

Partial Array Self-refresh in Linux

Henrik Kjellberg, E05
et05hk6@student.lth.se

March 7, 2010

Abstract

Partial Array Self-refresh is a technique to turn off the memory self-refresh from parts of a memory if it is not used, to reduce the power consumption during idle mode.

In this paper I will start with describing what Partial Array Self-refresh is. I will briefly describe the memory management in Linux and how it works. I will describe some different approaches that can be used to accomplish Partial Array Self-refresh in a Linux environment and then implement one of the approaches. I will also describe what else has to be done to achieve good results from Partial Array Self-refresh in the form of defragmentation and anti-fragmentation.

Contents

1	Introduction	6
2	Previous work	6
3	Aim of the thesis	7
4	Memory system overview	7
4.1	Paging	7
4.2	Non-Unified Memory Access	8
4.3	Virtual memory	8
4.4	Linux	8
4.4.1	The Buddy Allocator	9
4.4.2	The SLAB allocator	10
5	Design analysis	11
5.1	Disabling of allocations	12
5.1.1	Memory hotplug	12
5.1.2	Pre-validate power down	13
5.2	Defragmentation	13
5.3	Anti-fragmentation	14
6	My implementation	15
7	Evaluation	17
7.1	Partial Array Self-refresh (PASR) feasibility	17
7.2	Overhead	18
8	The future	19
9	Conclusions	21
10	Acknowledgements	22
	Appendix A: Tools	23
A.1	Procs	23
A.2	Pasr-info	25
A.3	Kpageflags	26
A.4	Kpagefree	26
A.5	Seq-file	27
A.6	Page-display	27
A.7	Emulator	28
A.7.1	User-mode Linux	28
A.7.2	QEMU	28
A.8	Fillmem	28
A.9	Time_count	28
A.10	LXR	29
	Appendix B: Graphics	29

Abbreviations

API Application Programming Interface

CPU Central Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

ISA Industry Standard Architecture

LKML Linux Kernel Mailing List

MMU Memory Management Unit

NUMA Non-Uniform Memory Access

PASR Partial Array Self-refresh

pfn Pageframe Number

RAM Random Access Memory

TLB Translation Lookaside Buffer

1 Introduction

One of the greatest challenges in mobile systems of today is battery time. All unnecessary power consumers of a mobile system have to be minimized. One such consumer is the dynamic memory.

Dynamic Random Access Memory (DRAM), is a volatile memory, where every bit of memory consists of a capacitor and a transistor. When the capacitor is charged, the bit is set to one, and when it is uncharged, the bit is set to zero.

The main advantage with the construction is that only two components are needed, which leads to a high memory density on the chip. The disadvantage is the physical properties of a capacitor as a component. The physical capacitor has leak currents, which drains the capacitor. The leakage itself does not lead to noticeable losses, but the data will get corrupted unless it is constantly refreshed by recharging the capacitor. The recharging is done by reading out and writing back all the data of the memory. During normal operation, the refresh is handled by the operating system. When in idle mode, on the other hand, the refresh is done by the memory itself, in a so called self-refresh mode, since the processor is powered down.

To refresh the whole memory if only a part of it is in use is a superfluous waste of power. That is why Partial Array Self-refresh (PASR) was introduced. There is no software implementation making use of PASR in the Linux kernel as of today. This thesis aims to present a possible implementation of PASR and evaluate its efficiency.

2 Previous work

In 2001, the Western Research Laboratory (WRL) in Palo Alto, California, did an investigation [24] on the power consumption of different parts of a mobile system. They found out that up to 35% of the power consumed during sleep mode, were used to maintain the dynamic memory. To handle this, they proposed a function for powering off the self-refresh for parts of the memory that were not in use. Today, such a function exists. It is called Partial Array Self-refresh, and is part of the JEDEC standards for LPDDR [15], LPDDR2 [17], DDR2 [16] and DDR3 [14].

In the classical PASR, Single ended PASR, the memory can be set to just update the lowest half or fourth of the memory, leaving the rest to be corrupted. Depending on the type of memory, one eighth and one sixteenth can be selected as well.

Modern memories are split into banks, enabling higher speed and features like pre-charging. In the new standard LPDDR2-S4 [17] the self-refresh of the memory banks can be powered down individually on a per-memory-bank basis as well as the traditional single ended. It also allows self-refresh to be turned off based on the segment. The segment is the upper bits of the row address. The size of a segment is a number of rows, depending on the density of the memory.

In 2007, T.Brandt, T.Morris and K.Darroudi released, in cooperation with Intel, an article[4] that describes a real implementation of PASR, in which the power consumption where measured. The results are found in Figure 1. The solution proposed in the article is based on the Single ended PASR, though they do propose the Bank-wise solution found in LPDDR2-S4. The LPDDR2-S4

Power Saving in Relation to PASR

PASR Ratio	Memory Retained	Self-Refresh power	PASR Power Savings
Full SR	64 MB	0.977mW	0%
1/2 PASR	32 MB	0.670mW	8%
1/4 PASR	16 MB	0.516mW	12%
1/8 PASR	8 MB	0.424mW	14%
1/16 PASR	4 MB	0.374mW	15%

Handheld w/64MB DRAM using ~4mW total in sleep mode²

Figure 1: Power Saving in Relation to PASR

memories are not found at the retailer yet, but are available from the manufacturer for large customers. Brandt et al. measured a noticeable saving of power, but large amounts of data were moved just before and after the self-refresh, so I would predict a little less savings. Besides the energy losses from moving data around, the time overhead for executing PASR is an important variable. The authors of the article are talking about a fraction of a second for taking the system back online after idle mode. This cannot be considered a satisfying solution.

3 Aim of the thesis

The goal of this thesis is to suggest how PASR is to be implemented in the Linux kernel, in one implementation or as several sub-problems. To do this, the work that has been done shall be investigated to see if it is possible to reuse existing solutions. The proposed solution shall be implemented in software and evaluated regarding execution overhead time and feasibility. The problems involved in achieving PASR shall be evaluated. Limitations like fragmentation and hardware restrictions shall be studied.

4 Memory system overview

4.1 Paging

The memory of a modern computer system is usually divided into blocks, called pageframes, with a fixed size. Usually a pageframe can hold 4KB, but that is not categorical. The data stored in a pageframe is called a page. The reason for this partitioning is primarily to reduce the fragmentation and to speed up the memory handling by dividing the memory into equally sized blocks.

Fragmentation is a problem that arises when two or more pages, or blocks of pages, gets allocated and one of them gets deallocated, leaving a gap of unused memory. The gap prevents larger blocks of memory from being contiguously allocated, or turned off using PASR.

The partitioning into pages is originally a pure software construction, but since the introduction of the Memory Management Unit (MMU), the hardware is involved in the page model as well. Apart from the actual memory chip, there

is a unit called MMU in the memory chain. The MMU takes a Virtual address and translates it into a physical address (more about this later).

Since the processor often uses the same part of the memory more than once, there is often a small cache as well, that stores translations from the MMU. The cache is called Translation Lookaside Buffer (TLB). The main task of the TLB is to be fast and to store the latest translations from the MMU.

4.2 Non-Unified Memory Access

Computers of today don't necessarily have just one processor. It's common with multi-processor platforms. It is also possible to have physical memory belonging to the different processors, leading to different access times for different parts of the memory space. This is called Non-Uniform Memory Access (NUMA). The introduction of NUMA leads to some complications since the access time increases drastically if the accessed memory doesn't belong to the executing processor. Due to this, allocations of memory belonging to other processors have to be avoided if possible.

4.3 Virtual memory

Virtual memory makes a process believe that it has all the addressable memory space available in Random Access Memory (RAM). This is done by translating virtual addresses into physical addresses when accessing the memory. It means that the virtual memory does not have to have a physical counterpart until it is accessed.

There are many advantages of using virtual memory. It simplifies the memory allocation when running several processes at once. It also gives the operating system a powerful, but yet simple way to control the access rights for sensitive parts of the memory.

Platforms with access to a hard drive or other non-volatile storage device can use the virtual memory system to move data that has not been in use for a while to the non-volatile storage device and then bring it back into RAM when it is being accessed. This allows using the same physical memory for more than one set of data. It can also allow some parts of the memory that by rights aren't free, to be set in low power mode using PASR.

4.4 Linux

Linux handles memory, just like many other systems, in page sized units. All pages of the system have descriptors, page-descriptors, describing their current state. All page-descriptors are stored in a large list called `mem_map`.

Each page-descriptor describes one physical page, located at one set of physical addresses. In most cases, though, the system works with virtual addresses instead of physical addresses. The virtual address space is divided into two parts, user-space and kernel-space. The kernel-space is strictly reserved for the kernel, device drivers and kernel extensions. User-space is where other processes can have their data.

User-space processes use virtual addresses handed out by the kernel. The kernel handles the translation between virtual and physical addresses by setting up and maintaining pagetables. Each process has its own pagetable, its set of

translations between virtual and physical addresses, which is loaded into the MMU when executing.

Linux supports NUMA and most of the memory management code of the kernel is influenced by it. To handle memory located on different processors, the memory is divided into nodes. One node represents one or more processors and the memory connected to it. On a computer with only one processor, all memory is located in the same node. This might seem confusing when working with Uniform Memory Access systems, but is used on such systems as well, due to portability.

The physical memory of each node is divided into zones. Just like the NUMA nodes, this partitioning has hardware restriction as basis. In most systems, there are up to three zones: `ZONE_DMA`, `ZONE_NORMAL` and `ZONE_HIGH`.

`ZONE_DMA` derives from hardware restrictions in the old Intel hardware Industry Standard Architecture (ISA)-bus, who's Direct Memory Access (DMA) has a 24bit address bus. A 24bit address enables addressing of up to 16MB. Because of that, data that shall be handled using ISA compatible DMA has to be located at the physical addresses 0 to 16MB.

`ZONE_HIGH` is based on a design decision. The idea is that all user-space processes shall have one part of the virtual address space dedicated to its own addresses and one part that belong to the kernel. The kernel part is common for all processes, while the rest is unique for each process. The user-space process cannot access the kernel data; it's just there to speed up context switches between user-space and kernel-space. Since the kernel addresses are always there, the MMU pagetable does not have to be flushed every time the kernel interrupts.

On a regular x86 system, the kernel is kept within 1GB of the available 4GB physical address space. From this 1GB, 128MB is used to map physical memory from the rest of the memory into kernelspace. Physical addresses from the end of the `ZONE_DMA` up to 896MB are therefore located in `ZONE_NORMAL` and if the system holds more than 1GB of memory, the rest of the memory is located in `ZONE_HIGH`, which is unavailable for the kernel.

Inside each zone, all the free memory of the system is listed using pointers to the corresponding page-descriptors. The page-descriptors of the free memory are located in either a small software cache, called Per-CPU pageframe cache, or in the so called Buddy lists.

4.4.1 The Buddy Allocator

The Buddy lists are part of the Buddy allocator, the main memory allocator of the Linux system. Each zone has five sets of 11 lists. The five sets correspond to the different levels of ability to migrate (see figure 11 on page 26).

Each list holds pointers to the first page-descriptor of all free blocks of 2^n pageframes within the zone, with n ranging from 0 to 10, representing the 11 lists. In other words, list no.0 contains all blocks of free single pageframes; list no.1 contains all blocks of two free pageframes and so on.

On system startup, all page-descriptors are located in the list of 2^{10} pageframes, the largest block. When a process wants some memory, the Buddy allocator starts by looking in the list of the size wanted, or the closest larger power-of-two. If that list contains a block, that block is allocated to the process. Otherwise the next larger power-of-two sized list is checked. If a block is

found there, it is split up in two, allocating one part (buddy) to the process and putting the other buddy in the list containing the smaller blocks.

When the process is done with its memory, it releases it to the Buddy allocator. The Buddy allocator checks in its lists to see if the buddy of the freed block is already free. If so, they are merged together and put in the list of the larger blocks. This avoids unnecessary fragmentation and an allocation is fairly fast.

The lists, though, have to be synchronized to avoid parallel processes to access it at the same time, creating race conditions. The synchronization is done with a spinlock, which unfortunately can be very time consuming.

To avoid having to take the spinlock, a cache of pageframes exists, the Per-CPU pageframe cache. The Per-CPU pageframe cache is an alternative to the Buddy allocator and can be used if only one page is needed. It is a list of free pageframes that are pre-allocated to one specific Central Processing Unit (CPU). Pages can be allocated from the Per-CPU pageframe cache with very short delay since no spinlock is needed for protection. Each zone has its own Per-CPU pageframe cache.

4.4.2 The SLAB allocator

Since memory often is allocated in smaller pieces than whole pages, the SLAB allocator was created. The SLAB allocator is a separate allocator that gets pages from the Buddy allocator and allocates smaller pieces from that.

The SLAB allocator consists of a list of caches. Each cache handles a specific type of object. There are a number of general caches, with preset sizes, and object specific caches for specific purposes.

All kernel threads can create their own caches for their own data types. One advantage of special purpose caches is that objects belonging to a specific purpose cache don't have to be reinitiated when freed and reallocated.

In every cache, there is one list of free, one list of partially free and one list of full slabs. A slab is usually one page in size and contains the objects of the specified type. Each cache also contains a CPU-local cache for fast allocations. See figure 2.

When an object is to be allocated, the SLAB allocator checks the specified cache for partially free slabs. If there is a partially free slab in the cache, an object is allocated from that slab. If not, the allocator checks in the list of free slabs. If a free slab exists, it is moved to the list of partially full slabs and an object is allocated.

When the last free object in a slab is allocated, the slab is moved to the list of full slabs. When the last object is freed from a slab, in similar ways the slab is moved to the list of free slabs.

When all the slabs of a cache are full, the SLAB allocator requests a new page from the Per-CPU pageframe cache.

To avoid undermining the work of the Buddy allocator, free pages have to be returned to the Buddy allocator at regular intervals. This is called reaping and is done by the `kswapd` and `cache_reap` kernel threads. When reaping the caches, the free slabs are returned to the Buddy allocator. A flag can be set to the cache to prevent the caches from being reaped.

The most common dynamic memory allocation function of the kernel is `kmalloc`. `Kmalloc` is actually an interface to the SLAB allocator. It allocates

memory from the closest larger general cache available.

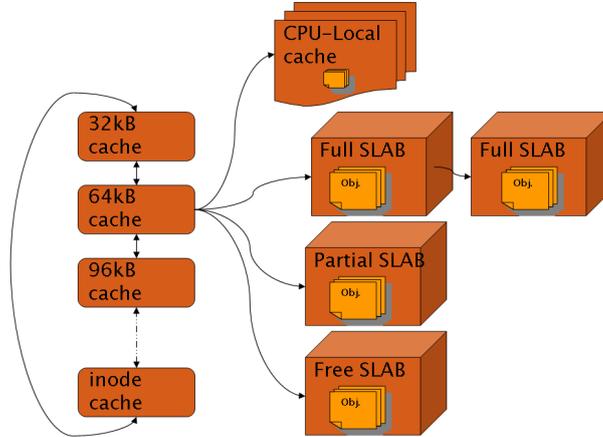


Figure 2: SLAB Allocator

5 Design analysis

To successfully implement PASR, a couple of decisions have to be made. The first one is to decide how to disable allocations of the memory region in software.

For some platforms, there is support for something called memory hotplug in the Linux kernel today. With memory hotplug, parts of the memory can be unplugged from the system by removing it from the list of available memory, the `mem_map` list, and the buddy lists, during operation. When this is done, the system cannot allocate memory from that region, and it's free to be set in low power mode using the Partial Array Self-refresh technique or by physically remove the memory module.

Another, more intuitive method, is to simply check so that there is no data in the region. This is done right before going into idle mode. If the system is set in low power mode and PASR is enabled atomically, no more allocations can be made inside the region because all application processes has been stopped, thus the system is safe. If the region is not free, the memory cannot be turned off using PASR without corrupting the data.

One other method that was investigated but later dropped was if the zone's found in the Linux kernel could be used. By setting up the zones to correspond to the regions that can be powered off using PASR, the code handling the zones can be used to see if there is any data in the zone. There are a couple of drawbacks with this solution. The first is that the zones exists for a reason. The zones is intended to be used to group memory depending on hardware restrictions like DMA controlled memory and software restrictions like kernel-space and user-space memory. The zones can't be used for both this and handling PASR. The second drawback is that the zones has got low water marks for the amount of free pages in the zone. If the number of free zones drops below that mark, the system will start freeing up memory using the `kswapd` kernel thread. Using the

zones to keep track of the regions would lead to smaller zones. Smaller zones would lead to a higher frequency of freeing and therefore unnecessary amounts of execution.

It's highly probable that putting the memory into low power mode using Single ended PASR will fail in a majority of the times if the memory is just checked for data. It's probable that some fragmented data will reside in the region of interest. To avoid this, an anti-fragmentation algorithm can be implemented. The goal with an anti-fragmentation algorithm is to allocate memory in a way that minimizes the fragmentation.

Unfortunately, no anti-fragmenting algorithm can be perfect since that would require a knowledge of the future. Since the future is not known, some kind of defragmentation is needed as well if optimal PASR is strived for. The memory that did get fragmented despite the anti-fragmentation, has to, if possible, be defragmented. This can be done by moving the fragmented data to safe memory or by freeing the fragmented data that is not needed anymore or can be recreated.

In order to save power, it's only needed to free the area and disable all allocations. Anti-fragmentations and defragmentations are optional and can be implemented after an evaluation of gain versus costs. The gain is the possibility to activate PASR on a larger part of the memory. The cost is mainly in the form of execution time, leading to an increased power consumption and latency.

5.1 Disabling of allocations

5.1.1 Memory hotplug

Memory hotplug is implemented on some platforms, mainly large server architectures. The main goal of memory hotplug is to be able to increase the amount of memory and to replace failing memory modules during operation. It can be used, though, to prevent allocation in a given region, enabling that region to be set in low power mode using PASR, as well.

In memory hotplug, the memory is removed by removing the corresponding page-descriptors from the zone. The operating system will not use physical memory at the given address range. When the page-descriptors are reinserted to the list, the memory is ready to be used again.

The memory hotplug algorithm is based on the assumption that every attempt to remove memory will succeed. When memory is to be removed, an attempt to isolate all the pages of the region is started. All allocated memory within the region is freed or migrated, and all Per-CPU pageframe caches are freed if possible. If any page, after a given number of tries to free or migrate, still can't be isolated, the page is considered unmovable and the removal is aborted. The migration can fail due to memory belonging to parts of the kernel or some DMA device driver. If the isolation succeeds, all page-descriptors of the region are removed from the zone and the region is no longer in use.

Removal of memory using memory hotplug is based on the partitioning made by the submodule sparsemem, whereby sparsemem has to be enabled as well.

The drawback of memory hotplug as memory removal method for PASR is that no evaluation is done before attempting to remove the memory. This leads to possibly large amounts of unnecessary work if the attempt fails. It is also a relatively complicated method for disabling allocations.

The advantage is that it is more portable and largely implemented into the Linux kernel as of today.

5.1.2 Pre-validate power down

To evaluate the possibilities to succeed, before trying to put the memory into low power mode using PASR, has the possibility to save time and energy, if the evaluation is effective. The more effective, the more profit is to gain on this method.

There are a couple of different ways to do this. One method, built for Single ended PASR, is proposed by Todd Brandt et al. in their article from 2007 [4]. Their solution uses a bitmap, with one bit per page in the system. When a bit is set to zero, the page is free and when a bit is set to one, the page is allocated. At the moment of powering down, the number of bits set to one is counted and the possible level of PASR is estimated.

Another method of evaluation is to have a counter for each region that can be set in low power mode using PASR. When the counter is zero, no page within the region is allocated. When a page within the region is allocated, the counter is incremented. The counters have to be checked in dependency order. If a region-counter has a low value, its pages might be possible to migrate or free. After the attempts to defragment, the evaluation has to be remade. After the last evaluation, when a certain part of the memory has been decided to be turned off, no more allocations must be done in that region.

5.2 Defragmentation

To achieve the optimal level of PASR, the pages outside a given safe region must be freed or moved. There are often pages outside the region even though there is enough free memory inside the region, due to fragmentation. The defragmentation can be done both in short terms and in long terms.

One short term defragmentation algorithm is the one proposed by Todd Brandt et al.[4] The bitmap described in chapter 5.1.2 is evaluated. A PASR boundry is set at the point where the amount of free pages below is the same as the allocated pages above. All pages above the boundry are moved into the free pages below the boundry, before going into low power mode. See figure 3 The pages that were moved have to be moved back before the system is fully powered up and restarted. Memory belonging to the kernel cannot be moved if this is to work, since it might be needed during power down and wake up. The stack and the pagetables of the MMU have to be placed in PASR safe memory. The main drawback of this method is, of course, the possibly large amount of data that has to be moved, at powering down as well as powering up. The defragmentation has to be redone every time as well. The main benefit is that the optimal level of PASR can be achieved.

A more sustainable defragmentation can be done by freeing as many pages as possible and migrate the rest. The main benefit of this is that the work done is not made undone at the next powering up, but is useful the next time the system is trying to power down into PASR as well. One implementation for doing this was published on the Linux Kernel Mailing List (LKML) by Mel Gorman at January 6 2010, in the patch "Memory Compaction v1" [11].

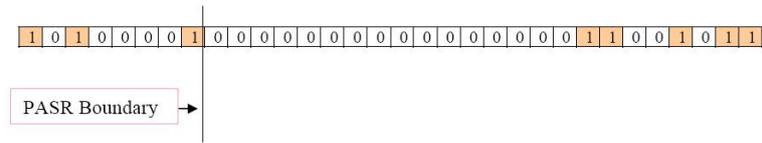


Figure 3: Bitmap as proposed by Todd Brandt et al.[4]

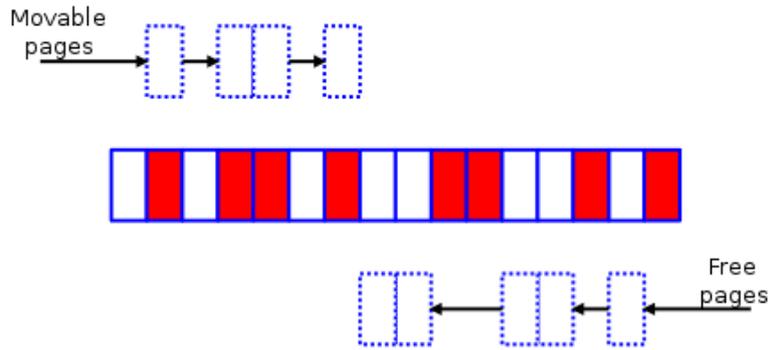


Figure 4: Memory compaction[6]

The basic idea behind memory compaction is very similar to the one that were used by Todd Brandt et al., but instead of keeping a list of all occupied pages at all time, the list is built at execution time. One algorithm searches for occupied pages, starting from one end. Another algorithm searches for free pages, starting from the other end. See figure 4.

Eventually, those two will meet. The page migration code is then used to move the occupied pages into the free pages found in the other end of the zone. The page migration code were originally designed for migration of data between nodes of NUMA systems but are today available to all systems. If all pages were movable or could be freed, the result is something like that of figure 5.

5.3 Anti-fragmentation

The idea behind anti-fragmentation is to avoid fragmentation at the time of allocation.

There is much work done within the Linux community on the topic of anti-fragmentation at the moment[10][9][12]. One of the algorithms that is implemented and in use today is zone movable. It means that a new zone,

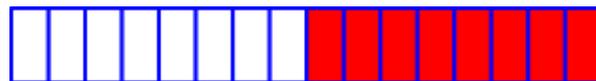


Figure 5: Memory compaction finished[6]

ZONE_MOVABLE[20], is created. Pages can be allocated from this zone by giving the `__GFP_MOVABLE` flag to the allocator. The zone is created with pages from the highest zone in the system, ex. `ZONE_HIGHMEM` if such is available. Movable data can be allocated both in the movable zone and in the normal zone, whilst normal pages, ie. pages not specified to be movable, cannot be allocated in the movable zone. All unmovable data will, due to this, stay within the boundaries of the normal zone, making it easier to power down the rest.

Another anti-fragmentation mechanism that is in use today is the migrate types mentioned in section 4.4.1. When booting the system, all pages are located in the set of Buddy lists that holds pages of the migrate type `MIGRATE_MOVABLE`. When an allocation is done for a page of any other migrate type, a block of tenth order, i.e. 2^{10} pages, is moved from the movable migrate type to the desired. During boot, the kernel allocates many long lived pages of memory. By forcing the kernel to reserve its memory, the unmovable, longlived pages are kept together, avoiding unnecessary fragmentation. Stacking pages of the same movability also makes defragmentation easier. Since pages of the same movability tend to have a similar lifetime, i.e. time from allocation to freeing, this grouping makes the Buddy allocators work more effective as well.

A more intuitive method, with Single Ended PASR in mind, is to allocate pages at the lowest physical address possible. The easiest way to do this is to continuously sort the lists of the Buddy allocator by the pageframe number. The problem with fragmentation due to allocations and deallocations still exists though, and it would probably be painfully slow as well.

The possibility to use the NUMA code for anti-fragmentation were investigated briefly. The idea were dropped, though, since NUMA introduces a huge amount of new code to the kernel. It would probably lead to prolonged execution times and increased power consumption.

The possibility to use the zones in a more application specific way were investigated as well. If the zones were used to split the memory into the PASR regions, the existing memory allocator would use the next zone as fallback when the first were full. The problem is that the zones keeps low water mark of free memory. When less than that amount of memory is free, the `kswapd` memory reclaiming function will launch. With increasing amount of zones comes smaller sizes of the zones. The smaller the zone is, the more often the `kswapd` reclaiming has to be performed [13].

6 My implementation

The Pre-validate power down approach was chosen because of the vast overhead of memory hotplug. The check was performed using counters for each region; the memory region is free when the counter is zero.

The memory is split up into regions using an array of a data structure, `struct pasr_region`, see figure 6. The struct contains the first Pageframe Number (pfn), the last pfn, the counter and a pointer to another `pasr_region`. A pfn is the physical address divided by the size of a page. The array is architecture specific and has to be redesigned for every new architecture. Each element of the array represents a region of the memory.

When the system is about to go into idle mode, the regions are checked one

after the other. If a region has the count value zero, but is depending on any other region, that region is checked recursively. When a region is found to be free, and any depending regions are free as well, the region is ready to be turned off.

```

/*
 * pasr_region will be put in an array, wich will have to be
 * sorted based on pfn.
 * start_pfn - is the first pfn of the region.
 * end_pfn is - the last pfn of the region.
 * depending - is an index pointing to another
 * struct pasr_region, upon wich this region is depending
 * to be able to turn of.
 */
struct pasr_region {
    unsigned long start_pfn;
    unsigned long end_pfn;
    atomic_t count;
    int depending;
};

```

Figure 6: Struct pasr_region

On a memory setup with Single ended PASR down to the level of 1/16, the array consists of five elements. The first element represents the region between 1/2 of the available memory up to the end. This region is not depending on any other region and can be turned off as soon as it's free, so its dependency field is set to `-NO_DEPENDENCY`. The second element represents the region between 1/4 and 1/2 of the memory. This region has its dependency field pointing to the first region, since it can only be put into low power mode using PASR if the first region, from 1/2 to the end, is powered down. The following regions are depending on the previous one in a similar way until reaching 1/16, the lowest level of PASR. When all but 1/16 of the memory is turned off, no more self-refresh can be powered down. The dependency field of the last region is therefore set to `-NOT_ABLE_TO_PASR` as an indication that it cannot be powered down using PASR.

On a memory setup with Bank selective PASR with 1/8 of granularity, the array consists of eight elements. One for each bank and all regions are independent of each other. Whenever a region counter is zero, that region can be turned off.

This design makes it possible to configure any memory setup, both systems with single memory as described above, and memory setups with more than one memory module, with different size and properties.

The counter operations has to be atomic, since an increment otherwise could occur at the same time as a check is being performed, leading to race problems. Hence the counter is implemented as an `atomic_t` value. To read and write to it, the Linux kernel atomic API is used.

The incrementing and decrementing, of course, has to be done whenever a page is allocated or freed. In kernel version 2.6.29, this is done in the file `Linux/mm/page_alloc.c`. The `page_alloc.c` is a large file that handles most of the low level memory allocations. It consists of both the Buddy allocator's code and the Per-CPU pageframe cache's code. It is a batch of about 45 functions that makes decisions about what and how to allocate and free the memory. Each one makes a decision and calls the next function, but in the end all requests ends up in one of three functions for allocations and one of two functions for freeing. The incrementing and decrementing of the region counters are performed in those five functions. The functions, `buffered_rmqueue`, `_rmqueue_smallest`, `_rmqueue_fallback`, `Free_hot_cold_page` and `__free_one_page` can be seen in the call-graph in figure 12 on page 30.

The problem with the counter is to initiate it to a correct value. When the system boots up, all memory is considered as reserved and is during the boot process freed and handed over to the Buddy allocator. The counters are initiated to its maximum value at an early stage of the boot process, before the memory is freed to the Buddy allocator. The counters are decremented when memory is freed to the Buddy allocator. In this way, the memory that is not freed up during boot process is still considered allocated and the rest is considered free.

7 Evaluation

To test the functionality of the implementation, a poisoning function was created. The function checks the configured memory regions to see if they are supposed to be empty. If so, it writes 0xAA in the whole region. If there were any valuable data in the region, the system would have behaved unpredictable. The function is implemented in a `cpu-idle` device driver module[19], so that the system executes it whenever it has nothing to do.

To give the system some dynamics, a simple application, `dynamics`, were written that randomly allocates and frees memory. A couple of `dynamics` processes, along with the poisoning `cpu-idle` module, were run overnight to ensure no vital data were overwritten.

The test succeeded and the system kept stable for both single ended PASR configuration and bank select PASR configuration.

7.1 PASR feasibility

To actually gain anything from PASR, it has to be feasible when in power-save mode. The biggest problem to encounter is the fragmentation of the memory.

A set of memory allocation applications were created to test the feasibility of the PASR and to study the fragmentation problem. The tools are called `fillmem_malloc` and `fillmem_static`. They are presented under Tools in Appendix A.

When first booting up the system, the tool `Page-display` found in chapter Tools in Appendix A generated the picture found in Figure 13a in Appendix B. The pale colours represents free pages and the rich colours represents allocated pages. As seen, there is a small amount of allocated data in the beginning of the memory. One small part is yellow, representing `MIGRATE_RESERVE` pages.

They cannot be moved and are probably holding kernel data. The rest of that block is green, representing movable pages. Near the end of the memory, there are a couple of data blocks of varying sizes. The blue ones are reclaimable, meaning the data can be removed if needed. The red ones are unmovable. Those are the ones creating problems for Single ended PASR, since that end of the memory is about to be corrupted. When using Bank selective PASR, this is not that much of a problem since the memory in the middle still can be powered down.

As can be seen in the pictures in the series of Figure 13a to Figure 13h, the user-space allocations keeps allocating mainly at the end of the memory. The allocations varies between movable and unmovable pages. The unmovable pages tends to fragment more than the movable. The reason for this is to me unknown. Even though there is no `ZONE_HIGH`, the user-space allocations still tends to allocate from the higher physical addresses. This is unfortunate if a Single ended PASR is to be implemented, since it blocks the upper half of the physical address space from being put into power save mode. If Bank selective PASR is to be implemented, though, this helps preventing fragmentation by spreading different allocations as far appart as possible. Larger areas can be freed by migrating if all movable pages are kept together, without unmovable pages in between. Allocations of the same type also tends to have a similar lifetime.

One thing that could be improved if Bank selective PASR is to be implemented is to implement the `ZONE_MOVABLE`. That would keep the red `MIGRATE_UNMOVABLE` blocks that starts to show up in figure 13g apart from the green `MIGRATE_MOVABLE` pages. The defragmentation gets simpler if the `MIGRATE_MOVABLE` pages are kept together.

To see if fragmented memory is handled in a satisfying way, a set of 512 processes were started, terminated and restarted a couple of times. For each step, an image of the memory usage were generated using the Page-display tool. Figure 14 shows how the 512 `fillmem_malloc` processes are started, closed and restarted. One can see how the same memory is reused each time.

Another test were performed to further study the usage of fragmented memory. A set of 512 processes were started. A set of 64 processes were then started. Those 64 processes fragmented the memory when the first 512 processes were terminated. When a new set of 512 processes were started, the fragmented free space were reused. In Figure 15, the steps are seen as 512 `fillmem_malloc` processes are started, another 64 are started, and then the first 512 are freed and later restarted. The two tests shows that the memory allocation of user-space data is handling an already fragmented memory well in terms of allocating new memory. What we can see is that the free memory is used in a last in, first out fashion, reusing the fragmented memory when possible.

7.2 Overhead

The idea of implementing PASR is to save power. To do that, the extra execution time and consequently the power spent enabling the PASR has to be kept at a minimum. To measure the time overhead of the counters and its maintaining, a simple program were written, see chapter A.9 in Appendix A. The program allocates two pages of memory in user-space using `malloc`, writes data to it and frees it again.

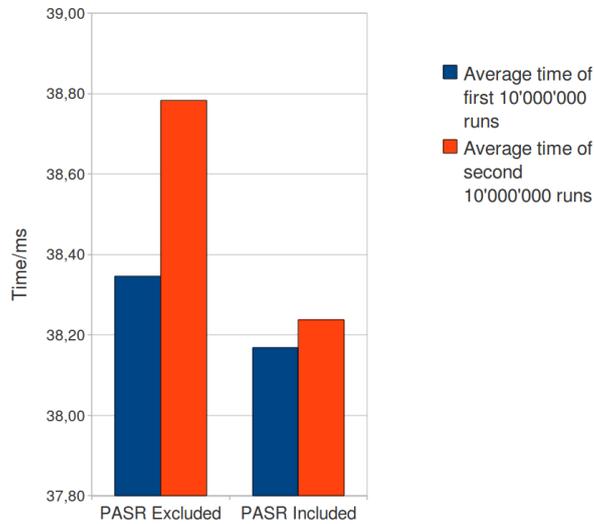


Figure 7: Time spent to maintain the PASR region counters

This is then repeated 10'000'000 times and the time spent on each allocation-writing-freeing cycle is measured using `gettimeofday[1]` and added up. In this way an average value is calculated for the time spent.

When the microsecond value overflows and the value of the second is increased, the time in microseconds won't be correct. Instead of spending time on calculating the correct time in microseconds when overflowing, those rare occasions were simply ignored and not added to the sum. The number of ignored values were varying between 380 and 395 out of 10'000'000 during the tests.

Figure 7 shows the average time from two executions of the `time_count` program run with PASR included and excluded from the compilation. As can be seen, the time to do a malloc of two pages size, write data to it and then free it, varies more from time to time than the extra time the PASR code introduces. This shows that the code that keeps track of occupied pages takes no measurable time.

On top of this, the time spent on putting the memory in and out of self-refresh has to be counted, as well as the time spent on the extra anti-fragmentation and defragmentation needed. The time spent on going in and out of self-refresh and activating PASR is mostly hardware dependent and can't be improved that much in software. The time spent on anti-fragmentation and defragmentation, on the other hand, is probably the ones that are the most time consuming in the PASR chain, and the ones that can be improved the most. It is outside the scope of this thesis to improve that time, though.

8 The future

The future for Partial Array Self-refresh is bright. Mobile systems of today needs to reduce their power consumption to gain battery time and compensate

for more powerful devices. The lower the total power consumption of a system gets, the greater the gain from Partial Array Self-refresh gets.

The solution proposed in this article is a fast and flexible way to check the memory usage for PASR, but some kind of anti-fragmentation and defragmentation can improve the results.

Defragmentation and anti-fragmentation is being worked hard at and has been so for some years now[6][9][10][11][12][20], since many large companies tries to optimize the memory hotplug functionality for their server architectures. Most of this work will be useful for PASR as well.

The screenshot from the Micron datasheet, from a fairly modern memory of 128MB, in figure 8 below shows the current consumption caused by self-refresh at different levels of PASR. Powering down the self-refresh from full array to 1/16 saves approximately 190 μ A at a temperature of 45°C.

Comparing these values with the values presented by Todd Brandt et al. in Figure 1[4] shows that the total power consumption has gone down slightly since Todd Brandt et al. did their article in 2007, but the gain of PASR is still significant. A modern cell phone has a current consumption of about 2mA[23] in idle mode giving a saving of almost 10% at a reduction of 190 μ A.

This saving might not lead to a prolonged battery time of 10% when using PASR. The saving is only valid when in idle mode and all but 1/16 of the memory is powered down. It is unusual to reach 1/16 PASR, although possible on a 128MB memory. 1/16 of 128MB is 8MB and the Linux kernel can run at a theoretical memory minimum of 4MB.



**1Gb: x16, x32 Mobile LPDDR SDRAM
Electrical Specifications – I_{DD} Parameters**

Table 9: I_{DD6} Specifications and Conditions

Notes 1–5, 7, and 12 apply to all the parameters/conditions in this table; V_{DD}/V_{DDQ} = 1.70–1.95V

Parameter/Condition	Symbol	Low Power	Standard	Units	
Self refresh: CKE = LOW; t _{CK} = t _{CK} (MIN); Address and control inputs are stable; Data bus inputs are stable	I _{DD6}	Full array, 85°C	1000	1200	μ A
		Full array, 45°C	500	750	μ A
		1/2 array, 85°C	750	900	μ A
		1/2 array, 45°C	440	730	μ A
		1/4 array, 85°C	600	750	μ A
		1/4 array, 45°C	380	680	μ A
		1/8 array, 85°C	550	750	μ A
		1/8 array, 45°C	350	620	μ A
		1/16 array, 85°C	500	700	μ A
		1/16 array, 45°C	330	540	μ A

Figure 8: Current consumption at different levels of PASR for a Micron 1Gb LPDDR SDRAM[21]

The most gain from PASR will be achieved if the hardware allows Bank selective PASR. When looking at the graphics in the Appendix B it becomes clear that as long as the memory layout of Linux is not changed, any attempt of performing Single ended PASR will require defragmentation to move the data from the half of the memory that will be powered down first. Bank selective PASR on the other hand will, in most cases, succeed to some level without further work.

One problem that can make PASR hard or even impossible to enable is the way some performance boosting memory controllers handles bank addresses. Some memory controller puts the bank select signal in the middle of the address, splitting and spreading the memory banks over the contiguous addresses.

When one address is accessed, it's probable that the address right after it will be accessed too in a short time. By putting the addresses on different banks, the next bank can be prepared for the next access, making the access pipelined. This is good when looking at pure performance numbers, but when it comes to power management, it puts some large sticks in the wheels. By putting the bank select signal in the middle of the address, the addresses on the memory side is not maintained on the processor side. That means that the lower half of the addresses on the memory is not the lower half of the addresses as they are presented to the processor and thus the organization of safe and unsafe memory in the operating system gets very complicated. It means that one safe block gets split up into several smaller blocks that are dependent.

Memory controllers that splits the banks by putting the bank number in the middle of the address often puts the segment number as the highest bits of the address instead of the banks. If so, a segment based PASR will work instead.

9 Conclusions

The PASR, originally proposed by Marc A. Viredaz and Deborah A. Wallach [24] at the Western Research Laboratory in Palo Alto, California, is today available in most memory hardware. The support in software, though, is still not widely spread. Conceptual implementations have been shown [4], but the Linux kernel for example lacks all support for PASR as of today.

The main thing to implement before PASR can be made possible is the unplugging algorithm. Besides that, anti-fragmentation and defragmentation is needed to get good results. There exists, though, some solutions to the anti-fragmentation and defragmentation today [11][20] and more work is done on the subject, with improvements coming every week [7].

Two alternative approaches were studied for the unplugging algorithm in this thesis, one where the memory to unplug is removed from the list of available memory using the Memory hotplug functions available in the kernel. The other where the system simply checks, atomically right before going into sleep mode, if the memory to unplug is free.

The later approach was selected and implemented using counters, one for each of the different regions that were to be powered down using PASR. The regions are represented by an array of objects that keeps track of the set of pageframe numbers that belongs to the region, how many pages within the region that are allocated and if the region is depending on any other region to be powered down. This approach is both time and space efficient. No time or memory space is spent on keeping track of what specific pages within the region that are allocated, just if some pages are. The check to see if a region is free is simply a matter of testing if the counter is zero or not.

The alternative to keep track of each and every page is about just as fast, but takes some more memory. To keep track of each page to know if it's allocated or not would take one bit per page of the system. That is totally unnecessary.

The evaluation shows that the overhead for keeping track of what regions are free is so negligible it can't be measured.

The feasibility of PASR is mainly depending on the type of PASR to use. Bank and segment selective PASR gives a high probability of finding some free regions to power down using PASR. Single ended PASR, on the other hand, is limited by the default memory layout of Linux, where user-space data is put separated from most kernel data. This separation is good in some cases, but prevents good results from Single ended Partial Array Self-refresh.

The memory controller has to be PASR compatible as well. Problems like memory banks being shattered over the full addressable area must be considered when choosing components.

10 Acknowledgements

References

- [1] gettimeofday. Linux Manpages.
- [2] linux/fs/proc/generic.c. Linux kernel source code.
- [3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, third edition, 2006.
- [4] Todd Brandt, Tonia Morris, and Khosro Darroudi. Analysis of the pasr standard and its usability in handheld operating systems such as linux. Technical report, Intel, 2007.
- [5] Jonathan Corbet. Driver porting: The seq_file interface. <http://lwn.net/Articles/22355/>, February 2003.
- [6] Jonathan Corbet. Memory compaction. *lwn.net*, Weekly Edition for January 7, 2010.
- [7] Mel Gorman. [patch 7/7] do not compact within a preferred zone after a compaction failure. LKML.
- [8] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.
- [9] Mel Gorman. [119/136] page-allocator: limit the number of migrate_reserve pageblocks per zone. LKML, October 2009.
- [10] Mel Gorman. [patch 08/20] move check for disabled anti-fragmentation out of fastpath. LKML, February 2009.
- [11] Mel Gorman. Memory compaction v1. Patch, January 2010.
- [12] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Proceedings of the Linux Symposium Volume One*, Ottawa, Ontario Canada, July 2006.
- [13] Dave Hansen, Mike Kravetz, Brad Christiansen, and Matt Tolentino. Hot-plug memory and the linux vm. In *Proceedings of the Linux Symposium*, volume One, July 2004.

- [14] JEDEC. *DDR3 SDRAM Specification*, jesd79-3b edition, September 2007.
- [15] JEDEC. *Low Power Double Data Rate (LPDDR) SDRAM Specification*, jesd209 edition, August 2007.
- [16] JEDEC. *DDR2 SDRAM Specification*, jesd79-2e edition, April 2008.
- [17] JEDEC. *Low Power Double Data Rate 2 (LPDDR2) SDRAM Specification*, jesd209-2 edition, March 2009.
- [18] M. Tim Jones. Access the linux kernel using the /proc filesystem. <http://www.ibm.com/developerworks/linux/library/l-proc.html>, March 2006.
- [19] Linux Kernel. [Linux/documentation/cpuidle/core.txt](http://www.linuxkernel.org/documentation/cpuidle/core.txt). Linux kernel Documentation.
- [20] Mel Gorman <mel@csn.ul.ie>. Create the zone_movable zone, July 2007.
- [21] Micron Technology. *Mobile Low-Power DDR SDRAM Datasheet*.
- [22] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. The linux kernel module programming guide. <http://www.tldp.org/LDP/lkmpg/2.6/html/x861.html>, May 2007.
- [23] Thomas Szepesi and Kin Shum. Cell phone power management requires small regulators with fast response. Planet Analog, January 2010.
- [24] Marc A. Viredaz and Deborah A. Wallach. Power evaluation of a handheld computer: A case study. Technical report, Western Research Laboratory, California, 2001.

Appendix A: Tools

A.1 Procfs

A tool, originally created to transfer information from the kernel to the user about processes, which has shown very useful, is procfs[18]. The procfs is a virtual file system, supported by the VFS[3] (Virtual File System). The VFS is a kernel layer, designed to give an abstraction layer between the operating system and the file system, enabling easy integration of different file systems.

Files in the /proc directory are not actual files on a physical storage media, but rather kernel functions that reads and writes data between user-space and kernel-space. When a user process queries the VFS for a file belonging to the procfs, the VFS in turn queries the function handling the specific file. That function, in turn, gathers the requested information and returns it to the user process.

Originally, procfs was designed to give the user information about processes but today it's used in a much more useful but inconsistent way. Today, the files in /proc are not restricted to give information about processes, but can give information about the system in total, hardware as well as software. A good example of this is the files found in /proc/acpi/. Those files give the user and user-space applications the possibility to read out and modify some basic

power management settings for the hardware, like available sleep modes and CPU clock downscaling.

There are a couple of other alternative virtual file systems inspired by `procfs`, which provides the same feature. The first that comes to mind is `sysfs`, a file system designed to hold information about the system devices and device drivers. It works much like `procfs` but is more structured when it comes to what goes where in the directory tree. The downside of `sysfs` is that it is more complicated than `procfs` to set up.

Another alternative is `debugfs`. A virtual file system that, as the name suggests, is intended for debugging and development. The reason that `procfs` were the one selected during this project is that the development were built from the `procfs` file `kpageflags` used in page-types described below.

```

PASR memory safety, fast count:
Region          Count
0                1575 / 2048
1                0 / 2048
2                0 / 4096
3                0 / 8192
4                3124 / 16384
Total           4699

PASR regions:
Region  Start pfn  End pfn  Total
0       0         2048    2048
1       2048      4096    2048
2       4096      8192    4096
3       8192     16384   8192
4     16384     32768  16384

Total count of page->_count, invalid pfn's included, slow count:
Region  page->_count
0       1575 / 2048
1       0 / 2048
2       0 / 4096
3       0 / 8192
4     3090 / 16384
Total   4665
Total, according to sumval2           4665
freeval = 13294

```

Figure 9: Pasr-info output

A.2 Pasr-info

One of the procs entries that has been developed and used during the project is `/proc/pasr-info`. The `/proc/pasr-info` entry provides information about the configured memory regions for PASR, the counter values and available memory in different parts of the system.

The output seen in figure 9 shows the PASR counters, the physical pages included in what region and the individual page descriptors values of `page->_count`. The `page->_count` value is supposed to tell if a page is allocated or not, but as seen in figure 9, there is a disparity in the fourth region between the amount of pages having `_count` as one or above and the value of the PASR counter. This disparity has been around as long as QEMU has been used and has had a static value depending on the available system memory. The reason for this has not been discovered during the project.

The `/proc/pasr-info` also shows the first pfn of all the buddy-lists. The output seen in figure 10 shows the first pfn in the buddy list of order 0 (i.e.

2⁰=1 page), zone normal and the migrate types found in figure 11 below. This is useable to see how fragmented the page allocator is at the moment.

Besides this, `/proc/pasr-info` displays some different values of used and unused memory, all to get a good overview of the utilized memory, collected at one spot.

Zone Normal	
Order 0	
Migratetype	First pfn
0	30773
1	29859
2	1562
3	36
4	4294963097

Figure 10: First pfn of each migratetype in zone normal

```
#define MIGRATE_UNMOVABLE      0
#define MIGRATE_RECLAIMABLE   1
#define MIGRATE_MOVABLE       2
#define MIGRATE_RESERVE       3
#define MIGRATE_ISOLATE       4 /* can't allocate from here */
```

Figure 11: Migrate types

A.3 Kpageflags

The `Kpageflags` `procs` entry gives a bitmask of flags for each page in the system memory. The file is intended for the tool `Page-types` found in `<Linux/Documentation/vm/page-types.c>`, but were during the project used for the tool `Page-display` described below. The bitmask were enhanced with three bits. Those bits tells the migrate type of the page, as described in figure 11.

A.4 Kpagefree

Another `procs` entry that has been developed during the project is `/proc/kpagefree`.

The purpose of `kpagefree` is to show what physical pages in the system that are allocated and which are free. This is later used by the tool `Page-display` to generate utilization graphs.

The information about what pages are allocated is generated from each page's counter `'_count'` found in `struct page`. When the counter is set to `-1`, the page is free. When it is set to `0`, one process is using it, and when it's larger than `0`, the page is shared between more than one process.

The entry was originally developed from the Kpageflags procfs entry. Due to problems during porting from User-Mode Linux to QEMU emulator environment, the original kpagefree produced an incorrect amount of data. The simplest solution to this problem was to rebuild the kpagefree using seq-file (see below).

A.5 Seq-file

To create a procfs entry that prints out a short string is pretty strait forward, but as the output grows larger, the problems arise. The Linux kernel has had numerous procfs entries through out the years with erroneous implementations. The main problem is to keep track of where in the output stream the user is trying to read. If the output fills the buffer used to transfer data between kernel-space and user-space, the buffer have to be emptied at user-space side before the printing on kernel-space side can continue. The kernel then has to keep track of where to continue. The same problem arises when a user-space application searches in the procfs entry or wants to read from a specific position.

This is, in the original procfs implementations done by providing an offset value as an argument when calling the function. There is also an argument, the pointer start, which is used to return where, within the buffer, the data is written. The problem here is that this pointer, through out the years, has been used for other purposes as well[2], which has made the procfs API very confusing.

A good solution, which was presented by Alexander Viro in the 2.6 kernel, is the seq_file interface[22][5]. The seq_file interface is a set of functions that makes it easy to create virtual files for procfs and other virtual file systems.

When using seq_file, stdio.h-like functions like seq_printf, seq_scanf, seq_putc and seq_getc are available for simple interfacing with user-space without worrying about offsets and sequential reads.

A.6 Page-display

Based on the page-types tool found in <Linux/Documentation/vm/page-types.c> and data produced by the procfs entry, a graphics generation tool was created. Page-display is an improved version of page-types, which, upon giving the flag -P at execution, not only opens the kpageflags procfs file, but also opens up the file kpagefree. It then uses information from both of these to generate a bitmap picture of XPM type.

The XPM file type is an open source bitmap format, often used in icons. The main reason for selecting XPM as output format is the simple construction. The XPM file is simply an array of strings, in plain C compatible format, with each pixel as a character. The characters can be freely selected, but has to be defined with a color code in the beginning of the file.

The picture generated is organized with one pixel for each physical page frame. The page frame numbers are ordered left to right, top to bottom, so that the first page frame is found in the top left corner, and the next is to its right.

In an early version of Page-display, the outputs were solely black and white. Black for occupied pages and white for free pages. In the current version, the

Kpageflags procs file is improved with three bits telling the current migrate-type of the page. This information is used to give colors to the picture. One color for each migrate-type and that color is then pale if the page is free and rich if the page is allocated.

A.7 Emulator

To be able to concentrate on the Linux kernel without having to worry about hardware problems and flashing, an emulator is preferred during early development.

A.7.1 User-mode Linux

One of the simpler emulated environments available is User-Mode Linux, or UML. UML is not a real hardware emulator but rather runs a Linux kernel as a single user-mode application. The user-mode Linux kernel then loads a root file system from a specified image file. The main advantages of UML are the extremely short compile- and boot times.

UML is built as an architecture, based on x86, that is selected during compilation. The drawback of this is that any coding done in UML is done in an x86-like architecture. When converting to target architecture, all code done in architecture specific code, has to be rewritten.

During development in UML, a problem with the memory was noticed. Even though the boot command specified a given memory size for the virtual machine, the system booted with different memory sizes every time. The reason for this seems to be that the host system allocates a varying amount of memory for the guest system when the guest is started.

A.7.2 QEMU

A more versatile and powerful solution is QEMU (Quick Emulator). QEMU is a real hardware emulator that can emulate a vast variety of CPU's; amongst x86 and ARM are two. It also provides a small amount of device models.

One feature of QEMU that were used during the project is the ability to stop the guest system at runtime and examine the emulated physical memory.

A.8 Fillmem

To gain as much usage as possible out of Page-display, the memory had to be used. Two small applications were written to allocate some memory in different ways, fillmem_malloc and fillmem_static.

Fillmem_malloc allocates 200 words of memory dynamically by using malloc.

Fillmem_static allocates 200 words of memory statically using an array.

A.9 Time_count

To measure the time that were spent on the tracking of occupied pages, a simple program were written. The program allocates two pages of memory in user-space using malloc, writes data to it and frees it again.

This is then repeated 10'000'000 times and the time spent on each allocation-writing-freeing cycle is measured using `gettimeofday[1]` and added up. In this way an average value is calculated for the time spent, that can be used to compare different kernel compilations.

A.10 LXR

A very useful tool when working with the Linux kernel is Linux Cross Reference, LXR. It is a tool that generates a webpage of the source code.

The webpage links objects together so that the definition and all occasions where the item is used are searchable. Variables as well as defines and functions are linked.

Pre-generated web pages of the mainline Linux source code is available at <http://lxr.linux.no/>. They were used to study the kernel source code during the project.

Appendix B: Graphics

In the figures 13 to 15 below, the rich colors are allocated pages and pale colors are free pages. Lowest addresses are located at top of the image, with increasing addresses from left to right, top to bottom.

- `MIGRATE_UNMOVABLE` is red
- `MIGRATE_RECLAIMABLE` is blue
- `MIGRATE_MOVABLE` is green
- `MIGRATE_RESERVE` is yellow
- `MIGRATE_ISOLATE` is black

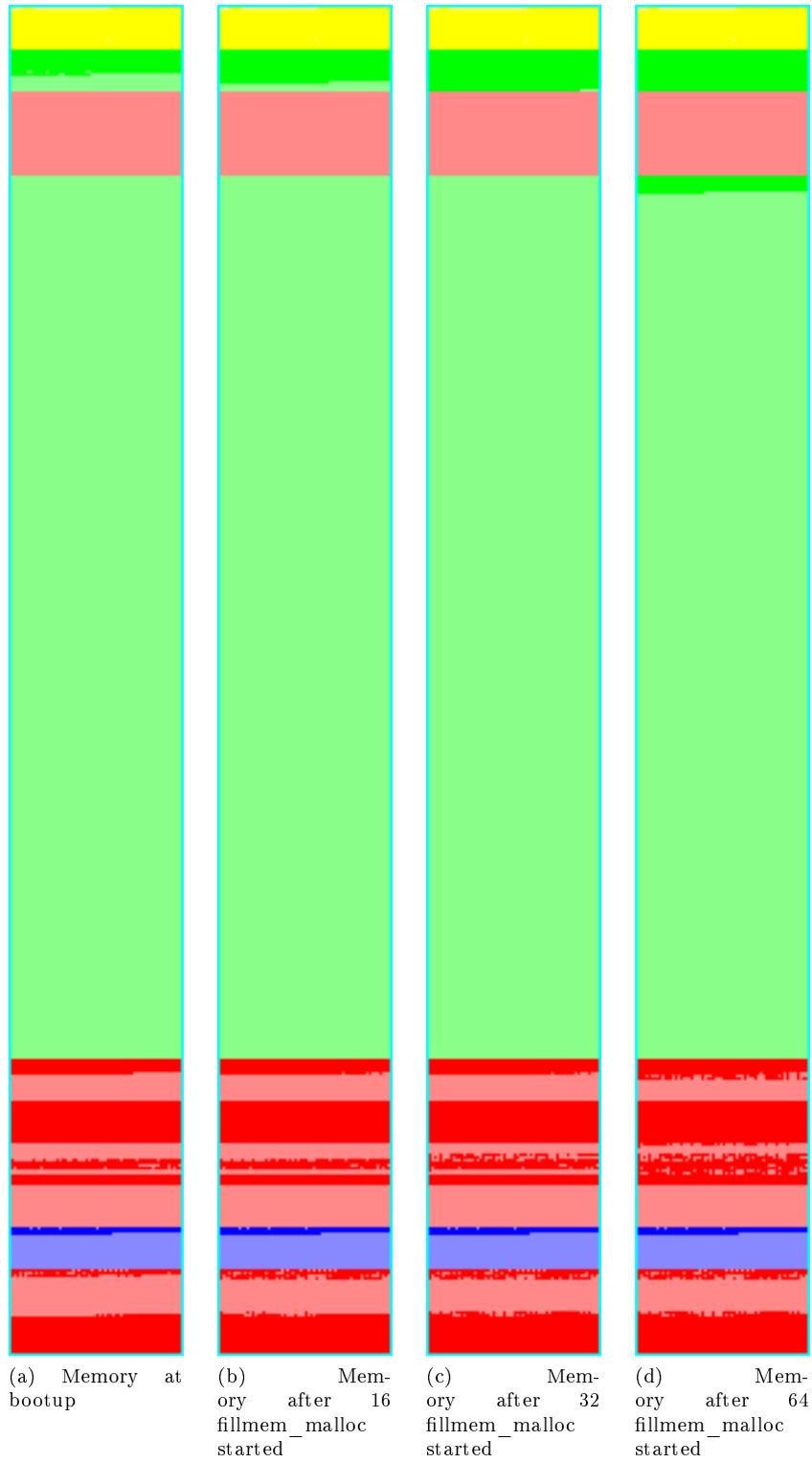


Figure 13: Memory from bootup to 1024 fillmem_malloc processes

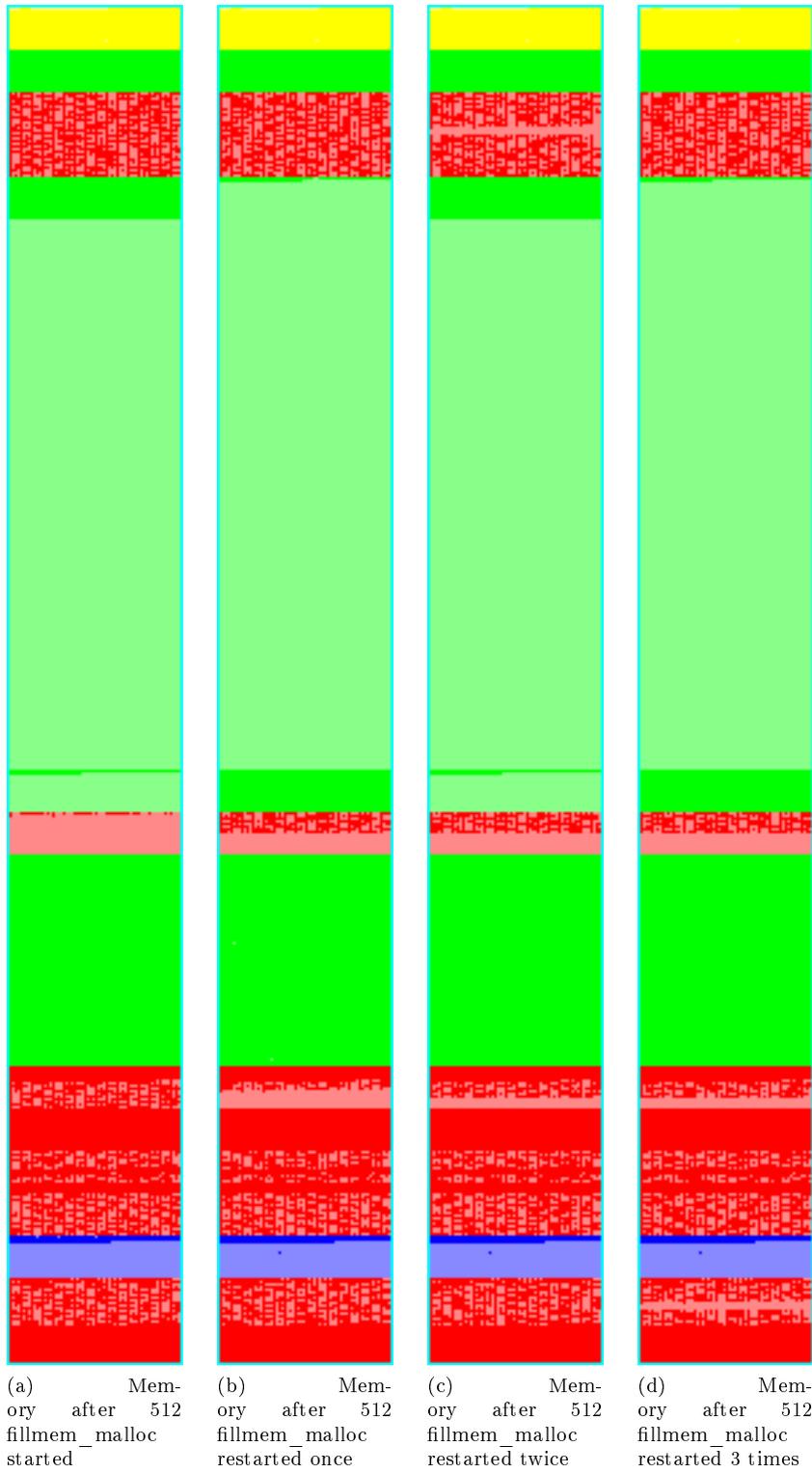


Figure 14: Reallocation of 512 fillmem_malloc processes

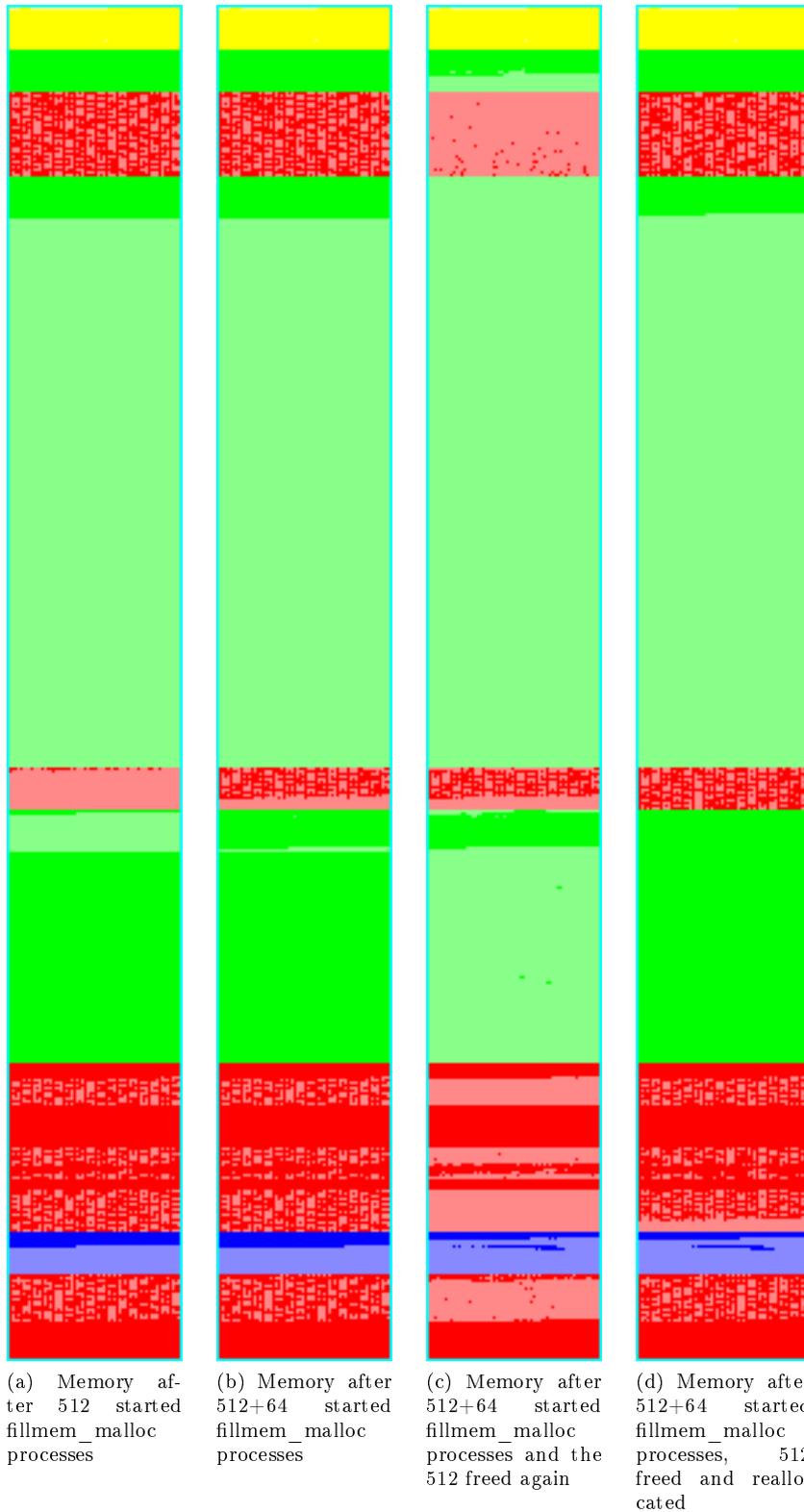


Figure 15: Reallocation with different numbers of fillmem_malloc processes