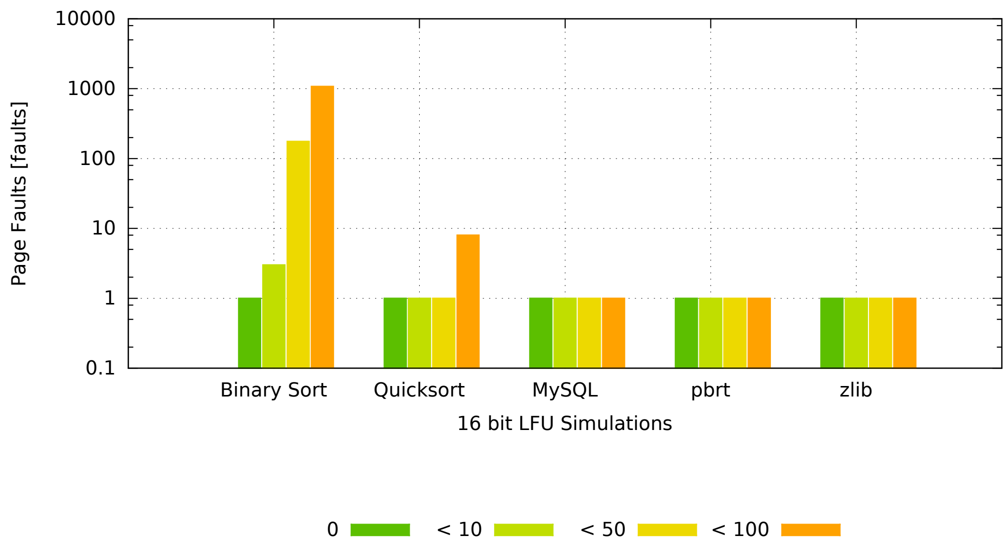

Figure 6.13: Number of page faults for which 0, < 10, < 50, and < 100 loaded pages have not been referenced since the last page fault. X-axis gives the benchmark. Y-axis gives the number of page faults with less than x pages without reference. The diagram covers only the 16 bit LFU simulation for each benchmark. It excludes early demand page faults populating the stack, for which no page replacement takes place.

On the one hand, the benchmark simulation altogether suggest that page replacement algorithms seemingly do not benefit from more accurate page access information. On the other hand, the results also suggests by implication that page replacement does not suffer from inaccurate page access information, e.g. provided by limited capacity 1 bit accessed flags. Figure 6.13 provides a potential explanation. It depicts how often the paging simulation's page replacement algorithms select a victim page which has been referenced since the last page fault. This happens only once for every 16 bit accessed flag LFU simulation. In other words, except for one page replacement taking place, every other replacement replaces a page which has not been referenced since the last page fault.

An overwhelming number of page faults happen on a simulated page stack that includes numerous pages without any reference since the last page fault, that is the last time the pages in the simulated page stack had been test-and-cleared. In case of a page without any reference within the current test-and-clear interval, the page access information is always correct, regardless of the accessed flag's storage capacity. In case of reference counting, even single bit accessed flags provide enough resolution to identify those pages. Accordingly, high capacity accessed flags pay off only if the number of references exceeds the resolution of lower capacity accessed flags.

Since our paging simulation selects only one victim at a time, the replacement algorithms look for the single page which comes last according to some replacement heuristic. Both LFU and LRU always select a victim without references if possible. Therefore, except for one, none of the page replacement decisions actually benefit from the more accurate page access information provided by higher capacity accessed flags since they select a page without reference anyway. They are able to identify this page even without more accurate information.

6.3.7 Conclusion

In order to assess whether or not page replacement suffers from inaccurate page access information, we ran paging simulations for our benchmarks and counted the total number of page faults. We simulated different benchmarks, subject to different page replacement algorithms which received differently accurate page access information from the accessed flags. However, we found that the accuracy of the page access information does not affect the quality of the page replacement decisions, embodied by the number of resulting page faults. Therefore, we concluded that page replacement in general does not suffer from inaccurate page access information.

Chapter 7

Conclusion

Page replacement refers to the process of selecting a currently loaded page victim to vacate its page frame in favor of a faulted page. Page replacement algorithms are tasked with selecting victims to push to external storage in a way that reduces the total number of page faults the system experiences. Offline paging algorithms employ assumptions about page reference behavior of processes in order to extrapolate future reference behavior from past reference behavior. Since the paging algorithm does not observe reference behavior directly, it relies on the page table entries used for address translation to cache page access information until it is able to process it. However, the IA-32e architectures page table entry format reserves only a single bit for said purpose, introducing temporal as well as counter inaccuracies into the page access information page table entries supply to the paging algorithm.

In this work, we analyzed how the page tables inability to accurately count references between two consecutive paging algorithm invocations alters the operating system's, strictly speaking the paging algorithm's, perception of page reference behavior. In order to reconstruct the operating system perception, we traced and later on replayed the interactions between the operating system and the IA-32e page table entries using full system simulations and Simutrace. We concluded that a single bit of access information cache is generally insufficient to provide the operating system with an accurate perception of reference behavior. We also concluded that moderately increasing the number of bits in the page table entry format reserved for counting page references not only not helps to overcome the inaccurate perception but potentially worsens it. Only a counter capacity of 16 bit allows the operating system to acquire an accurate perception of page reference behavior.

Furthermore, we analyzed how the inaccurate perception of page reference behavior affects the performance of paging algorithms, namely the number of page faults it generates for a given reference string, page table entry format and number of physical page frames. To this end, we employed a paging simulation very similar to the Mattson stack algorithm. However, our simulation had the paging algorithm rely on simulated page table entries to provide access information in order to reconstruct the stack order of the Mattson stack algorithm. We found that inaccurate page access information, such as provided by IA-32e page table entries, does not negatively affect the paging algorithm's replacement decisions as we could not reduce the number of page faults by providing more accurate page access information.

We explained our findings as follows: The paging algorithm's assumption about page reference behavior, e.g., frequently referenced pages are more likely to be used in the future as infrequently referenced pages, imposes a priority on loaded pages to which pages should remain in memory. Page table entries that fail to count page references for frequently referenced pages because of saturation affect the ranking order of high priority pages only. This is largely because, for infrequently referenced pages, the page table entries provide perfectly accurate access information. However, the page replacement algorithm is tasked to single out a single victim page. It finds this victim among the low priority, that is infrequently referenced or entirely unreferenced pages. Hence, we concluded that paging algorithms do not benefit from more accurate page access information.

7.1 Future Work

We showed that more accurate page access information fails to improve the performance of demand paging algorithm according to Aho et al. [3] because it is not needed in order to single out the most suitable victim page. However, our simulation model did not yet cover the time requirements of paging algorithms. Our simulation allowed the paging algorithm to retrieve access information from every loaded page and to reevaluate the entire priority ranking order, a potentially time consuming operations increasing the algorithms response time. It remained unclear whether or not the simulation presents different results in case the paging algorithm is only allowed to evaluate the access information of a limited set of loaded pages, thereby reducing its response time in favor of accuracy.

Another aspect of paging our simulation did not cover are asynchronous pushes. The definition of demand paging algorithms presented in [3] requires the memory to be kept as utilized as possible. Although this potentially reduces the number of page faults, it at least increases the page wait time since it forces the paging algorithm to process page faults synchronously. Before a faulted page can be brought into memory, first a victim must be selected and pushed to external storage in order to free a page frame. However, this is not always desirable or even suitable. Therefore, other paging algorithms opt to start evicting multiple pages once the number of free page frames drop below some threshold. Operating systems such as Linux perform periodic page frame reclaiming in order to keep the number of free page frames above the threshold [6, 707ff.], thereby selecting multiple pages as victims at a time. When evicting multiple victims at a time, page replacement algorithms potentially must resort to higher priority pages, pages whose page access information actually benefits from an extended page table entry format.

Another aspect not covered in this work is non-uniform memory access (NUMA). NUMA is an organization scheme for memory in multiprocessor systems where each processor provides its local memory to a shared address space of physical page frames. However, this scheme implies that the processors access different parts of the address space with different speed, depending on the actual location of the memory. An operating system running on a system that does not employ NUMA is tasked with separating pages into two sets: pages in use, which therefore should remain in fast memory, and pages currently not in use, which may be pushed to slow external storage. However, in a NUMA scenario there are at least three potential locations for pages: fast memory attached to the processor the process is scheduled to run on, not so fast memory attached to another processor, and slow external storage. Those three locations imply three sets of pages. Since the assumptions about page reference behavior, which paging algorithms employ, are designed to predict future reference probabilities, they may also be suitable to separate pages into said three sets. In order to separate pages of the first and second set, the operating system depends on the accuracy of the entire page ranking, not only on the order of the least prioritized pages at the bottom of the ranking. Hence, it requires accurate page access information on both less frequently referenced pages as well as frequently referenced pages, potentially benefiting from an extended page table entry format.

Bibliography

[1] The gnu c library (glibc). http://www.gnu.org/software/libc/. accessed 2014-18-07.

[2] Sysbench: a system performance benchmark. https://launchpad.net/sysbench. accessed 2014-18-07.

[3] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. *J. ACM*, 18(1):80–93, January 1971. http://doi.acm.org/10.1145/321623.321632.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966. http://dx.doi.org/10.1147/sj.52.0078.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1247360.1247401.

[6] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[7] Oracle Cooperation. Mysql. http://www.mysql.com/. accessed 2014-18-07.

[8] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968. http://doi.acm.org/10.1145/363095.363141.

[9] Linux Kernel Documentation. Memory resource controller. https://www.kernel.org/doc/Documentation/cgroups/memory.txt. accessed 2014-18-07.

[10] John Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Commun. ACM*, 4(10):435–436, October 1961. `http://doi.acm.org/10.1145/366786.366800`.

[11] Thorsten Gröninger. On statistical properties of duplicate memory pages. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, October31 2013. `http://os.ibds.kit.edu/`.

[12] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962. `http://comjnl.oxfordjournals.org/content/5/1/10.abstract`.

[13] Intel, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, May 2011.

[14] Intel, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, September 2013.

[15] Intel, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, September 2013.

[16] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):pp. 81–93, 1938. `http://www.jstor.org/stable/2332226`.

[17] Andi Kleen. Linux x86_64 mm layout, July 2004. `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt`. accessed 2014-18-07.

[18] Canonical Ltd. Ubuntu. `http://ubuntu.com`. accessed 2014-18-07.

[19] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.

[20] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Sally A. Mckee, Bronis R. De Supinski, and Andy Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Trans. Program. Lang. Syst.*, 29(2), April 2007. `http://doi.acm.org/10.1145/1216374.1216380`.

[21] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970. `http://dx.doi.org/10.1147/sj.92.0078`.

[22] Paul Menage. Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`. accessed 2014-18-07.

[23] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22(2):297–306, June 1993. `http://doi.acm.org/10.1145/170036.170081`.

[24] Matt Pharr and Greg Humphreys. pbrt. `http://http://www.pbrt.org/index.php`. accessed 2014-18-07.

[25] Marc Rittinghaus. Runtime benefits of memory deduplication. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, July5 2012. `http://os.ibds.kit.edu/`.

[26] Greg Roelofs and Mark Adler. zlib. `http://www.zlib.net/`. accessed 2014-18-07.

[27] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985. `http://doi.acm.org/10.1145/2786.2793`.

[28] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 84–97, New York, NY, USA, 2003. ACM. `http://doi.acm.org/10.1145/859618.859629`.

[29] M.T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, April 2007.

[30] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 177–188, New York, NY, USA, 2004. ACM. `http://doi.acm.org/10.1145/1024393.1024415`.