MTD Based Compressed Swapping for Embedded Linux.

Alexander Belyakov, alexander.belyakov@intel.com

http://mtd-mods.wiki.sourceforge.net/MTD+Based+Compressed+Swapping

Introduction and Motivation

Memory is a critical resource in many embedded systems. Increasing memory often increases packaging and cooling costs, size, and energy consumption. The total RAM requirements for applications in the mobile phone market are doubling or even tripling each year [1]. As embedded systems support new applications, their working data sets often increase in size, exceeding original estimates of memory requirements. Significant hardware redesign may increase time-to-market and design costs.

The flash memory device is a critical component in building embedded systems because of its non-volatility, shock-resistant, and power economic nature. A typical embedded multimedia system such as high-end cellular phone usually contains DRAM, NOR flash, and NAND flash memory devices, where DRAM is used as a working memory, NOR flash as code storage and NAND for non-volatile data storage.

The basic idea of this work is to enable flash memory as swap area to store compressed unused memory pages. The key requirement is solution simplicity without hardware redesign and significant Linux kernel changes.

What are the key advantages of having compressed swap on top of flash memory? *First* – more virtual memory becomes available for applications with the same amount of RAM. *Second* – more memory becomes available for file-cache increasing file-backed application performance. *Third* – power consumption becomes lower as we use power economic flash instead of RAM part. *Next* – whole design cost may become lower because flash is usually considered to be cheaper than DRAM and we need less space because of compression. *Next* point seems to be quite debatable. There is an opinion among kernel developers that swap *must* be enabled in Linux, because “even for systems that don't need the extra memory space, swap can actually provide performance improvements by allowing unused memory to be replaced with often-used memory. It is a magical property of swap space, because extra RAM doesn't allow you to replace unused memory with often used memory. The theory holds true no matter how much RAM you have.” [2]

The price for all these nice things is performance degradation, because flash is slower than RAM and because compression eats CPU time. Another flash media type related issue is erase cycles required for generic NOR and NAND parts. There is no way to rewrite data without erasing whole eraseblock which is most likely much larger than kernel page size. And the last issue to mention is flash wear out problem. There is very limited number of write-erase cycles allowed for each eraseblock. Normally it is about 10^5 cycles.

Media types

We used three types of underlying media to evaluate compressed swap solution.

NAND

Swapping to NAND is quite difficult because all the flash issues mentioned earlier holds true. To handle explicit erase requirements and wear out issue it is necessary to implement caching layer or use special sector manager with garbage collector, bad block manager and wear leveling algorithms. As an

option one may use swapfile on top of existing NAND filesystem which supports swapping. But even having all this stuff might not be enough, because swapping subsystem still writes too much and wears out flash too quickly.

PCM

PCM is phase change memory. It is relatively new flash type which has NOR-like interface. The most important new features of PCM from swapping point of view are high cycling endurance (up to 10^8 rewrite cycles), bit alterable writes, and quite promising performance. Bit alterable writes make swapping very simple eliminating all erase related stuff.

RAM

The most simple media is RAM itself. Not much to say here – we just allocate memory and put compressed swapped pages there. Key benefits are extra virtual memory with minor performance impact and without hardware design changes.

Swapping to RAM is well known technique and there are several already working solutions. The one of most recent is compressed caching for Linux solution [8]. It seems compcache idea is quite similar to ours if we are speaking about swapping to RAM. One may consider our solution as a kind superset for compcache solution.

Related works

The idea of memory compression is not new. Early techniques for reducing the RAM requirements of embedded systems were mostly hardware-based. For example, there was an idea to insert a hardware compressor/decompressor unit between the cache and RAM [3, 4]. These solutions require significant hardware changes and cannot be easily introduced into existing embedded designs.

Swap compression itself is also known technique [5-7]. For example, researches from Spain [5] developed and tested mechanism of compressing swap pages and keeping them in a swap cache whenever possible. Depending on workload applications and data compression ratio they got benchmark speedup between 1.2 and 2.1. Their approach changes Linux virtual memory management subsystem internals and relays on hard-disk enabled systems.

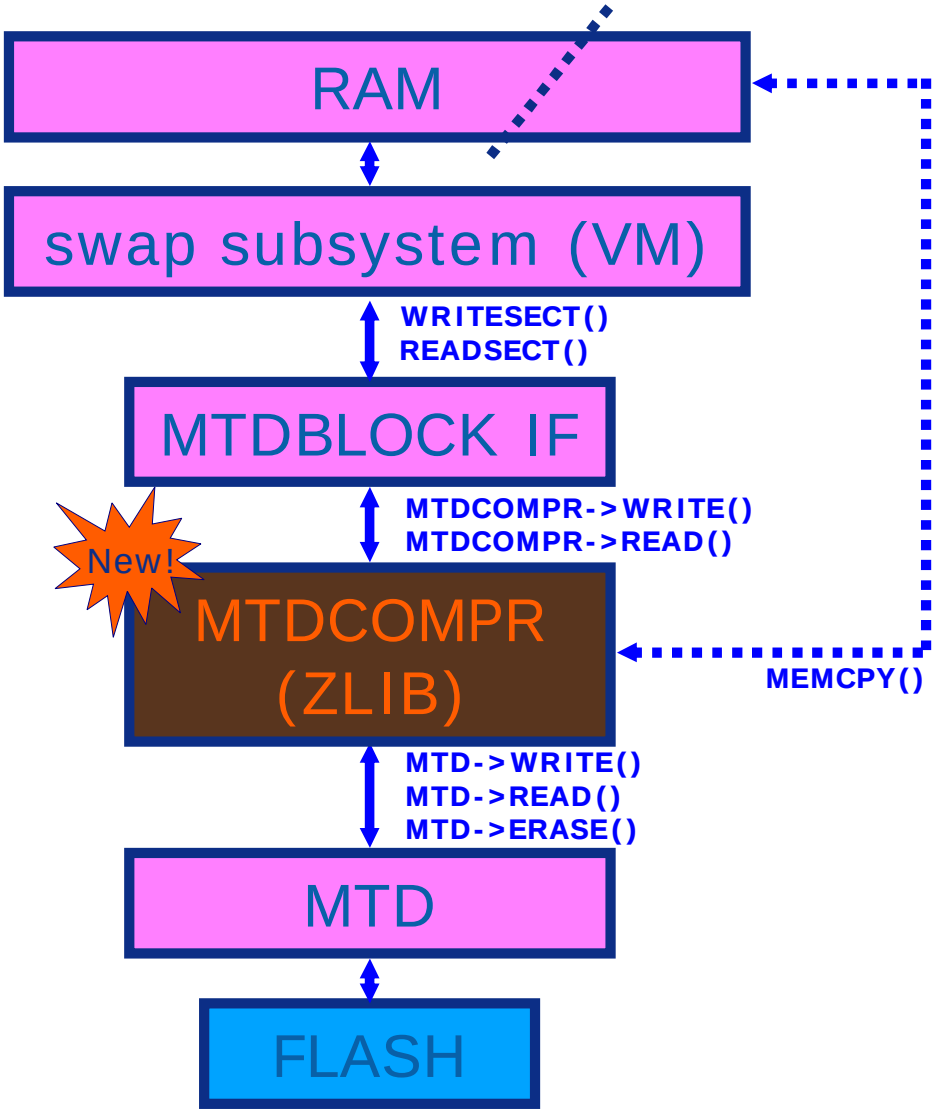
Another related work [6] deals with compression of swapped pages and storing them into the RAM device. Their prototype divided the RAM into two portions: one containing compressed data pages and the other containing uncompressed data pages as well as code pages. This solution was tested by the authors on disk-less embedded systems. Experimental results indicated that solution is capable of doubling the amount of RAM available for applications with negligible performance of power consumption overhead.

From very high level point of view our solution is quite similar to this one (as well as to most recent compcache solution), except we are able to keep compressed swapped pages not only in RAM, but also on flash.

Another related work [7] is trying to keep compressed swapped pages on NAND flash memory. The authors created swap file on YAFFS partition and changed Linux virtual memory management subsystem swapping algorithm. Developed swapping algorithm is based on CFLRU (Clean First LRU) method and employs some additional features such as selective compression and delayed swapping. This algorithm helps enhancing the stability of filesystem by reducing number of writes. Experimental results show that proposed solution can reduce number of writes by 50-70% without affecting the system performance much. Reduced DRAM size helps in lowering the system implementation cost as well as power consumption.

MTD compression layer (prototype)

The goal of this work was to lower RAM requirements of embedded systems through enabling effective swapping to flash with minor software changes. The basic idea was to create MTD (memory technology device) compression layer and use it as swap area via mtdblock interface. We called this new layer MTDCOMPR just to give an idea of what is this layer for. MTDCOMPR creates new MTD device on top of existing one. MTD flag for the new device is always set to MTD_NO_ERASE, so mtdblock itself doesn't perform caching or flash erase. The only function of mtdblock interface is sector read/write operations translation to mtdcompr read/write calls.

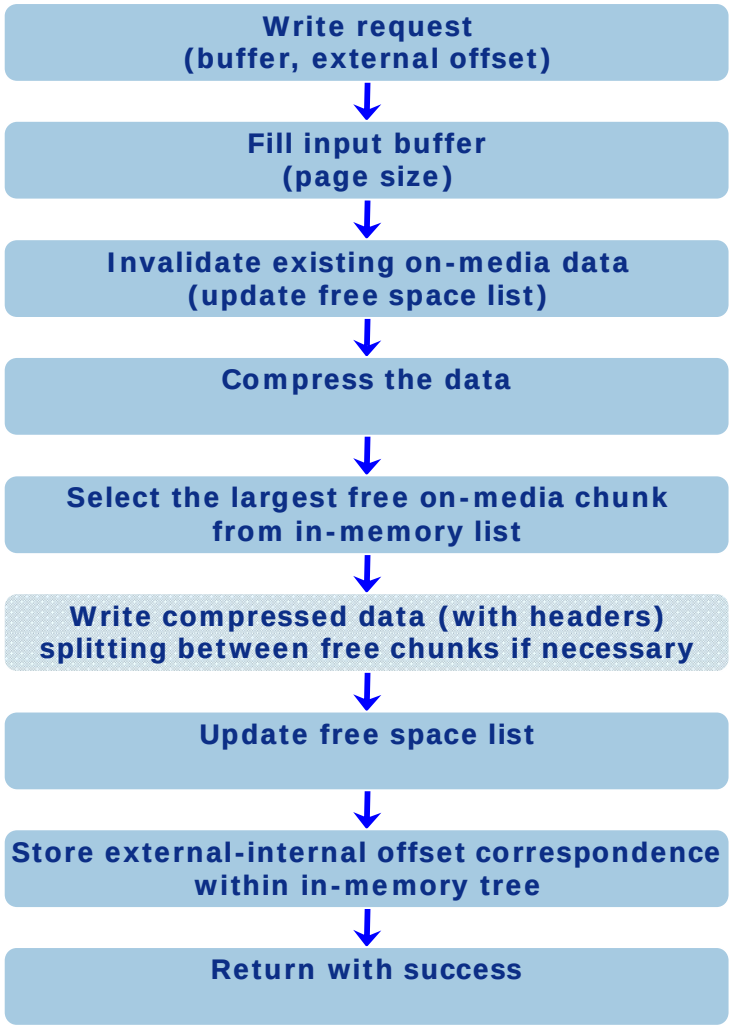


We picked kernel zlib library as a main swapped pages compressor. Experimenting with MTDCOMPR on desktop PC running standard GUI enabled applications, it was found that swapped pages have less than 25% compression ratio with best speed compression method (i.e. pages require four times less memory if compressed). The same results have been achieved by researched worked earlier with RAM compression techniques [6].

MTDCOMPR data flow

Writing to swap

Assume virtual memory subsystem decided to swap out single page. Write request falls to MTDCOMPR via mtdblock interface. Using MTDCOMPR exclusively for swapping allows us to use kernel page sized buffer. By default mtdblock splits all incoming requests by 512 bytes chunks, so MTDCOMPR waits until whole page is in buffer.



Next step is to seek if page with such external offset has been previously swapped out and writted to the media. If such a page does exist we invalidate the old one by marking flash space it holds as free. MTDCOMPR keeps in-memory sorted list with flash free space fragments information.

Now we are ready to compress data. Do it.

Next step is to pick the largest free space fragment from in-memory list to write compressed data there. Such a rotation creates a kind of so called random wear leveling where pages with the same external offset is stored by different media offsets all the time.

Before actual writing we split compressed data into chunks of free space fragment size if needed and apply headers. Then perform writing.

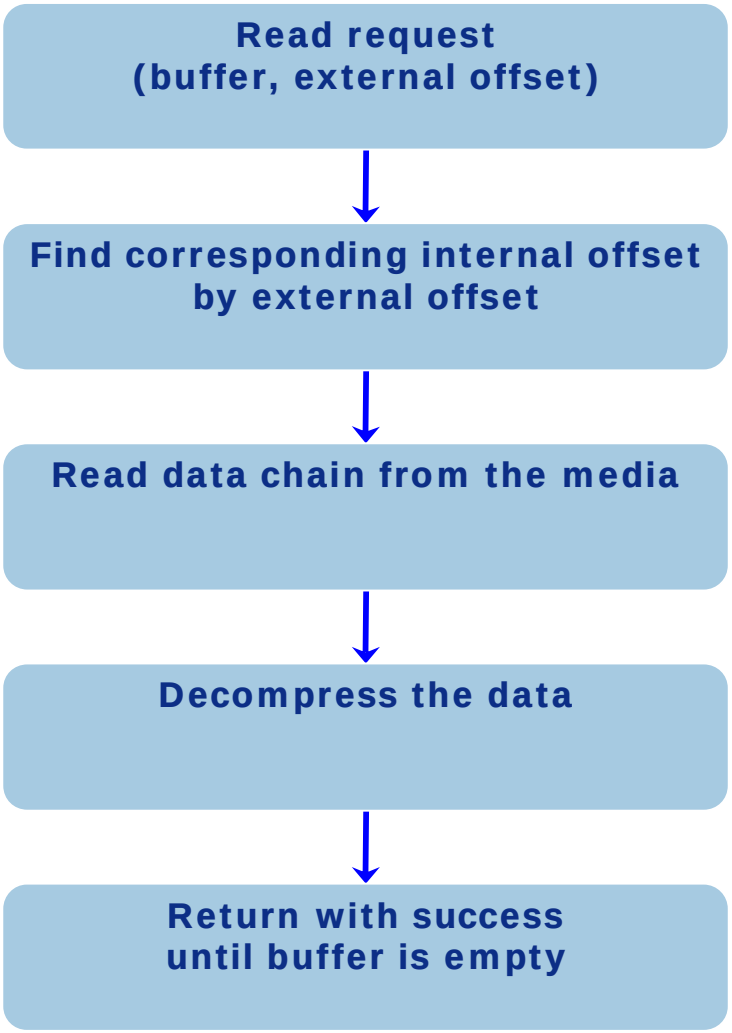
There is one important note. Such writing algorithm is fine for RAM and PCM, but not for NAND. Because NAND flash requires explicit erase before rewriting the data. So MTDCOMPR has optional caching layer similar to one of mtdblock. It is quite simple and thus brings significant performance overhead because writing small piece of data may turn into read-erase-write sequence altering whole sector (eraseblock). Actually to make swapping on NAND viable it is necessary to use sector manager or FTL (flash translation layer) with garbage collector, bad block manager and wear leveling stuff. Another option is to use one of existing NAND filesystem like YAFFS. Anyway using NAND as swap requires a lot of things to think about including virtual memory manager changes lowering number of writes [7].

If writing succeeds we update free space list and store external-internal page offset correspondence within in-memory tree.

After that we happily return write success status to the mtdblock caller.

Reading page back

If virtual memory subsystem wants the page back, MTDCOMPR receives read request from mtdblock. Using read-from offset we search in-ram tree for corresponding on-media offset. Then MTDCOMPR reads the data chain from the media and decompress it to the provided buffer.



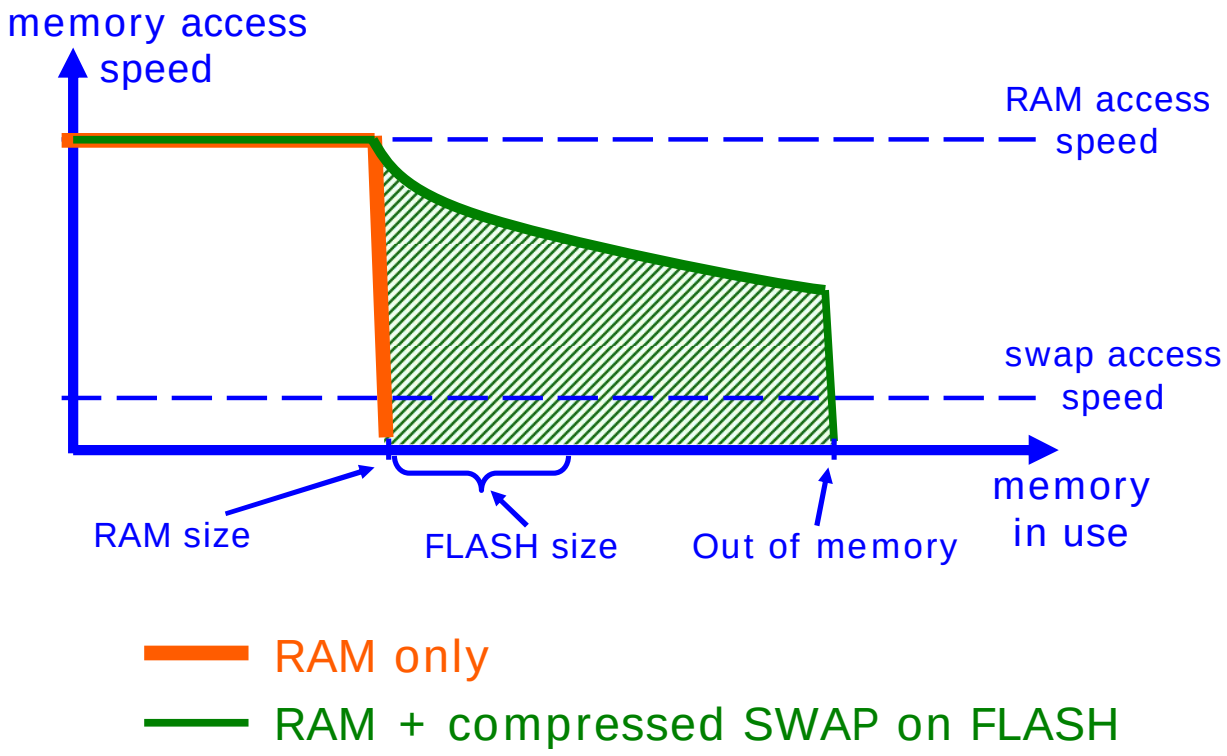
As page read request can span several read requests from mtdblock, we read and decompress whole page after the first subrequest, and then pass the data to the mtdblock caller with several read requests without actual reading from media.

Performance expectations (theory)

Compressed swap performance and memory increase depends on several factors. They are flash read/write times, flash partition size, erase requirements, CPU, compressor, data compression rate etc. But the generic behavior is quite common.

Flash memory

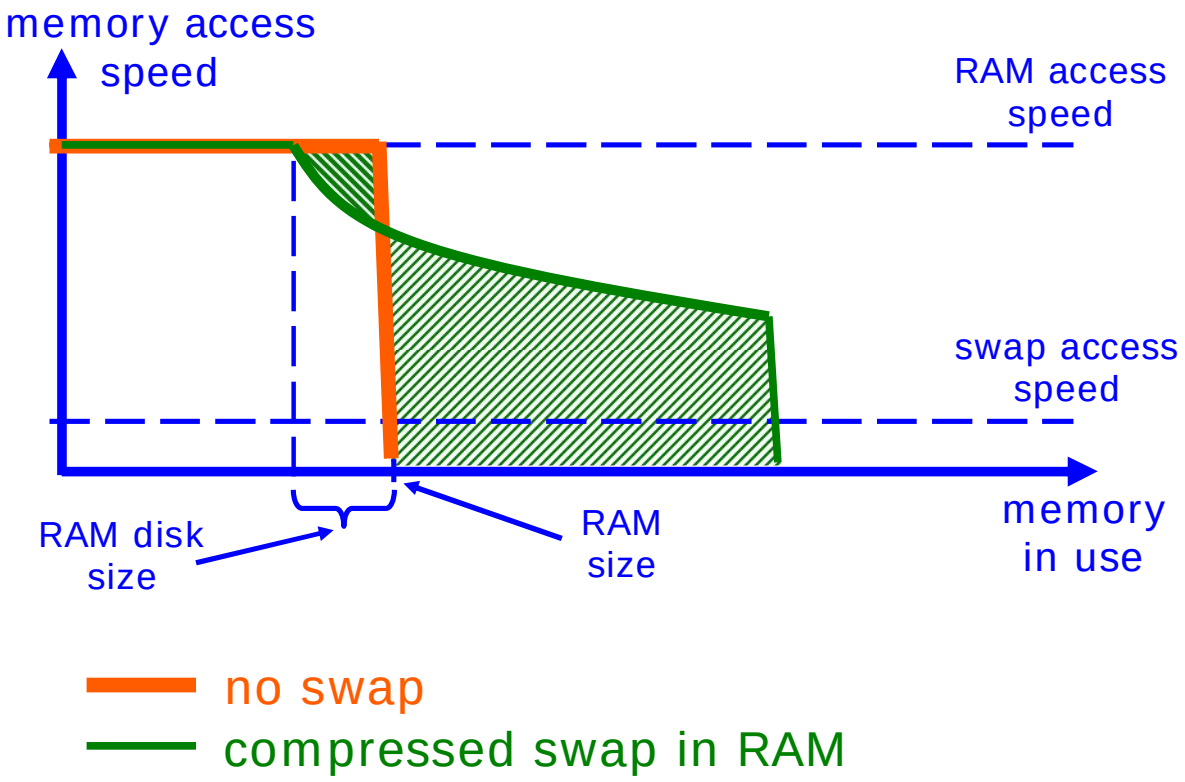
Speaking about swapless system we see almost flat memory access time until system uses all available RAM. After that system most likely gets out-of-memory exception. For system with compressed swap things are different. There is also almost flat performance until used memory amount hits RAM size. After that threshold memory access performance gets lower and stays somewhere between pure RAM and swapped pages access speed, because some pages are still in RAM and some pages are swapped. Out-of-memory event occurs later depending on data compression ratio.



Note there is known way to control swapping intensity – swappiness parameter which can be set via procfs interface. In particular this parameter changes the threshold where systems start swapping.

RAM

System with compressed swap in RAM starts to swap earlier because some amount of RAM is already allocated to store compressed pages. As for the rest of behavior it is quite similar to swapping to flash, except RAM is faster.



Performance (experiments)

Performance measurement technique

We used MTDCOMPR module with 2.6.23.8 kernel and ran it on Mainstone II (PXA271) development board limiting its RAM amount to 32 Mbytes. JFFS2 rootfs is placed on M18 volume. MTDCOMPR module itself gathers swap read/write average performance as well as compression and decompression throughput. This information can be accessed via procfs interface.

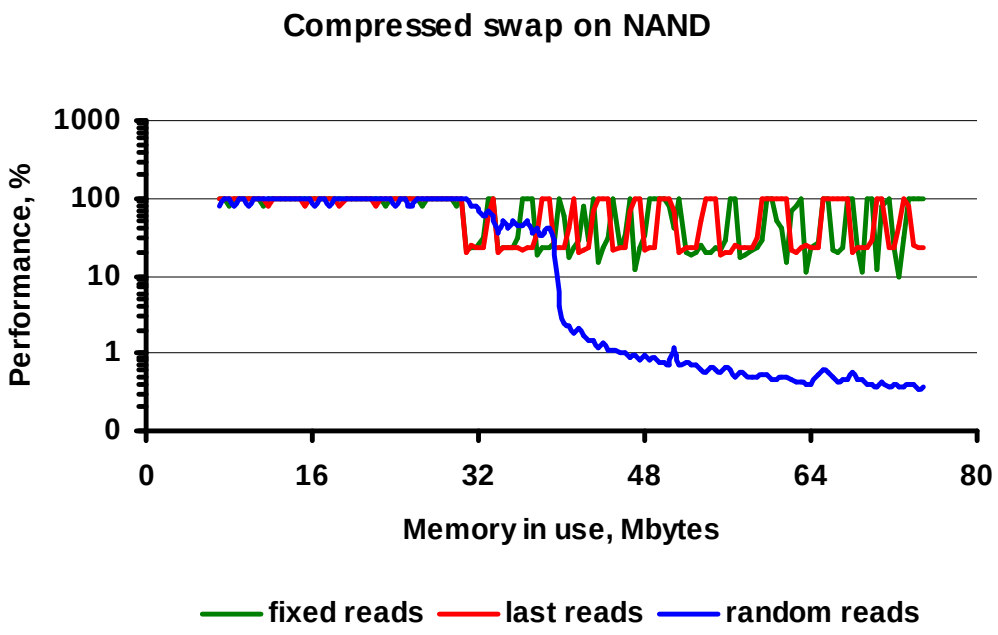
To measure the performance on user-mode level we were using simple benchmarking application. The application allocates memory chunk by chunk filling them with data. After chunk allocation it tries to read some of previously allocated data measuring average performance. There are three cases: first - data to read picked each time randomly ("random reads"); second – we randomly pick data chunk and read it several times ("fixed reads"); third – we read several times the last allocated chunk of data ("last reads").

"Random reads" case is quite far from reality, because application doesn't access all the allocated memory randomly. Normally applications work with selected (its own) piece(s) of memory. "Fixed read" and "last read" cases describe such a behavior quite well.

As a key performance indicator we used memory access speed chart depending on total amount of allocated by the application memory.

NAND

A lot of erase related troubles comes if we are speaking about NAND. Some of them have been solved in [7] by using NAND file system and changing swapping algorithm. We were focused on PCM and RAM based solution, but still experimented a bit with NAND. One of purposes was to give an impression about difficulties related to having swap on generic flash like NOR and NAND.

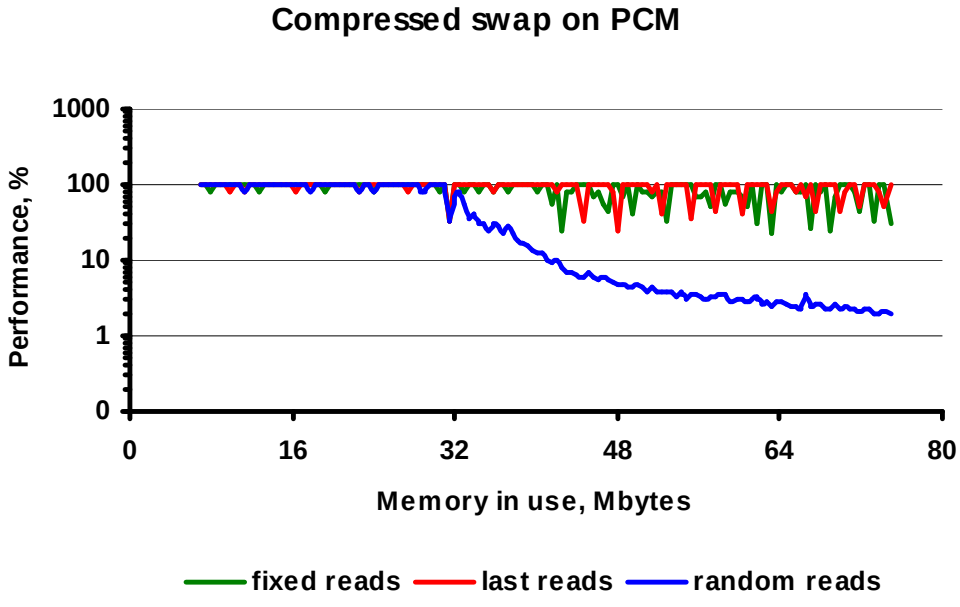


"Fixed reads" case average performance is about 52% of pure RAM performance. In case of "random reads" the performance quickly drops to the less than 0.5% level.

Main conclusion from experiment is that NAND is not appropriate media for such a simple solution. To make it viable it is necessary to use quite complex sector manager (FTL) with compression or NAND filesystem which does support swapping. But even using such complex software it still may require to change virtual memory subsystem to lower number of swap read/writes [7].

PCM

Phase change memory is great embedded solution's media for swapping, because of its bit alterable writes. It is possible to write data without explicit sector erase. From other side this memory behaves as generic NOR flash.

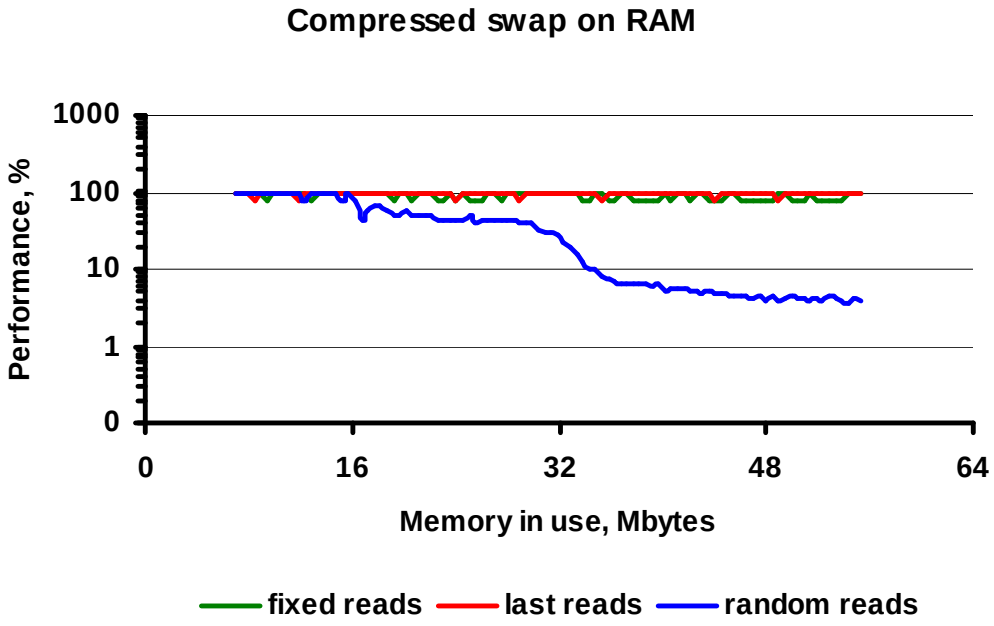


For the first PCM generation “fixed reads” benchmark showed performance at the level of 80% from one for pure RAM. In case of “random reads” the performance slowly drops to the less than 10% level.

In our experiments MTDCOMPR increased amount of memory available for applications by 2.5 times. Having 32 Mbytes of RAM and 16 Mbytes of PCM we allocated more than 70 Mbytes of memory for benchmarking application.

RAM

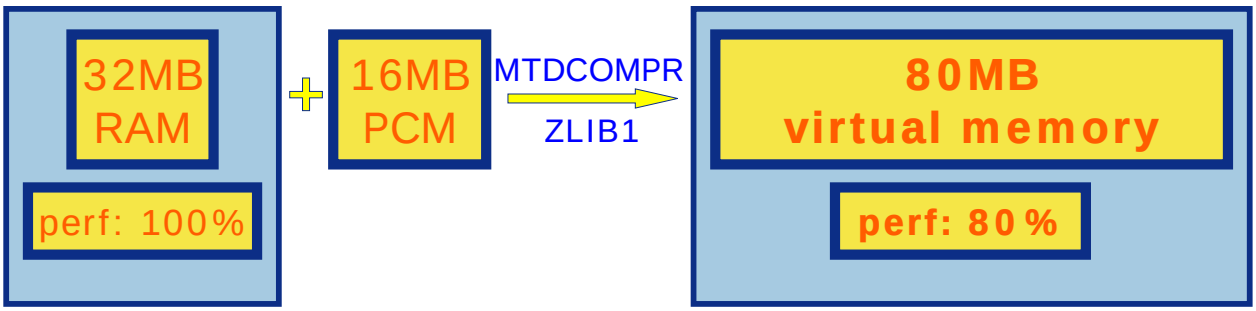
Storing compressed swapped pages in RAM provides more virtual memory for applications with quite small performance and power consumption change [6].



“Fixed reads” case average performance is about 89% of pure RAM performance. In case of “random reads” the performance drops to about 10% level. Note that the amount of memory available for applications was almost doubled in experiments with swap in RAM.

Conclusion

MTDCOMPR being relatively simple Linux kernel driver allows dramatically increasing virtual memory available for applications with moderate performance impact. Having more virtual memory with the same amount of RAM allows reducing embedded design cost and power consumption. Or, from the other hand, allows running applications with exceptional memory requirements.



PCM or RAM as swap area solution appears to be quite simple because no explicit erase is required. PCM enabled swapping is hardware dependent solution, RAM enabled – is not.

For NAND significant software changes may be needed including usage of FTL or NAND file system, as well as virtual memory subsystem change.

References

[1] M. Yokotsuka, "Memory motivates cell-phone growth," *Wireless Systems Design*, Apr. 2004.
[2] Nick Piggin, "LKML"
[3] B. Tremaine, et al., "IBM memory expansion technology," IBM Journal of Research and Development, vol. 45, no. 2, Mar. 2001.
[4] L. Benini, et al., "Hardware-assisted data compression for energy minimization in systems with embedded processors," in Proc. Design, Automation & Test in Europe Conf., Mar. 2002.
[5] T. Cortes, Y. Becerra, and R. Cervera, "Improving Application Performance through Swap Compression," in Proc. USENIX Conf., June 1999.
[6] Lei Yang, Robert P. Dick, Haris Lekatsas, Srimat Chakradhar, "CRAMES: Compresses RAM for Embedded Systems", in CODES + ISSS'05, Sept. 2005.
[7] Sangduck Park, Hyunjin Lim, Hoseok Chang, and Wonyong Sung, "Compressed Swapping for NAND Flash Memory Based Embedded Systems"
[8] Compressed Caching for Linux, http://code.google.com/p/compcache/