Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz Universität Hannover

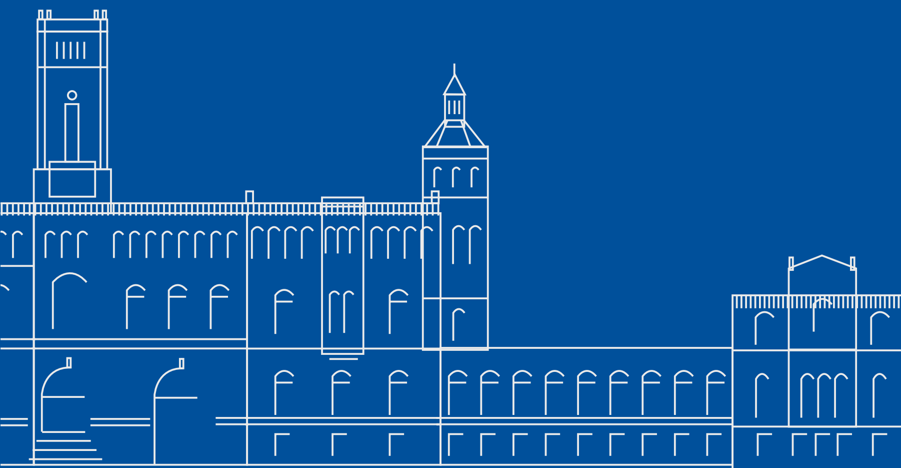
Tobias Landsberg

Analyzing and Optimizing TLB-Induced Thread Migration Costs on Linux/ARM

Masterarbeit im Fach Informatik

5. November 2018

Please cite as:
Tobias Landsberg, "Analyzing and Optimizing TLB-Induced Thread Migration Costs on Linux/ARM" Master's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, November 2018.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Analyzing and Optimizing TLB-Induced Thread Migration Costs on Linux/ARM

Masterarbeit im Fach Informatik

vorgelegt von

Tobias Landsberg

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**Björn Fiedler, M.Sc.**
	Christian Dietrich, M.Sc.

Beginn der Arbeit:	**5. Mai 2018**
Abgabe der Arbeit:	**5. November 2018**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Tobias Landsberg)
Hannover, 5. November 2018

ABSTRACT

Modern multi-core processors have a variety of caches, some of which are core-local. Therefore, a thread migration to another CPU core by the operating system can lead to an increased cache miss rate, which results in a slower and less efficient execution of the same instruction stream. The Translation Lookaside Buffer (TLB) has a particular large influence on this because a TLB miss results in two, often not cached, memory accesses.

TLB migration is supposed to solve this problem by migrating the contents of the TLB together with the rest of the thread context to the new core. However, this was not supported by the hardware of common processor architectures like x86 or ARM, so a software implementation was necessary. Since x86 did not offer access to the TLB, this thesis focused on the ARM-Architecture, which allowed at least direct read access. Write access was performed by exploiting a side effect on the TLB caused by an instruction to preload data at a given address.

However, since initial measurements indicated that this approach was not going to be fast enough to cause any overall speedup, a second approach exploring the effects of minor hardware modifications (using gem5) was pursued. This resulted in time savings of over 4.9 µs when using a modified Linux and running a thread on four different cores accessing 56 Pages with a stride of at least 68 KiB.

The results suggested that the examined ways of accessing the TLB using existing methods are not sufficient to implement TLB migration in a way that achieves any overall speedup. However, minor hardware modifications would suffice in some cases and could be taken into account when designing processors in the future.

KURZFASSUNG

Moderne Mehrkernprozessoren besitzen verschiedenste Arten von Zwischenspeichern, von denen einige Prozessorkernlokal sind. Migriert das Betriebssystem einen Programmfaden auf einen anderen Kern, kann dies zu einer hohen Rate von Zwischenspeicherfehlzugriffen führen, die wiederum zu einer langsameren und ineffizienteren Ausführung des selben Befehlsstromes führt. Einen besonders großen Einfluss darauf hat der Übersetzungspuffer (TLB), da ein TLB-Fehlzugriff zu zwei ggf. nicht zwischengespeicherten Speicherzugriffen führt.

TLB-Migration soll dieses Problem lösen, indem der Inhalt des TLBs zusammen mit dem restlichen Kontext des Fadens auf den neuen Kern migriert wird. Da dies jedoch nicht von gängigen Prozessorarchitekturen wie x86 und ARM in Hardware implementiert wurde, musste die Migration in Software geschehen. Da x86 keine Möglichkeit bot, auf den TLB zuzugreifen, wurde sich in dieser Arbeit auf die ARM-Architektur beschränkt, die zumindest einen lesenden Zugriff erlaubte. Für den schreibenden Zugriff wurde ausgenutzt, dass die Instruktion zum Vorladen einer Speicheradresse den Seiteneffekt hatte, den TLB zu füllen, wenn dies notwendig ist.

Erste Messungen legten nahe, dass dieser Ansatz nicht schnell genug sein würde, um eine Geschwindigkeitsverbesserung zu erzielen, weswegen zudem untersucht wurde, ob durch leichte Anpassung der Hardware (mithilfe von gem5) bessere Werte erzielt werden können. Es stellte sich heraus, dass sich dadurch, unter Verwendung eines angepassten Linux, Verbesserungen von über 4,9 µs erzielen lassen, wenn ein Faden auf vier verschiedenen Kernen auf 56 Seiten zugreift, die mindestens 68 KiB auseinander liegen.

Insgesamt folgte, dass die vorhandenen Mittel der Hardware nicht ausreichten, um durch TLB-Migration einen Geschwindigkeitsvorteil zu erzielen. In Zukunft würde es jedoch reichen, die Hardware minimal anzupassen, um in zumindest einigen Fällen eine Verbesserung der Geschwindigkeit bei der Migration eines Fadens zu erzielen.

CONTENTS

Abstract **v**

Kurzfassung **vii**

1 Introduction **1**

2 Fundamentals **3**
2.1 Translation Lookaside Buffer (TLB) . . . 3
2.2 Platform . . . 4
2.3 Related Work . . . 6

3 Architecture: TLB Migration **7**
3.1 The Concept of TLB Migration . . . 7
3.2 TLB Data Layout . . . 7
3.3 TLB Access Path . . . 8

4 Analysis: TLB Migration **11**
4.1 TLB Read Access . . . 11
4.2 The `pld` Instruction . . . 12
4.3 Migration Process . . . 15

5 Architecture: Hardware Modifications **21**
5.1 TLB Data Layout . . . 21
5.2 TLB Access Path . . . 21

6 Analysis: Hardware Modifications **25**
6.1 Cortex-A7 vs. gem5 . . . 25
6.2 TLB Interfaces . . . 27
6.3 Migration Process . . . 28

7 Architecture: Linux Integration **31**

8 Analysis: Linux Integration **33**
8.1 Migration Validation . . . 33
8.2 End-to-end-Migration . . . 34

9 Conclusion **37**

A Appendix **41**
A.1 Cortex-A7 41
A.2 gem5 45
A.3 Linux Integration 49

Lists **51**
List of Acronyms 51
List of Figures 53
List of Tables 55
List of Listings 57
Bibliography 59

1 INTRODUCTION

Modern processors usually have more than one CPU core and a variety of caches to speed up memory accesses by orders of magnitude. Some of these caches are core-local. Thus, an operating system should avoid switching the core a thread is running on whenever possible. However, sometimes it is imperative, e.g., to load balance the system or to satisfy constraints imposed by the scheduler. This process, called thread migration, introduces additional costs, some of them directly by saving the context and restoring it on another core, some of them indirectly. The latter are caused by cache misses due to the loss of data in the core-local caches. In these cases the CPU core has to wait for the slow memory to be able to continue with its task.

The Translation Lookaside Buffer (TLB) behaves quite similar to these caches but is special in its purpose. Instead of caching data or instructions, it caches translations from virtual to physical addresses (see Section 2.1). In short, the TLB caches two memory accesses instead of just one, resulting in a miss being twice as slow. However, since it is core-local and therefore lost during thread migration, it is also a starting point to reduce the indirect migration costs.

In this thesis it is explored whether migrating the TLB as part of thread migration is a suitable approach to reduce TLB-induced thread migration costs. The underlying idea is to handle the contents of the TLB belonging to a thread as part of its context and therefore migrating it to the target core. In the best case scenario, the TLB hit rate after the migration is comparable to the one before. To assess this approach, TLB migration will be implemented on a modern processor architecture and eventually integrated into a operating system of choice, in this case Linux. For this purpose, the ARMv7 architecture was chosen because it allows direct read access to the TLB, unlike x86-based processors. Write access is performed through exploitation of a side effect on the TLB caused by an instruction to preload data at a given address.

The thesis is structured as follows:

Chapter 2 introduces the reader to all additional fundamentals required to understand the subsequent chapters, e.g., how the TLB works and detailed information about the used hardware, and also gives an overview about related work. The following chapters come in pairs each separated in an architecture and analysis part. Chapter 3 and Chapter 4 focus on TLB migration on an ARM processor. To explore the idea of possible hardware modifications, Chapter 5 and Chapter 6 cover this approach with help of a simulator. After discussing TLB migration, Chapter 7 and Chapter 8 address the integration into Linux. Chapter 9 concludes this thesis.

2 FUNDAMENTALS

This chapter explains the fundamentals required to understand this thesis. It starts with an overview of the TLB's functionality and simultaneously explains the concepts of virtual memory and paging. Furthermore, it describes the hardware platform used and how it can be simulated. In the end, other related work is mentioned.

2.1 Translation Lookaside Buffer (TLB)

To grasp the TLB, an understanding of virtual memory is indispensable. Virtual memory is a concept to isolate multiple running processes from each other where instead of one global address space with direct access to the main memory, each process can have its own restricted virtual address space managed by the operating system. It is usually implemented via paging and is a feature of most modern operating systems. Paging is a technique where the virtual address space is segmented into chunks of memory, called pages. Each page of virtual memory is mapped to a page frame in physical memory. The size of a page can vary between architectures and even on a per-page basis with 4 KiB being used most of the time. Therefore, a 4 KiB page is assumed from now on for an easier understanding. As a consequence, the number of possible pages is huge (1 048 576 for 4 Gibit of memory) and therefore requires efficient management, even more so when taking into account that each process has its own pages, which is why the mapping can change with every context switch. This is often achieved using multi-level page tables. However, understanding two-level page tables as shown in Figure 2.1 is sufficient. A virtual address is made up of three parts as depicted, in this case two 10 bit table indices (PT1 and PT2) for the top- and second-level page tables and a 12 bit offset. To translate a virtual address to a physical one, a process called table walk is carried out by the Memory Management Unit (MMU). Whenever a memory access is performed, the MMU accesses the index PT1 of the top-level page table which contains the address of the second-level page table in which index PT2 has the address of the sought page frame. At last the offset in the offset field is added to the page frame's address, resulting in the final physical address. For example, when translating an arbitrary virtual address like `0xffc00248`, it has to be separated in its parts as described in Figure 2.1a, which results in PT1 = 1023, PT2 = 0 and Offset = 584. Afterwards, the MMU looks up the top-level page table at index 1023, which points to a second-level page table. This table contains the page frame's address, a physical address, at index 0. Finally, the offset of 584 is added to this address. Thus, one memory access in software generates effectively three accesses in hardware as opposed to a single one without virtual memory.

Without extra hardware virtual memory would be hardly used at all because of its performance penalty. The TLB however, as part of the MMU, diminishes this performance penalty by caching address translations. Because most programs only use a small number of pages, it is possible to

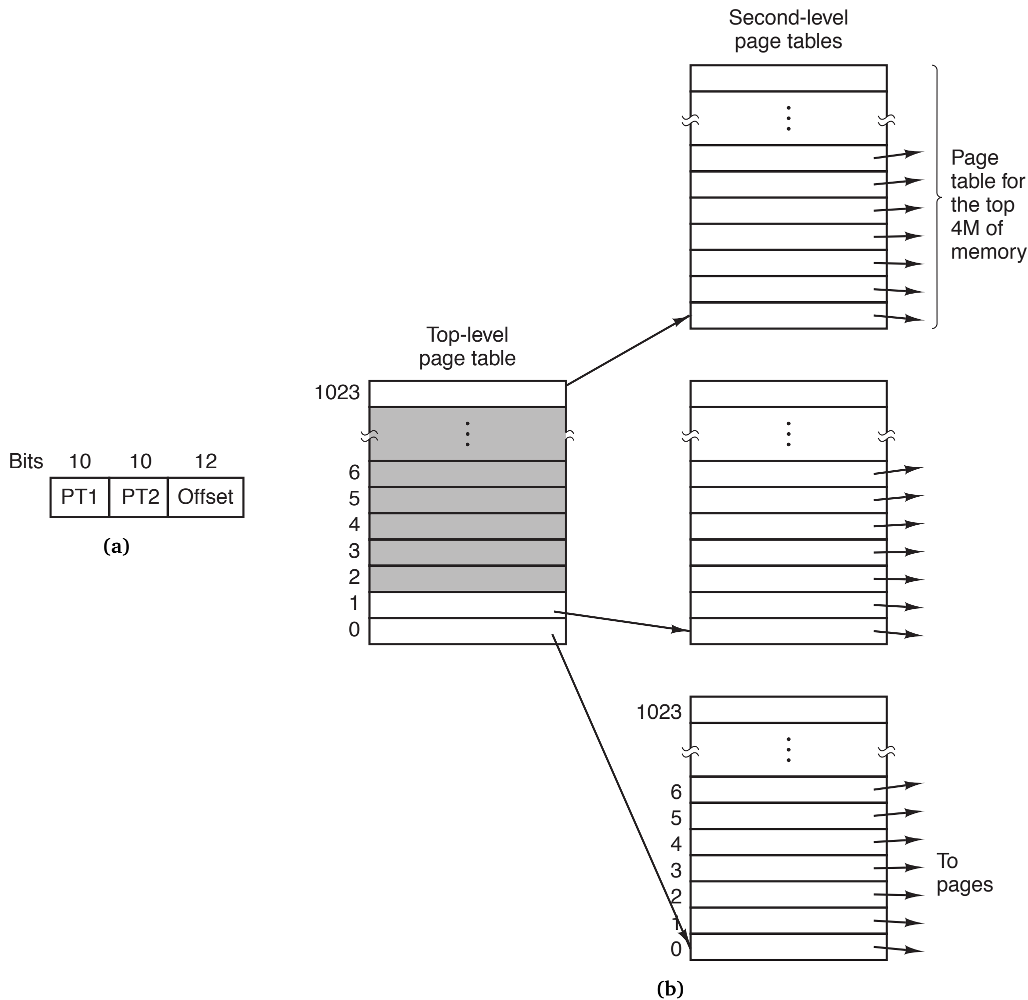


Figure 2.1 – (a) A 32 bit virtual address when using two-level page tables. (b) Two-level page tables [TB15, p. 206].

cache their translations with a small TLB (rarely more than 256 entries). So whenever the MMU has to translate an address, it checks the TLB first. If the TLB has a matching, valid entry, a table walk is not necessary [TB15, pp. 194–207].

2.2 Platform

The Raspberry Pi is a series of single-board computers and is used for this thesis because it provides a cost efficient and easy way to work with an ARM processor. More precisely, the Raspberry Pi 2 Model B is used, which is equipped with a 900 MHz quad-core ARM Cortex-A7 CPU [Ras15]. The Cortex-A7 is an ARMv7-A 32 bit processor with an in-order pipeline. It has a 64 kB 4-way set associative L1 cache with a pseudo-random cache replacement policy and a 1 MB 8-way set associative L2 cache with the same replacement policy [ARM13, pp. 1-5–2-6]. Figure 2.2 gives an overview of the processor. The Cortex-A7's TLB has a two-level hierarchy. The first level are two

10-entry fully associative TLBs, one for data and one for instructions (Harvard architecture), called micro TLB. The second level is a 256-entry 2-way set-associative TLB (von Neumann architecture), called main TLB [ARM13, p. 5-5].

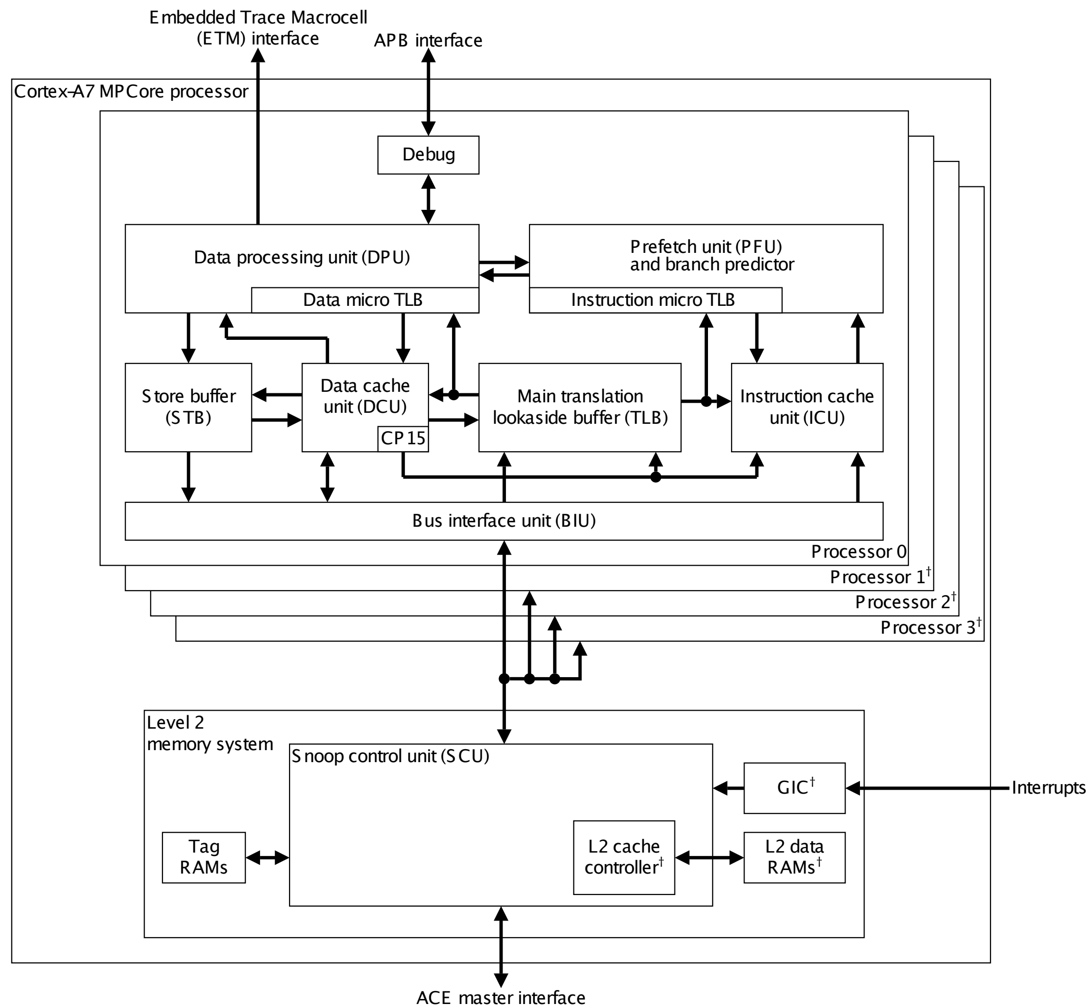


Figure 2.2 – A top-level diagram of the Cortex-A7 MPCore processor. It shows the two micro TLBs and the unified main TLB as part of each core. Both have a direct connection to the L1 cache [ARM13, p. 2-2].

In Chapter 5 hardware modifications are evaluated and therefore a simulator is required. The best solution is to simulate the ARMv7-A architecture using gem5. gem5 "is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture", which is written in C++ and Python [gem18]. It is open-source, licensed under a Berkeley-style open source license. The *Full-System mode*, which runs a complete operating system, supports the architectures Alpha, SPARC, x86 and ARM with CPU models of varying speed and level of detail, ranging from a simple one-CPI CPU to detailed models of in-order and out-of-order CPUs. In this thesis, the *TimingSimple* CPU model is used. It is based on the *AtomicSimple* (the

simplest) CPU model but simulates the timing of memory accesses, which is the key part for this thesis. Conveniently, the ARM models are able to boot an unmodified Linux. In addition, gem5 supports a *System-call Emulation mode*, in which it can run binaries using a Linux emulation, however, this mode is not used in this thesis [gem18; Bin+11].

2.3 Related Work

The impact of core-local TLBs in a multi-core environment is not a field without previous research. It has already been established that these do not offer the best performance.

Lustig, Bhattacharjee, and Martonosi found that Instruction TLB miss rates are almost negligible for PARSEC and SPEC workloads, while Data TLB miss rates are at 1.97 % and 5.19 % respectively. To fix this problem, they proposed two solutions: Inter-Core Cooperative (ICC) TLB prefetchers and Shared Last-Level (SLL) TLBs. By using ICC TLB prefetchers, which communicate information about strided access patterns among cores, or SLL TLBs, it is possible to decrease the number of Data TLB misses significantly. ICC prefetches can reduce the misses by up to 90 % and SLL TLBs by up to 79 % for parallel workloads. At last, they combined both approaches to improve the TLB hit rates slightly further [LBM13].

Reza and Byrd also showed that migrating the TLB can lead to a higher TLB hit rate when running a subset of SPEC CPU2006. They also showed that an age-based predictor can reduce the number of unnecessarily migrated entries significantly. However, they saw a more promising impact on performance when focusing on last-level cache misses and therefore followed that path [RB13].

Srikantaiah and Kandemir introduced Synergistic TLBs, which only require minimal hardware changes. Synergistic TLBs support capacity sharing, translation migration and translation replication, which means they can evict entries into the TLBs of other cores to use their storage and migrate or replicate them back, when necessary. Overall, a reduction of TLB misses by up to 44.3 % for SPEC applications was achieved [SK10].

Therefore, the idea of sharing entries between the TLBs of a multi-core processor is not new, but none of the presented approaches tried to implement it mainly in software, which is what is tried to achieve in this thesis.

3

ARCHITECTURE: TLB MIGRATION

This chapter describes the concept of TLB migration and how its building blocks are implemented on the ARM Cortex-A7 processor. Note that some of the instructions needed are implementation defined and therefore will not work on every ARM or even ARMv7-A CPU [ARM14, pp. A8-524, B3-1378].

3.1 The Concept of TLB Migration

As already stated, thread migration comes with a performance penalty due to cold caches, resulting in an increased number of cache misses but cannot be avoided sometimes. TLB migration is supposed to reduce this penalty by copying the relevant TLB entries to the TLB of the thread's new CPU. The relevance of these entries is determined by characteristics such as validity, associated address space and whether or not it was accessed since the last migration, if available.

Overall, a TLB migration consists of the following steps:

1. Extracting all entries from the local TLB because there is no way to tell which ones are relevant in advance (at least on the Cortex-A7).
2. Filtering out all entries which are irrelevant for the current thread, e.g., invalid entries or entries for other processes are dispensable.
3. Storing the remaining entries in memory so that they can be restored in the future.
4. Populating the new core's TLB by using the information stored in the memory the step before.

3.2 TLB Data Layout

The Cortex-A7 provides 86 bit of data for each TLB entry as listed in Table 3.1. To implement TLB migration, only the fields *Valid*, *VA* and *ASID* are relevant. The *Valid* field indicates whether or not an entry is valid. An invalid entry should, of course, be ignored. The Address Space Identifier (ASID) is used to determine whether an entry belongs to the address space of the current process. Note that it is impossible to differentiate between different threads of the same process as they usually share the same address space and therefore the ASID, which means that an entry of another thread could incorrectly be migrated sometimes. However, other than a little overhead that unfortunately cannot be avoided, no harm is done, because all (security-)restrictions imposed by the hardware

and operating system are still intact. If an entry belongs to another address space, it is ignored. The *VA* field is used to calculate the virtual address [ARM13, pp. 6-12 et sqq.].

The *VA* field is 13 bit wide, which is not enough to encode a 32 bit (virtual) address. However, it belongs to a page which is at least 4 KiB (2^{12} B) aligned and therefore only 20 bit are necessary. Moreover, the TLB is 2-way set-associative with a total of 256 entries, which means that each of the two ways has 128 entries and each address is stored in either one of those. Whenever a new entry is added to the TLB (or any 2-way set-associative cache, for that matter), the N least significant bits (after adjusting for the alignment), in this case seven (the number of bits required to index 128 entries), are used to determine its index. Therefore, the index itself represents the remaining 7 bit of the address, obviating the need to store these bits.

Bits	Name	Description
[0]	**Valid**	Valid bit, when set to 1 the entry contains valid data.
[3:1]	Size	This field indicates the VMSA v7 or LPAE TLB RAM size.
[4]	NS, walk	Security state that the entry was fetched in.
[17:5]	**VA**	Virtual address.
[25:18]	VMID	Virtual Machine Identifier.
[33:26]	**ASID**	Address Space Identifier.
[34]	nG	Not global.
[37:35]	AP or HYP	Access permissions from stage-1 translation and HYP mode flag.
[39:38]	HAP	Hypervisor access permissions from the stage-2 translation.
[40]	NS, descriptor	Security state allocated to memory region.
[68:41]	PA	Physical Address.
[69]	PXN	Privileged Execute Never.
[70]	XN1	Stage-1 translation Execute Never bit.
[71]	XN2	Stage-2 translation Execute Never bit.
[77:72]	Memory Type and shareability	Information about memory type and shareability.
[81:78]	Domain	Only valid if the entry was fetched in VMSAv7 format.
[83:82]	S1 Size	The stage 1 size that gave this translation.
[85:84]	S2 Level	The stage 2 level that gave this translation.

Table 3.1 – The data fields of a main TLB descriptor. The fields relevant for TLB migration are highlighted [ARM13, pp. 6-12 et sqq.].

3.3 TLB Access Path

The access path to the TLB of the Cortex-A7 is separated into a read and a write path. Read access is provided using four registers which are part of the system coprocessor interface but only to the main TLB. There is no way to access the micro TLB (and therefore the Data TLB) directly. A write to the write-only “TLB Data Read Operation Register” encoded as shown in Table 3.2 fills the read-only “Data Register 0” through “Data Register 2” with the data described in Section 3.2. Therefore, it takes a total of four instructions to read all data for a single TLB entry as Listing 3.1 demonstrates [ARM13, p. 6-9].

Unfortunately, a closer look into the Cortex-A7 Reference Manual reveals that TLB access is only available in secure privileged modes, which are part of the ARMv7-A Security Extensions. These extensions define two security states, *Secure state* and *Non-secure state*. Additionally, the

Bit-field	Description
[7:0]	TLB index
[30:8]	Unused
[31]	TLB way

Table 3.2 – The encoding for the TLB Data Read Operation Register. Writing accordingly encoded data to the register will populate the TLB Data Read Operation Registers [ARM13, p. 6-11].

```
1 mcr p15, 3, r0, c15, c4, 2 // write to operation register
2 mrc p15, 3, r0, c15, c0, 0 // extract entry from data register 0-2
3 mrc p15, 3, r1, c15, c0, 1
4 mrc p15, 3, r2, c15, c0, 2
```

Listing 3.1 – Reading a TLB entry using the system coprocessor interface.

Raspberry Pi's closed source bootloader boots Linux in *Non-secure state* with no way to get into *Secure state*. However, after reverse engineering the GPU firmware, which contains the bootloader (the BCM2836's boot process includes that the GPU starts the CPU), the bootloader can be patched to boot the CPU in the *Secure state*. Linux boots successfully in the *Secure state* in which it will be evaluated from now on. It is outside of this thesis' scope to modify Linux to switch between the states [ARM14, pp. A2-38, B1-1156][ARM13, p. 6-9][jam15; Kom13; dom15].

Another thing to keep in mind is ASID rollover. ASID rollover is the mechanism with which Linux handles the fact that the ASID is only 8 bit wide and will become more relevant later on. An 8 bit ASID allows the simultaneous usage of 256 distinct global address spaces, presenting a major limitation. Therefore, Linux needs to reassign ASIDs every time this limit is exceeded to guarantee that there are always enough ASIDs available. In other words: Every time 255 (ASID 0 is reserved) processes were executed, the ASID could have changed. The quintessence is that ASIDs are not constant and TLB migration needs to circumvent that [Dea15].

To gain write access, another workaround is necessary as the provided access path is read-only. ARMv7 defines an instruction to preload data which will likely be used in the near future called `pld`. The Cortex-A7's implementation of this instruction starts a cache linefill for a given address, which, in order to be fetched from the memory, has to be translated into a physical address, affecting the TLB's state in adding the appropriate entry [ARM13, p. 6-8]. The convenient circumstance that the TLB entry persists even after the instruction is finished allows to exploit this method to write to the TLB and since the instruction only needs a virtual address as parameter, no other data has to be stored when reading the TLB. Furthermore, since a new entry is created, it is guaranteed that the ASID is up-to-date. The downside is that the instruction performs a superfluous memory access as the data it preloads is not needed. However, this access happens in the background and therefore should be negligible when it comes to execution times.

In order to use this access as sensibly as possible, another, more indirect, approach is to preload the page descriptor instead of data inside this specific page. Because a table walk can be performed on cached page descriptors, the locality of those data structures in memory can be utilized [ARM13, p. 5-3]. This comes with the penalty of calculating the address of the second-level descriptor for each page. Although the top-level page table's address can be read from a register (so the address of the top-level descriptor can be calculated without a memory access), reading the data takes an additional memory access. In the best case this lookup is already cached. All in all, this approach

really depends on how the cache works and it is difficult to make any statements on its effectiveness without taking any measurements.

4 ANALYSIS: TLB MIGRATION

In this chapter the different components of TLB migration are analyzed. Each measurement is repeated 2500 times and taken using the Performance Monitoring Unit's cycle counter [ARM13, p. 11-1]. The displayed values are the resulting medians as average values are distorted by extreme outliers due to memory refresh. Interrupt and fast interrupt requests are disabled (this applies to all future benchmarks). To prevent dynamic frequency clocking from influencing the results, `arm_freq_min` is set to 900 MHz, the Pi's base clock, which guarantees that the processor runs at the same clock speed the whole time [Ras18].

4.1 TLB Read Access

To understand the whole migration process, it is useful to look at its parts first. As described in Section 3.3, reading the TLB is achieved by just register operations. To verify that expectation, a benchmark is used which measures the time of N TLB read operations. The results are shown in Figure 4.1. Moreover, additional measurements showed that a write to the operation register takes 4 cycles, while a read from a data register takes 1 cycle, making up a total of 7 cycles that are required to read a single entry.

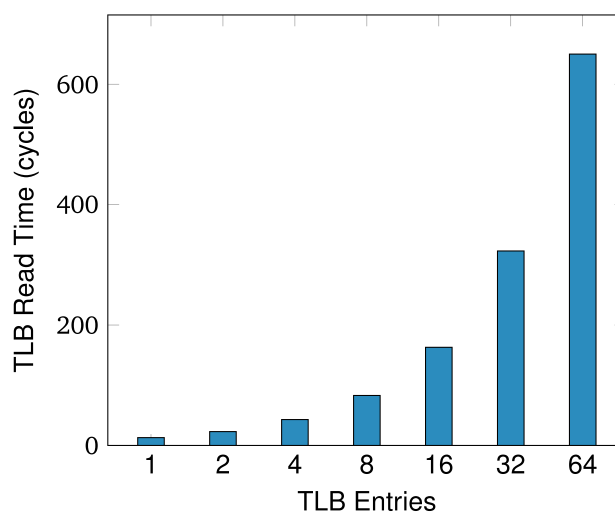


Figure 4.1 – The number of cycles required to read different numbers of entries from the TLB without storing them in memory.

When looking at the results, the access time for one TLB entry with 13 cycles is higher than the expected 7 cycles. However, this is easily explained by the loop around the register accesses used to iterate over them and the update of the data for the operation register taking up a significant proportion of the number of cycles. When accessing more entries, 10 cycles for each additional TLB entry are added, which seems reasonable when the access itself takes 7 cycles. The only exception is at 64 entries where the read takes 650 cycles instead of 643 cycles, but generally the access times scale, as expected, linearly with the number of accessed entries.

4.2 The `pld` Instruction

The other important part of the migration process is writing to the TLB using the `pld` instruction. The benchmark is a simple loop over a `pld` instruction preloading *N* different consecutive pages with clean data caches and TLB, as well as data prefetching disabled. The naive expectation is that only a single cycle is required per page as the data itself is fetched in the background. However, reading the reference manual closely reveals that the “instruction retires as soon as its linefill is started”, which can be interpreted such that the table walk happens before the instruction retires. In that case, multiple hundred of cycles for each instruction are expected because a table walk can cause two memory accesses [ARM13, p. 6-8]. A memory access can be assumed to take about 150 cycles on average but can vary a lot in reality. Having said this, the actual results of the microbenchmark are shown in Figure 4.2.

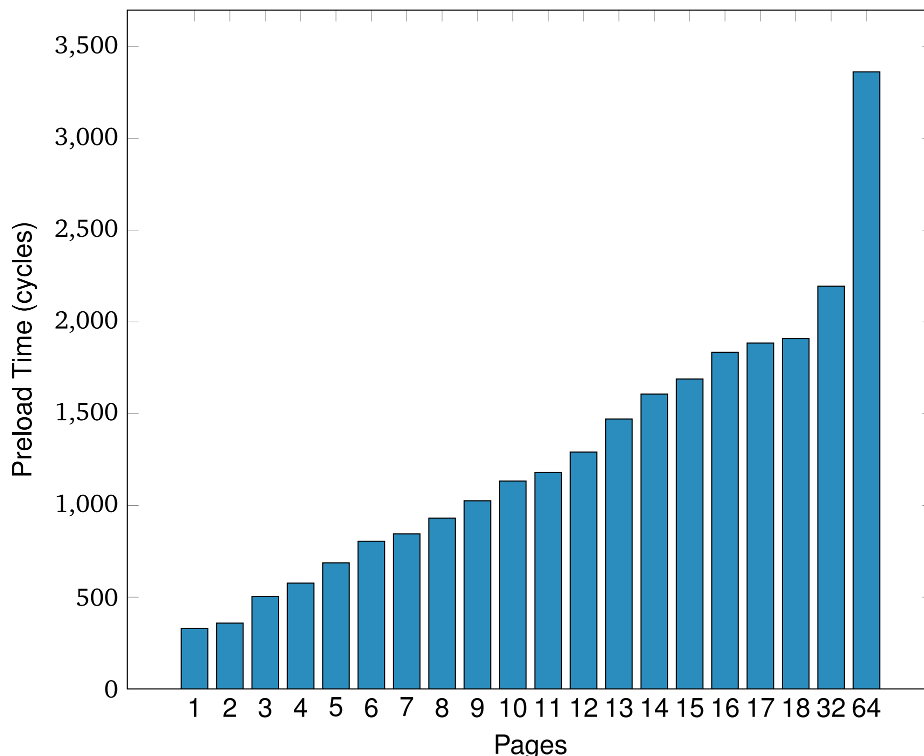


Figure 4.2 – The number of cycles required to preload different numbers of consecutive pages using the `pld` instruction.

The first thing to be noticed is that the `pld` instruction takes 329 cycles for a single page, the duration of two memory accesses. This suggests that the table walk does indeed happen in the foreground. For two pages the instruction takes virtually the same time with 359 cycles and not at all the expected doubled time. However, this can be explained by taking a look at the memory layout of page tables described in Section 2.1. Generally speaking, top-level page descriptors can always be assumed to be cached (except for the first page). For the sake of this explanation, it is assumed that the first page belongs to the first descriptor in the second-level page table and the second page to the second descriptor. This means that the descriptors have consecutive locations in memory, as shown in Figure 4.3. Whenever the first descriptor is accessed during the table walk for the first page, both descriptors are loaded into the same cache line (assuming the appropriate alignment). Since the MMU can perform a table walk in cache, the table walk for the second descriptor does not require any memory accesses. The only memory access remaining, the one for the data itself, is handled in the background. This means that when performing accesses to two or more consecutive pages, the second and following accesses are much faster because the whole cache line of 64 B, or 16 page descriptors, is used [ARM13, pp. 6-2, 7–2].

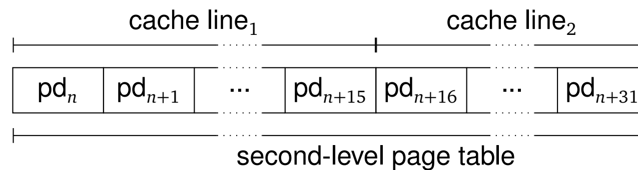


Figure 4.3 – The page descriptors' memory layout in a second-level page table. Whenever a page descriptor is accessed, the other 15 descriptors belonging to that cache line are loaded into cache as well (assuming the appropriate alignment). So accessing the 32 page descriptors manifests itself in just two memory accesses.

Having said this, this effect should also apply when preloading three pages, which is too slow with 503 cycles. Interestingly, this pattern repeats, sometimes less obviously, about every two extra pages. To get a better understanding of the phenomenon, the number of cache accesses and misses are measured as well using the Performance Monitoring Unit [ARM13, p. 11-10]. It should be noted that cache accesses during table walk handled by the MMU seem to not be reported. The results in Figure 4.4 demonstrate that the caching effects do indeed work as expected. The first page generates three cache misses: for the top- and second-level table descriptors and the page itself. For every additional page only one extra cache miss is added, because the descriptors are already in cache. This is not the case for 13 pages, presumably because the descriptors are distributed over a cache line border, which has to happen every 16th page. But since the cache seems to work as intended, the reason for the slowdown when accessing three pages has to lie somewhere else.

Obviously, hardware is finite and so has to be the number of simultaneous data preloads which can be started by the `pld` instruction. The numbers indicate that this limit seems to be at two, which would explain the observed pattern. As depicted in Figure 4.5, the first two background accesses could be handled at the same time, which corresponds with the results. The third access would have to wait before the first access has finished, which fits the data, too. The fourth access would have to wait for the second access to have finished, but since the time between the first and second access can be assumed to be about the same as the time between the third and fourth, the effect is negligible. Deviations could be explained by variations of the access times causing the pipeline to get a little disorganized, e.g., extra wait time or no wait time at all where it is expected.

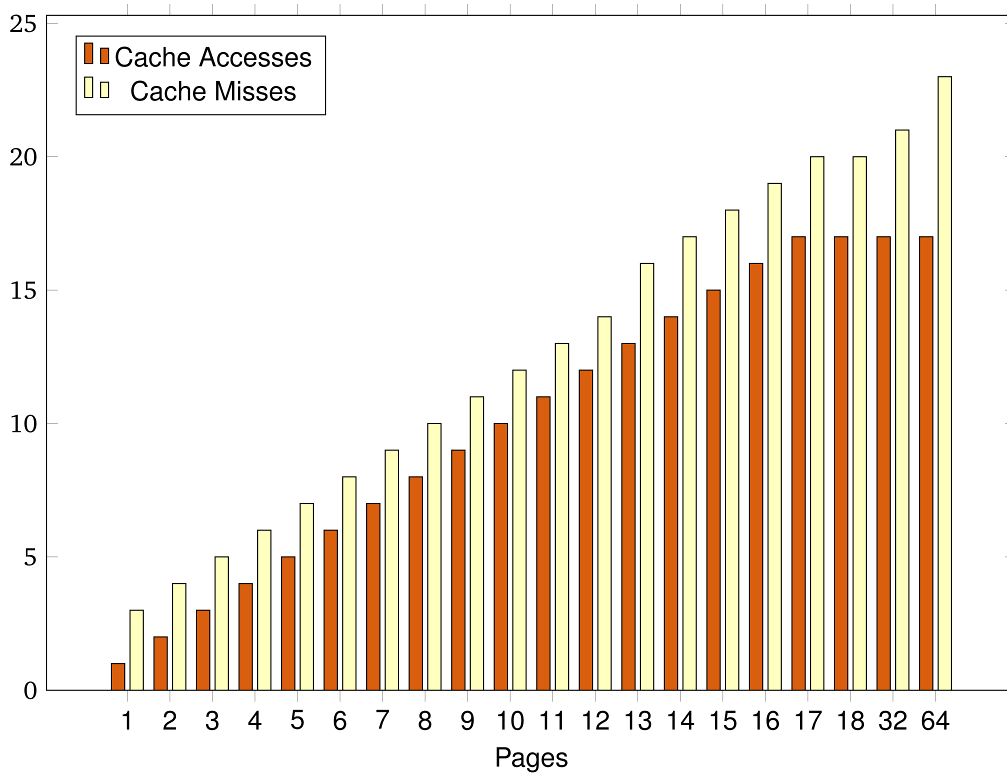


Figure 4.4 – The number of cache accesses and misses when preloading different numbers of consecutive pages using the pld instruction. Note that cache accesses of the MMU are not accounted for.

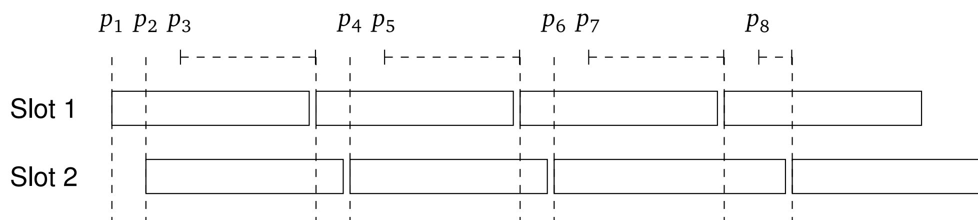


Figure 4.5 – The expected behavior in case two pld instructions are executable at the same time. The first two pld instructions, p_1 and p_2, could be executed as soon as they are dispatched, p_3 has to wait until one has finished. Assuming a constant execution time, p_4 could be executed instantaneously afterwards. Since p_6 takes longer, p_8 has to wait.

Furthermore, Figure 4.4 shows the number of accesses and misses not growing as expected for an increased number of pages. In fact, the count of cache accesses is the same for 17, 18, 32 and 64 pages. This correlates with the slower than expected growing preload times in Figure 4.2, which indicates that the pld instruction does not work as expected for large numbers of pages. However, it is not contradictory to the at the same time increasing count of cache misses as they could originate in the MMU. Adjusting the stride of the memory accesses or adding NOP instructions offers no additional insight into what happens. Since ARMv7 does not require the instruction to do

anything and its effects in the Cortex-A7 are non-functional, it is reasonable to assume some kind of simplification in its implementation [ARM14, p. A8-528]. More on that in Section 4.3.

To conclude, the usage of the `pld` instruction is without alternative but rather subpar because it performs the table walk in the foreground and generates an unnecessary memory access that sometimes adds overhead.

4.3 Migration Process

After examining both components of the TLB migration, the whole process is simulated. To determine its effectiveness, it is necessary to have some kind of benchmark, for which it is important to understand the factors affecting its performance. The most obvious factor is the number of accessed pages because, as explained in Section 2.1, it is possible to save up to two memory accesses per page. Therefore, more pages mean more memory accesses to save. The second and more complicated factor is the pattern in which the memory accesses are performed because the table walk can work with page descriptors located in cache. As explained in Section 4.2, when performing the table walk for the second of multiple consecutive pages, the page descriptor is most likely already cached, speeding up the access by an order of magnitude and diminishing the possible performance gains.

Therefore, two benchmarks, implemented as loadable kernel modules for Linux, were developed to represent two extremes. One with a completely sequential access pattern and one with some space between each accessed page. Since a cache line is 64 B wide, it fits 16 page descriptors managing 64 KiB memory in total, or 4 KiB each. Hence, a 64 KiB stride guarantees a cache miss on the second-level page table and represents a less densely packed memory. However, it also causes the same 8 of the 128 possible indices in the TLB to be used all the time (16 is a factor of 128). To prevent that, the stride is increased by one to 17 pages in total resulting in a 68 KiB stride. The benchmarks, called 4k- and 68k-benchmark from now on, work as follows:

1. Flushing all data caches, including the TLB, to simulate a thread migration without TLB migration.
2. Accessing N pages, using the given access pattern.
3. Storing the relevant TLB entries in memory to restore them later.
4. Flushing all data caches, including the TLB, again to simulate a thread migration.
5. Restore the TLB, using the data stored in memory.
6. Accessing the N pages, using the same access pattern again.

By comparing the duration of step 2 and step 6 the speedup can be determined. Step 2 resembles a thread execution after thread migration without TLB migration since all caches are empty, while step 6 corresponds to an execution with TLB migration since the TLB has been restored and therefore memory accesses should be faster. However, the gains come at the cost of the duration of step 3 and 5. The difference between costs and speedup makes up the overall speedup.

TLB migration starts with extracting all relevant data from the TLB and storing it in memory to save its current state. Figure 4.6 shows the cycles required to perform this task for different numbers of pages and both benchmarks. The high number of cycles it takes to perform the task stem from the fact that all 256 TLB entries have to be read before discarding the irrelevant ones. The cycle count increases slightly with an increased number of pages because more addresses have to be stored in

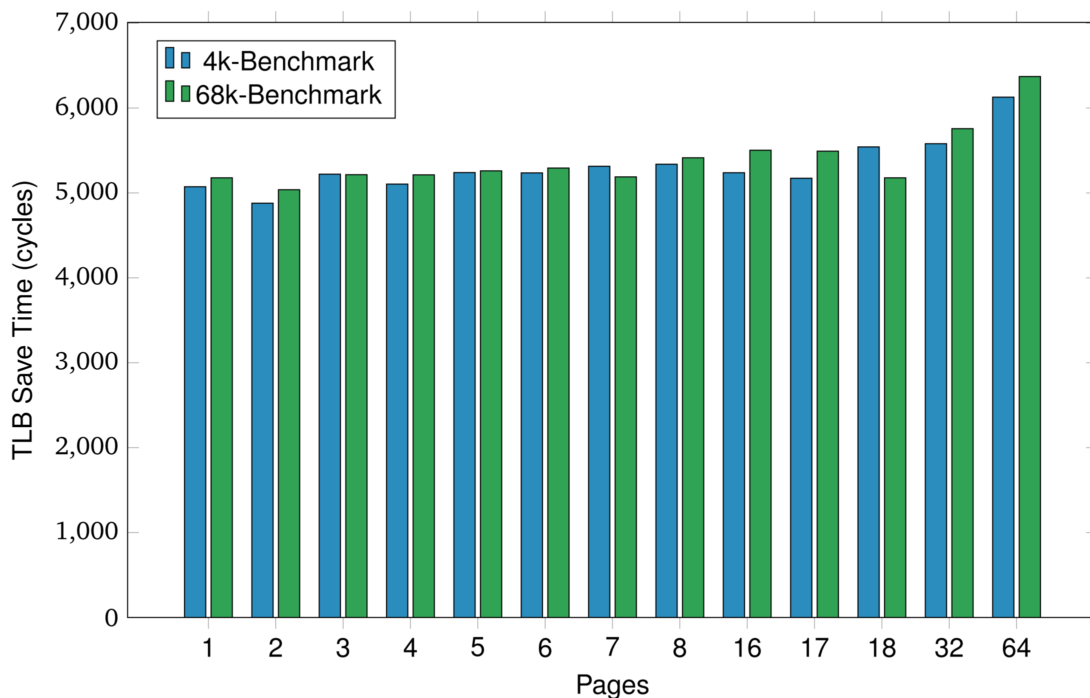


Figure 4.6 – The number of cycles required to read all TLB entries and store the relevant ones in memory for different numbers of pages. The measurements are taken using the 4k- and 68k-benchmarks.

memory. But since there are inevitably more TLB entries for the current address space than the ones used for the benchmark, an additional entry and therefore sequential write for the page used in the benchmark does not make much of a difference. The results are more or less constant read times around 5250 cycles. Only for a significant increase in pages some effects are observable. Since Section 4.1 suggests that about 2800 cycles are required to just read the entries from the TLB, an additional 2450 cycles to filter and, most notably, store the data in memory seem reasonable.

The second step is restoring the TLB, which includes reading the addresses from memory and writing them to the TLB with help of the `pld` instruction. The number of cycles required for this step in both benchmarks are depicted in Figure 4.7. The 4k-benchmark cannot be directly compared with the results from Section 4.2 due to the mentioned additional entries in the address space and the fact that the addresses which will be preloaded have to be fetched from memory. Both of which result in an increased number of cycles. Additionally, since the duration of a memory access tends to fluctuate a lot, more memory accesses mean more fluctuation. However, there is some useful information in this data which manifests itself for a higher number of pages. The time needed to restore the TLB in the 68k-benchmark is significantly longer as no table walk can be completed from cache. Furthermore, the previously observed abnormalities for more than 16 pages are not yet visible.

After analyzing the costs of the migration, it is to discuss whether or not it is worthwhile under the given circumstances. The memory access times (step 2) with flushed caches and for different numbers of pages in Figure 4.8 demonstrate where the time can be saved. The migration has the side effect of bringing the first 64 B of a page into cache because it uses the `pld` instruction, so the

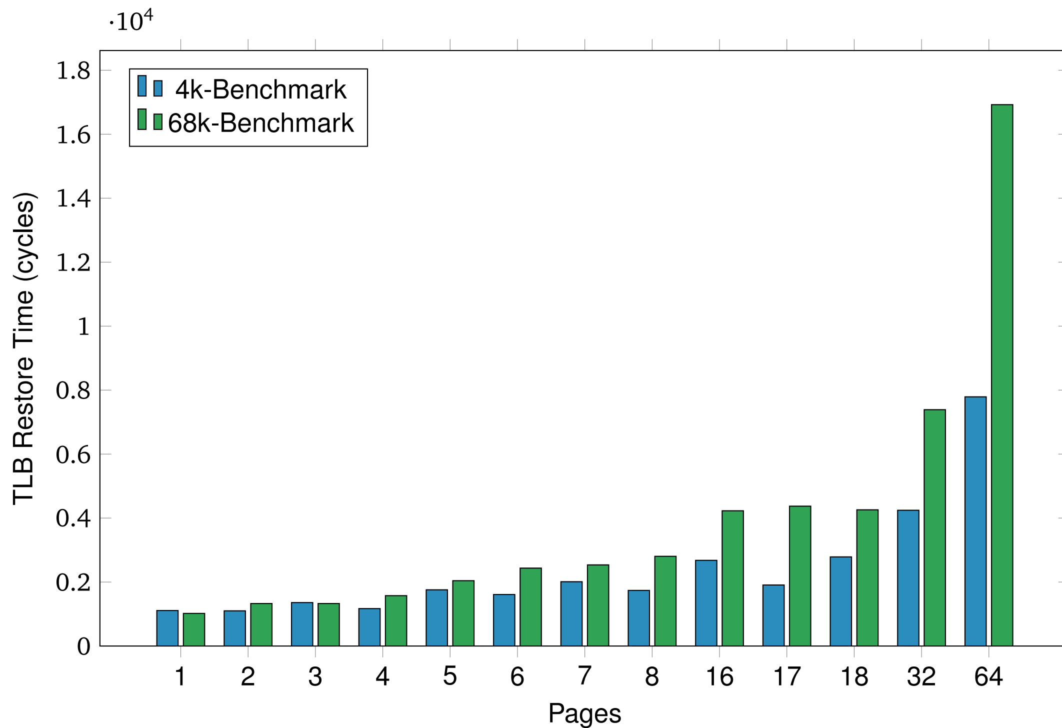


Figure 4.7 – The number of cycles required to load the TLB entries from memory and write them to the TLB for different numbers of pages. The measurements are taken using the 4k- and 68k-benchmarks.

memory accesses are performed with an offset into the page. The figure gives a general overview of how the memory access times compare. There are quite a few similarities to what was discussed in Section 4.2. So in short: One page access generates three memory accesses (first-level page table, second-level page table and the address itself) and therefore takes 429 cycles. The value for two page accesses stands out a little because for some reason an extra access occurs. Additionally, five, six and seven pages take virtually the same time with 972 cycles, 981 cycles and 985 cycles in the 4k-benchmark, which can be explained by data prefetching. The Cortex-A7 is able to detect fixed stride memory access patterns of three or more accesses and prefetches them in the background [ARM13, p. 6-8]. Since this would also happen with real world workloads, it is deliberately left enabled. However, setting the data prefetching interference aside, it becomes clear that for a higher number of pages the accesses in the 68k-benchmark take about twice as long, e.g., 3469 cycles vs. 6445 cycles for 32 pages. As explained before, this is because the table walk generates a cache miss and the resulting additional memory access for every page in the 68k-benchmark. Overall, the conclusion is quite simple: When assuming that the first-level descriptor is always in cache, only one memory access (for the second-level descriptor) for every 16th page can be saved in the 4k-benchmark. In the 68k-benchmark this is possible for every page.

After exploring where the speedup can be achieved, the speedup itself is discussed in Figure 4.9 while ignoring the costs for now. The improvements in the 4k-benchmark fluctuate around an equivalent of one to two memory accesses for up to 16 pages. For 64 pages the performance gains are higher with about five accesses as the number of required cache lines for all page descriptors are higher, too. Interestingly, a slowdown occurs for exactly 18 pages. However, this is about the number

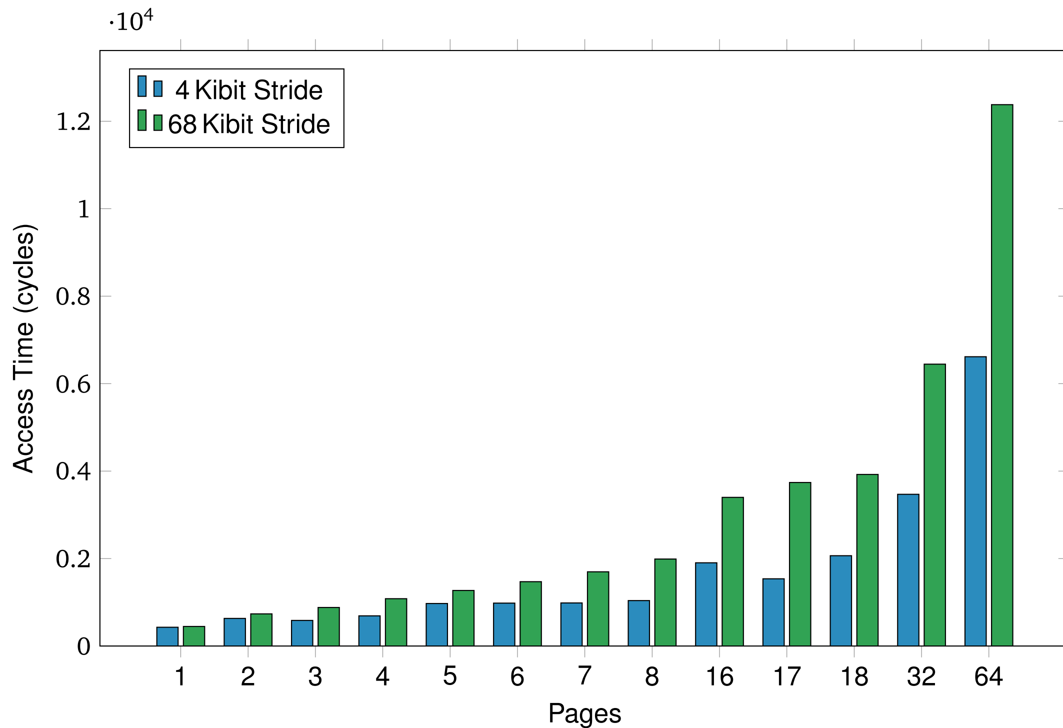


Figure 4.8 – The number of cycles required to access different numbers of pages with cold caches and strides of 4 Kibit and 68 Kibit.

of pages where the `pld` instruction shows the aforementioned strange behavior (see Section 4.2). The 68k-benchmark shows more promising results. The number of cycles saved increases with every extra page (with the exception of 18 pages). But most interestingly, the speedup exists also for a larger number of pages where the behavior of the `pld` instruction is inconclusive. The maximum speedup is achieved for 64 pages with 6225 cycles, which is with just a little under 100 cycles/page a little less than expected, but taking data prefetching into account, it seems reasonable. Overall, the migration process seems to work.

Altogether, the migration itself is working and delivering expected improvements regarding memory access times. However, the only case in which the gains come close to compensating for even the cost of saving the TLB, is for 64 pages in the 68k-benchmark where restoring the TLB takes 16 924 cycles.

The indirect approach presented in Section 3.3 could solve this problem. Since it is clear that it will not be possible to achieve a speedup in the 4k-benchmark, only the 68k-benchmark is considered from now on. Saving the TLB is unchanged and therefore is not discussed again. Restoring the TLB, however, looks promising (see Figure 4.10). The difference between the direct and indirect approach increases with the number of pages. For one page they are almost equal with 1019 cycles and 1098 cycles respectively, but for 64 pages the indirect approach is 2.5 times faster. The reason for this difference is the fact that instead of using only the side effects of the `pld` instruction, data relevant to the benchmark is preloaded. And since this data is inside the paging data structures, all memory accesses - table walk and preload - are highly localized maximizing caching effects. This means a high cache load, but, at least for this benchmark, these effects are negligible as Figure 4.11

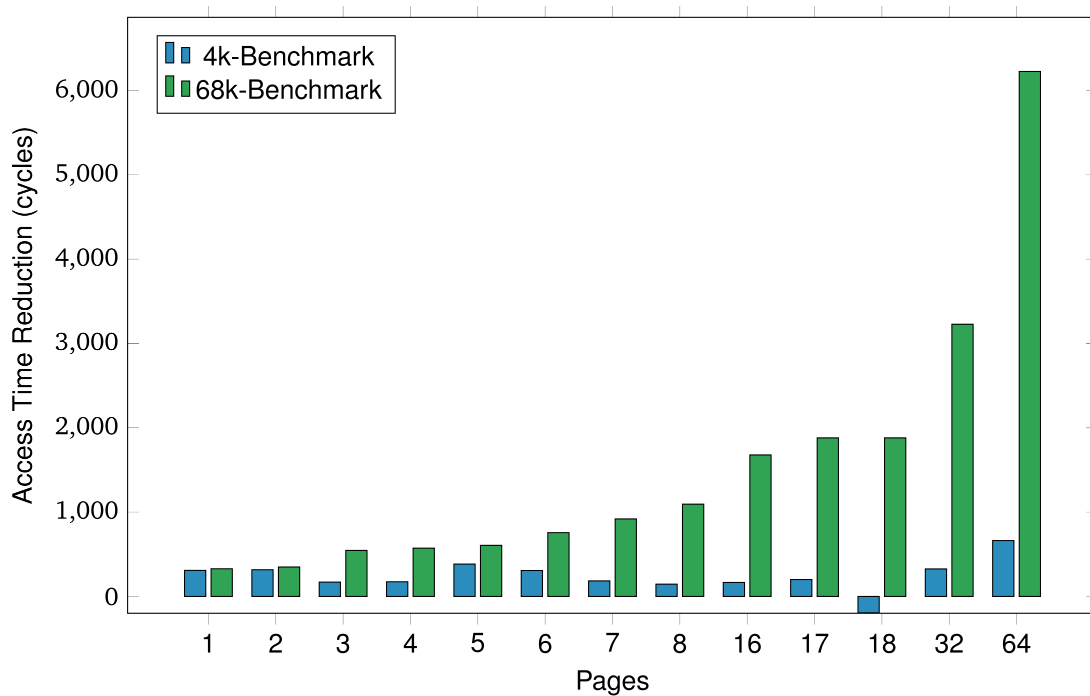


Figure 4.9 – The number of cycles that the memory access times are reduced by through TLB migration for different numbers of pages. The measurements are taken using the 4k- and 68k-benchmarks.

demonstrates. The improvement in memory access times is very similar to the ones using the direct approach. To summarize, the indirect approach is faster due to the more efficient usage of the `pld` instruction. However, just like before, the improvement of 6323 cycles are barely enough to compensate for the cost of saving the TLB taking 6321 cycles.

All in all it becomes clear that the use of the `pld` instruction causes major problems and is not sufficient for TLB migration. Additionally, speeding up the TLB read access would be nice as it is not possible to achieve an overall speedup for a small number of pages otherwise.

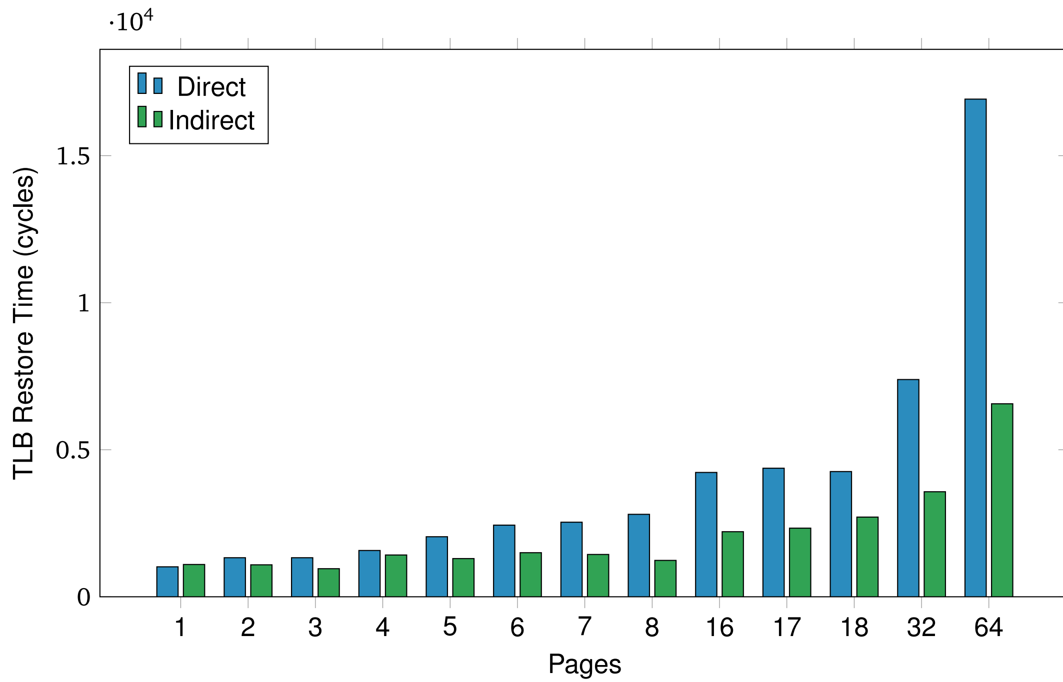


Figure 4.10 – The number of cycles required to load the TLB entries from memory and write them to the TLB, using the direct and indirect migration approaches. Measured using the 68k-benchmark for different numbers of pages.

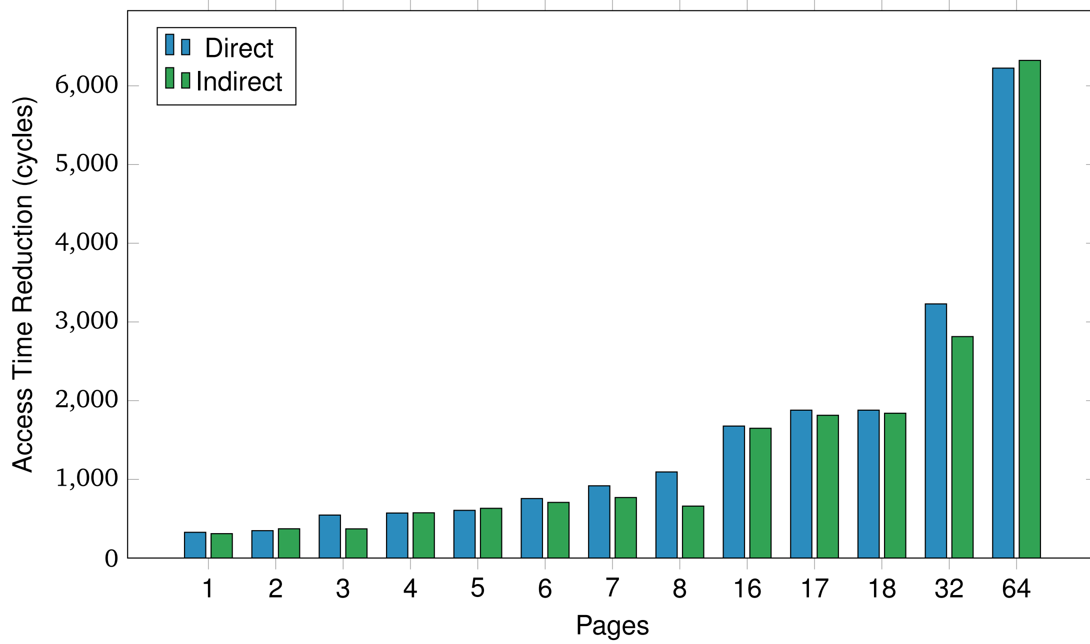


Figure 4.11 – The number of cycles that the memory access times are reduced by through the direct and indirect TLB migration approaches. Measured using the 68k-benchmark for different numbers of pages.

5 ARCHITECTURE: HARDWARE MODIFICATIONS

As the previous approaches to implement TLB migration seem nowhere near fast enough to provide an overall speedup, this chapter presents a different approach which modifies the underlying hardware. However, since real hardware modifications are not practical, the simulator gem5 is modified instead. Because the problems are caused by the time needed to interface with the TLB, mainly when using the `pld` instruction, the goal is to speed up this interface.

5.1 TLB Data Layout

gem5 does not provide an interface to the TLB like the Cortex-A7 does, which is why it needs to be implemented (more in Section 5.2). Since gem5's internal TLB entry representation differs from the Cortex-A7's one and the gem5 data layout needs to provide sufficient information to recreate a gem5 TLB entry, the new layout differs as well.

In the context of this thesis three data layouts are designed, each of which serving a different purpose. Version 1's function is to be as similar as possible to the Cortex-A7 layout regarding its size. Version 2 is a smaller variant of version 1 achieved by a more sophisticated encoding. For example, the page size, or page shift factor, takes up 7 bit in gem5 but is a binary decision in practice, which means only 1 bit is needed. Furthermore, an analysis of occurring values shows that the lookup level for example has in theory five possible values in gem5, but since the simulated processor only has L1 and L2 caches, 1 bit is sufficient. The end result is a size reduction by nearly a third. Lastly, version 3 is a data layout for TLB entry streaming (more in Section 5.2) where some fields like *asid* or *valid* can be omitted to further reduce its size. Table 5.1 gives an overview of the different versions.

5.2 TLB Access Path

The read path for the version 1 data layout mostly resembles the Cortex-A7's. A write to the write-only "TLB Data Read Operation Register" fills "Data Register 0" through "Data Register 2" with the data described in Section 5.1. However, the operation register encoding differs because gem5 does not implement the main TLB and compensates for it with larger micro TLBs (64 instead of 10 entries) which are fully associative with a least recently used replacement policy. Since the used benchmark profits mostly from Data TLB migration, and Instruction TLB migration does not promise a significant impact, the focus of the analysis lies on the Data TLB [LBM13]. Therefore, only an index is required, which simplifies the encoding for the TLB Data Read Operation Register to said index.

Name	Size V1	Size V2	Size V3	Description
asid	8	8	0	Address Space Identifier
valid	1	1	0	Valid
N	7	1	1	Page shift factor
v_addr	20	20	20	Virtual address
p_addr	20	20	20	Physical address
ap	8	2	2	Access permissions
attributes	12	2	2	Memory attributes formatted for PAR
innerAttrs	3	2	2	Inner attributes
outerAttrs	1	1	1	Outer attributes
domain	3	2	2	Domain type
mtype	2	1	1	Memory type
nonCacheable	1	1	1	Not cacheable
shareable	1	0	0	Shareable
xn	1	1	1	Execute Never bit
lookupLevel	3	1	1	Lookup level
global	1	1	1	Global
Total	92	64	55	

Table 5.1 – A size comparison of the different TLB data fields (sizes in bit).

Version 2 works similarly but without the need for Data Register 2 as the data fits in two registers, reducing memory accesses by a third when reading and writing the entries to/from memory.

Since the usage of the `pld` instruction for the write access to the TLB on the Cortex-A7 is the main reason for why the implementation is too slow, the new write path has to be different. A better instruction would be one which behaves like `pld` without the redundant memory access in addition to the table walk. However, to achieve the highest possible performance, utilizing the fact that the read path provides sufficient information to create the TLB entries directly, without additional memory accesses like a table walk, is a more promising option. It has the slight downside that only storing the virtual address is not sufficient anymore. An additional problem is the expiry of TLB entries while they are in memory, which has not only functional consequences but also security implications. Fixing this is out of scope of this thesis and is therefore not realized. The simplest way to create TLB entries is to enable write access on the read path by reversing the current procedure. Writes to the three (version 1) or two data registers (version 2) respectively, prepare the data. A write of `0xffffffff` to the operation register commits the entry in version 1. A constant is sufficient, because specifying an index is not necessary and in fact would create new problems as current information about the TLB's content would be required to determine an appropriate index. In version 2, the write to the last data register already commits the entry to minimize operations, which means that the correct write order is important.

Nonetheless, the problem of reading the whole TLB to determine which entries to migrate still persists. To solve this, the hardware has to filter the entries by itself, resulting in a required interface change. This has the side benefit of the hardware being able stream out all the entries without requiring a request in form of an index, reducing the work to be done even further, although this assumes a more special purpose usage of the interface as it removes the capability to read specific entries or the unfiltered TLB. Writing the ASID to the operation register initiates a stream of all valid TLB entries matching this ASID, terminated by zeroed data registers. As a bonus, the validity and ASID of each entry are already known, obviating the need to include them in the data stream, resulting in data layout version 3. Moreover, as filtering is performed in hardware, the software does

not have to interpret the data. Also the ASID rollover is taken care of. Where it was necessary to update each entry every time before writing it, it is now handled implicitly during the initialization of the stream. In other words, the migration now consists of reading data from the TLB, storing it in memory and writing it back when necessary. Therefore, the obvious way to increase performance further is to minimize the data to be moved. To achieve that, a used bit for TLB entries is introduced. However, this should not have any significant impact on the benchmark as all pages are accessed. The last implemented improvement is to byte pack the data stream, resulting in version 4 of the TLB interface. Out of the 64 bit (two 32 bit data registers) only 55 bit are needed to encode an entry, allowing the migration to save 1 B for each entry, 64 B (or one cache-line) in total. When writing to the TLB, the stream has to be terminated by zeroed data registers to signal the hardware when to start unpacking the stream.

To summarize, in this chapter four TLB interfaces using three data layouts were designed. Each of it to be faster than its predecessor by minimizing the transfered data and required register operations.

6 ANALYSIS: HARDWARE MODIFICATIONS

After designing the new TLB interfaces, it is time to evaluate their caused speedups. This chapter gives an overview of how conclusive the results from gem5 are, before the interfaces themselves are compared. Eventually, the final results of the gem5 modifications are discussed. gem5 does not implement the Performance Monitoring Unit, so the Linux-provided `ktime_t` is used, which offers nanoseconds-resolution time [GM16]. Admittedly, the resulting numbers cannot be expected to be of the same precision as before. The shown results of previous benchmarks are converted from cycles to nanoseconds. To maintain comparability, all results are medians of 2500 runs, just like previously. Since the 4k-benchmark will not achieve any speedup, the focus is on the 68k-benchmark.

6.1 Cortex-A7 vs. gem5

To evaluate the previously designed TLB interfaces, the simulated system as in Listing 6.1 is used. The specific values are chosen to resemble the Cortex-A7 as closely as possible.

```
./build/ARM/gem5.opt configs/example/fs.py \
  --machine-type=VExpress_GEM5_V1 \
  --dtb-file=system/arm/dt/armv7_gem5_v1_4cpu.dtb \
  --kernel=../gem-linux/vmlinux \
  --num-cpu=4 \
  --sys-clock='900MHz' \
  --cpu-clock='900MHz' \
  --caches \
  --l2cache \
  --num-l2caches=1 \
  --l2_size='512kB' \
  --l1d_size='32kB' \
  --l1i_size='32kB' \
  --cpu-type=TimingSimpleCPU
```

Listing 6.1 – The system configuration for gem5 used to replicate the Cortex-A7.

Having said this, comparisons between simulator and real hardware are always problematic because simulators can only estimate the real hardware's behavior. As pointed out before, the way the TLB is implemented in gem5 differs significantly from the Cortex-A7's implementation. To get an approximation of how the interfaces and the simulator compare with the Cortex-A7,

the 68k-benchmark is run in the simulator using the version 1 TLB interface, which replicates the Cortex-A7's. The results in Figure 6.1 suggest that gem5 behaves at least similar to the Cortex-A7 when it comes to memory accesses. The Cortex-A7 needs 497 ns to access one page with flushed caches, while the simulation takes 414 ns for the same operation. Up until eight pages both need about the same time with sometimes one being faster, sometimes the other. For more than eight pages, the Cortex-A7 is increasingly faster compared to the simulator. This effect is unfortunate but can at least presumably be explained by data prefetching as the access times in gem5 seem to scale linearly, e.g., accesses to 64 pages take nearly exactly twice as long with 18 414 ns as accesses to 32 pages with 9156 ns. This is not the case on the Cortex-A7 where accesses to 64 pages take 13 754 ns and to 32 pages 7161 ns. The memory access times to migrated pages behave accordingly.

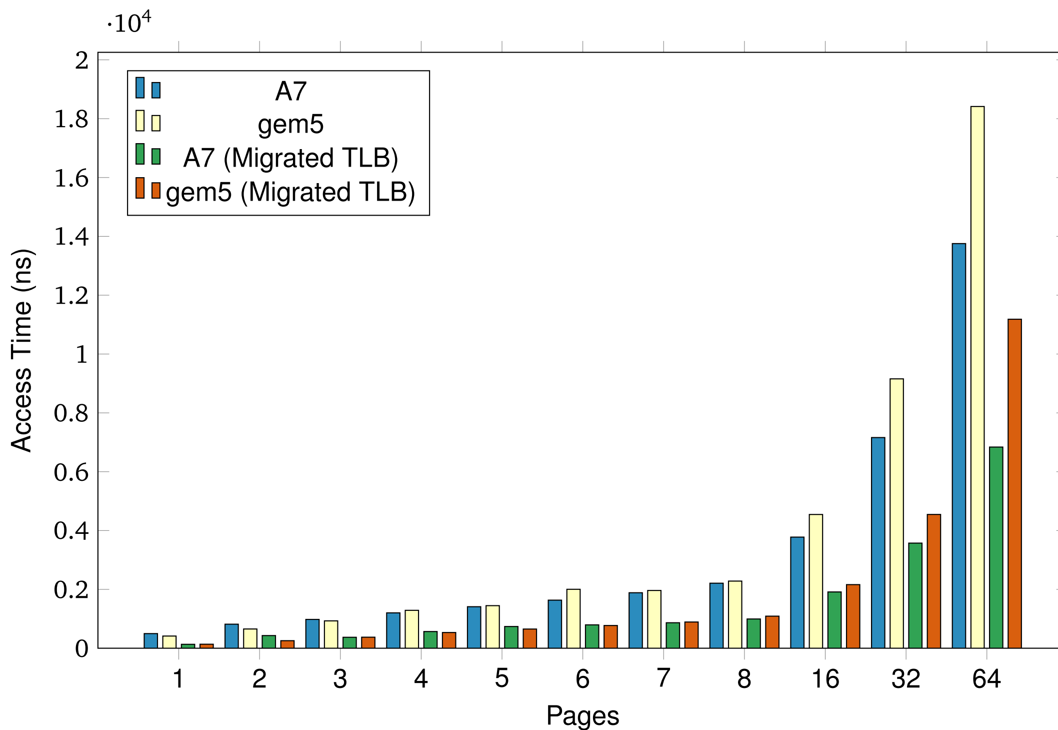


Figure 6.1 – The memory access times for a 68 KiB stride and different numbers of pages. The measurements are taken once with cold caches and once with a migrated TLB on the Cortex-A7 and in gem5.

When it comes to interfacing with the TLB, restoring is ignored (for this comparison) as the `pld` instruction is not implemented in gem5 the way it works on the Cortex-A7. Figure 6.2 shows the time spent saving the TLB. Just like before, a large fraction of the total time is required to read all entries. At first glance, it seems as if the gem5 TLB interface is twice as fast as the Cortex-A7's. However, the A7 has four times the number of entries as gem5, so in fact it is the other way around. Additionally, the access times increase faster with the number of pages in gem5, presumably because of data prefetching as well.

All in all, the simulator resembles the real hardware acceptably when it comes to memory access times. Only the slow performance of the TLB interface is problematic but becomes less relevant with version 3 and 4 of the interface and irrelevant when comparing the interfaces. However, it should not be forgotten that it is still just a simulator, so the results should be interpreted with caution.

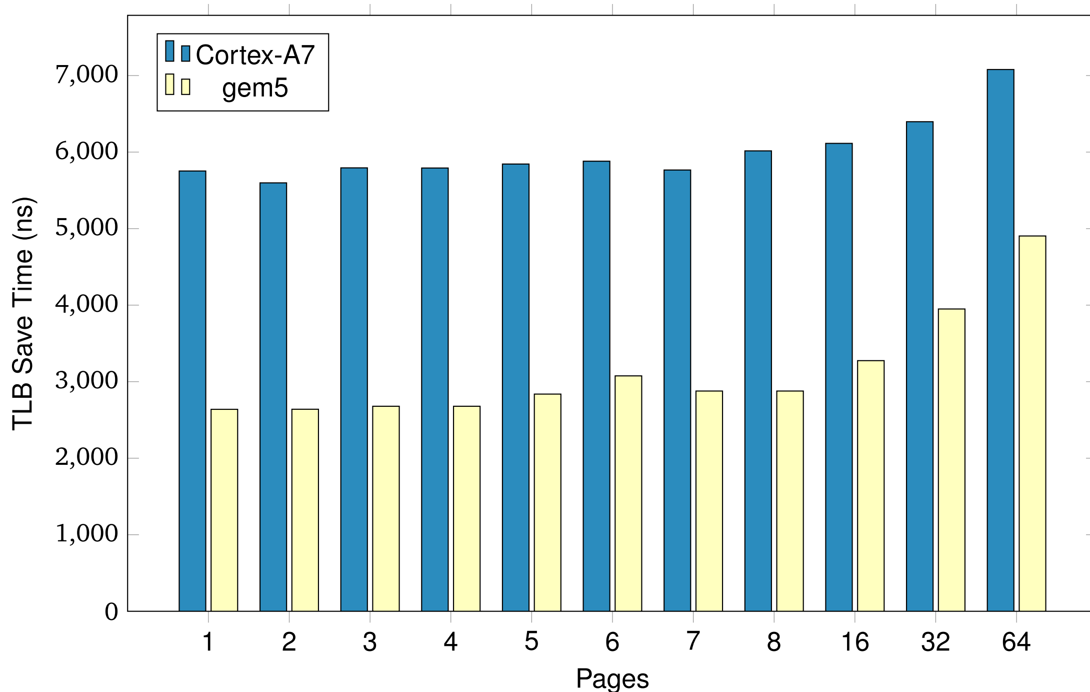


Figure 6.2 – The time required to read all TLB entries and store the relevant ones in memory for different numbers of pages. Measured using the 68k-benchmark.

6.2 TLB Interfaces

In order to compare the performance of the different TLB interface designs, the 68k-benchmark is run another time for each of them. Since the memory access times do not differ significantly between the runs, which is expected when everything works as intended, the only interesting times are the ones of the TLB save and restore steps of the benchmark.

Reading the TLB and storing its relevant entries takes at least about 2600 ns using the version 1 and 2 interfaces for a single page, as seen in Figure 6.3. Remember, all entries of the TLB have to be read. For 64 pages it takes 4904 ns and 3712 ns respectively. The difference comes mostly from memory accesses: For each entry, version 2 needs 4 B less space, resulting in 256 B less data to be stored in memory. Version 3 and 4 are, with about 800 ns for a single page, significantly faster as the filtering happens "in hardware" and therefore only the relevant entries have to be transfered in addition to the reduced number of register operations, which is caused by the streaming approach. The byte packing of version 4 saves 1 B/entry, which is why the difference becomes clearer after 32 pages, when a whole word is saved. At 64 pages the differences between the versions are smaller because the hardware filtering has only a small effect since almost no entry should be filtered out.

Restoring the TLB, as illustrated in Figure 6.4, looks very similar, apart from the fact that only the relevant entries have to be written and not all 64. Therefore, the differences to the streaming approach are less significant because filtering in hardware does not apply when writing. Overall, it applies that the less data has to be transfered, the faster the transfer. An interesting exception seems to be version 4 for small number of pages. For some reason this interface seems to be slower than version 3 and for seven pages even slower than version 2. The only difference which could lead to

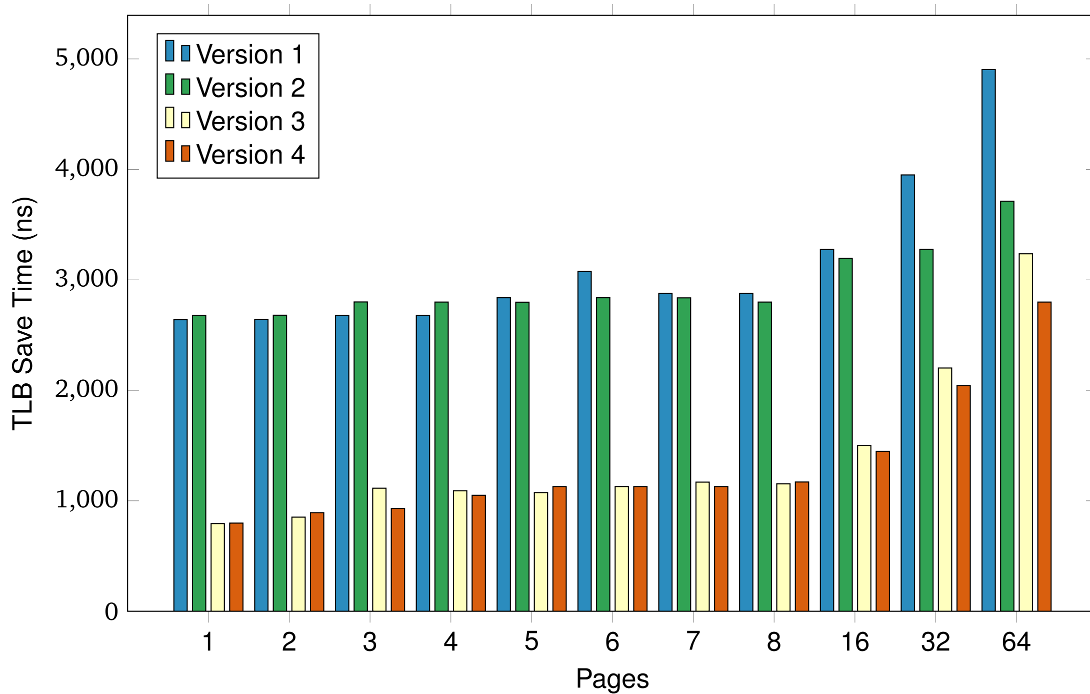


Figure 6.3 – The time required to read all TLB entries and store the relevant ones in memory for different numbers of pages. Measured using the 68k-benchmark for all TLB interface versions.

this is the fact that the stream has to be terminated by two register writes in interface version 4. For more pages, the traffic reduction could outweigh the operations. However, the differences seem to be too high for just two register writes. But other than that, the different TLB interfaces work as expected.

6.3 Migration Process

After verifying that the version 4 TLB interface is indeed the fastest (at least for a high number of pages), it is time to measure whether it can produce a speedup when performing TLB migration. As explained before, the 68k-benchmark is used. With the TLB interface and memory access times discussed in the chapters before, the only remaining question is the overall speedup shown in Figure 6.5.

For a single page TLB migration is not reasonable as it slows down the process by 1357 ns. This slowdown shrinks with an increased number of pages with some fluctuations caused by the fact that the data is calculated by measurements of four different processes. At 16 pages the slowdown is down to 273 ns. From 21 pages onwards TLB migration is an overall improvement of the performance by 86 ns and ends up with 1953 ns at 64 pages.

All in all, the improvements of the TLB interface are working, making TLB migration feasible. However, the results should be taken with a grain of salt because they were acquired in a simulator using a very artificial benchmark. Moreover, the speedup occurs for a number of pages where data prefetching effects are observable on the Cortex-A7.

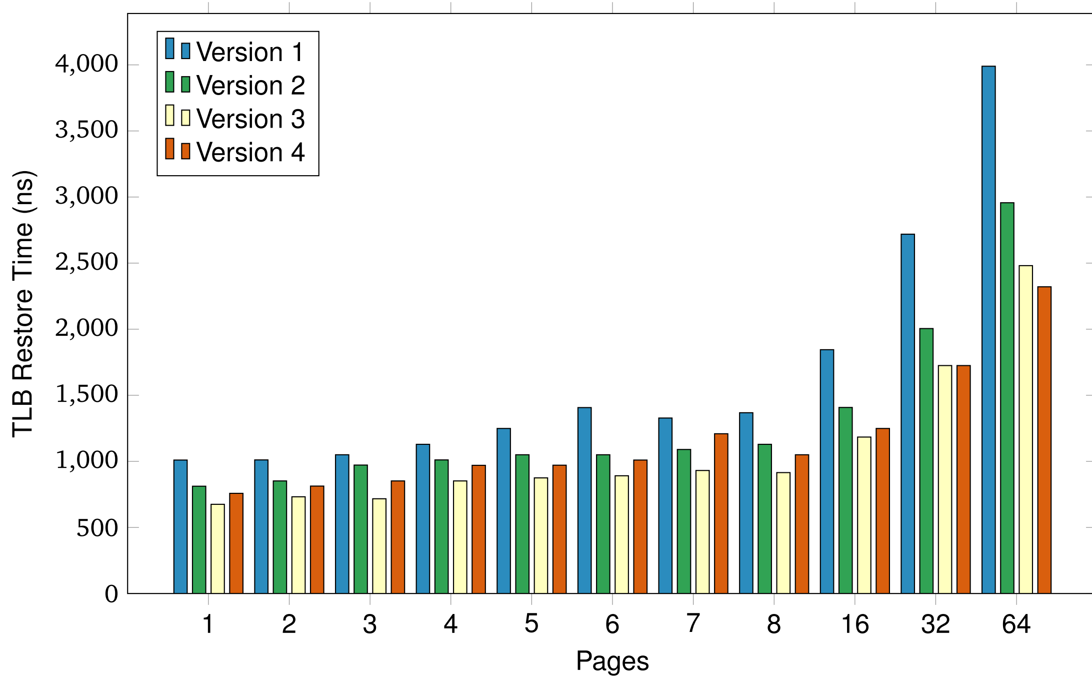


Figure 6.4 – The time required to read all relevant entries from memory and write them to the TLB for different numbers of pages. Measured using the 68k-benchmark for all TLB interface versions.

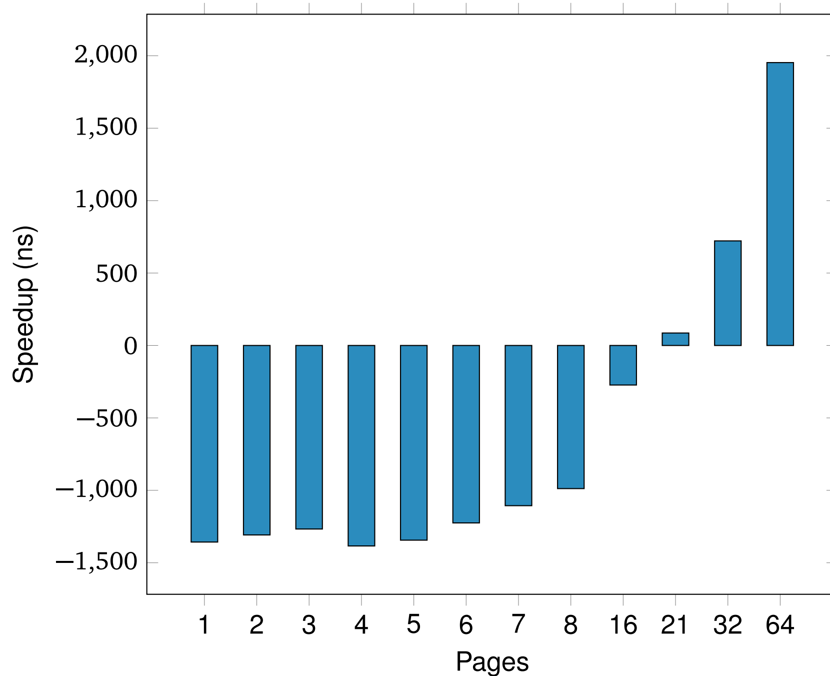


Figure 6.5 – The overall speedup of TLB migration in the 68k-benchmark for different numbers of pages, using interface version 4.

7 ARCHITECTURE: LINUX INTEGRATION

After exploring on how to save and restore the TLB's contents, the question is when to do it and where to store the data in context of an integration into Linux. All this work is done on each platform's own fork of Linux 4.14 [Fou][gem].

The first step is to understand how Linux manages processes and threads because the distinction is quite relevant for the TLB migration as all threads of a process share the same address space but do not necessarily require the same pages and thus TLB entries. When scheduling, Linux does not differentiate between processes and threads but rather works with tasks that represent both. They are managed by a structure called `task_struct`. This abstraction allows the scheduler to be much simpler because it can handle both the same way. Therefore, the implementation of TLB migration has to work with the same abstraction [TB15, p. 740].

There are basically two moments in which it is reasonable to store the TLB in memory: Every time a task gets displaced or whenever a task is migrated to another CPU core. The former has the advantage of a "warmer" TLB, meaning that the fewest entries have already been replaced. However, it also means that this process is performed several times a second, generating a lot of overhead. Moreover, it is expected that a thread migration only occurs once in a while, meaning that the process is performed unnecessarily most of the time. Therefore, storing the TLB during thread migration seems to be the better choice as it does not affect the context switch itself and it is guaranteed that the work is only done when needed, even if it means that some relevant TLB entries could already have been replaced. In practice, this leads to modifying the `move_queued_task()` function, which is called whenever a task is migrated. As the name states, the function works only with queued (not running) tasks. When a task is running, a stopper task has to be started to force the other task off the CPU. Afterwards, the same function is called [Tor17, ll. 899–909].

For storing the data, the `task_struct` seems suitable. It exists for each task and will probably already reside in cache, in particular in the TLB. Moreover, it eliminates the need to allocate additional heap space. This results in lower access times. However, it also means that there is only limited space for TLB entries, making it reasonable to limit their count. For this thesis, the limit is set to the size of the TLB, effectively removing the limit, but it could be part of future work to determine an optimal value.

Restoring the TLB after migration can be done at similar moments as when it is saved: directly after the migration or as part of the first context switch in `finish_task_switch()`. Since in this case both solutions result in the same amount of work to be done, the decision comes down to which option would restore the TLB at the latest possible moment. As the first context switch is temporally much closer to the execution of a task, in fact as close as it can be, this moment is chosen.

Packing all parts together results in the following changes to be made:

1. Add fields to the `task_struct` to make room for the TLB entries and a flag to indicate that the TLB can be restored.
2. Modify `move_queued_task()` to extract the relevant TLB entries, store them in the `task_struct` and set the flag so the TLB gets restored the next time the task is running.
3. Check whether the flag is set in `finish_task_switch()` and restore the TLB if needed, deleting the flag afterwards.

8 ANALYSIS: LINUX INTEGRATION

In this chapter the Linux integration is validated by observing the effects of TLB migration on memory access times when it is performed as part of the kernel. Afterwards, a migration process is benchmarked. The discussed results are median values measured in a gem5 simulation because a speedup is not achievable on the Cortex-A7.

8.1 Migration Validation

To ensure that the migration is working as intended, a benchmark is developed which measures the access times for a single page while migrating from one CPU core to another. The exact steps are as follows:

1. Flushing all data caches, including the TLB, to start from a clean slate.
2. Accessing one page.
3. Flushing the data caches (excluding the TLB) to simulate other tasks trashing the shared caches.
4. Migrating to the next CPU core by allowing the kernel to run the benchmark on the target core only.
5. Starting over with step 2.

This way, the benchmark is forced to run in a round-robin fashion on all cores. It ends once it ran on every core twice because without TLB migration the TLB is filled on the first run of each core, thus, on the second run the access times can benefit from the TLB enabled speedup. It would be possible to migrate more often, however, once the TLBs are filled, there are no significant differences between a thread migration with or without TLB migration regarding memory access times.

Figure 8.1 shows that without TLB migration the workload generates two memory accesses taking 335 ns every time it runs on a core for the first time. This is expected as most of the time the top-level descriptor is already cached, which means there is one access for the second-level page table and one for the page itself. As soon as it runs on a core a second time, the number of memory accesses drops to one, taking only 176 ns. TLB migration should ensure that the second core already benefits from TLB induced speedups. The data confirms this with the first run taking 336 ns and every following run taking 177 ns. It should be noted that results look almost too good to be true, which is caused by the fact that they were gathered using a simulator.

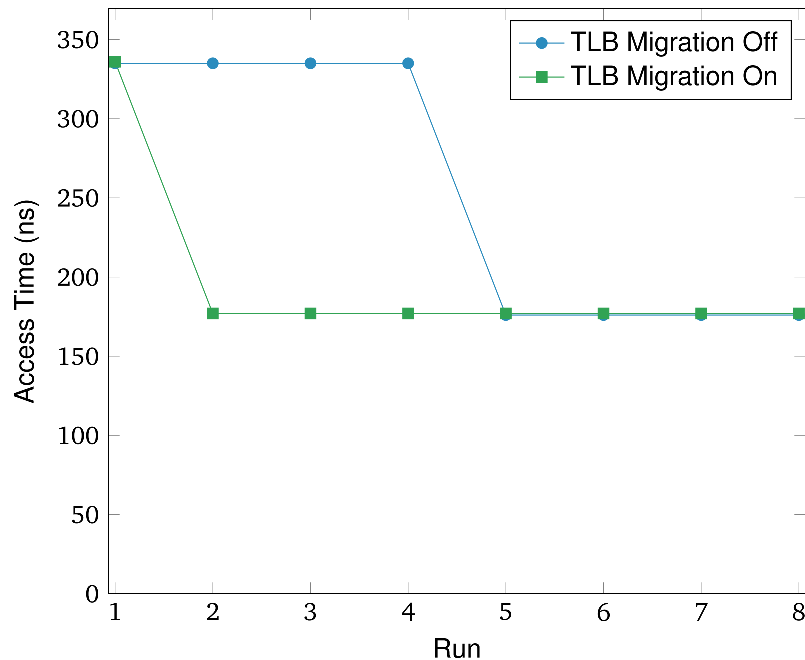


Figure 8.1 – Memory access times for a single page with flushed data caches. A migration to the next CPU core is performed after each run.

8.2 End-to-end-Migration

After validating the migration process and measuring the time needed for its different components in previous benchmarks, it is time to see how it behaves when integrated into Linux. Therefore, a benchmark is developed which works similarly to the 68k-benchmark. The workload consists of accessing N pages with a stride of 68 KiB to reduce the effects caching can have because, as before, an overall speedup is not realistic otherwise. Other than that, it is very similar to the previous benchmark and consists of the following steps:

1. Flushing all data caches, including the TLB, to start from a clean slate.
2. Accessing N pages with a stride of 68 KiB.
3. Flushing the data caches (excluding the TLB) to simulate an arbitrary workload that generates memory accesses.
4. Migrating to the next CPU core by allowing the kernel to run the benchmark on the target core only.
5. Starting over with step 2.

As explained in Section 8.1, performing a TLB migration after a task ran on the target core already minimizes the probability of a speedup. Since this benchmark's purpose is to show that an overall speedup is possible under some circumstances, it is chosen to only run on every core once, so four runs are performed in total. The other differences to the previous benchmark are that instead

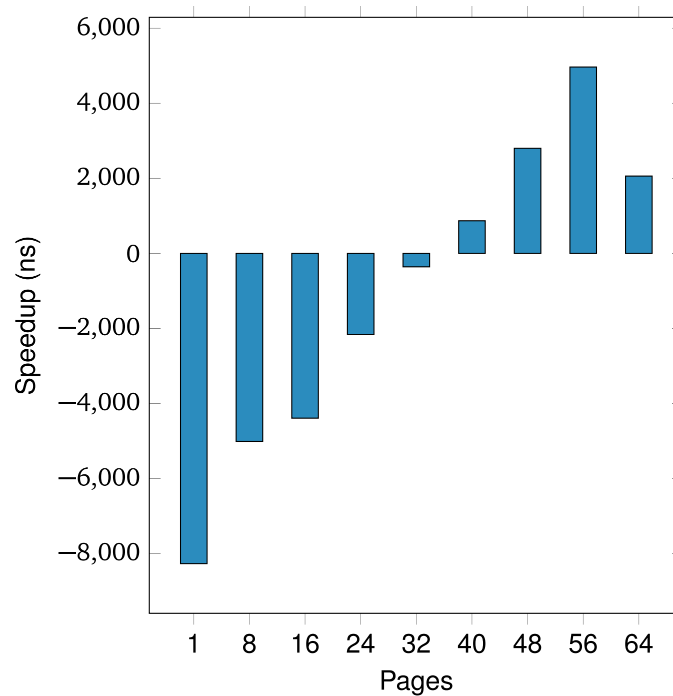


Figure 8.2 – Speedup through TLB migration in Linux when running on four different CPU cores back to back for different numbers of pages. Memory accesses are performed with a 68 Kibit stride.

of the access time, the whole process is measured with only 1000 iterations where performed as the process takes a lot of time and more iterations are not really necessary in a simulator as seen before.

The results in Figure 8.2 confirm the expectations. For a small number of pages migrating the TLB is actually slower. For a larger number, i.e., 40 pages and more, the migration leads to an overall speedup. This is a higher number of pages than in Section 6.3, which is no surprise because those measurements where taken in a more controlled and simpler environment. The maximum is at 56 pages with over 4900 ns being saved. However, this is only about 1 % of the total time of 353 830 ns. A noticeable occurrence is the decrease of the gains at 64 pages, in contrast to the general trend. This is most likely caused by the fact that a higher number of TLB entries get replaced by kernel specific entries because the TLB fits only 64 of them. This leads to the extra memory access being slower.

To conclude, if the number and stride of memory accesses are suitable, TLB migration can, in fact, lead to an overall speedup on a simulated, modified processor.

9 CONCLUSION

In this thesis, the concept of TLB migration was evaluated from a performance point of view. The performance was quantified by measuring memory access times for predetermined sets of pages. The findings show that using side effects of the `pld` instruction, an instruction for prefetching memory, is not worthwhile on the ARMv7 Cortex-A7 processor, not least because the instruction performs the table walk in the foreground. In addition, reading the contents of the TLB takes a significant portion of the migration's duration, especially when migrating a small number of entries because the whole TLB has to be read to determine the relevant entries. A second approach bringing relevant paging data structures into cache using said instruction was also evaluated. It was up to 61.6 % faster than the first approach but overall still losing 6561 cycles compared to no migration at all when accessing 64 pages with a stride of 68 KiB. This stride was necessary to prevent cached page descriptors from reducing the possible gains.

To accommodate this, a new TLB interface was developed and evaluated using the gem5 hardware simulator. Unlike the Cortex-A7's, it supports direct write-access and hardware-side filtering of the TLB, obviating the need to read all of its entries. Furthermore, streaming and byte-packing the data to minimize the register operations required are supported. These modifications allowed improvements of up to 1953 ns through TLB migration in the simulator when accessing 64 pages with a stride of 68 KiB.

Having found a case where TLB migration showed performance improvements, Linux, particularly the scheduler and the thread migration function, was modified to make use of this technique. Measurements show that when accessing 56 pages with a stride of 68 KiB on all cores in a round-robin fashion, the whole process is 1.4 % or 4967 ns faster with TLB migration. Speedups where observable from 40 pages onward.

Overall, it seems that TLB migration improves the performance only for very specific use cases. However, it remains uncertain how it affects processes in a real world environment instead of just a small set of benchmarks.

Future work on this approach can be separated into two categories. On one hand, there are many opportunities to expand on the work done in this thesis. First of all, a predictor could reduce the number of TLB entries to be migrated by predicting which ones will be used in the future. Additionally, a fix for the expiring TLB entries stored in memory could be implemented by modifying the kernel's memory management. To judge the current approach appropriately, a look at real world task sets would be necessary. The results can help to find the optimal amount of memory to reserve in the `task_struct` for TLB entries. If it seems reasonable with the new insights, bringing the hardware modifications from the simulator to the real hardware while in the same process specifying hardware costs of the modifications and fixing the *Secure state* workaround, could be a

next step. Besides, modifying `fork()` or providing user space access to TLB migration could be used to improve OpenMP [Ope].

On the other hand, this thesis approached TLB migration from a performance point of view. However, the technique could be used for other purposes as well, e.g., to reduce jitter in real-time applications.

All in all, there are many opportunities for further research to be pursued in the future.

A

APPENDIX

A.1 Cortex-A7

TLB Entries	Cycles
1	13
2	23
4	43
8	83
16	163
32	323
64	650

Table A.1 – The number of cycles required to extract different numbers of entries from the TLB without storing them in memory. Data of Figure 4.1.

Pages	Cycles	Cache Accesses	Cache Misses
1	329	1	3
2	359	2	4
3	503	3	5
4	577	4	6
5	687	5	7
6	805	6	8
7	845	7	9
8	931	8	10
9	1,025	9	11
10	1,133	10	12
11	1,179	11	13
12	1,291	12	14
13	1,471	13	16
14	1,607	14	17
15	1,689	15	18
16	1,835	16	19
17	1,885	17	20
18	1,910	17	20
32	2,195	17	21
64	3,363	17	23

Table A.2 – The number of cycles required to preload different numbers of consecutive pages using the `pld` instruction, and their corresponding numbers of cache accesses and hits. Data of Figures 4.2 and 4.4.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	429	120	1,109	5,073
2	631	315	1,100	4,879
3	585	416	1,358	5,221
4	690	517	1,170	5,104
5	972	589	1,757	5,240
6	981	673	1,610	5,236
7	985	802	2,010	5,314
8	1,039	894	1,738	5,338
16	1,903	1,737	2,678	5,238
17	1,537	1,336	1,908	5,173
18	2,065	2,260	2,786	5,542
32	3,469	3,144	4,244	5,580
64	6,615	5,952	7,790	6,128

Table A.3 – The results of the 4k-benchmark for different numbers of pages. The number of cycles required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 4.6 to 4.9.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	447	120	1,019	5,178
2	735	387	1,328	5,038
3	881	335	1,328	5,215
4	1,082	510	1,574	5,213
5	1,270	664	2,042	5,260
6	1,471	715	2,436	5,293
7	1,697	779	2,536	5,189
8	1,989	895	2,804	5,414
16	3,399	1,722	4,228	5,503
17	3,739	1,860	4,372	5,492
18	3,923	2,044	4,258	5,178
32	6,445	3,216	7,388	5,757
64	12,379	6,154	16,924	6,371

Table A.4 – The results of the 68k-benchmark for different numbers of pages. The number of cycles required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 4.6 to 4.11, 6.1 and 6.2.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	410	168	1,217	5,001
2	596	314	1,275	5,212
3	754	531	1,219	5,191
4	803	539	982	5,251
5	1,000	777	1,287	5,248
6	959	712	1,339	5,185
7	1,079	895	1,455	5,399
8	1,135	1,014	1,205	5,408
16	1,841	1,783	1,946	5,313
17	2,177	1,830	1,736	5,527
18	2,215	1,842	1,810	5,367
32	3,681	3,657	1,999	5,671
64	6,669	5,993	2,960	6,001

Table A.5 – The results of the 4k-benchmark for different numbers of pages and when using the indirect TLB migration approach. The number of cycles required to access those pages with clean and migrated TLB, to save the TLB in memory and to restore it from memory.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	464	154	1,098	5,206
2	716	344	1,087	5,219
3	903	532	955	5,344
4	1,124	549	1,421	5,251
5	1,348	716	1,299	5,419
6	1,445	738	1,499	5,396
7	1,646	877	1,439	5,329
8	1,439	779	1,237	5,217
16	3,711	2,062	2,213	5,617
17	3,695	1,881	2,334	5,679
18	4,353	2,513	2,710	5,692
32	6,377	3,563	3,572	5,604
64	12,497	6,174	6,563	6,321

Table A.6 – The results of the 68k-benchmark for different numbers of pages and when using the indirect TLB migration approach. The number of cycles required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 4.10 and 4.11.

	Direct TLB Migration		Indirect TLB Migration	
Pages	4k-Benchmark	68k-Benchmark	4k-Benchmark	68k-Benchmark
1	-5,873	-5,870	-5,976	-5,994
2	-5,663	-6,018	-6,205	-5,934
3	-6,410	-5,997	-6,187	-5,928
4	-6,101	-6,215	-5,969	-6,097
5	-6,614	-6,696	-6,312	-6,086
6	-6,538	-6,973	-6,277	-6,188
7	-7,141	-6,807	-6,670	-5,999
8	-6,931	-7,124	-6,492	-5,794
16	-7,750	-8,054	-7,201	-6,181
17	-6,880	-7,985	-6,916	-6,199
18	-8,523	-7,557	-6,804	-6,562
32	-9,499	-9,916	-7,646	-6,362
64	-13,255	-17,070	-8,285	-6,561

Table A.7 – The overall speedup in cycles for different numbers of pages, calculated from data gathered in the benchmarks for the direct and indirect TLB migration approaches.

A.2 gem5

Bits	Name	Description
[7:0]	asid	Address Space Identifier
[8]	valid	Valid bit, when set to 1 the entry contains valid data
[15:9]	N	Page shift factor
[35:16]	v_addr	Virtual address
[55:36]	p_addr	Physical address
[63:56]	ap	Access permissions
[75:64]	attributes	Memory attributes formatted for PAR
[78:76]	innerAttrs	Inner attributes
[79]	outerAttrs	Outer Attributes
[82:80]	domain	Domain type
[84:83]	mtype	Memory type
[85]	nonCacheable	Not cacheable
[86]	shareable	Shareable
[87]	xn	Execute Never bit
[90:88]	lookupLevel	Lookup level
[91]	global	Global

Table A.8 – The gem5 TLB Data Fields Version 1. Used in Table 5.1.

Bits	Name	Description
[7:0]	asid	Address Space Identifier
[8]	valid	Valid bit, when set to 1 the entry contains valid data
[9]	N	Page shift factor
[29:10]	v_addr	Virtual address
[49:30]	p_addr	Physical address
[51:50]	ap	Access permissions
[53:52]	attributes	Memory attributes formatted for PAR
[55:54]	innerAttrs	Inner attributes
[56]	outerAttrs	Outer Attributes
[58:57]	domain	Domain type
[59]	mtype	Memory type
[60]	nonCacheable	Not cacheable
[61]	xn	Execute Never bit
[62]	lookupLevel	Lookup level
[63]	global	Global

Table A.9 – The gem5 TLB Data Fields Version 2. Used in Table 5.1.

Bits	Name	Description
[0]	N	Page shift factor
[20:1]	v_addr	Virtual address
[40:21]	p_addr	Physical address
[42:41]	ap	Access permissions
[44:43]	attributes	Memory attributes formatted for PAR
[46:45]	innerAttrs	Inner attributes
[47]	outerAttrs	Outer Attributes
[49:48]	domain	Domain type
[50]	mtype	Memory type
[51]	nonCacheable	Not cacheable
[52]	xn	Execute Never bit
[53]	lookupLevel	Lookup level
[54]	global	Global

Table A.10 – The gem5 TLB Data Fields Version 3. Used in Table 5.1.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	414	136	1,010	2,639
2	654	256	1,011	2,640
3	930	375	1,050	2,679
4	1,288	534	1,129	2,679
5	1,447	652	1,249	2,838
6	2,003	772	1,407	3,076
7	1,963	891	1,328	2,878
8	2,282	1,090	1,368	2,878
16	4,546	2,162	1,845	3,275
32	9,156	4,547	2,719	3,950
64	18,414	11,183	3,990	4,904

Table A.11 – The results of the 68k-benchmark for different numbers of pages, using TLB interface version 1. The time in nanoseconds required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 6.1 to 6.4.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	414	136	812	2,679
2	653	296	852	2,680
3	971	454	972	2,800
4	1,210	535	1,011	2,799
5	1,526	692	1,050	2,798
6	1,765	811	1,050	2,838
7	2,043	970	1,090	2,837
8	2,242	1,050	1,129	2,799
16	4,467	2,242	1,408	3,195
32	9,196	4,627	2,005	3,276
64	18,216	11,103	2,957	3,712

Table A.12 – The results of the 68k-benchmark for different numbers of pages, using TLB interface version 2. The time in nanoseconds required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 6.3 and 6.4.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	318	119	675	794
2	692	295	732	852
3	916	360	717	1,114
4	1,250	574	852	1,090
5	1,471	637	875	1,074
6	1,805	851	891	1,129
7	2,004	931	931	1,169
8	2,306	1,153	915	1,153
16	4,482	2,138	1,184	1,502
32	9,077	4,547	1,725	2,202
64	18,057	10,865	2,481	3,236

Table A.13 – The results of the 68k-benchmark for different numbers of pages, using TLB interface version 3. The time in nanoseconds required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 6.3 and 6.4.

Pages	Cold Access	Migrated Access	Restoring TLB	Saving TLB
1	321	122	758	798
2	693	296	813	892
3	931	415	852	931
4	1,209	573	970	1,050
5	1,448	692	971	1,129
6	1,725	811	1,010	1,129
7	2,202	970	1,209	1,129
8	2,283	1,051	1,050	1,170
16	4,666	2,242	1,249	1,448
21	6,096	2,997	1,447	1,566
32	9,275	4,785	1,725	2,043
64	18,176	11,103	2,321	2,799

Table A.14 – The results of the 68k-benchmark for different numbers of pages, using TLB interface version 4. The time in nanoseconds required to access those pages with a clean and a migrated TLB, to save the TLB in memory and to restore it from memory. Data of Figures 6.3 to 6.5.

Pages	Speedup
1	-1,357
2	-1,308
3	-1,267
4	-1,384
5	-1,344
6	-1,225
7	-1,106
8	-988
16	-273
21	86
32	722
64	1,953

Table A.15 – The overall speedup in nanoseconds for different numbers of pages, calculated from data gathered in the 68k-benchmarks for the version 4 TLB interface. Data of Figure 6.5.

A.3 Linux Integration

Run	No TLB Migration	TLB Migration
1	335	336
2	335	177
3	335	177
4	335	177
5	176	177
6	176	177
7	176	177
8	176	177

Table A.16 – Memory access times in nanoseconds for a single page with flushed data caches. A migration to the next CPU core is performed after each run. Data of Figure 8.1.

Pages	No Migration	Migration	Delta
1	289,139	297,405	-8,266
8	299,352	304,359	-5,007
16	307,658	312,047	-4,389
24	318,505	320,669	-2,164
32	329,511	329,869	-358
40	338,525	337,656	869
48	347,909	345,108	2,801
56	358,797	353,830	4,967
64	368,726	366,664	2,062

Table A.17 – Time spent to access different numbers of pages in nanoseconds when running on four different CPU cores back to back. Memory accesses are performed with a 68 Kibit stride. Data of Figure 8.2.

LIST OF ACRONYMS

ASID Address Space Identifier

ICC Inter-Core Cooperative

MMU Memory Management Unit

SLL Shared Last-Level

TLB Translation Lookaside Buffer

LIST OF FIGURES

2.1 Paging Using Two-level Page Tables 4
2.2 Cortex-A7 MPCore Processor Top-level Diagram 5

4.1 TLB Read Times 11
4.2 `pld` Times 12
4.3 Page Descriptor Memory Layout 13
4.4 `pld` Cache Accesses and Misses 14
4.5 `pld` Dual-slot Behavior 14
4.6 TLB Save Times 16
4.7 TLB Restore Times 17
4.8 Memory Access Times 18
4.9 Memory Access Time Reductions 19
4.10 TLB Restore Times (Direct vs. Indirect) 20
4.11 Memory Access Time Reductions (Direct vs. Indirect) 20

6.1 Memory Access Times (Cortex-A7 vs. gem5) 26
6.2 TLB Save Times (Cortex-A7 vs. gem5) 27
6.3 TLB Save Times (gem5 Comparison) 28
6.4 TLB Restore Times (gem5 Comparison) 29
6.5 Overall Speedup (gem5) 29

8.1 TLB Migration Validation in Linux (gem5) 34
8.2 Linux Speedup (gem5) 35

LIST OF TABLES

3.1 Main TLB Descriptor Data Fields 8
3.2 TLB Data Read Operation Register Encoding 9

5.1 gem5 TLB Data Field Comparison 22

A.1 TLB Read Times 41
A.2 `pld` Benchmark 42
A.3 4k-Benchmark 42
A.4 68k-Benchmark 43
A.5 4k-Benchmark (Indirect) 43
A.6 68k-Benchmark (Indirect) 44
A.7 Overall Speedup 44
A.8 gem5 TLB Data Fields Version 1 45
A.9 gem5 TLB Data Fields Version 2 45
A.10 gem5 TLB Data Fields Version 3 46
A.11 68k-Benchmark (gem5 TLB Version 1) 46
A.12 68k-Benchmark (gem5 TLB Version 2) 47
A.13 68k-Benchmark (gem5 TLB Version 3) 47
A.14 68k-Benchmark (gem5 TLB Version 4) 48
A.15 Overall Speedup (gem5) 48
A.16 TLB Migration Validation in Linux (gem5) 49
A.17 Linux Speedup (gem5) 49

LIST OF LISTINGS

3.1 TLB Read Operations 9

6.1 gem5 System Configuration 25

REFERENCES

[Bin+11] Nathan Binkert et al. "The Gem5 Simulator." In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[Dea15] Will Deacon. *Linux 4.14 ARM MM Context Source Code*. Dec. 3, 2015. URL: https://github.com/torvalds/linux/blob/v4.14/arch/arm/mm/context.c (visited on 09/18/2018).

[dom15] dom. *The Raspberry Pi 2 Q&A thread*. Feb. 17, 2015. URL: https://www.raspberrypi.org/forums/viewtopic.php?p=697474#p697474 (visited on 10/22/2018).

[Fou] Raspberry Pi Foundation. *Raspberry Pi Linux*. URL: https://github.com/raspberrypi/linux.

[gem] gem5. *gem5 Linux*. URL: https://gem5.googlesource.com/arm/linux.

[GM16] Thomas Gleixner and Ingo Molnar. *Linux 4.14 ktime Header File*. Dec. 25, 2016. URL: https://github.com/torvalds/linux/blob/v4.14/include/linux/ktime.h (visited on 10/11/2018).

[jam15] jamesh. *Where can I find the Start.ELF source code?* Jan. 22, 2015. URL: https://www.raspberrypi.org/forums/viewtopic.php?t=97358#p676069 (visited on 10/22/2018).

[Kom13] Albert-Ludwigs-Universität Freiburg Lehrstuhl für Kommunikationssysteme. *RaspberryPI und sein Bootloader*. Oct. 23, 2013. URL: https://lab.ks.uni-freiburg.de/projects/linux_from_scratch/wiki/RaspberryPI_und_sein_Bootloader (visited on 10/22/2018).

[LBM13] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs." In: *ACM Trans. Archit. Code Optim.* 10.1 (Apr. 2013), 2:1–2:38. ISSN: 1544-3566. DOI: 10.1145/2445572.2445574. URL: http://doi.acm.org/10.1145/2445572.2445574.

[Ope] OpenMP. *OpenMP. The OpenMP API specification for parallel programming*. URL: https://www.openmp.org/ (visited on 10/15/2018).

[RB13] Sajjid Reza and Gregory T Byrd. "Reducing Migration-induced Misses in an Oversubscribed Multiprocessor System." In: *Parallel Processing Letters* 23.01 (2013), p. 1350006.

[SK10] Shekhar Srikantaiah and Mahmut Kandemir. "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors." In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 313–324. ISBN: 978-0-7695-4299-7. DOI: 10.1109/MICRO.2010.26. URL: https://doi.org/10.1109/MICRO.2010.26.

[TB15] A.S. Tanenbaum and H. Bos. *Modern Operating Systems*. Pearson, 2015. ISBN: 9780133591620.

[Tor17] Linus Torvalds. *Linux 4.14 Core kernel scheduler code and related syscalls*. Sept. 29, 2017. URL: https://github.com/torvalds/linux/blob/v4.14/kernel/sched/core.c (visited on 10/22/2018).

[ARM13] ARM. *Cortex™-A7 MPCore™. Technical Reference Manual*. Apr. 11, 2013. URL: https://developer.arm.com/docs/ddi0464/f (visited on 08/31/2018).

[ARM14] ARM. *ARM® Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. May 20, 2014. URL: https://silver.arm.com/download/download.tm?pv=1603196 (visited on 09/03/2018).

[Ras15] Raspberry Pi Foundation. *Raspberry Pi 2 Model B*. 2015. URL: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/ (visited on 09/13/2018).

[Ras18] Raspberry Pi Foundation. *Overclocking options in config.txt*. Sept. 18, 2018. URL: https://www.raspberrypi.org/documentation/configuration/config-txt/overclocking.md (visited on 09/18/2018).

[gem18] gem5. *The gem5 Simulator*. Apr. 29, 2018. URL: http://gem5.org/Main_Page (visited on 09/13/2018).