Großer Beleg

Support for on-chip memory in Fiasco

Daniel Molka

August 16, 2007

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Marcus Völp

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den August 16, 2007

Daniel Molka

Acknowledgement

I want to thank all members of the operating systems group. Especially my supervisor Marcus Völp for all his support and patience, Alexander Warg for his assistance with Fiasco(arm) and Neal Walfield for his hints to improve my english writing and document structure.

Contents

List of Figures

1 Introduction

In real-time systems, the worst case execution time (WCET) is very important. It is needed for reserving CPU-time. But most hardware is optimised to improve average case performance. Caches are such an optimisation. They improve average performance, but increase worst case execution times and make the estimation of the WCET more complex. In embedded real-time systems, scratch pad memory is used to improve the predictability of the WCET. In this work, it is investigated how scratch pad memory can be used in a microkernel based operating system.

The rest of this chapter explains the technical background needed to understand this document. One paragraph explains caches, why they are needed, how they work, and what their advantages and disadvantages are. Feel free to skip it if you are familiar with caches. Another paragraph introduces scratch pad memory (SPM) and explains the differences between caches and SPM. Subsequently, the consequences of those two paragraphs are summarised. Finally, the goal of this work is formulated and the outline for the rest of this document is given.

1.1 Cache

A big performance problem in modern computers is the increasing gap between processor clock rate and memory clock rate. This has not been a problem until the early 1990th. 386, the early 486, and the first Pentium™ processors operated at the same clock rate as the memory. As the infrastructure did not improve as fast as the CPUs, later processors started to operate at an internal clock rate higher than the surrounding system. Currently processors with 3 or more GHz have to work with memory that internally operates at 266 MHz (DDR2-1066). The latency of the fastest currently available modules is 25 cycles (4-4-4-12-1T) referring to the "effective" 1066 MHz. This translates to a delay of more than 70 CPU cycles for a main memory access (about 120 cycles with common 5-5-5-15-1T DDR2-800). If the memory controller is implemented in the chipset, the latency increases further.

Caches are fast on-chip memories (SRAM) used to hide this latency. They store code and data recently used by the CPU. Most processors use relatively small separate caches for either code or data, which are located close to the instruction fetch or load-store unit respectively. This, so called, Level-1 cache typically can be accessed with less than 3 cycles delay. It is usually combined with a bigger Level-2 cache used for both, code and data. In contrast to the L1-cache, which is integrated in the core of the CPU, L2 caches are usually implemented as a separate block on the chip. Because of this, their latency is higher (typically 10 to 15 cycles). Caches are managed by hardware and cannot be seen by software, apart from the speedup. They are implemented as content addressable memory. This means that the content is tagged with the memory address it belongs to. Figure 1.1 shows how caches are implemented in principle.

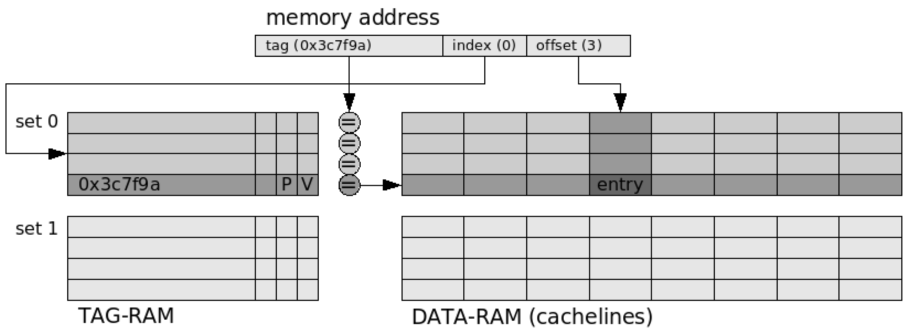

Figure 1.1: basic cache architecture (4-way set-associative)

The *TAG-RAM* stores the address tags and additional information about the corresponding cacheline (if it contains a valid entry, if its data has been modified, coherency information, etc.). The TAG-RAM usually is divided into several sets. The cached data is stored in the *DATA-RAM*. To find an entry in the cache, the requested memory address is typically split into three parts, *tag*, *index* and *offset*. At first the index is used to select a set within the TAG-RAM. Then the tag is compared to the entries in the selected set (a special case is the, so called, fully-associative cache, which has only one set and needs no indexing). If a valid entry is found that is equal to the searched tag, the offset is used to locate the requested data within the selected cacheline. When a memory location is accessed that is not found in the cache, a cache miss is said to occur. The hardware then chooses an entry in the cache that will be removed. The chosen *cacheline* is written back to memory if needed and the requested data is copied into the freed cacheline. The replacement decisions have to be quick. Many architectures implement algorithms that try to remove the least recently used entry (LRU). Others use simple approaches like round robin or random replacement that do not need additional counters. In all cases it can happen that code or data that will be accessed soon is removed from the cache. For round robin and random replacement this is obvious. LRU usually works quite well [11], but if an application starts reusing code and data, which were not accessed for a while, LRU might throw it away just before it is needed again.

In general, caches improve the average performance, as frequently used code and data usually can be accessed at low latency. On the other hand, the worst case execution time (WCET) can increase. As a complete cacheline is written back to main memory and another one is loaded, the worst case penalty for a cache miss is usually much higher than the memory latency would be without caches. This results in an increase in the span between average case execution time and worst case execution time.

1.2 Scratch Pad Memory

Scratch pad memory (SPM) is a different type of on-chip memory (SRAM). It is visible as part of the physical address space and can be accessed directly. The hardware implementation is much easier than for caches. The SPM only consists of DATA-RAM. The TAG-RAM, the replacement logic, and the comparators for checking the tags are not required. Hardware only needs to know the size of the SPM and the start address in the physical address space. This information is either hardwired or stored in registers. When a physical address is accessed, the hardware checks if the address is within the SPM-region. If that is the case, the start address is subtracted from the requested address (usually the size of the SPM is a power of 2 and the start address is size-aligned, so that subtraction is a cheap bitwise AND with size-1). The offset is used for addressing within the SPM. If the requested address is not in the SPM, main memory is accessed as usual. Figure 1.2 shows the difference in the memory layout between caches and SPMs.

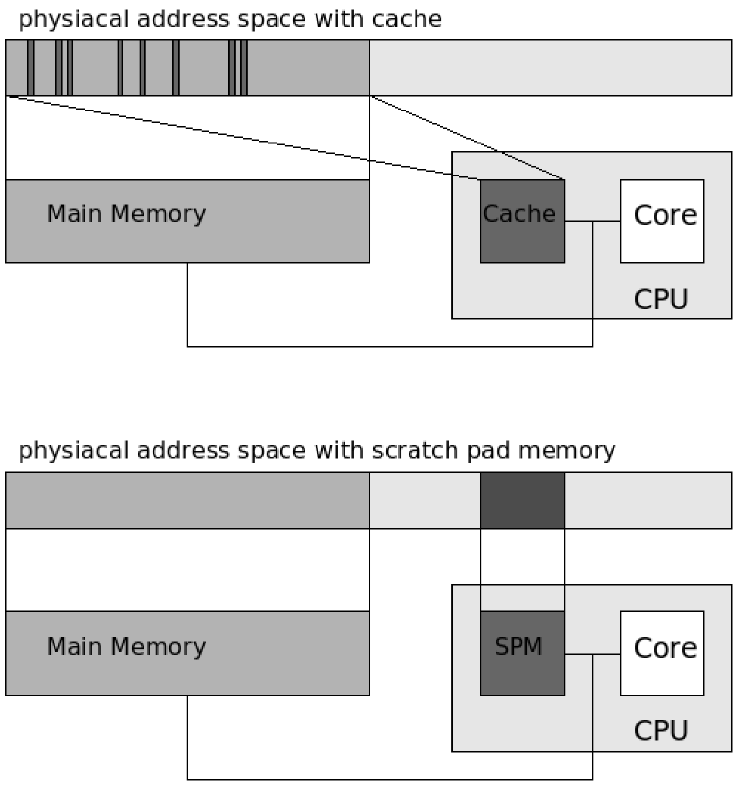
physiacal address space with cache
Main Memory
Cache
Core
CPU
physiacal address space with scratch pad memory
Main Memory
SPM
Core
CPU

Figure 1.2: memory layout: cache vs. scratch pad memory

When using SPM to store a physical address, accesses to this address will always have a short latency, which is in the order of accessing a cache of the same size. Whereas the latency of cached memory can vary at runtime. In contrast to caches, the content of scratch pad memory is not managed by hardware. There is no automatic buffering of frequently used code and data. Therefore, SPMs do not automatically improve performance. They have to be managed by software. Operating systems and applications are responsible for explicitly placing code and data in this fast on-chip memory.

SPMs are currently used in several embedded processores [2, 1], as the simplified chip design reduces energy consumption and production costs. In embedded systems the fast memory is usually managed by the compiler. The compiler based optimisation will be discussed in more detail in chapter 2. These approaches have in common, that the whole SPM is used exclusively by one application. To do the optimisation the compiler needs to know the available amount of scratch pad memory.

SPM can be used together with caches. This allows real-time applications, which need deterministic memory access times, to use the SPM, whereas the rest of the software benefits from the automatic speedup of caches.

1.3 Consequences

Hardware managed caches improve well and automatically the average performance of applications. But in the context of real-time systems the increased WCET results in lower system utilisation. In particular, in multitasking environments it is often hard to predict if a certain memory location is in the cache. The cache, which does not distinguish memory used by different applications, is shared between all applications. Usually it cannot be avoided that applications share cache lines and destroy each other's cache working set. This contention on the cache can have significant influence on the runtime behaviour. As a result the prediction of WCET can become badly complex when using caches. Often WCET has to be determined under the assumption that all memory accesses cause cache misses. Those pessimistic WCETs lead to bad utilisation, as more CPU time than actually required has to be reserved.

In contrast to that, the predictable timing of scratch pad memory can be used to increase performance without decreasing predictability at the cost of having to explicitly manage the on-chip memory.

1.4 Mapping Database (Mapdb)

Fiasco uses an recursive address space model [9]. Each application can map any of its memory pages to another application, if the receiver agrees. The receiver has access to this page afterwards. The mapdb is a datastructure in the kernel that stores information, to which address spaces a physical page is mapped directly or indirectly. This is necessary to allow later revocation of access permission.

1.5 Goal

The primary goal of this work is to design and implement a mechanism to manage scratch pad memory in the Fiasco microkernel. It should be possible to share the SPM between applications. Applications should be able to place their code and data (e.g., by using the existing compiler based approaches) as well as kernel code and data into the scratch pad memory at runtime. The existing approaches typically do not allow sharing the SPM and use only compile time decisions.

The policy decision, when and which memory is moved to SPM, should be implemented in user-level servers or in the applications. Only the basic mechanism will be implemented in the kernel. With this mechanism the user level managers must be able to:

- control which application can access which part of the SPM
- consistently move application code and data between main memory and SPM
- move kernel code and data between main memory and SPM
  - prepare kernel such that relevant parts can be moved
  - Mechanism to control which application can move which kernel code and data

Strategies are required to decide if an application can allocate a part of the SPM, and if an application can move certain kernel code and data to the SPM. This work investigates the required mechanism, strategies left for future work. The solution we propose is to use virtual memory to replace main memory with SPM transparently.

1.6 Outline

The remainder of the document is structured in 5 chapters

- Related work - Brief discussion of some other cache, SPM, and WCET related work
- Design - Overview over the system, reasoning for design decisions
- Implementation - Describes the implemented mechanisms in detail.
- Evaluation - Measurements proving that the mechanisms work, overhead and benefit estimation
- Conclusion - summarises results of the work

2 Related Work

This chapter compares the solution proposed here with existing approaches to use caches and scratch pad memory in the context of real-time systems.

2.1 Caches and WCET Estimation

It is not easy to enforce that WCET-critical code and data resides in the cache when needed. One approach to improve this is cache colouring [8]. This mechanism assigns all physical memory, which could end up in the same cacheline, to a certain application. If only one application uses a certain colour then other applications cannot evict. But this procedure is complex, wastes memory as all available memory is partitioned, requires detailed knowledge about the cache architecture, and does not work if contiguous memory is required. Also it is not trivial to write applications in a way that they do not destroy the own cache content. In contrast to that, individual pages of scratch pad memory can be assigned to applications(e.g., using the existing mapping mechanism), without having to partition main memory.

Other approaches for improving cache predictability can be found in literature [7, 15]. Currently those techniques are not widely used as their performance penalty is too high, the implementation costs are not tolerable, or they cannot be used because of missing hardware support.

2.2 Compiler based SPM Usage

As already mentioned in the previous chapter, the usage of the SPM in embedded systems is typically managed by the compiler. There exist two principles, static and dynamic placement, for placing code and data in scratch pad memory. In the static approach [12], it is decided at compile-time what to store in on-chip memory and what remains in main memory. Single objects, even single variables can be placed in the fast memory region. This method is easy to implement (apart from the placement strategy) but does not allow adaption to changing requirements during the execution of an application. In particular, it does not allow to adapt to changing amounts of available SPM. The solution presented here enables applications to change the placement at runtime.

There are also dynamic compiler based methods [14, 5, 6]. These methods allow changing the placement of objects at runtime through the insertion of instructions that move code or data between on-chip and off-chip memory. This results in a better utilisation of the fast memory, but complex analysis is required to find the optimal placement.

The copying introduces runtime overhead. One Method is based on timestamps. Whenever the compiler decides to move something into or out of the SPM, a new timestamp is used. The linker uses the timestamps to figure out where a referenced object is placed at a certain point in the program. This technique is not dynamic in the sense that the application can decide at runtime, which objects are stored in the SPM (e.g., based on the value of a variable). All decisions are made at compile time, it is then hardcoded in the binary when objects are replaced (e.g., after entering function f() move its local data to the SPM). Our mechanism allows to use runtime information to decide what should be placed in the SPM. Another feature introduced with the presented solution is that applications can modify other application's address spaces.

As the size of the scratch pad memory is limited it has to be figured out what should be put into the fast memory to achieve the best performance. The advantage of the existing methods is that the compilers do that automatically by analysing the source code.

The SPM is currently typically used in single application scenarios, but can be partitioned in multitasking environments [10]. Each application can then use a certain part of the SPM exclusively. Only static partitions are supported in [10], whereas the assignment of SPM memory to applications can change at runtime with the mechanism described here.

2.3 SPM and WCET Estimation

The effect of scratch pad memory on worst case execution time has already been investigated. The results presented in [18] show that the predictable timing of memory accesses allow to estimate WCET better. In [13] is shown that it is not only possible to predict it better, the WCET can also be reduced significantly by placing the code of the critical path in fast on-chip memory.

2.4 Virtual Memory

As virtual memory causes unpredictability too (TLB misses, Pagefaults), it is typically not used in embedded real-time systems. Hardware typically used in desktops and laptops currently does not feature SPMs. So it has to our knowledge not yet been researched how to use scratch pad memory in a system that uses virtual memory.

3 Design

This chapter explains how the mechanism is designed and why it is done that way. First basic requirements and design goals are listed. Several possible solutions are discussed and a trade-off is made that leads to the selected model. The last paragraph describes the implementation design.

3.1 Basic Requirements

As the mechanism is meant to be used in a multiapplicative environment, the isolation between address spaces is essential. In contrast to embedded systems, we cannot rely on applications that do not harm each other. Fiasco uses virtual memory to provide isolated address spaces. The mechanism for managing the SPM have to work together with virtual memory. This implies that direct accesses to physical memory are impossible. To enable applications to access the SPM, it has to be (partially) mapped into the applications virtual address space.

3.2 Design Goals

We express the following goals to achieve with our design.

1. Allow to share the SPM between applications without destroying the isolation of the address spaces
2. Support sharing of SPM regions between applications
3. Give applications the possibility to move kernel code and data to the SPM
4. Maintain a consistent view to shared objects for all application (before,) **while**, and after moving these objects
5. Preserve the real-time capabilities of the kernel (keep blocking times low)
6. Allow delegation of SPM-management responsibility
7. SPM management independent from other memory management strategies
8. Allow to move code and data objects of arbitrary size to the SPM
9. Keep the kernel implementation small
10. Introduce as little runtime overhead as possible

The list is ordered according to their priority. The first 4 are regarded to be mandatory.

3.3 Basic Design Decisions

3.3.1 How to Use Virtual Memory

As already mentioned, we have to use virtual memory. There are basically three options for using the existing mapping mechanism [9].

- Linear mapping of the whole SPM into all address spaces
- Allow all applications to use the whole SPM using time sharing
- Mapping single pages to arbitrary locations in certain address spaces

Mapping the whole SPM to all applications is of course the simplest solution. If the SPM would be used that way, the existing compiler based approaches could be used to manage the SPM. But this would not allow to control access to individual parts of the SPM. This violates the first design goal. Additionally, it would cause dependencies such that application that use the same part of the SPM cannot run in parallel.

Time sharing would require to save and restore the SPM-state, when switching between applications that use it. It is likely that SPM sizes increase in the future, so this could become to costly.

The 3rd solution allows to give certain applications exclusive access to a certain amount of scratch pad memory, using the access permission bits in the page table. This solution provides sharing SPM without saving and restoring the SPM-state. With this method the only remaining dependency is that there has to be enough on-chip memory for all applications. The allocation can be done with the granularity of the smallest hardware page size.

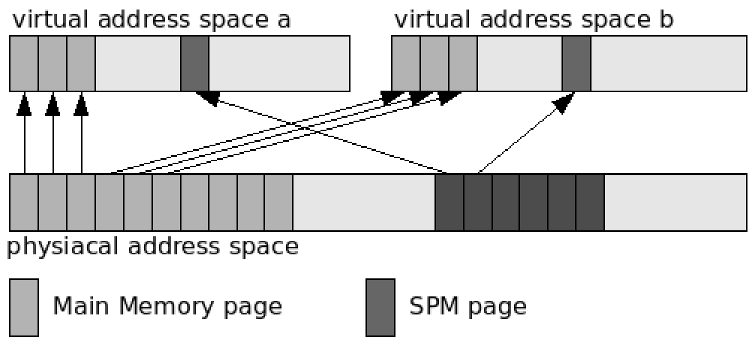

Figure 3.1: allocating scratch pad memory using virtual memory

The assignment of smaller blocks is not possible without destroying isolation between the address spaces, as access permission to the memory can only be controlled with page size granularity.

The mapping of single pages was chosen to be used as it is necessary to provide isolation within the SPM, which is the design goal with the highest priority. Time sharing provides isolation as well, but it cannot be foreseen how costly saving and restoring the SPM-state might become in the future. Another advantage is that mappings can be used to implement shared memory as well. On the other hand the relatively high granularity is problematic for the placement of small objects.

3.3.2 kernel- vs. user-level mechanism

The recursive address space model[9, 16] could be used to implement the data transfers SPM at user level and without additional kernel support. Pagers could manage their application's address spaces by mapping pages and provide interfaces that applications could use to request modifications to this mappings. But this would not allow to move pages in the kernel address space, as it is not controlled by a pager. As this is a major design goal, it was necessary to implement the mechanisms in the kernel.

3.4 Use Cases

1 Applications restructure their address space by exchanging pages from main memory with pages from the SPM, using the exchange mechanism from the kernel. This is the most flexible kind of using the SPM. It allows to move arbitrary code and data into the SPM without changes in the virtual address space (no update of jump targets or PC-relative addresses required).

2 Applications move frequently used kernel code and data to the SPM, using the exchange mechanism. This could be used to speedup frequently used code path such as ipc. As the kernel address space is shared, all applications are affected by those replacements. As a consequence strategies are required to decide which application can replace which parts in the kernel address space.

3 Applications use pages from the SPM as free memory and dynamically allocate new objects in it or move existing objects to the SPM. This allows finegrained placement of application objects and is more flexible than existing dynamic approaches, as the decisions can be made at runtime depending on the amount of available scratch pad memory. Compared to the first alternative, this use case either needs the objects, which should be moved, to be addressed indirectly or the code has to be self modifying to access objects at their new location after the movement.

4 The address space (excluding the kernel address space) of an application is managed by its pagers. Pagers can easily replace pages by unmapping them, copying them to a different location, and establishing new mappings. This can be done completely in user-level without additional kernel support and could be used to optimise existing applications without modifying them. Another use case of this is to take the existing compiler based techniques and use pagers to construct virtual address spaces that look like the physical address space expected by the applications.

3.5 Design of the Mechanism

The use cases 3 and 4 do not need additional kernel support. This paragraph describes how the exchanges, used by the use cases 1 and 2, work. Once the mechanism is available, it might be convenient to use it for use case 4 as well. This mechanism only allows to exchange complete pages, so the code should be restructured into a replacement-friendly form (pagesize aligned sections containing code or data that makes sense to be moved as a whole) to use this efficiently. To improve average performance the most frequently used kernelcode and -date should be put into a separate section, which can be placed in SPM. To improve the WCET, all critical paths should be put into a section for easy replacement.

3.5.1 User Address Space

Figure 3.2 shows the starting point for the exchange procedure. An application has its code and data in pages backed with main memory mapped into its virtual address space. In addition the application has a SPM page mapped, which is not used yet.

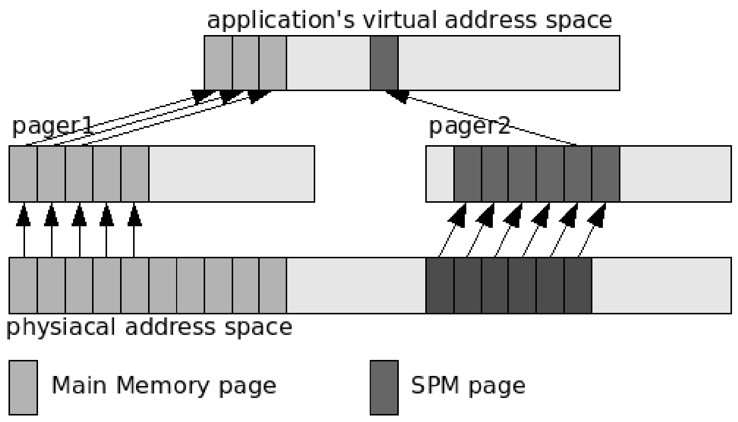

Figure 3.2: memory layout before exchanging pages

There are several possibilities for exchanging the pages.

1. only update caller's address space
2. update all address spaces (global exchange)
3. update address space of caller and children in the pager-hierarchy (local exchange)

The first solution is the simplest, it would just require to copy the data, update the page table of the calling task and modify the corresponding entries in the mapdb[16]. This method would only affect the calling task, it would not work if the memory that should be moved to the SPM is mapped further. In other words it violates the 2nd and 4th design goal. It has to be ensured that all applications, which use this physical page, are affected by the operation and use the new location after the page is moved.

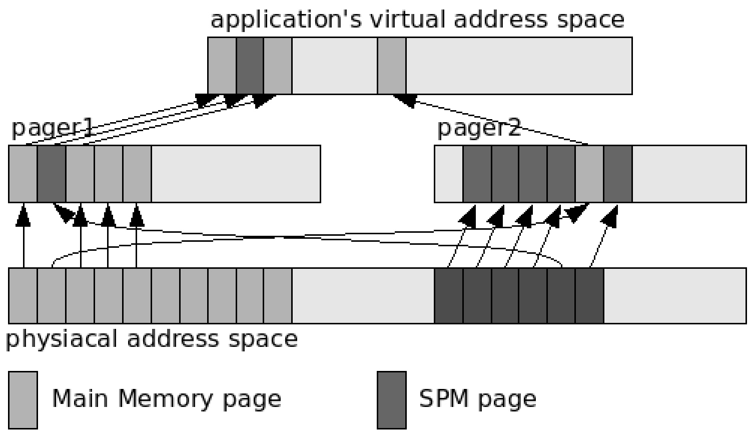

Figure 3.3: exchange pages globally

Figure 3.3 shows the global solution. The changes affect all applications, in particular, all pagers. This method is the safest solution, as there cannot occur any problems with shared pages. Everything is replaced at the bottom level of the recursive address spaces [9, 16]. Pagefaults will allways find the right pager, as the application's view on the memory regions does not change. So the consistency requirement is fulfilled. On the other hand, it is the most expensive solution as all pagers, which are involved into the mapping of a physical page, have to be modified. This has the disadvantage that superpage mappings in the lowlevel pagers have to be split. To avoid the overhead of splitting the mappings at runtime, one would has to refrain from using superpage mappings. In summary, this solution introduces a lot of overhead. Another problem would be that it is not possible to write a pager that manages the whole SPM area. As can be seen in the picture *pager2* has no relation to the SPM-page after it has been exchanged by the application. As a consequence pager2 has no possibility to get the original page back. So this solution is not the best choice as it is not useful for implementing strategies.

To allow pagers to control the SPM area, including a possibility to revoke (unmap) access rights if an application violates the protocol, a linear mapping is required that is not affected by exchanging pages. Figure 3.4 on the next page shows how linear mappings could be preserved.

But this approach would limit system design to a central manager for the SPM. This would work for now, but may not be flexible enough in the future. It could only be used as a simple lowlevel manager that provides memory for higher level managers and a way for them to get their pages back in the case that an application violates the highlevel managers protocol. The lowlevel part could be included into sigma0.

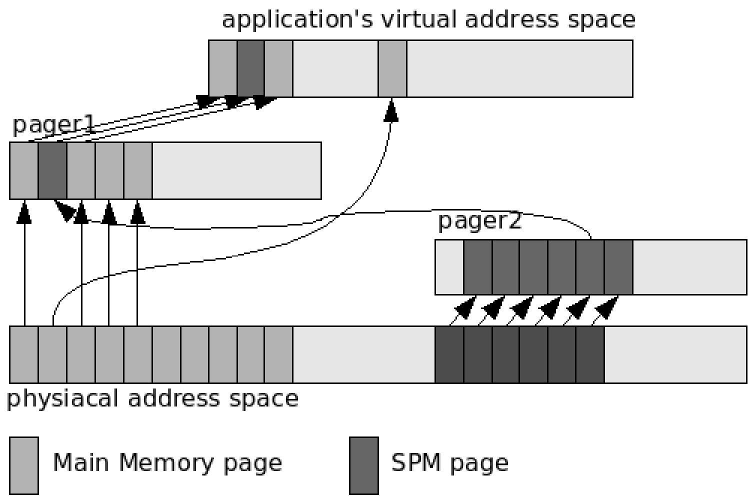

Figure 3.4: special case for SPM-pager

The approach shown in Figure 3.4 also has the problem that mappings in lowlevel pagers have to be modified. But those updates are not allways necessary. As the pagers usually only provide memory and do not care what the applications do with it, they do not need to be involved in the exchange process. If the page is not shared with other applications it is sufficient that the application that exchanges pages itself and all applications that receive mappings from this task are modified. Figure 3.5 shows this scenario.

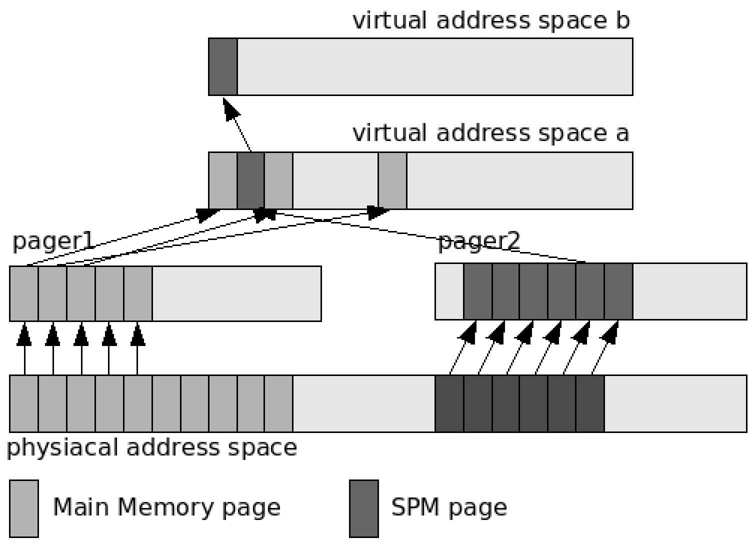

Figure 3.5: exchange pages locally

This method has the advantage that the lowlevel pages are not affected. This makes the implementation faster as less pagetables have to be traversed and there are fewer changes in the mapping database. This approach does not depend on the callers position in the mapping tree, as the pagers are not affected. Only the callers subtree in the mapdb has to be updated. Because of this, it might be possible to guarantee an upper bound for the execution time of the exchange. Also it is possible to work with superpage mappings in the lower levels of the pager hierarchy. On the other hand, there are a couple of disadvantages. Figure 3.6 shows an example where an application replaces a page that is used by another application too.

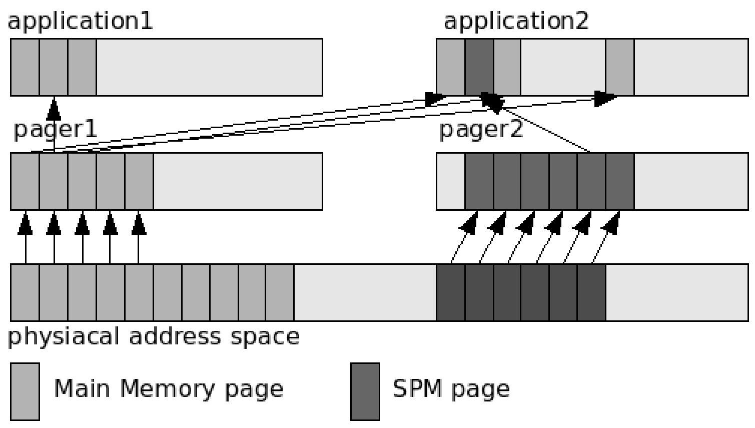

Figure 3.6: local exchanges and shared memory

The second application is not affected by the local change in the other task and continues to use the old copy. This violates the requirement of a consistent view of all applications sharing the page. This inconsistency can however be tolerated. If both applications need to be affected by the exchange the task that provides the shared memory has to perform the exchange what would affect both applications. It is not the responsibility of the kernel to prevent unexpected behaviour when applications use the exchange mechanism in a wrong way.

The second problem with local exchanges is that the region map of the task changes. Figure 3.6 shows that a virtual address that belongs to the area of *pager1* includes a page of *pager2* after the exchange operation. There are 2 solutions for that. Updating the region list is not one of them, because even if the right pager receives the pagefault message it does not know about a mapping to the new virtual address. Updating the pagers internal data structures isn't an option as the pagers internals are not known. The first solution is simple. Just ignore it and assume that there won't be pagefaults. If both pagers will never unmap something until the application frees it explicitly there is no problem. The page tables are updated during the exchange, so there won't be pagefaults afterwards. But if one of the pagers might do an unmap this simple solution is not enough. Another solution is to use one pager that combines both functionalities like shown in Figure 3.7.

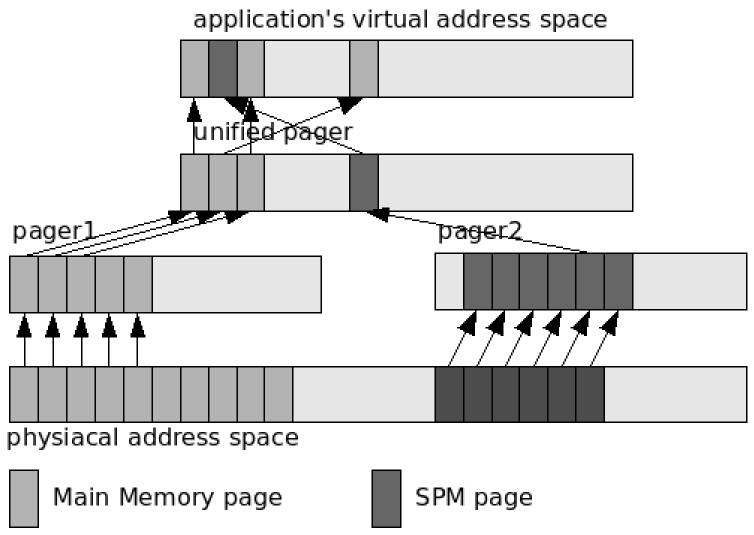

Figure 3.7: unified pager

This still needs communication between the pagers to avoid data loss caused by an unexpected unmap, but the implementation costs amortise among all applications that use this pager.

The 3rd approach: Local exchanges were chosen for implementation because it is the most flexible approach. The other 2 possibilities are special cases of this more general method. If sigma0 exchanges something it is a global change affecting all tasks. With this mechanism each pager can perform changes in its subtree of the recursive address spaces what is more flexible than having one central instance as shown in Figure 3.3. Problems remain with local exchanges. Programmers have to be careful in using this feature on shared memory locations, but this can be handled by allowing only the pager of the shared memory area to do exchanges. The problem with the region map requires new pagers/ dataspace managers to be implemented.

3.5.2 Kernel Address Space

In the current implementation, the kernel address space is shared between all applications, so only globally visible exchanges can be done in there. The problem with that is that applications can influence the runtime behaviour of other applications, when they replace kernel code and data. Frequently exchanging pages in kernel address space could possibly be used for denial-of-service attacks what has to be avoided. One way to avoid it completely is to disallow applications to modify the kernel address space at all, and leave the decision, if a certain page is moved to the SPM, to the kernel. But the kernel cannot know what code and data is currently needed by the applications.

One possibility to limit the application's influence on each other would be not to exchange pages in kernel space. Instead, applications could grant pages to the kernel. The kernel then decides, which page to use and prefers the fastest memory location in its decision. So if an application gives a fast page to the kernel it will be used. If another application later on tries to move it back to a slow memory location this will fail as the kernel would prefer the SPM page. However, this solution adds much memory overhead. Lists of available pages have to be maintained for each kernel page. This approach also wastes resources as many applications will give pages to the kernel to speed up the same frequently used code. So this method is not the best choice.

Another possibility is to allow application to only exchange certain pages in the kernel address space. Allowing them to only exchange kernel data that is related to them (their TCB, page directory, and page tables). This will avoid the influence on other applications by making their pages temporary unavailable during the exchange. For the kernel code and its global data structures it could be decided during startup or when allocating memory what is moved to the SPM and what remains in the main memory. Applications would have no influence on the kernel code and global data (e.g., mapdb, page tables for the kernel address space, kdir, kip, etc.) in this case.

Another solution would be to allow applications to only move code and data from main memory into the SPM and never in the other direction. This ensures that a kernel-page that is in the SPM stays in the SPM. But then it is impossible for an application to revoke the SPM pages, therefore that isn't a good idea.

This problem has not been solved yet. In the current implementation all applications can move any page in the kernel, affecting all other applications. The strategy which application can replace which parts of the kernel address space can later be implemented in user level servers. Capabilities could be used to delegate permissions that authorise to move parts of the kernel address space.

which would then be the only application that would be allowed to modify the kernel address space. Alternatively it could give capabilities to applications allowing them to exchange a certain page.

3.5.3 Summary

The exchange mechanism allows applications to manage the usage of SPM on their own. Programmers can optimise applications manually. In addition, pagers can modify the address space of their application. This could be used to place code and data of unmodified applications in the SPM. Instead of modifying the running code, the translation from virtual addresses to physical addresses is used to transparently relocate memory to the SPM.

On the other hand, it is only possible to exchange whole pages. The mechanism alone cannot be used to relocate smaller objects. However, it can be combined with existing compiler based methods to achieve that.

3.5.4 Other Use Cases

The replacement feature could be used for other purposes too.

- Currently the Mapped_allocator uses physical memory that is mapped linearly into the kernel address space. With the replacement feature sigma0 could exchange the physical pages used by the allocator. As memory used by applications could be replaced by other pages anyway, sigma0 could use the mechanism to free any physical page (given their is a free one to replace with). This could be used to free all pages from a memory module to allow exchanging the physical device.

- Embedded DRAM (eDRAM) is another type of fast memory that might be available in several architectures soon. It could be managed in the same way we manage scratch pad memory
- Another property of scratch pad memory is that, in contrast to caches, operations on it are not observable by other applications. Placing security critical code and data in scratch pad memory might be useful to improve security in some cases (e.g., AES encryption tables).

3.6 Implementation Design

This paragraph explains how the mechanism is implemented in general. Details are discussed in the *Implementation* chapter.

The exchange procedure exchanges the virtual addresses of 2 pages. It only copies code and data in one direction. Copying in both directions would require additional memory for temporarily storing the content of a page and would take longer. Overwriting existing code and data has to be avoided. Because of this it is only allowed to replace an existing page with an empty page.

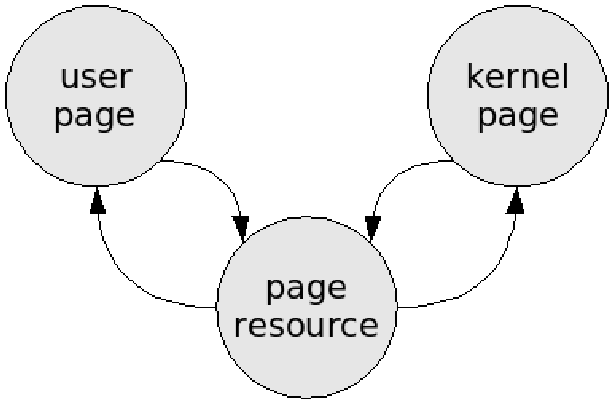

Figure 3.8: page state

To distinguish between empty and used pages, we record for each page if and how it is currently used. *Page resources* are unused pages and can be used to replace existing pages. As they might contain old code and data from other applications or from the kernel, they are mapped without rights. *User pages* are pages used by user applications. *Kernel pages* are pages used by the kernel.
Further information is needed per page:

- the address space to which the page belongs is needed to control if an application is allowed to do certain operations on the page (e.g., only the owner of a user page can destroy it, what converts it into a page resource)
- the location of the page (main memory, SPM, eDRAM, memory mapped hardware, etc.) might be necessary to implement useful strategies for automatic management
- information if it is allowed to replace the page is required to avoid movement of pages that have to be located at a certain physical address (e.g., DMA)

The interface provides the following functions:

- **create_user_page**(address) - Used by applications to convert a page resource into a user page that they can access directly (e.g., for finegrained placement of indirectly addressed objects). The pointer *address* locates the page resource in the applications virtual address space. The kernel implementation finds the corresponding physical page and updates its state. The type of the page is updated to *user page* and it is stored that the page now belongs to the address space of the system call invoking thread.
- **delete_user_page**(address) - Converts user pages that are no longer needed back into page resources. The kernel ensures that user pages can only be destroyed by threads in the address space to which they belong.
- **replace_page**(target,resource,size,count) - The function that exchanges pages. *target* and *resource* are virtual addresses. The physical pages mapped contiguous starting at *target* (old pages) are replaced with the physical pages mapped starting at *resource* (new pages). *size* is the pagesize *target* and *resource* are mapped with (needed to distinguish pages at a superpage boundary). *count* is the number of pages that are replaced.
  Exchanging pages involves:
  - copying the content
  - updating the mapdb
  - updating the page tables of all affected applications
  - updating the state of the affected physical pages

  After the call *target* points to the new pages containing the old content and *resource* points to the old pages. If pages in the user address space are replaced the old pages stay user pages for performance reasons. They can be converted back into page resources with the delete_user_page function. If *target* is in kernel address space, the old pages will be automatically converted back into page resources.

4 Implementation

This chapter describes the implementation in detail, how the required information is stored and how the mechanism is implemented. As ARM is the only architecture that supports scratch pad memory and that is supported by Fiasco, Fiasco(arm) [17] was chosen to be extended to support directly addressable on-chip memory.

4.1 ARMs Scratch Pad Memory Implementation

Several ARM processors provide scratch pad memory, which is called *Tightly coppled memory* (TCM) by ARM. Implementations differ in supported size and flexibility. Several processors have the TCM at a fix location in the physical address space (e.g., ARM966E-S[2]) while others allow to place it somewhere in the physical address space(e.g., ARM926EJ-S[1]). For convenience it is expected in this work that the TCM can be placed somewhere in the physical address space. Adaption for TCMs at a fixed address is possible but would require changes in the physical memory layout.

TCMs are implemented as a Harvard architecture, divided in an instruction TCM (ITCM) and data TCM (DTCM). The ITCM is executable, readable and writeable. The DTCM is readable and writeable but not executable. An instruction fetch on an DTCM address will read from the main memory (if available). The DTCM is located near the load-store unit together with the L1 data cache. The ITCM is located near the instruction fetch unit. It is accessable from the data side of the CPU because it has to be writeable to fill it with code. Also read- and write-access is required to allow using PC-relative addressed data. The ITCM is optimised for accesses of the instruction fetch unit. Accesses to the ITCM from the Load-Store unit may have longer latencies. Therefore it should be avoided to put real-time critical data in the ITCM.

4.2 Data Structure

The following data structure is used to store all relevant information about a page.

```
class Frame_description
{
  unsigned int properties;
  Frame_description * small_pages;
  Address mapped_addr1;
  Address mapped_addr2;
  Address mapped_addr3;
  Address pdir[4];
};
```

The integer properties stores multiple information. A bitfield could be used for that too, but this would make the syscall implementation more complex.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I	Space												Pagesize				U	Loc.		D	T	U	Type			U		M	X	W	R

Figure 4.1: properties of physical pages

I The invalid Bit, is not actually used for pages. It is reserved as an invalid value (0x80000000) is needed to indicate errors and handle special cases.

Space This information is needed for user pages to store the information, which address space is the owner of a page. A thread is only allowed to replace or destroy a user page if the address space it belongs to is the owner of this page (currently not checked for replacements). The number of tasks (address spaces) is currently limited to 2048 in the v2 interface and 256 in the x0 version (according to fiasco/src/abi/l4_types.cpp). Therefore, 11 Bit are currently sufficient for the address space ID. The 12 Bits used here allow up to 4096 address spaces.

Pagesize The size of the described page as index into an array of page sizes.

- **0** superpage
- **1** biggest subpage
- ...

Loc. The Location of the physical page

- **0** undefined
- **1** main memory
- **2** ITCM
- **3** DTCM

D Flag that indicates if the page is part of a page directory.

T Flag that indicates if the page contains page tables.

Type the type of the page

- **0** undefined
- **1** page resource
- **2** user page
- **3** kernel page

Rights the maximal rights that can be set for this page. This can be limited by hardware (e.g., pages in the DTCM can never be executable, pages of memory mapped hardware devices might not be replaceable) or by software (e.g., it could be necessary that certain areas are made non-replaceable by the kernel)

- **M** Flag that indicates if it is allowed to replace (move) this page.
- **XWR** execute-/write-/readable

U Currently unused Bits that are reserved for later extensions.

The pointer small_pages points to an array of Frame_descriptions for subpages if something is mapped with a smaller pagesize. The mapped_addr1-3 are used for kernel pages that can be mapped to multiple locations in the kernel address space. This information is necessary because the mapping database is currently not used for the kernel address space. If one of the virtual addresses is replaced with another page, all virtual addresses, where the page is mapped to, have to be updated. The original page will be freed. 2 Addresses would be sufficient. One for the Mapped_allocator address and another one for the address where vmem_alloc maps it. The 3rd address was introduced while porting the implementation to x86 to handle the superpage mapping of the cpu_page. As the cpu_page is now mapped as 4K page the third address is not needed any longer. If a page is mapped to more than 3 locations in the kernel address space (e.g., zero page) it will be marked as non-replaceable. The array pdir contains the addresses of the corresponding page directories if the page contains page tables. As 4K pages are used on ARM 4 page tables that have a size of 1K fit into one page. Thus, in the worst case 4 addresses have to be stored.

4.2.1 Initialisation

The datastructure is initialised at startup right after the initialisation of Kmem_alloc to ensure that the Mapped_allocator can be used for dynamic memory allocation of the datastructures. The first step is to walk through the memory regions in the Kip. Conventional memory is initialised as user page located in main memory with all rights available. Reserved memory is initialised as kernel page in main memory with all rights available. Bootloader memory is initialised as user page in main memory with all rights available. Other memory is initialised as undefined. The region from 48MB to 56MB is initialised as page resource and is currently used to emulate the TCM area, as Qemu [3] does not provide a TCM implementation and there was no TCM hardware available for testing. To support real hardware the corresponding implementation of CPU::init() would have to be extended to initialise the TCM and add a corresponding memory region to the Kip. Existing mappings in the kernel address space are found by analysing Mem_layout::phys_to_pmem and the Kernel page directory.

4.3 Interface

The Interface seen by the applications provides the following functions:

```
L4_INLINE l4_umword_t
l4_change_page_state(l4_umword_t page,
                     l4_umword_t size,
                     l4_umword_t new_state);
```

This function allows to directly change the properties of the specified page. *page* is the virtual address of the page that should be modified. *size* determines wheather a superpage or a small page should be updated. This value is used to decide what page to modify if a superpage-size aligned address is given. *new_state* is the new state of the page. If it is an allowed modification it will be used as replacement for the integer properties in the Frame_description belonging to the page.
Disallowed modifications for applications are:

- changing the state of kernel pages
- changing information if a page contains page tables or a page directory (user pages can never contain such data)
- changing the type from user_page to page_resource (=destroy_user_page) if the address space does not own this page
- update information about the physical location of a page (main memory or TCM, cannot change at runtime)
- change the maximal rights of a page (those values have nothing to do with the current rights an application has on a page. The current rights are determined by the page table entry for read, write and execute and in case of moving pages by the space information in the Frame_description)

if *new_state* is set to INVALID_FRAME (0x80000000) the function returns the current state of the specified page.

```
L4_INLINE l4_umword_t
l4_create_user_page(l4_umword_t page,
                    l4_umword_t size);
```

Special case of l4_change_page_state. It converts a page resource into a user page.

```
L4_INLINE l4_umword_t
l4_delete_user_page(l4_umword_t page,
                    l4_umword_t size);
```

Another special case. It converts a user page back into a resource. (will fail if the specified user page does not belong to the callers address space)

```
L4_INLINE l4_umword_t
l4_replace_page(l4_umword_t old_page,
                l4_umword_t new_page,
                l4_umword_t size,
                l4_umword_t count,
                l4_umword_t rights);
```

This function performs the exchange. It replaces a number of pages specified by *count* of the given *size* at the virtual address *old_page* with the same amount of pages at the virtual address *new_page*. The old pages can be user pages or kernel pages. The new pages have to be page resources. "rights" determines what rights are needed by the application on that page what is necessary to avoid that code is moved into the DTCM or data is moved to ITCM. The application has to define that because it cannot be figured out from the page table entry of the old page that has only 2 access permission Bits (User accessable and writeable). This inconvenience could be hidden by providing separate functions for code and data. Also more special use cases of l4_change_page_-state could be implemented if there are more common use cases (e.g., changing the space information).

4.4 Kernel Implementation

2 syscalls where added to implement the interface:

- sys_change_page_state implements l4_change_page_state() and its special cases
- sys_replace_page implements l4_replace_page()

4.4.1 sys_change_page_state

The implementation of changing the state of a page is quite simple. It just looks up the physical address in the invoker's page directory and changes the corresponding Frame_description if the modifications are allowed.

4.4.2 sys_replace_page

4.4.2.1 Preconditions

If one of the following conditions is not fulfilled the exchange is aborted before anything is modified.

- addresses have to be aligned to the given size
- old memory region and new memory region must not overlap
- pages at the new location have to be page resources
- the maximal rights of pages at the new location have to be greater or equal than the rights needed by the old pages
- pages at the old location have to be moveable
- page resources have to be in the user address space (the application has to supply them, page resources in the kernel address space cannot be used by applications, currently all pages in kernel address space are kernel pages)

- everything has to be mapped (currently pagefaults are handled, but that is only because there is currently no way for applications to request page resources and ensure they are mapped. This will be removed when pagers that provide page resources are available. At the moment roottask only maps the resources on a pagefault)
- currently it is not supported to replace pages that are split into smaller pages.
- if a page directory should be replaced it has to be moved completely (4 4K pages on ARM) and not in combination with other pages
- the area that should be replaced has to be physically contiguous (required for page directories on ARM)
- when moving user pages the invoker's address space has to own them (not implemented yet as sigma0/roottask do not care about changing space information when mapping memory)
- when moving kernel pages additional checks might be necessary to test if an application is allowed to move the pages. But that has not been done yet.

4.4.2.2 Consistency

It is required to preserve a consistent view for all applications on the exchanged pages. Currently cli-sti is used to make the exchange atomic. This ensures consistency but introduces blocking times that decrease the real-time capability. cli-sti was chosen because it is easy to implement. Alternative implementations that preserve the real-time capabilities are discussed later on. To keep the blocking time short the replacement of multiple pages is not done in one big block. Instead each page of the specified size is exchanged separately.

4.4.2.3 User Page Exchange

exchanging a user page involves the following steps:

- cli
- update Frame_descriptions
- copy content using memcpy
- update mapdb and page tables of tasks that received a mapping of this page from the invoker
- update page table of caller
- sti

Updating Frame_descriptions means to copy all variable information from the old page to the new one. The location of the page (main memory or TCM) the maximal rights and the size of the page are not changed as those are properties referring to the physical page itself, not to the content. The state of the old pages is not changed. If the caller owns these pages it can destroy them by calling l4_destroy_user_page. Currently the content is copied using the standard memcpy function. For user pages the mapdb is used to find out, which applications have access to the replaced page. The page tables of all children in the mapping tree are updated before copying the mapping tree to the new parent mapping. Finally, the page table of the caller is updated to the new physical addresses.

4.4.2.4 Kernel Page Exchange

exchanging a kernel page involves the following steps:

- cli
- update Frame_descriptions
- copy content using memcpy
- update page table (page tables for the kernel address space are shared, so this automatically affects all applications)
- update corresponding page directories if page tables are moved
- copy changes in kernel page directory to all page directories if necessary
- sti

The Frame_descriptions of the new pages are updated in the same way as for user pages. The old pages are converted into page resources immediately to avoid that kernel pages are mapped into the user address space. The content is copied like in the preceding case.

The mapping database is not affected by changes in the kernel address space. If a page resource is used to replace a kernel page the pager that provided the page resource still sees the mapping into the application's virtual address space, where the page resource was mapped. But the application has a different page in its page table at this location. The pager could unmap the page, but not destroy the contained kernel pages. A possible solution for this problem could be to not convert the old pages into page resources, but let their type remain kernel page. This would make them unusable for the application (user accessable Bit is zero) after the exchange. In the case of an unmap the pages could be automatically exchanged back before the unmap. But this would cause problems when using it together with the approach to disallow applications to replace fast memory with slow memory in the kernel address space. To avoid this it could be to distinguish between page resources that can be unmapped and page resources that are not allowed to be unmapped. If only page resources that cannot be unmapped are accepted to replace kernel pages the problem cannot occur.

In the common case only the corresponding entry in the affected page table has to be updated. As the page tables for the kernel address space are shared, all applications are affected. TLB entries have to be invalidated to make these changes effective.

4.4.2.5 Special Cases

When replacing kernel pages it can happen that page tables are moved. In this case the page directories the page tables belong to have to be updated. The needed address of the page directory and the index into the page directory is stored in the array pdir in the Frame_description when allocating a page table. When the kernel page directory was modified the changes have to be copied to all address spaces.

Another special case are page directories. When the currently used page directory is exchanged the page directory base register in the CPU has to be updated.

4.5 Consequences

Using the mechanism requires further modifications of the kernel:

- To allow fine grained replacement all superpage mappings in the kernel address space have been replaced by 4K mappings.
- The kernel used linear mappings to speed up translations from physical to virtual addresses and vice versa in some cases. The translations Mem_layout::phys_to_pmem() and Mem_layout::pmem_to_phys that only add or subtract an offset, are now only used during startup. Later on these simple translations had to be replaced by page table walks when translating virtual to physical addresses and lookups in the Frame_descriptions (mapped_addr1) when translating from physical to virtual addresses.
- To allow goal oriented replacements the kernel code should be restructured into sections including functionality, which belongs together. We started to put everything related to ipc into a separate section, and all mapping related functions into another section. Unfortunately, many "section type conflicts" limited the success. Looks like the build system combining the sources for the different architectures causes these problems. If the code would be distributed in different sections, applications could move functionality they need into the TCM. Another possibility would be to put all frequently used kernel code into one section and special cases into another section. Applications that do not cause the kernel to handle special cases could then rely on high performance as the part of the kernel they use is in the fast memory. However, if the special cases running in the slow main memory need locks or use cli-sti, the kernel might be blocked for longer periods.

4.6 Problems and Limitations

The current prototypical implementation proves that it is possible to use virtual memory to assign scratch pad memory to applications. Remaining problems are:

4.6.1 Blocking Times

The biggest problem in the implementation is the use of cli-sti. As the real-time benefits were a major design goal, it is not a good solution to have a long blocking time including memcpy, mapdb modification, and page table changes. The blocking times are discussed in detail in the *Evaluation* chapter. The measured delays could be tolerated for multimedia applications, which are the most common soft real-time applications. But the usability for more sophisticated use cases is limited. A way to reduce the blocking time is to avoid cli-sti periods.

4.6.1.1 Optimisation for Exchanging Read-Only Memory

When the replaced pages are read-only, atomic exchange is not necessary. There is no consistency problem on read-only memory, because it makes no difference, which copy is used during the exchange procedure. Therefore the mapdb and page tables need not be updated atomically together with the memcpy as long as the old page is not removed before all page tables and the mapdb are updated. This works for read-only user and kernel pages.

4.6.1.2 Approaches for Optimising Writable Memory in User Address Space

For writeable pages the situation is different. It has to be ensured that the old copy is used by all affected applications before the replacement and all applications use the new pages after the exchange. Non-atomic exchanges can only be allowed if a consistent view can be maintained during the exchange. One possible way to do that would be to update the old and new location when a write occurs. This has the risk that the memcpy is copying the written address at that time and has the old content in a register, what would cause a lost update. To allow this approach the memcpy had to be split into smaller parts (e.g., 128 Byte blocks) writes to the currently copied block had to be delayed. To implement that write permission had to be revoked. When a pagefault occurs, the faulting write operation had to be emulated, instead of handling the pagefault by restoring rights. Program execution would continue after the faulting instruction, still without write permission. With this method a frequently written location would cause many pagefaults. A simpler solution would be to just revoke write permission and stop handling pagefaults for this page during the copy. To reduce the number of pagefaults after the exchange the original rights could be restored.

4.6.1.3 Approaches for Optimising Writable Memory in Kernel Address Space

The pagefault handling based approaches only work for writable user pages. Removing cli-sti when replacing writable kernel pages is difficult. Revoking write permission to certain data structures might block the system. One possibility to solve that would be the usage of finegrained locking. Before moving a page the included data had to be locked. After the replacement is complete, the lock can be released.

4.6.1.4 Summary

In all possible variants writes to a page, which is replaced at that time, will cause a penalty. For user data this overhead could be tolerated and avoided if applications just do not write to pages that are replaced at that time (exchange in a critical section). The actual problem is with kernel data. As those pages are a shared resource, applications cannot ensure that it is not written by other applications syscalls(e.g., an application moves the mapdb while another application performs a map operation). This could only be avoided if an application can only move kernel data exclusively used by itself.

Exchanges in the user address space could be done without the exchange mechanism. Applications could do the copy on their own and map the new pages to the old location themselves. If a pager wants to replace a page of a client it could simply unmap it and copy the content. When the client accesses the page the next time it will cause a pagefault and the new page is mapped (pager can delay handling this fault when the copying is not finished). This would avoid the cli-sti in the kernel. But it would take longer as multiple syscalls and pagefaults are involved. When using the syscall for the exchange there is only one kernel entry and no pagefaults have to be handled.

4.6.2 Multiple Replacements

It cannot be avoided that multiple applications try to move the same page into the SPM. With its separate address spaces of applications the problem can only occur if pages are shared. This case can be ignored as applications that share a writeable location have to cooperate anyway. They have to maintaine consistency on their own. In the case of readonly data the only problem is the waste of resources. Applications could be modified to do the exchange once, at a level in the pager hierarchy where the replacement affects all participating applications.

For kernel memory this has already been discussed in the *Design* chapter. The proposed solution was to only allow applications to move kernel data that is related to them and decide during startup, which kernel code is stored in the TCM and what is stored in main memory. This solves the problem because no 2 applications move the same kernel pages. But it limits flexibility. Another solution might be to allow only one application to do exchanges in the kernel address space depending on the current situation. This could be used to implement the complex handling of lists of available physical pages for a kernel page in user space and keep the implementation in the kernel simple. Additionally this would allow reusing page resources for other purposes, if multiple applications try to replace the same page.

4.6.3 Consistency of Shared Memory

Consistency needs not to be maintained, when writeable shared pages in the user address space are replaced. As mentioned before, this problem has to be handled by the applications by doing the exchange in the pager that provides the shared memory instead of local exchanges in the applications.

4.6.4 Superpages

Exchanging superpages is currently only possible if a superpage mapping is not split into smaller mappings in higher levels of the pager hierarchy. The exchange would be done under cli-sti, so it is currently not useful at all. The only solution for that seems to be to split the mapping at the level where it should be replaced and exchange the small pages solitary. After all pages are replaced, a new superpage mapping could be established until a level in the pager hierarchy is reached where the pages are mapped with smaller granularity. Pagers can do that without additional kernel support for the user address space. In the kernel address space superpage mappings are not used any more.

4.6.5 Memory Overhead

Storing all locations a kernel page is mapped to in the Frame_description is not as efficient as it could be. Since most pages should be user pages, which do not need this information, the overhead could be reduced by using the mapdb for the kernel address space as well. The mapdb needs less memory and would be more flexible as there is no limitation how often a page can appear in the kernel address space (e.g., the zero page might need more than 3 addresses). Another waste of memory are the 4 addresses per page reserved for finding the page directories. As only few pages contain page tables this overhead is not acceptable. This should be put into a separate data structure limited to the memory region (Mapped_allocator) where page tables are allocated. The pointer to small pages is not needed for the smallest page size.

4.6.6 Non-executable DTCM

An annoying feature of the ARM architecture is that instruction fetches in the DTCM area are access the main memory. Additionally it cannot be derived from a page table entry if it contains code or data. There are only 2 access permission bits in the page table entry (user-accessable, writable). The user-accessable-Bit is used for read- and executable information, which therefore cannot be distinguished. This makes it necessary to specify what rights are needed on a page when moving it to avoid code being placed in the DTCM. To get rid of this, the DTCM could be placed at a physical address where main memory is available too. When moving a page to the DTCM it could be temporarily disabled and the content could be copied to the main memory too. Then the data in this region could be accessed fast within SPM and instruction fetches would read from main memory. But it would require to copy twice. When placing the DTCM at a physical address where no main memory is available an instruction fetch will cause an exception that would have to be handled (e.g., by moving the page back to main memory).

4.7 Port to x86

For easier debugging and evaluation the mechanism has been ported to the x86 architecture. The implementation only differs in some architecture specific details. The size of a page table on x86 is 4K, the small page size. Because of this, only 1 page directory address has to be stored per page. The parts of the code that use this information had to be modified to support x86. Two special cases of pages on x86 are not handled, cpu_-page and service_page. Both are marked to be non-replaceable. They contain hardware related information. Moving them could result in unexpected behaviour. There might be more pages directly accessed by the hardware that could cause problems, when replacing them without updating special CPU registers appropriately. The cpu_page is now mapped as 4K page to avoid superpage mappings in kernel address space. Currently there is no indication that there will be a x86 processor with scratch pad memory in the near future. So this part is currently only useful for development and debugging.

5 Evaluation

As no ARM hardware with TCM was available for doing measurements, evaluation was done on a 800MHz AMD Duron and the instruction set simulator Simics [4]. The AMD Duron was only used to measure the execution time of the syscalls. Simics was used to generate memory traces to prove that the mechanism works.

5.1 Implementation Costs

The current implementation adds about 2000 LOC to the kernel (1500 generic, 250 x86, 250 ARM) including debugging code and instrumentation for doing measurements. The necessary code part is about 1000 LOC.

5.2 Overhead

The superpage mappings in the kernel are replaced by 4K mappings. Because of this additional memory is required for page tables. The memory overhead of 0,1% for the page tables is negligible(one 4K page table per 4M region on x86, one 1K page table per 1M region on ARM). Runtime overheads may occur because superpage entries in the TLB are no longer used. The contention on the 4K entries is higher what potentially causes more TLB misses. The magnitude of this effect depends on the number of TLB entries, their replacement strategy, and the load on the system. Estimating it was way to complex to be covered by this work. Further memory is required for the additional data structure used to store the state of the physical pages. On ARM the size of the datastructure is 36 Byte per frame for both supported page sizes. The memory required for the 4K pages in a 1M region is 9216 Byte (256 pages * 36 Byte/page). Because of the currently used aligned allocation 12288 Bytes are allocated per superpage. The following table shows the overhead for 64 MB of memory on ARM:

	needed memory	allocated memory
Frame_descriptions for 64 superpages	2304 Byte	4096 Byte
Frame_descriptions for small pages	589324 Byte	786432 Byte
Total	578 KB	772 KB

On x86 24 Bytes are sufficient per frame. 24576 Bytes are needed for the 1024 small pages per 4M region. As this is a multiple of the page size no additional overhead is caused by aligned allocation. The following table shows the overhead for 64 MB of memory on x86:

	needed memory	allocated memory
Frame_descriptions for 16 superpages	384 Byte	4096 Byte
Frame_descriptions for small pages	393216 Byte	393216 Byte
Total	384 KB	388 KB

As described in the previous chapter much of the information in the data structure is only needed for kernel pages. When the mapping data base would be used for the kernel address space the 3 addresses for mappings in the kernel address space would not be necessary. This would save 12 Byte per frame (33% on ARM, 50% on x86) in the Frame_descriptions. It would require additional memory in the mapping data base, but only for kernel pages. The pointers to the page directory are only needed for page tables. Storing this information in a separate data structure would save another 16 Byte on ARM and 4 Byte on x86. Finally the pointer to smaller pages is not needed for the smallest page size. With all these optimisations the memory overhead could be reduced to 8 Byte for each superpage plus 4 Byte per small page.

	needed memory	allocated memory
ARM		
Frame_descr. for 64 superpages	512 Byte	4096 Byte
Frame_descr. for small pages	65536 Byte (64*1024)	262144 Byte(64*4096)
Total	65 KB	260 KB
x86		
Frame_descr. for 16 superpages	128 Byte	4096 Byte
Frame_descr. for small pages	65536 Byte	65536 Byte
Total	65 KB	68 Byte

The alignment overhead on ARM could be reduced as well. To sum up, the current implementation of the data structure causes a memory overhead of up to 1,18%. With the mentioned optimisations, this overhead could be reduced to about 0,1%

Fiasco maps Allocator memory linearly and physically contiguous at startup. This was used for easy address translations before adding the exchange mechanism. The possibility that pages might be replaced requires using the page tables for translating virtual to physical addresses and looking up the reverse translations in the Frame_descriptions when translating physical to virtual addresses. The simple translation functions from class Mem_layout cannot be used any longer. Those lookups of course take longer than adding or subtracting an offset. The runtime overhead caused by this could not be measured, because analysing the kernel code to figure out how often those functions are used at runtime was to complex. Page table lookups and accessing the Frame_descriptions cannot cause pagefaults (memory from Mapped_allocator), so there is an upper bound for each lookup. But the total overhead depends on the frequency of such translations. Memory allocated from Vmem_alloc is not affected by this, as it is not mapped linear, translations are complex anyway. So it is expected that this does not add much overhead.

If it turns out that this runtime overhead introduced by TLB misses and the following page table walk cannot be tolerated, the dimension of the waiting times might be reduced significantly by placing the page tables and Frame_descriptions itself in the fast on-chip memory. This matter remains to be investigated in future work.

5.3 runtime of syscalls

The costs of the syscalls are as follows (800MHz AMD Duron):

Call	min cycles	max cycles	average
create a 4K user page (idle)	19416	21511	20106
create a 4K user page (under load)	23513	24373	23931
delete a 4K user page (idle)	2185	2228	2215
delete a 4K user page (under load)	2405	2481	2430

The costs for creating user pages include a memset to 0 what explains the longer runtime. Clearing the page is needed as the page resources possibly contains code and data from a previous use, which could reveal kernel internal secrets if not cleared. Creating user pages was measured on page resources that were already mapped into the applications virtual address space as that is the way it is meant to be used. (Creating user pages on invalid addresses is currently supported, but thats just a workaround as the roottask only maps pages when pagefaults occur. Later on applications should request page resources in advance from another pager.) Creating and deleting user pages is not critical. It does not block the kernel as it runs with enabled interrupts. Furthermore requesting page resources from a pager will produce so much overhead (ipc, mapping) that this additional overhead can be ignored.

Call	min cycles	max cycles	average
replace 4K page of user data (idle)	34504	37067	35700
replace 4K page of user data (under load)	67286	68502	67849
replace 4K page of user code (idle)	32243	40419	37444
replace 4K page of user code (under load)	65180	75267	67228
replace 4K page of kernel data (idle)	35550	39774	36630
replace 4K page of kernel data (under load)	54559	55987	55477
replace 4K page of kernel code (idle)	32657	46161	36359
replace 4K page of kernel code (under load)	55524	68861	58295

Replacements in the kernel address space perform better under load. A possible explanation for this behaviour is that replacements in kernel address space might benefit from global TLB entries, whereas the TLB entries for the user address space are flushed when switching to another application. The measurements for replacing pages in the user address space were done with pages, which had not been mapped any further. Replacing pages, which are mapped to other applications will cause additional overhead for updating the mapping data base and all affected page tables. The figures for kernel data refer to the common case. Page tables and page directories are special cases of kernel data. Replacing them causes additional overhead.

special page to be replaced	min cycles	max cycles	average
page table for user address space (idle)	31338	37717	33729
page table for user address space (under load)	60003	68757	64437
page table for kernel address space (idle)	40911	43727	42658
page table for kernel address space (under load)	85884	93801	90097
applications page directory (idle)	30971	43619	35521
applications page directory (under load)	59363	69458	64424
kernel page directory (idle)	32838	35775	34092
kernel page directory (under load)	58594	59993	59271

Replacing page tables requires the corresponding page directory to be updated. Locating the raw page directory entry and updating it causes overhead. Additionally, the affected virtual address has to be flushed from the TLB, as the translation becomes invalid. When replacing a page table that belongs to the kdir all existing page directories have to be updated. This involves walking through the space registry and causes much overhead. In the test scenario 4 address spaces were active (roottask, sigma0, the test application, and the application producing the load). With more tasks running the penalty could be even higher. When replacing a page directory, it has to be checked if it is currently used by the CPU. If that is the case, the page directory base register has to be reloaded.

The problem with the current implementation of the replacement is the usage of cli-sti to ensure atomic exchanges. Therefore, it was tested how long the cli-sti period can be. This is important as long blocking times have a negative effect on the real-time capabilities.

4K page of	min (idle)	max (idle)	avg (idle)	min (load)	max (load)	avg (load)
user data						
cli-sti	28025	32529	28254	30209	33258	32497
copy	22210	23335	22311	19393	22409	22002
frame_desc	898	1307	913	1089	1353	1132
mapdb	3625	5557	3743	7196	7806	7411
page table	776	1259	787	800	988	831
user code						
cli-sti	25082	25747	25291	29572	38063	30694
copy	19297	19489	19367	19273	27600	20162
frame_desc	882	1258	931	1086	1143	1102
mapdb	3668	3884	3758	7153	7620	7455
page table	776	777	776	798	890	814

4K page of	min (idle)	max (idle)	avg (idle)	min (load)	max (load)	avg (load)
kernel data						
cli-sti	22097	23366	22440	23709	24185	24005
copy	19255	19758	19525	19220	19412	19314
frame_desc	901	1087	995	1135	1307	1207
page table	646	879	651	858	877	863
kernel code						
cli-sti	18566	32535	22951	23373	34049	25427
copy	15795	29595	20050	18964	29371	20765
frame_desc	898	1108	983	1115	1300	1205
page table	615	1721	664	854	1839	963
user page table						
cli-sti	25038	31565	26513	27365	34721	31055
copy	22089	28643	23547	22074	29353	25731
frame_desc	898	1110	913	1095	1100	1100
page directory	175	395	185	419	602	573
page table	566	566	566	777	777	777
kernel page table						
cli-sti	48196	55883	52977	50532	57852	54173
copy	22217	29593	26952	21674	28697	25189
frame_desc	900	934	912	1101	1112	1107
page directories	23115	23820	23227	24168	24524	24369
page table	567	567	567	779	795	781
user page directory						
cli-sti	25222	32898	29022	27864	35095	31455
copy	21622	28824	25332	21518	28638	25044
frame_desc	905	956	929	1081	1092	1090
page table	1003	1062	1012	1221	1291	1243
pdbr	365	608	388	754	899	790
kernel page directory						
cli-sti	22226	25123	24828	24437	27189	26806
copy	18992	21916	21677	19130	22027	21650
frame_desc	891	1039	900	1078	1246	1097
page table	627	663	635	852	863	858
pdbr	183	343	190	400	437	430

It can be seen that the longest operatins within the cli-sti are the memcpy and, in case of moving page tables of the kernel address space, updating all page directories. When replacing pages from the user address space, the update of the mapping data base is significant too. The mapdb update will take even longer when the pages that are replaced are mapped to other applications, as more page tables have to be updated then and it becomes necessary to copy mapping trees.

	min	max	median	variation
cli-sti	18566	57852	38209	51%
cli-sti (no special pages)	18566	38063	28314	34%
copy	15795	29371	22583	30%
update frame_descriptions	882	1353	1118	21%
update mapdb	3625	7806	5715	36%
update page table	566	1839	1202	53%
update page directories (kernel pt)	23115	24524	23819	3%
update page directories (user pt)	175	602	388	55%
update pdbr (kdir)	183	437	310	41%
update pdbr (user pd)	365	899	632	42%

The table shows the over all variation of the exchange systemcall. Most of the variation depends on the current load on the system and on the exact use case (replacing page tables of kernel address space is the worst case that needs much longer than all other use cases). The memcpy shows a high variation in runtime, which is independent from the work load. The reason for that has not been found so far. The variations might be related to unpredictability caused by the hardware (e.g., branch prediction, limitations in the memory controller). Cache and TLB do not show higher miss rates when copying lasts longer, so it seems that this behaviour is not related to the cache or TLB. Tests with pingpong showed deterministic behaviour, but pingpong uses its own implementations and does not test the standard *memcpy() [string.h]*, which is used for copying in the current implementation. The worst case measured was a cli-sti period of 57852. That is about 75 µs on the machine used for the measurements. Without the special cases the longest non-preemptable period is about 50 µs (up to 37 µs only for the memcpy).

5.4 Benefit

Simics tracing facility was used to evaluate the potential benefits of using the SPM. Memory traces were produced for a test routine within the test application. In the first run nothing was moved to the (emulated) scratch pad memory region, the second run was done with the used data moved to SPM. The third run was done with the code moved to scratch pad memory. The last run was done with code and data in scratch pad memory.

run	hit rate
unmodified	0,00%
data in SPM	30,18%
code in SPM	68,97%
code and data in SPM	98,52%

As expected, the SPM is not used at all, if code and data is not explicitly moved there. The figures for only moving data or code to the SPM are application specific and not representative for other scenarios. The figures show that about 70% of the memory accesses in the used method are caused by instruction fetches and about 30% are caused by reading or writing data. When code and data is moved to the SPM, the application runs completely in the fast memory. The reason for the missing 1,5% is the scheduling in the kernel (periodic timer interrupts) that was recorded by the tracer too. In this scenario the application does not use any other systemcalls and no other applications were running.

The figures prove that the mechanism can be used by applications to place code and data in the SPM. The properties of the SPM ensure that the code and data inside the SPM can allways be accessed at low latency. But there are other sources of uncertainty.

- Before accessing the SPM the physical address is needed, but TLB misses cannot be easily eliminated.
- Many CPUs feature branch prediction. The branch history tables, used to implement that, are influenced by all applications what causes uncertainty in fully preemptable systems.

Because of the TLB misses, it is unlikely that the existing results on predictability of SPM in embedded systems are still valid, when virtual memory is used. Because of missing time it was not possible to investigate that further.

6 Conclusion and Future Work

The goal of this work was to implement a basic mechanism for managing scratch pad memory in the Fiasco microkernel and to show that it is useful for reducing the WCET and for predicting it better. The mechanism has been implemented and evaluated. The effect on performance and the predictability of worst case execution times is not clear yet. This chapter summarises what has been archived in this work, and gives an outlook to future work based on the achieved state.

6.1 Results

- The most important result is that it is possible to use Fiascos mapping mechanism to manage the access of applications to scratch pad memory. Pagers can use fast memory pages to speed up applications. They can automatically improve the performance of unmodified applications by moving the pages, which the applications use most frequently, into the on-chip memory. Furthermore pagers could be used to combine the existing compiler based approaches with virtual memory by constructing appropriate virtual address spaces, which are partially backed by SPM. Alternatively, applications can use the provided syscalls to manage their address space on their own. That enables the programmer to use additional knowledge about an application to use the SPM more efficient. In contrast to the existing dynamic methods, the decisions about the placement of code and data can be done at runtime, depending on the currently available amount of scratch pad memory. This could possibly be used for quality of service management.
- To allow moving kernel code and data to the SPM at a reasonable granularity superpage mappings cannot be used in the kernel address space. Exchanging superpages in user address space is not necessary, they can be split into mappings of small pages before exchanging them.
- Although the latency of an SPM access is always low, TLB misses could increase the overall latency. As a consequence currently no statement can be made about the effect of scratch pad memory on WCET prediction when using virtual memory. address apace identifiers could possibly be used to partition the TLB. If a real-time task has exclusive access to a certain TLB partition TLB misses become predictable. A better solution would be to have a second completely programmable buffer that stores the translations for the pages currently located in the SPM. Upon creation of a user page, the virtual and physical address of the page could be stored in that buffer together with an address space identifier. When moving kernel pages its virtual to physical translation could be stored as global entry. This

would guarantee fast address translation for the pages in the SPM. Shared memory would require multiple entries per SPM-page or the virtual addresses would have to be identical in all address spaces the page is mapped to.

- Replacements in the user address space can be handled using pagers without additional kernel functionality. The exchange can be implemented by doing an unmap, copying the content to new location, and establishing a new mapping. This, however, involves ipc and causes pagefaults. In contrast to that, the kernel mechanism could possibly guarantee an upper bound for the execution time of replacements. Copying, updating the Frame_descriptions and changing one page table entry in the active page directory surely have an upper bound. On the other hand updating all page directories, as required when replacing page tables for the kernel address space, and modifications in the mapdb are potentially unbounded. Nothing can be done about the page tables, so there is no upper bound for that. If replacements in kernel address space, as proposed, involve a user level server there are no upper bounds for exchanges in kernel address space anyway. But there might be a chance for exchanges in user address space. The measurements in the *Evaluation* chapter indicate that there is an upper bound for the mapdb update. Under the assumption that the page that is replaced is not mapped to other applications, the mapdb update includes 2 insertions, 2 deletions and a fixed number of lookups of mappings. If an upper bound is found for that, then there is one for exchanging exclusively used pages in user address space. This would enable real-time applications to move pages within the critical path.
- As with the used method replacements can only be done with page size granularity, the scratch pad memory cannot completely replace caches. Applications that are not aware of the scratch pad memory need caches to show a good performance.
- Splitting the SPM into an instruction part and a data part, as done in the viewed ARM processor, reduces the latency of the memory as both parts can be placed close to the CPU-units that mainly access them. On the other hand, it makes managing the memory more difficult, as it has to be distinguished between code and data. A unified scratch pad memory would be easier to use. If the SPM is split into data and instruction part, an access permission bit in the page table entries that allows to distinguish code and data would be useful (e.g., NX-Bit).

6.2 Design Goal Review

1. Allowing to share the SPM between applications without destroying the isolation of the address spaces is achieved by using the existing mapping mechanism that ensures isolation.
2. Sharing of SPM regions between applications is supported, but it has to be used carefully to prevent inconsistency.
3. applications can move kernel code and data into the SPM. But currently there is no mechanism implemented to control which applications can move which pages.

4. The consistent view of all application to shared objects is preserved for the kernel address space. As mentioned above the mechanism has to be used carefully with shared memory in user address space.
5. The real-time capabilities of the kernel currently suffer from the usage of cli-sti. However, there are possibilities to reduce the blocking times in most cases.
6. Delegation of SPM-management is possible. Pager-hierarchies can be used to implement multiple strategies.
7. As the pagers are not affected by exchanges of their applications, the existing memory management is not influenced by the added functionality.
8. Replacing code and data objects of arbitrary size is not supported in the kernel address space. In the user address space, applications can manage that on their own by using the existing compiler based approaches or simply copy data and update pointers to indirectly addressed objects.
9. Because of the debugging code and instrumentation for measurements, the implementation is currently relatively huge.
10. The runtime of the syscalls is relatively long. Alternative copy methods (e.g., using SSE if available) could improve that.

6.3 Possible Uses of the Exchange Mechanism

Exchanging physical pages might be interesting for other use cases too.

- Categorising pages into *page resources*, *user pages*, and *kernel pages* could be used for dynamic allocation of kernel memory from sigma0.
- Because of the address space protection between applications, operations on the scratch pad memory are not observable by other applications. This property could improve security of some cryptographic algorithms.
- The replacement feature could also be used by sigma0 to free certain memory areas. If there is enough free memory available, all memory belonging to a memory module could be freed to allow replacing the physical device.
- The same mechanism could be used to manage embedded DRAM.

6.4 Future Work

So far only the basic mechanism has been implemented. It has been shown that the mechanism for moving code and data to the SPM works. If this can be used to improve performance or predict worst case execution times better, is not sure yet. The results presented in the chapter *Related Work* indicate that this is the case, but it remains to be investigated if this is destroyed by using virtual memory.

If it turns out that SPM can be used to improve performance or predictability, then the next step would be to implement the userlevel part that manages the allocation and usage of the scratch pad memory.

Some open issues remain to be solved:

- The kernel address space has to be protected against unwanted replacements (currently all applications can move any page of the kernel address space).
- Splitting (and reestablishing) superpage mappings is not yet implemented.
- ARM provides a DMA controller that could be used to fill the TCM in the background.

The implementation has to be cleaned up and optimised. Replacing the cli-sti by a preemptable protection mechanism is the most important challenge here.

6.5 Outlook

The final system based on the mechanisms presented in this work could lock like shown in Figure 3.8

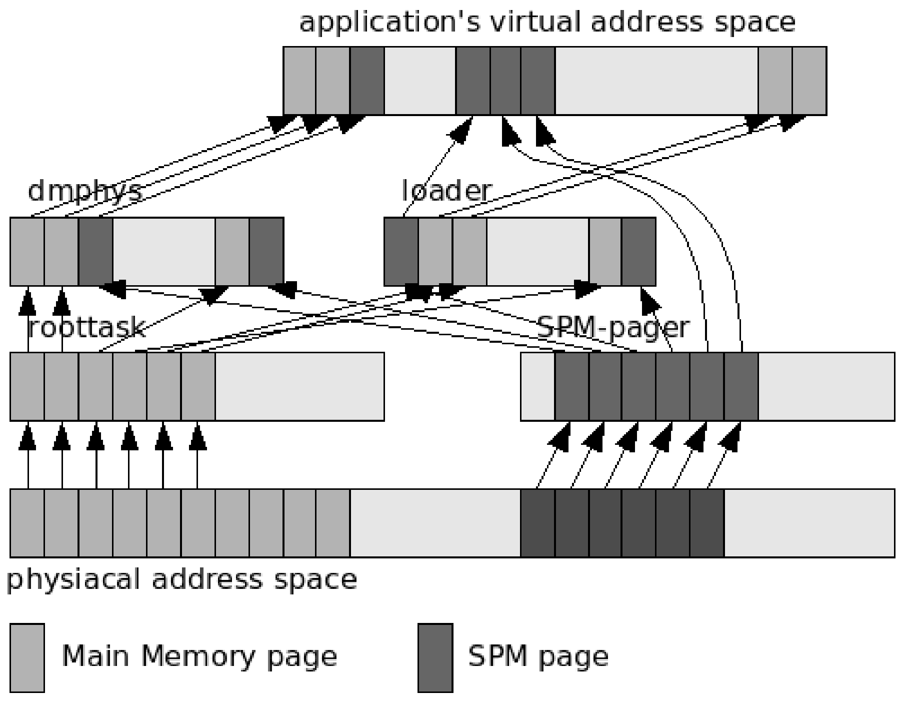

Figure 6.1: system overview

In this example, *loader* and *dmphys* use pages from the *SPM-pager* to replace the original pages, received from *roottask*. Both presume that the lowlevel pagers do not unmap any page. The exchanges done by the loader and dmphys are transparent to the applications. Additionally, the applications could receive SPM-pages from the *SPM-pager* or a higher level pager, which is based on this simple lowlevel pager. Those pages could be used explicitly by the applications.

Glossar

cli - instruction that disables the handling of interrupts, can be used together with sti to implement short non-preemptable sections in the kernel (longer sections should not be protected by cli-sti as this increases interrupt latencies)

content addressable memory - memory that is not addressed using the physical location but using a kind of key that is stored together with the content to identify the right entry

page directory base register - register in the CPU that stores the address of the currently used page directory

scratch pad memory - fast directly addressable on-chip memory

sti - instruction that enables interrupt handling

thread control block - data structure in the kernel used to store the state of a thread

tightly coupled memory - another term for scratch pad memory used by ARM

translation lookaside buffer - little buffer in the CPU that stores the physical addresses for recently accessed virtual addresses to reduce the latency of address translations (most caches use the physical address to locate the corresponding entry, so the physical address is needed fast)

DRAM - dynamic random access memory, based on capacitors, loses information when not refreshed regularly

eDRAM - embedded DRAM, DRAM that is integrated in the CPU

LOC - lines of code

LRU - least recently used, replacement strategy used in caches, tries to replace the content that has not been used for the longest time

SPM - see scratch pad memory

SRAM - static random access memory, semiconductor memory based on transistors, holds information without refreshes

TCB - see thread control block

TCM - see tightly coupled memory

ITCM - part of the TCM used for instructions

DTCM - part of the TCM used for data

TLB - see translation lookaside buffer

WCET - worst case execution time, the maximal runtime a piece of code needs if everything that can go wrong goes wrong (cache misses, TLB misses, interrupts, lock contention, etc.).

Bibliography

[1] Arm926ej-s technical reference manual. http://www.arm.com. 4, 21

[2] Arm966e-s technical reference manual. http://www.arm.com. 4, 21

[3] Qemu. http://fabrice.bellard.free.fr/qemu/. 23

[4] Simics. http://www.virtutech.com/. 33

[5] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization, 2006. 7

[6] Mahmut Kandemir, lsmail Kadayif, and Ugur Sezer. Exploiting scratch-pad memory using presburger formulas, 2001. 7

[7] David B. Kirk and Jay K. Strosnider. Smart (strategic memory allocation for real-time) cache design using the mips r3000, 1990. 7

[8] J. Liedtke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems, 1997. 7

[9] Jochen Liedtke. L4 reference manual, 1996. 4, 10, 11, 13

[10] O. Ozturk, M. Kandemir, and I. Kolcu. Shared scratch-pad memory space management, 2006. 8

[11] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Predictability of cache replacement policies, 2006. 2

[12] Stefan Steinke, Christoph Zobiegala, Lars Wehmeyer, and Peter Marwedel. Moving program objects to scratch-pad memory for energy reduction, 2001. 7

[13] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory, 2005. 8

[14] Sumesh Udayakumaran, Aangel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions, 2006. 7

[15] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability, 2003. 7

[16] Marcus Völp. Design and implementation of the recursive virtual address space model for small scale multiprocessor systems, 2002. 11, 12, 13

[17] Alexander Warg. Software structure and portability of the fiasco microkernel, 2003. 21

[18] Lars Wehmeyer and Peter Marwedel. Influence of onchip scratchpad memories on wcet prediction, 2004. 8