Transparent Superpages for FreeBSD on ARM

Zbigniew Bodek
Semihalf, The FreeBSD Project
zbb@{semihalf.com, freebsd.org}

Abstract

This paper covers recent work on providing transparent superpages support for the FreeBSD operating system on ARM. The concept of superpages mechanism is a virtual memory optimization, which allows for efficient use of the TLB translations, effectively reducing overhead related to the memory management. This technique can significantly increase system's performance at the interface between CPU and main memory, thus affecting its overall efficiency.
The primary goal of this work is to elaborate on how the superpages functionality has been implemented on the FreeBSD/arm and what are the results of its application. The paper presents real-life measurements and benchmarks performed on a modern, multiprocessor ARM platform. Actual performance achievements and areas of application are shown. Finally, the article summarizes the possibilities of future work and further improvements.

1 Introduction

ARM technology becomes more and more prevailing, not only in the mobile and embedded space. Contemporary ARM architecture (ARMv7 and the upcoming ARMv8) is already on a par with the traditional PC industry standards in terms of advanced CPU features like:

- MMU (with TLB)
- Multi-level Cache
- Multi-core
- Hardware coherency

Performance and scalability of the ARM-based machine is largely dependent of these functionalities. Majority of the modern ARM chips is capable of running complex software and handle multiple demanding tasks simultaneously. In fact, general purpose operating systems have become the default choice for these devices.
The operating system (*kernel*) is an essential component of many modern computer systems. The main goal of the kernel operations is to provide runtime environment for user applications and manage available hardware resources in an efficient and reasonable way. Memory handling is one of the top priority kernel services. Growing requirements of the contemporary applications result in a significant memory pressure and increasing access overhead. Performance impact related to the memory management is likely to be at the level of 30% up to 60% [1]. This can be a serious issue, especially for the system that operates under heavy load.

Today's ARM hardware is designed to improve handling of contemporary memory management challenges. The key to FreeBSD success on this architecture is a combination of sophisticated techniques that will allow to take full advantage of the hardware capabilities and hence, provide better performance in many applications. One of such techniques is transparent superpages mechanism.
Superpages mechanism is a virtual memory system feature, whose aim is to reduce memory access overhead by making a better use of the CPU's Memory Management Unit hardware capabilities. In particular, this mechanism provides runtime enlargement of the TLB (translation cache) coverage and results in less overhead

related to memory accesses. This technique had already been applied on i386 and amd64 architectures and brought excellent results.

FreeBSD incorporates verified and mature, high-level methods to handle superpages. Work presented in this paper introduces machine-dependent portion of the superpages support for ARMv6 and ARMv7 on the mentioned OS.

To summarize, in this paper the following contributions have been made:

- Problem analysis and explanation
- Introduction to possible problem solutions
- Implementation of the presented solution
- Validation (benchmarks and measurements)
- Code upstream to the mainline FreeBSD 10.0-CURRENT

The project was sponsored by Semihalf and The FreeBSD Foundation. The code is publicly available beginning with FreeBSD 10.0.

2 Problem Analysis

In a typical computer system, memory is divided into few, general levels:

- CPU cache
- DRAM (main memory)
- Non-volatile backing storage (Hard Drive, SSD, Flash memory)

Each level in the hierarchy has significantly greater capacity and lower cost per storage unit but also longer access time. This kind of design provides best compromise between speed, price and capabilities of the contemporary electronics. However, the same architecture poses a number of challenges for the memory management system.

User applications stored in the external, non-volatile memory need to be copied to the main memory so that CPU can access them. The operating system is expected to handle all physical memory allocations, segments transitions between DRAM and external storage as well as protection of the memory chunks belonging to the concurrently running jobs. Virtual memory system carries these tasks without any user intervention. The concept allows to implement various, favorable memory management techniques such as on-demand paging, copy-on-write, shared memory and other.

2.1 Virtual Memory

Processor core uses so called *Virtual Address* (VA) to refer to the particular memory location. Therefore, the set of addresses that are 'visible' to the CPU is often called a *Virtual Address Space*. On the other hand there is a real or *Physical Address Space* (PA) which can incorporate all system bus agents such as DRAM, SoC registers, I/O.

Virtual memory introduces additional layer of translation between those spaces, effectively separating them and providing artificial private work environment for each application. This mechanism, however, requires some portion of hardware support to operate. Most application processors incorporate special hardware entity for managing address translations called *Memory Management Unit* (MMU). Address translation is performed with the *page* granulation. Page defines VA$\longrightarrow$PA translation for a subset of addresses within that page. Hence, for each resident page in the VA space exists exactly one frame in the physical memory. For the CPU access to the virtual address to succeed MMU has to provide the valid translation to the corresponding physical frame. The translations are stored in the main memory in the form of virtually indexed arrays, so called *Translation Tables* or *Page Tables*.

To speed up the translation procedure *Memory Management Unit* maintains a table of recently used translations called *Translation Lookaside Buffer* (TLB).

2.1.1 TLB Translations

Access to the pages that still have their translations cached in the TLB is performed immediately and implies minimal overhead related to the access completion itself. Other scenarios result in a necessity to search for a proper translation in the Translation Tables (presented in the Figure 1) or, in case of failure, handling the time consuming exception. TLB is therefore in the critical path of every memory access and for that reason it is desired to be as fast as possible. In practice, TLBs are fully associative arrays of size limited to several dozens of entries. In addition, operating systems usually configure TLB entries to cover the smallest available page size so that dense page granulation, thus low memory fragmentation could be maintained. Mentioned factors form the concept of *TLB coverage*, which can be described as the amount of memory that can be accessed directly, without TLB miss. Another substantial TLB behavior can be observed during frequent, numerous accesses to different pages in the memory (such situation can occur when a large set of data is being computed). Because a lot of pages is being touched in the process, free TLB entries become occupied fast. In order to make room for subsequent translations some entries need to be evicted. TLB evictions are made according to the eviction algorithm which is implementation defined. However, regardless of the eviction algorithm, significant paging traffic can cause recently used translations to be evicted even though they will need to be restored in a moment. This phenomenon is called *TLB trashing*. It is associated directly with the TLB coverage factor and can seriously impact system's performance.

2.1.2 Constraints and opportunities

It is estimated that performance degradation caused by the TLB misses is at 30-60%.

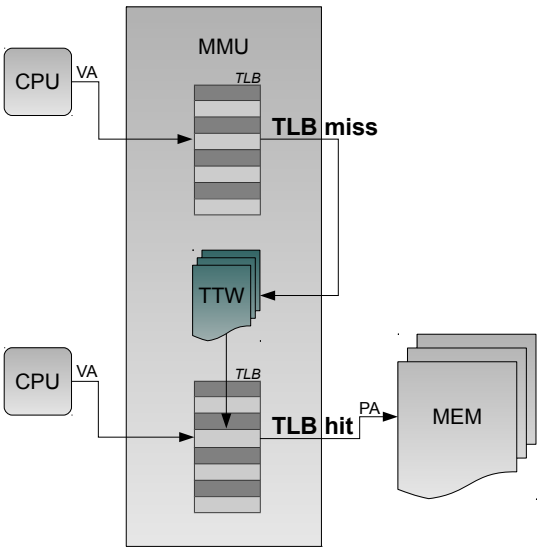

Figure 1: Memory access with TLB miss.

That is at least 20%, up to 50% more than in 1980's and 1990's [1]. TLB miss reduction is therefore expected to improve memory bandwidth and hence overall system performance, especially for resource-hungry processes. Reducing the number of TLB misses is equivalent to TLB coverage enhancement. Obvious solutions to achieve that would be to:

◦ Enlarge the TLB itself.
However, bigger translation cache means more logic, higher complexity and greater energy consumption that still may result in a little improvement. To sustain satisfying TLB characteristics with the currently available technologies, translation buffers can usually hold tens up to few hundreds of entries.

◦ Increase the base page size.
Majority of the microprocessor architectures support more than one page size. This gives the opportunity to cover larger memory areas consuming only a single entry in the TLB. However, this solution has a major drawback in the form of increased fragmentation and hence, inefficient memory utilization. The application may need to access very limited amount of memory but placed in a few, distinct locations. If the small pages were used as a base allocation

unit, less memory is reserved and more physical frames are available for other agents. On the other hand using superpages as a main allocation unit results in a rapid exhaustion of available memory for new allocations. In addition, single page descriptor contains only one set of access permissions and page attributes including *dirty* and *referenced* bits. For that reason, the whole dirty superpages needs to be written back to the external storage on page-out since there is no way to determine which fraction of the superpage has been actually written. This may cause serious disk traffic that can surpass the benefit from reducing TLB misses.

◦ Allow user to choose the page size.
In that case, the user would have to be aware of the memory layout and requirements of the running applications. That approach could be as much effective for some cases as it will be ineffective for any other. In fact, this method contradicts the idea of the virtual memory that should be a fully transparent layer.

2.1.3 Universal Solution

Reduction of the TLB miss factor has proven to be a complex task that requires support from both hardware and operating system sides. OS software is expected to provide low-latency methods for memory layout control, superpage allocation policy, efficient paging and more.

FreeBSD operating system offers the generic and machine independent framework for transparent superpages management. Superpages mechanism is a well elaborated technology on FreeBSD, which allow for runtime page size adjustment based on the actual needs of the running processes. This feature is already being successfully utilized on i386 and amd64 platforms. The observed memory performance boost for those architectures is at 30%. These promising numbers encouraged to apply superpages technique on another, recently popular ARM architecture. Modern ARM revisions (ARMv6, ARMv7 and upcoming ARMv8) are capable of using various page sizes allowing for superpages mechanism utilization.

3 Principles of Operation

Virtual memory system consists of two main components. The machine-independent VM manages the abstract entities such as address spaces, objects in the memory or software representations of the physical frames. The architecture-dependent `pmap(9)`, on the other hand, operates on the memory management hardware, page tables and all low-level structures. Superpages framework affects both aspects of the virtual memory system. Therefore, in order to illustrate the main principles of superpages mechanism, relevant VM operations are described. Then the specification of the Virtual Memory System Architecture (VMSA) introduced in ARMv6/v7-compliant processors is provided along with the opportunities to take advantage of the superpages technique on that architectures.

3.1 Reservation-based Allocation

VM uses `vm_page` structure to represent physical frame in the memory. In fact, the physical space is managed on page-by-page basis through this structure [2]. In the context of superpages, `vm_page` can be called the base page since it usually represents the smallest translation unit available (in most cases 4 KB page). Operating system needs to track the state and attributes of all resident pages in the memory. This knowledge is a necessity for a pager program to maintain an effective page replacement policy and decide which pages should be kept in the main memory and which ought to be discarded or written back to the external disk.

Files or any areas of anonymous memory are represented by virtual objects. `vm_object` stores the information about related `vm_pages` that are currently resident in the main memory, size of the area described by this object, pointer to shadow objects that hold private copies of modified pages and other information [3]. At system boot time, kernel detects the number of free pages in the memory and assigns them `vm_page` structures (except for pages occupied

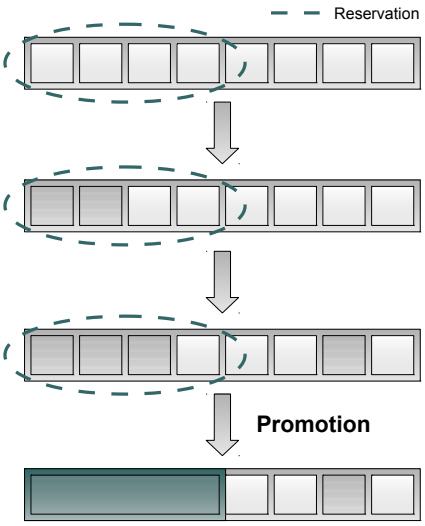

Figure 2: Basic overview of the reservation-based allocation.

by the kernel itself). When the processes begin to execute and touch memory areas they generate page faults since no pages from the free list have been filled with relevant contents and assigned to the corresponding object. This mechanism is a part of the on-demand paging and implies that only requested (and further utilized) pages of any object are cached in the main memory. Superpages technique relies on this virtual memory feature and is in a way its extension. When the reservation-based allocation is enabled (`VM_NRESERVLEVEL` set to non-zero value) and the referenced object is of superpage size or greater, VM will reserve a continuous physical area in memory for that object. This is justified by the fact that superpage mapping can translate a continuous range of virtual addresses to the range of physical addresses within a single memory frame. Pages within the created area are grouped in a population map. If the process that refers to the object will keep touching subsequent pages inside the allocated area, the population map will eventually get filled up. In that case, the related memory chunk will become a candidate for promotion to a superpage. The mechanism is briefly visualized in the Figure 2.

Not all reservations can be promoted even though the underlying pages satisfy the continuity requirements. That is because the single superpage translation has only one set of attributes and access permissions for the entire area covered by the mapping. Therefore, it is obvious that all base pages within the population map must be consistent in terms of all settings and state for promotion to succeed. In addition, superpages are preferred to be promoted read-only unless all base pages have already been modified and are marked 'dirty'. The intention is to avoid increased paging traffic to the disk. Since there is only one modification indicator for the whole superpage, there is no way to determine which portion of the corresponding memory has been actually written. Hence, the entire superpage area needs to be written back to the external storage. Demotion of the read-only superpage on write attempt is proven to be a more effective solution [1]. Summarizing, to allow for the superpage promotion, the following requirements must be met:

- The area under the superpage has to be continuous in both virtual and physical address spaces
- All base mappings within the superpage need to have identical attributes, state and access permissions

Not all reservations can always be completed. If the process is not using pages within the population map then the reservation is just holding free space for nothing. In that case VM can evict the reserved area in favor of another process. This proves that the superpages mechanism truly adapts to the current system needs as only active pages participate in the page promotion.

3.2 ARM VMSA

Virtual Memory System Architecture introduced in ARMv7 is an extension of the definition presented in ARMv6. Differences between those revisions are not relevant to

this work since backward compatibility with ARMv6 has to be preserved (ARMv6 and ARMv7 share the the same `pmap(9)` module).

ARMv6/v7-compliant processors use Virtual Addresses to describe a memory location in their 32-bit Virtual Address Space. If the CPU's Memory Management Unit is disabled, all Virtual Addresses refer directly to the corresponding locations in the Physical Address Space. However, when MMU is enabled, CPU needs additional information about which physical frame to access when some virtual address is used. Both, logical and physical address spaces are divided into chunks - pages and frames respectively. Appropriate translations are provided in form of memory resident Translation Tables. Single entry in the translation table can hold either invalid data that will cause Data/Prefetch abort on access, valid translation virtual⟶physical or pointer to the next level of translation. ARMv7 (without Large Physical Address Extension) defines two-level translation tables.

L1 table consists of 4096 word sized entries each of which can:

- Cause an abort exception
- Translate a 1 MB page to 1 MB physical frame (section mapping)
- Point to a second level translation table

In addition, a group of 16 L1 entries can translate a 16 MB chunk of virtual space using just one, supersection mapping.
L1 translation table occupies 16 KB of memory and needs to be aligned to that boundary.

L2 translation table incorporates 256 word sized entries that can:

- Cause an abort exception
- Provide mapping for a 4 KB page (small page)

Similarly to L1 entries, 16 L2 descriptors can be used to translate 64 KB large page by a single TLB entry. L2 translation table takes 1 KB of memory and has to be stored with the same alignment.

Recently used translations are cached in the unified TLB. Most of the modern ARM processors have additional, 'shadow' TLBs for instructions and data. These are designed to speed-up the translation process even more and are fully transparent to the programmer. Usually, TLBs in ARMv6/v7 CPUs can hold tens of entries so the momentary TLB coverage is rather small. An exceptional situation is when pages bigger than 4 KB are used.

3.2.1 Translation Process

When a TLB miss occurs MMU is expected to find a mapping for the referenced page. The process of fetching translations from page tables to TLB is called a Translation Table Walk (TTW) and on ARM it is performed by hardware.

For a short page descriptor format (LPAE disabled), translation table walk logic may need to access both L1 and L2 tables to acquire proper mapping. TTW starts with L1 page directory whose address in the memory is passed to the MMU via Translation Table Base Register (TTBR0/TTBR1). First, 12 most significant bits of the virtual address (VA[31:20]) are used as an index to the L1 translation table (page directory). If the L1 descriptor's encoding does not indicate otherwise the section (1 MB) or supersection (16 MB) mapping is inserted to the TLB and translation table walk is over. However, if L1 entry points to the L2 table then 8 subsequent bits of the virtual address (VA[19:12]) serve as an index to the destination L2 descriptor in that table. Finally the information from L2 entry can be used to insert small (4 KB) or large (64 KB) mapping to the TLB. Of course, invalid L1 or L2 descriptor format results in data or prefetch abort depending on the access type.

3.2.2 Page Table Entry

Both L1 and L2 page descriptors hold not only physical address and size for the related pages but also a set of encoded attributes that can define access permissions, memory type, cache mode and other. Page descriptor format is programmable to some extent, depending on enabled features and overall CPU/MMU settings (access permissions model, type extension, etc.). In general, every aspect of any memory access is fully described by the page table entry. This also indicates that any attempt to reference a page in a different manner than allowed will cause an exception.

4 Superpages Implementation for ARM

The paragraph elaborates on how the superpages mechanism has been implemented and operates on ARM. Main modifications to the virtual memory system have been described along with the explanation of the applied solutions.

4.1 Superpage size selection

First step to support superpages on a new architecture is to perform VM parameters tuning. In particular, reservation-based allocation needs to be enabled and configured according to the chosen superpages sizes.

Machine independent layer requires two parameters declared in `sys/arm/include/vmparam.h`:

- `VM_NRESERVLEVEL` - specifies a number of promotion levels enabled for the architecture. Effectively this indicates how many superpage sizes are used simultaneously.
- `VM_LEVEL_{X}_ORDER` - for each reservation level this parameter determines how many base pages fully populate the related reservation level.

At this stage a decision regarding supported superpage sizes had to be made. 1 MB section mapping has been chosen for a superpage whereas 4 KB small mapping has remained a base page. This approach has a twofold advantage:

1. Shorter translation table walk when TLB miss on the area covered by a section mapping.

   In that scenario, TTW penalty will be limited to one memory access only (L1 table) instead of two (L1 and L2 tables).

2. Better comparison with other architectures.

   i386 and amd64 can operate on just one superpage size of 2/4 MB. Similar performance impact was expected when using complementary page sizes on ARM.

Summarizing, VM parameters have been configured as follows:

`VM_NRESERVLEVEL` set to 1 - indicates one reservation level and therefore one superpage size in use.
`VM_LEVEL_0_ORDER` set to 8 - level 0 reservation consists of 256 (1 « 8) base pages.

4.2 `pmap(9)` extensions

The core part of the machine dependent portion of superpages support is focused on the `pmap` module. From a high-level point of view, VM "informs" lower layer when the particular reservation is fully populated. This event implies a chance to promote a range of mappings to a superpage but promotion itself still may not succeed for various reasons. There are no explicit directives from VM that would influence superpages management. `pmap` module is therefore expected to handle:

- promotion of base pages to a superpage
- explicit superpage creation
- superpage demotion
- superpage removal

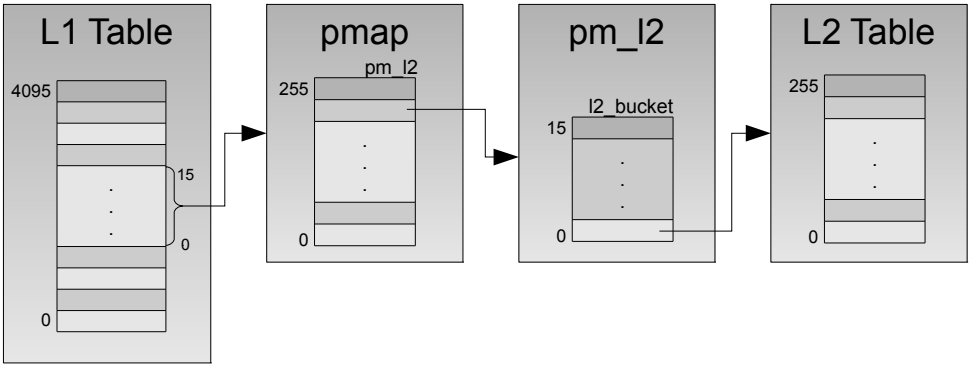

Figure 3: Page tables and kernel structures organization.

4.2.1 Basic Concepts

`pmap(9)` module is responsible for managing real mappings that are recognizable by the MMU hardware. In addition it has to control the state of all physical maps and pass relevant bits to the VM. Main module file is located at `sys/arm/arm/pmap-v6.c` and is supplemented by the appropriate structure definitions from `sys/arm/include/pmap.h`. Core structure representing physical map is `struct pmap`.

During virtual memory system initialization `pmap` module allocates one L1 translation table for each fifteen user processes out of maximum pool of *maxproc*. L1 entries sharing can be achieved by marking all L1 descriptors with the appropriate domain ID. Architecture defines 16 domains of which 15 are used for user processes and one is reserved for the kernel. This design can reduce KVM occupancy as each L1 table requires 16 KB of memory which is never freed. Each `pmap` structure holds `pm_l1` pointer to the corresponding L1 translation table meta-data (`l1_ttable`) which provides table's physical address to move to the `TTBR` on context switch as well as other information used to allocate and free L1 table on process creation and exit.

Figure 3 shows the page tables organization and their relation with the corresponding kernel structures. L1 page table entry points to the L2 table which collects up to 256 L2 descriptors. Each L2 entry can map 4 KB of memory. L2 table is allocated on demand and can be freed when unused. This technique effectively saves 1 KB of KVA per each unused L2 table.
`pmap`'s L2 management is performed via `pm_l2` array of type `struct l2_dtable`. Each of `pm_l2` fields holds enough L2 descriptors to cover 16 MB of data. Hence, for each 16 L1 table entries, exists one `pm_l2` entry. `l2_dtable` structure incorporates 16 elements of type `struct l2_bucket` each of which describes single L2 table in memory. In the current `pmap-v6.c` implementation, both `l2_dtable` and L2 translation table are allocated in runtime using `UMA(9)` zone allocator. `l2_occupancy` and `l2b_occupancy` track the number of allocated buckets and L2 descriptors accordingly. `l2_bucket` can be deallocated if none of 256 L2 entries within the L2 table is in use. Similarly, `l2_dtable` can be freed as soon as all 16 `l2_buckets` within the structure are deallocated.

Additional challenge for the `pmap` module is to track multiple mappings of the same physical page. Different mappings can have different states even if they point to the same physical frame. When modifying physical layout (page-out, etc.) it is necessary to take into account wired, dirty and other attributes of all pages related to a particular physical frame. The described functionality is provided by us-

ing pv_entry structures organized in chunks and maintained for each pmap in the system. When a new mapping is created for any pmap, the corresponding pv_entry is allocated and put into the PV list of the related vm_page.

Superpages support required to provide extensions for the mentioned mechanisms and techniques. Apart from implementing routines for explicit superpage management the objective was to make the existing code superpages aware.

4.2.2 Promotion to a Superpage

The decision whether to attempt promotion is based on two main conditions:

- vm_reserv_level_iffullpop() - indicates that physical reservation map is fully populated
- l2b_occupancy - implies that (aligned) virtual region of superpage size is fully mapped using base pages

Both events will most likely occur during new mapping insertion to the address space of the process. Therefore the promotion attempt is performed right after successful pmap_enter() call.

The page promotion routine (pmap_promote_section()) starts with the preliminary classification of the page table entries within the potential superpage. At this point the decision had to be made which pages to promote and which of them should be excluded from the promotion. In the presented implementation, promotion to a superpage is discontinued for the following cases:

- *VA belongs to a vectors page*
  Access to a page containing exception vectors must never abort and should be excluded from any kind of manipulation for safety reasons. Every abort in this case would result in nested exception and fatal system error.
- *Page is not under PV management*
  With Type Extension (TEX) disabled, page table entry has not enough room to store all the necessary status bits. For that reason pv_flags field from the pv_entry structure holds the additional data including bits relevant for the promotion to a superpage.
- *Mapping is within the kernel address space*
  On ARM, kernel pages are already mapped using as much section mappings as possible. The mappings are then replicated in each pmap.

Page table entry in the L2 under promotion is also tested for reference and modification bits as well as permission to write. Superpage is preferred to be a read-only mapping to avoid expensive, superpage-size transitions to a disk on page-out. Therefore it is convenient to clear the permission to write for a base page if it has not been marked dirty already. All of the mentioned tests apply to the first base page descriptor in the set. This approach can reduce overhead related to the unsuccessful promotion attempt since it allows to quickly disregard invalid mappings and exit. However if the first descriptor is suitable for the promotion then the remaining 255 entries from the L2 table still need to be checked

Apart from the above mentioned criteria the area under superpage must satisfy the following conditions:

1. *Continuity in the VA space*
2. *Continuity in the PA space*
   Physical addresses stored in the subsequent L2 descriptors must differ by the size of the base page (4 KB).
3. *Consistency of the pages' attributes and states*

When all requirements are met then it is possible to create single 1 MB section mapping for a given area. It is important that during promotion process L2 table zone is not being deallocated. Corresponding `l2_bucket` is rather stashed to speed-up the superpage demotion in the future.

The actual page promotion can be divided into two stages:

- `pmap_pv_promote_section()`
  At this point `pv_entry` related to the first `vm_page` in a superpage is moved to another list of PV associated with the 1 MB physical frame. The remaining PV entries can be deallocated.
- `pmap_map_section()`
  The routine constructs the final section mapping and inserts it to the L1 page descriptor. Mapping attributes, access permissions and cache mode are identical with all the base pages.

Successful promotion ends with the TLB invalidation which flushes old translations and allows MMU to put newly created superpage to the TLB.

4.2.3 Explicit Superpage Creation

Incremental reservation map population is not always a necessity. In case of a mapping insertion for the entire virtual object it is possible to determine the object's size and its physical alignment. The described situation can take place when `pmap_enter_object()` is called. If the object is at least of superpage size and VM has performed the proper alignment it is possible to explicitly map the object using section mappings.

`pmap_enter_section()` has been implemented to create a direct superpage mappings. The routine has to perform preliminary page classification similar to the one in `pmap_promote_section()`. This time however, it is not necessary to check any of the base pages within the potential superpage since they do not exist yet. Bits that still need to be tested are:

- *PV management status*
- *L1 descriptor status*
  The given L1 descriptor cannot be used for a section mapping if it is already a valid section or it is already serving as a page directory for a L2 table.

Direct insertion of the mapping involves a necessity to allocate new `pv_entry` for a 1 MB frame. This task is performed by `pmap_pv_insert_section()` which may not succeed. In case of failure the superpage cannot be mapped, otherwise section mapping is created immediately.

4.2.4 Superpage Demotion and Removal

When there is a need to page-out or modify one of the base pages within the superpage it is required to destroy a corresponding section mapping. Lack of any mapping for a memory region that is currently in use would cause a chain of expensive `vm_fault()` calls. Demotion procedure (`pmap_demote_section()`) is designed to overcome this issue by recreating L2 translation table in place of the removed L1 section.

There are two possible scenarios of the superpage demotion:

1. Demotion of the page created as a result of promotion.
   In that case it is possible to reuse the already allocated `l2_bucket` that has been stashed after the promotion. This scenario has got two major advantages:
   - No need for any memory allocation for L2 directory and L2 table.
   - If the superpage attributes have not changed then there is no need to modify or fill the L2 descriptors

2. Demotion of the page that was directly inserted as a superpage.
   This implies that there is no stashed L2 table and it needs to be allocated and created from scratch. Any allocation failure results in an immediate exit due to speed restrictions. Sleeping is not an option.

The demotion routine has to check if the superpage has exactly the same attributes and status bits as the stashed (or newly created) L2 table entries. If not then the L2 entries need to be recreated using current L1 descriptor. PV entries also need to be allocated and recreated using `pv_entry` linked with the 1 MB page. Finally when the L2 table is in place again, the L1 section mapping can be fixed-up with the proper L1 page directory entry and the corresponding translation in the TLB ought to be flushed.

The last function used for superpage deletion is `pmap_remove_section()`. It is used to completely unmap any given section mapping. Calling this function can speed-up `pmap_remove()` routine if the removed area is mapped with a superpage and the size of the space to unmap is at least of superpage size.

4.2.5 Configuration and control

At the time when this work is written, superpages support is disabled by default in `pmap-v6.c`. It can be enabled in runtime during system boot by setting a loader variable:

```
vm.pmap.sp_enabled=1
```

in `loader.conf` or it can be turned on during compilation time by setting:

```
sp_enabled
```

variable from `sys/arm/arm/pmap-v6.c` to a non-zero value.

System statistics related to the superpages utilization can be displayed by invoking:

```
sysctl vm.pmap
```

command in the terminal. The exemplary output can be seen below:

```
vm.pmap.sp_enabled: 1
vm.pmap.section.demotions: 258
vm.pmap.section.mappings: 0
vm.pmap.section.p_failures: 301
vm.pmap.section.promotions: 1037
```

`demotions` – number of demoted superpages
`mappings` – explicit superpage mappings
`p_failures` – promotion attempts that failed
`promotions` – number of successful promotions

5 Results and benchmarks

The functionality has been extensively tested using various benchmarks and techniques. The performance improvement depends to a large extent on the application behavior, usage scenarios and amount of available memory in the system. Processes allocating large areas of consistent memory or operating on big sets of data will benefit more from superpages than those using small, independent chunks.

Presented measurements and benchmarks have been performed on Marvell Armada XP (quad-core ARMv7-compliant chip).

5.1 GUPS

The most significant results can be observed using the *Giga Updates Per Second* (GUPS) benchmark. GUPS measures how frequently system can issue updates to randomly generated memory locations. In particular it measures both memory latency and bandwidth. On multi-core ARMv7 platform, measured CPU time usage and real time duration dropped by 34%. Number of updates performed in the same amount of time has increased by 52%.

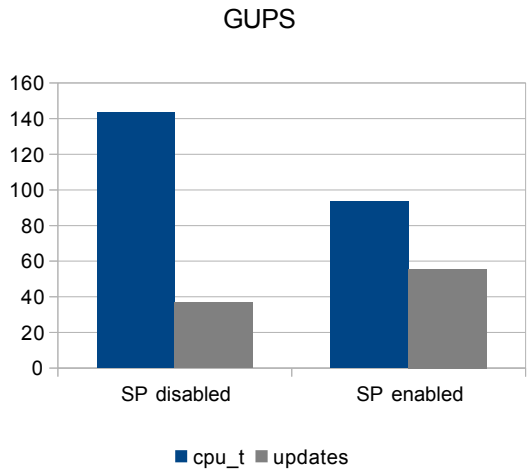

Figure 4: GUPS results.CPU time used [s], number of updates performed [100000/s].

5.2 LMbench

LMbench is a popular suite of system performance benchmarks. It is equipped with the memory testing program and can be used to examine memory latency and bandwidth. Measured memory latency has dropped by 37,85% with *superpages* enabled. Memory bandwidth improvement varied depending on the type of operation and was in the range from 2,26% for `mmap` reread to 8,44% for memory write. It is worth noting that *LMbench* uses *STREAM* benchmark to measure memory bandwidth which uses floating point arithmetic to perform the operations on memory. Currently FreeBSD does not yet support FPU on ARM what had a negative impact on the results.

Mmap reread [MB/s]	*Bcopy (libc) [MB/s]*	*Bcopy (hand) [MB/s]*	*superpages*
645,4	305,4	432,3	
660,0	312,4	446,9	✓

Table 1: LMbench. Memory bandwidth measured on various system calls.

Mem read [MB/s]	*Mem write [MB/s]*	*Mem latency [ns]*	*superpages*
681	3043	238,8	
696	3300	148,4	✓

Table 2: LMbench. Memory bandwidth and latency measured on memory operations.

The results summary is shown in Tables 1 and 2. Table 3 on the other hand shows the the percentage improvement of the parameters with the best test results.

Mem write %	*Rand mem latency %*
8,44	37,85

Table 3: LMbench. Percentage improvement of the selected parameters.

5.3 Self-hosted world build

Using superpages helped to reduce self-hosted world build when using GCC. The results are summarized in Table 4. The time needed for building the whole set of user applications comprising to the root file system has dropped by 1 hour 22 minutes (20% shorter). No significant change has been noted when using CLANG.

GCC	*CLANG*	*superpages*
6h 36min	6h 16min	
5h 14min	6h 15min	✓

Table 4: Self-hosted `make buildworld` completion time.

5.4 Memory stress tests

Presented functionality has been also tested in terms of overall stability and reliability. For that purpose two popular stress benchmarks have been used:

- *forkbomb:* `forkbomb -M`
  Application can allocate entire available memory using `realloc()` and access this memory.

- *stress:* `stress -vm 4 -vm-bytes 400M` Benchmark imposes certain types of compute stress on the system. In this case 4 processes were spinning on `malloc()`/`free()` calls, each of which working on 400 MB of memory.

No anomalies or instabilities were detected even during long runs.

6 Future work

The presented functionality has significant impact on system's performance but does not cover all of the hardware and OS capabilities. There are possible ways of improvement.

Adding support for additional 64 KB page size will further increase the amount of created superpages, enabling a smoother and more efficient process for the promotion from 4 KB small page to 1 MB section. In addition, a larger number of processes will be capable of taking advantage from superpages if the required population map size is smaller.

In addition, current `pmap(9)` implementation uses PV entries to store some information about the mapping type and status. This implies the necessity to search through PV lists on each promotion attempt. `TEX` (Type Extension) support would allow to move those additional bits to the page table entry descriptors and lead to reduction of the promotion failure penalty.

7 Conclusions

Presented work has brought the transparent superpages support to the ARM architecture on FreeBSD. The paper described virtual memory system from both OS and hardware points of view. System's bottle-necks and design constrains have been carefully described. In particular the work has elaborated on the TLB miss penalty and its influence on the overall system performance.

Mechanisms implemented during the project met their objectives and provided performance gain on the interface between CPU and memory. This statement has been supported by various tests and benchmarks performed on a real ARM hardware. Test results vary between different benchmarks but improvement can be observed in all cases and is at 20%.

Introduced superpages support has been committed to the official FreeBSD SVN repository and is available starting from revision 254918.

8 Acknowledgments

Special thanks go to the following people:

Grzegorz Bernacki and Alan Cox, for all the help and mentorship.
Rafał Jaworowski, mentor of this project.

Work on this project was sponsored by Semihalf and The FreeBSD Foundation.

9 Availability

The support has been integrated into the mainline FreeBSD 10.0-CURRENT and is available with the FreeBSD 10.0-RELEASE. The code can also be downloaded from the FreeBSD main SVN repository.

References

[1] Juan E. Navarro, *Transparent operating system support for superpages*, 2004

[2] The FreeBSD Documentation Project, *FreeBSD Architecture Handbook*, 2000-2006, 2012-2013

[3] Marshall Kirk McKusick, *The Design and Implementation of the FreeBSD Operating System*, 2004