Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Evaluating effects of cache memory compression on embedded systems

Anderson Farias Briglia
Nokia Institute of Technology
anderson.briglia@indt.org.br

Allan Bezerra
Nokia Institute of Technology
allan.bezerra@nokia.com

Leonid Moiseichuk
Nokia Multimedia, OSSO
leonid.moiseichuk@nokia.com

Nitin Gupta
VMware Inc.
ngupta@vmware.com

Abstract

Cache memory compression (or compressed caching) was originally developed for desktop and server platforms, but has also attracted interest on embedded systems where memory is generally a scarce resource, and hardware changes bring more costs and energy consumption. Cache memory compression brings a considerable advantage in input-output-intensive applications by means of using a virtually larger cache for the local file system through compression algorithms. As a result, it increases the probability of fetching the necessary data in RAM itself, avoiding the need to make low calls to local storage. This work evaluates an Open Source implementation of the cache memory compression applied to Linux on an embedded platform, dealing with the unavoidable processor and memory resource limitations as well as with existing architectural differences.

We will describe the Compressed Cache (CCache) design, compression algorithm used, memory behavior tests, performance and power consumption overhead, and CCache tuning for embedded Linux.

1 Introduction

Compressed caching is the introduction of a new level into the virtual memory hierarchy. Specifically, RAM is used to store both an uncompressed cache of pages in their 'natural' encoding, and a compressed cache of pages in some compressed format. By using RAM to store some number of compressed pages, the effective size of RAM is increased, and so the number of page faults that must be handled by very slow hard disks is decreased. Our aim is to improve system performance. When that is not possible, our goal is to introduce no (or minimal) overhead when compressed caching is enabled in the system.

Experimental data show that not only can we improve data input and output rates, but also that the system behavior can be improved, especially in memory-critical cases leading, for example, to such improvements as postponing the out-of-memory activities altogether. Taking advantage of the kernel swap system, this implementation adds a virtual swap area (as a dynamically sized portion of the main memory) to store the compressed pages. Using a dictionary-based compression algorithm, page cache (file-system) pages and anonymous pages are compressed and spread into variable-sized memory chunks. With this approach, the fragmentation can be reduced to almost zero whilst achieving a fast page recovery process. The size of Compressed Cache can be adjusted separately for Page Cache and Anonymous pages on the fly, using `procfs` entries, giving more flexibility to tune system to required use cases.

2 Compressed Caching

2.1 Linux Virtual Memory Overview

Physical pages are the basic unit of memory management [8] and the MMU is the hardware that translates virtual pages addresses into physical pages address and vice-versa. This compressed caching implementation, CCache [3], adds some new flags to help with compressed pages identification and uses the same lists used by the PFRA (Page Frame Reclaiming Algorithm). When the system is under a low memory condition, it evicts pages from memory. It uses Least Recently Used (LRU) criteria to determine order in which

to evict pages. It maintains two LRU lists—active and inactive LRU lists. These lists may contain both page-cache (file-backed) and swap-cache (anonymous) pages. When under memory pressure, pages in inactive list are freed as:

- Swap-cache pages are written out to swap disks using `swapper_space writepage()` (`swap_writepage()`).
- Dirty page-cache pages are flushed to filesystem disks using filesystem specific `writepage()`.
- Clean page-cache pages are simply freed.

2.1.1 About Swap Cache

This is the cache for anonymous pages. All swap cache pages are part of a single `swapper_space`. A single radix tree maintains all pages in the swap cache. `swp_entry_t` is used as a key to locate the corresponding pages in memory. This value identifies the location in swap device reserved for this page.

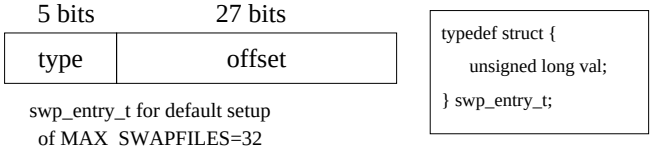

Figure 1: Fields in `swp_entry_t`

In Figure 1, **'type'** identifies things we can swap to.

2.1.2 About Page Cache

This is the cache for file-system pages. Like swap cache, this also uses radix-tree to keep track of file pages. Here, the offset in file is used as the search key. Each open file has a separate radix-tree. For pages present in memory, the corresponding radix-node points to `struct page` for the memory page containing file data at that offset.

2.2 Compressed Cache Overview

For compressed cache to be effective, it needs to store both swap-cache and page-cache (clean and dirty) pages. So, a way is needed to transparently (i.e., no changes required for user applications) take these pages in and out of compressed cache.

This implementation handles anonymous pages and page-cache (filesystem) pages differently, due to the way they are handled by the kernel:

- For anonymous pages, we create a **virtual swap**. This is a memory-resident area of memory where we store compressed anonymous pages. The swap-out path then treats this as yet another swap device (with highest priority), and hence only minimal changes were required for this kernel part. The size of this swap can be dynamically adjusted using provided `proc` nodes.
- For page-cache pages, we make a corresponding page-cache entry point to the location in the compressed area instead of the original page. So when a page is again accessed, we decompress the page and make the page-cache entry point back to this page. We did not use the 'virtual swap' approach here since these (file-system) pages are never 'swapped out.' They are either flushed to file-system disk (for dirty pages) or simply freed (for clean pages).

In both cases, the actual compressed page is stored as series of variable sized 'chunks' in a specially managed part of memory which is designed to have minimum fragmentation in storing these variable-sized areas with quick storage/retrieval operations. All kinds of pages share the common compressed area.

The compressed area begins as few memory pages. As more pages are compressed, the compressed area inflates (up to a maximum size which can be set using `procfs` interface) and when requests for these compressed pages arrive, these are decompressed, and corresponding memory 'chunks' are put back onto the *free-list*.

2.3 Implementation Design

When a page is to be compressed, the radix node pointing to the page is changed to point to the **chunk_head**—this in turn points to first of the **chunks** for the compressed page and all the chunks are also linked. This `chunk_head` structure contains all the information

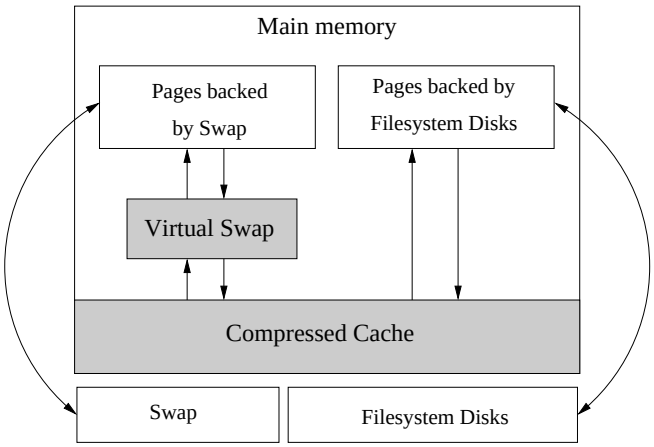

Figure 2: Memory hierarchy with Compressed Caching

required to correctly locate and decompress the page (compressed page size, compression algorithm used, location of first of chunks, etc.).

When the compressed page is accessed/required later, page-cache/swap-cache (radix) lookup is done. If we get a `chunk_head` structure instead of a page structure on lookup, we know this page was compressed. Since `chunk_head` contains a pointer to first of the chunks for this page and all chunks are linked, we can easily retrieve the compressed version of the page. Then, using the information in the `chunk_head` structure, we decompress the page and make the corresponding radix-node points back to this newly decompressed page.

2.3.1 Compressed Storage

The basic idea is to store compressed pages in variable-sized memory blocks (called *chunks*). A compressed page can be stored in several of these chunks. Memory space for chunks is obtained by allocating 0-order pages at a time and managing this space using chunks. All the chunks are always linked as a doubly linked list called the *master chunk list*. Related chunks are also linked as a singly linked list using the related-chunk list; e.g., all free chunks are linked together, and all chunks belonging to the same compressed page are linked together. Thus all chunks are linked using master chunk list and related chunks are also linked using one of related-chunk list (e.g. free list, chunks belonging to same compressed page).

Note that:

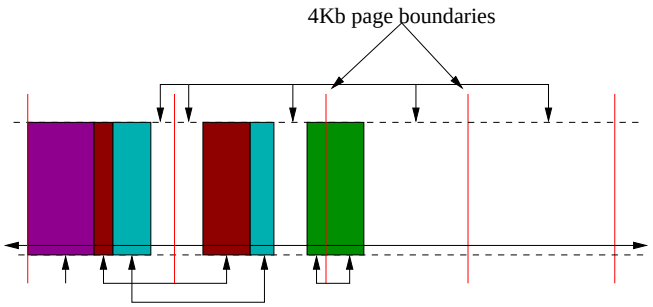

Figure 3: A sample of compressed storage view highlighting 'chunked' storage. Identically colored blocks belong to the same compressed page, and white is free space. An arrow indicates related chunks linked together as a singly linked list. A long horizontal line across chunks shows that these chunks are also linked together as a doubly lifnked list in addition to whatever other lists they might belong to.

- A chunk cannot cross page boundaries, as is shown for the 'green' compressed page. A chunk is split unconditionally at page boundaries. Thus, the maximum chunk size is `PAGE_SIZE`.
- This structure will reduce fragmentation to a minimum, as all the variable, free-space blocks are being tracked.
- When compressed pages are taken out, corresponding chunks are added to the free-list and physically adjacent free **chunks are merged together** (while making sure chunks do not cross page boundaries). If the final merged chunk spans an entire page, the page is released.

So, the compressed storage begins as a single chunk of size `PAGE_SIZE` and the free-list contains this single chunk.

An LRU-*list* is also maintained which contains these chunks in the order in which they are added (i.e., the 'oldest' chunk is at the tail).

2.3.2 Page Insert and Delete

Page Insert: The uncompressed page is first compressed in a buffer page. Then a number of free chunks are taken from free list (or a new page is allocated to get a new chunk) according to the size of the compressed page. These chunks are linked together as a singly

linked related-list. The remaining space from the last chunk is added back to the free list, and merged with adjacent free chunks, if present. The entry in the page-cache radix tree is now made to point to the `chunk_head` allocated for this compressed page. Pages that are not compressible (size increases on compression), are never added to CCache. The usual reclaim path applies to them.

Page Delete: When a page is looked up in memory, we normally get a `struct page` corresponding to the actual physical page. But if the page was compressed, we instead get a `struct chunk_head` rather than a `struct page` at the corresponding node (identified by the `PG_compressed` flag set), which gives a link to first chunk. Since all related chunks are linked together, compressed data is collected from all these chunks in a separate page, and then decompression is done. The freed chunks are merged with adjacent free chunks, if present, and then added to the free list.

2.4 Compression Methods

In the fundamental work [5], a number of domain-specific considerations are discussed for compression algorithms:

1. Compression must be lossless, i.e., one-to-one mapping from compressed and decompressed forms, so exactly the same, original data must be restored.

2. Compression must be in memory, so no kind of external memory can be used. Additionally, the following properties of typical memory data can be exploited to achieve good compression:

   - it is word or double-word aligned for faster handling by the processor;
   - it contains a large number of integers with a limited range of values;
   - it contains pointers (usually with integer size), mostly to the same memory region, so the majority of information is stored in the lower bits;
   - it contains many regularity sequences, often zeros.

3. Compression must incur a low overhead. Speed matters for both compression and decompression. A compression algorithm can also be asymmetric since typically decompression is required for many read operations, therefore making it important to have as cheap a decompression as possible. Small space overhead is also required since this overhead memory has to be allocated when the system is already running low on memory.

4. Compressible data, on average, should incur a 50% size reduction. If a page can not be compressed up to 50%, it increases the complexity of the procedure for handling compressed pages.

Thus, general purpose compression algorithms may not be good for our domain-specific compression requirements, due to high overhead. In this case we can choose low overhead compression, which takes into account the majority of in-memory data regularities and produces a sufficient compression ratio. We can also balance between compressibility and overhead by using several compression algorithms $\{A1..AN\}$ sequentially. Assuming that probabilities to compress a page are $\{P1..PN\}$ and compression times are $\{C1..CN\}$, the average compression time can be estimated as

$$C = \sum C_1 * P_1 \tag{1}$$

Note that since non-compressible pages can exist, we can determine that

$$\sum Pi < 1.0 \tag{2}$$

Thus, the best result from the speed versus compressibility point of view will be obtained by minimizing C time at compile- or run-time. The simplest way is to apply first the fastest algorithm, then the second fastest, and so on, leaving the slowest as last. Typically this scheme will work pretty well if $C1 \ll CN$; nevertheless, any runtime adoption can be used. Originally the following compression methods were used for cache memory compression [5]:

- WK in-memory compression family of algorithms as developed by Paul Wilson and Scott F. Kaplan. These algorithms are based on the assumption that

the target system has 32-bit word size, 22 bits of which match exactly, and 10 bits differ in values which will be looked for. Due to this assumption, this family of algorithms is not suitable for 64-bit systems, for which another lookup approach should be used. The methods perform close to real-life usage:

  - WK4x4 is the variant of WK compression that achieves the highest (tightest) compression by itself by using a 4x4 set-associative dictionary of recently seen words. The implementation of a recency-based compression scheme that operates on machine-word-sized tokens.
  - WKdm compresses nearly as tightly as WK4x4, but is much faster because of the simple, direct-mapped dictionary of recently seen words.

- miniLZO is a lightweight subset of the very fast Lempel-Ziv (LZO) implementation by Markus Oberhumer [9], [15]. Key moments can be highlighted from the description and make this method very suitable for our purpose:

  - Compression is pretty fast, requires 64KB of memory.
  - Decompression is simple and fast, requires no memory.
  - Allows you to dial up extra compression at a speed cost in the compressor. The speed of the decompressor is not reduced.
  - Includes compression levels for generating pre-compressed data, which achieves a quite competitive compression ratio.
  - There is also a compression level which needs only 8 KB for compression.
  - Algorithm is thread-safe.
  - Algorithm is lossless.
  - LZO supports overlapping compression and in-place decompression.
  - Expected speed of LZO1X-1 is about 4–5 times faster than the fastest zlib compression level.

3 Experimental Results

This section will describe how the CCache was tested on an OMAP platform device—the Nokia Internet Tablet N800 [6]. The main goal of the tests is to evaluate the characteristics of the Linux kernel and overall system behavior when CCache is added in the system.

As said before, this CCache implementation can compress two types of memory pages: anonymous pages and page-cache pages. The last ones never go to swap; once they are mapped on a block device file and written to the disk, they can be freed. But anonymous pages are not mapped on disk and should go to swap when the system needs to evict some pages from memory.

We have tests with and without a real swap partition, using a MMC card. Our tests intend to evaluate CCache against a real swap device. So, we decided to use just anonymous pages compression. This way we can better compare CCache performance against a system with real swap. For the comparison, we measured the following quantities:

- How many pages were maintained in memory (CCache) and did not go to swap (avoiding I/O overhead).
- Changes in power requirements.
- Comparison of different compression algorithms: compress and decompress times, compression ratio.

3.1 Test Suite and Methodology

We used a mobile device with embedded Linux as testing platform. The Nokia Internet Tablet N800 has a 330Mhz ARM1136 processor from Texas Instruments (OMAP 2420), 128MB of RAM, and 256MB of flash memory used to store the OS and applications. The OMAP2420 includes an integrated ARM1136 processor (330 MHz), a TI TMS320C55x DSP (220 MHz), 2D/3D graphics accelerator, imaging and video accelerator, high-performance system interconnects, and industry-standard peripherals [4]. Two SD/MMC slots, one internal and another external, can be used to store audio/video files, pictures, documents, etc.

As we can see on device's specification, it is focused on multimedia usage. As we know, multimedia applications have high processor and memory usage rates. Our tests intend to use memory intensively, hence we can measure the CCache impact on system performance. The tests were divided into two groups: tests with real applications and tests using synthetic benchmarks. Tests with real applications tried to simulate a high memory consumption scenario with a lot of applications being executed and with some I/O operations to measure the memory and power consumption behavior when CCache is running. Synthetic tests were used to evaluate the CCache performance, once they provided a easy way to measure the time spent on the tests.

Tests with real applications consist of:

- Running 8 or 9 Opera browsers and load some pages through a wireless Internet connection.
- Playing a 7.5MB video on Media player.
- Opening a PDF document.

We used an application called **xautomation** [12] to interact with the X system through bash scripts from the command line. Xautomation controls the interface, allowing mouse movements, mouse right and left clicks, key up and down, etc. Reading the `/proc/meminfo` file, we have the memory consumption, and some graphics related to the memory consumption can be plotted. They will be shown and commented on in the following sections.

The tests using synthetic benchmarks were executed using MemTest [11]. MemTest is a group of small tests to evaluate the stability and consistency of the Linux memory management system. It contains several tests, but we used just one, since our main goal is to measure the CCache performance: **fillmem**. This test intends to test the system memory allocation. It is useful to verify the virtual memory system against the memory allocation operation, pagination, and swap usage. It has one parameter which defines the size of memory allocated by itself.

All the tests, using real applications or synthetic benchmarking, were applied using a pre-configured scenarios, depending on what would be measured. Basically we have scenarios with different RAM memory sizes, with or without a *real* swap partition, and with or without CCache added to the system.

Memory behavior tests evaluate the memory consumption against time and the OOM killer interaction on the scenarios discussed before. Performance tests used MemTest [11] to measure the total time of the `fillmem` execution. Power consumption tests evaluate the impact of CCache usage on power consumption, since it is crucial on mobile devices with embedded systems.

3.1.1 Tuning for Embedded Linux

One of the most important constraints in embedded systems is the storage space available. Therefore implementing mechanisms to save some space is crucial to improve the embedded OS.

The N800's OS, also known as Internet Tablet OS 2007 (IT OS 2007), is based on the linux-omap [10] kernel and has some features customized to this device. One of these, the most relevant in this case, is the swap system behavior. The Linux kernel virtual memory subsystem (VM) operation can be tuned using the files included at `/proc/sys/vm` directory. There we can configure OOM-killer parameters, swap parameters, writeout of dirty data to disk, etc. On this case, just swap-related parameters were modified, since the evaluation must be as close as possible to reality—in the other words, the default system configuration.

The swap system behavior on the device's kernel is configured as if a swap partition were not present in the system. We configured two parameters to make the test execution feasible under low-memory scenarios: `/proc/sys/vm/swappiness` and `/proc/sys/vm/min_free_kbytes`.

- **swappiness** [7] is a parameter which sets the kernel's balance between reclaiming pages from the page cache and swapping process memory. The default value is 60.
- **min_free_kbytes** [7] is used to force the Linux VM to keep a minimum number of kilobytes free.

If the user wants the kernel to swap out more pages, which in effect means more caching, the `swappiness`

parameter must be increased. The `min_free_kbytes` controls when the kernel will start to force pages to be freed, before out-of-memory conditions occur.

The default values for `swappiness` and `min_free_kbytes` are **0** and **1280**, respectively. During our tests, we had to modify these values since the CCache uses the default swap system, and with the default values, the CCache would not be used as expected. Another fact is that we added a real swap partition, and the `swappiness` must be adjusted to support this configuration. Increasing the `swappiness` to its default value, 60, we noticed that more pages were swapped and the differences between the CCache usage and a real swap partition could be measured.

During our tests the available memory was consumed so quickly that CCache could not act. The OOM killer was called, and the test was killed. To give more time to CCache, the `min_free_kbytes` was increased. Another motivation is that pages taken by CCache are marked as pinned and never leave memory; this can contribute to anticipating the OOM killer call. It happens because the CCache pages are taken out from the LRU list and due to this, the PFRA (Page Frame Reclaiming Algorithm) cannot select such pages to be freed.

Other parameters that must be adjusted are the `max_anon_cc_size` and `max_fs_backed_cc_size`. These parameters set the max size for anonymous pages and the max size for page-cache pages, respectively. The CCache will increase until the max size is reached. Note that those parameters are set in number of pages (4KB) and the pages are used to store the chunks lists and the metadata needed to manage those chunks. These parameters are exported by CCache via `/proc/sys/vm/max_anon_cc_size` and `/proc/sys/vm/max_fs_backed_cc_size`. We have one limitation here: `max_anon_cc_size` and `max_fs_backed_cc_size` can be configured only one time; dynamic re-sizing is not permitted yet.

As previously mentioned, the tests covered the anonymous pages compression; therefore only `max_anon_cc_size` was used. After the initialization, the IT OS 2007 has about 50M of free memory available, from a total of 128M. After making some empirical tests, we decided to set `max_anon_cc_size` to about 10% of device's total free memory (5 MB), or in other words, 1280 pages of 4 KB each. Since CCache pages are pinned in memory and adaptive resizing is not yet implemented, we chose to have only a small percentage of memory assigned to ccache (10%). With a higher percentage, we might end-up calling the OOM killer too soon.

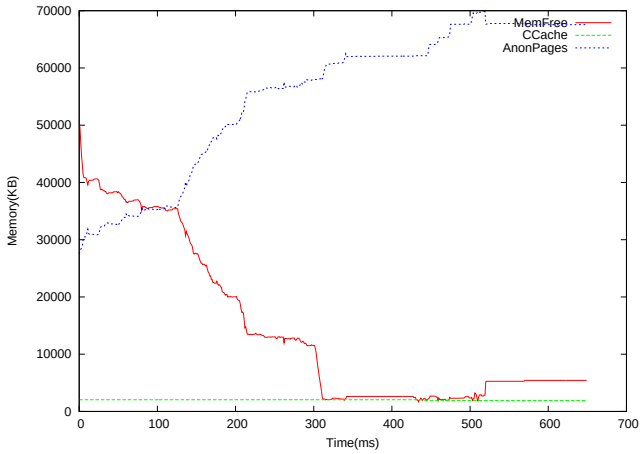

Figure 4: CCache x time: max_anon_cc_size = 1024, Mem = 128M

3.2 Memory Behavior Tests

The main goal of these tests is to see what is happening with the memory when CCache is compressing and decompressing pages. To execute the tests we used real application scenarios, using applications provided by the default installation of the N800's distribution.

Using a bash script with XAutomation [12], we can interact with the X server and load some applications while another program collects the memory variables present in `/proc/meminfo` file. Figure 4 shows the memory consumption against time on a kernel with CCache and `max_anon_cc_size` set to 1024 pages (4MB).

As we can see in Figure 4, the CCache consumption was very low once the `swappiness` parameter was configured with default value of 1. Therefore more pages are cached in memory instead of going to swap, even if the available free memory is low. Actually this behavior is expected since the system doesn't have swap for default. On Figure 5, we limited the memory to 100M at boot, which caused increased memory pressure on the system. The figure shows that the OOM killer was called at 600 ms, and CCache did not have time to react.

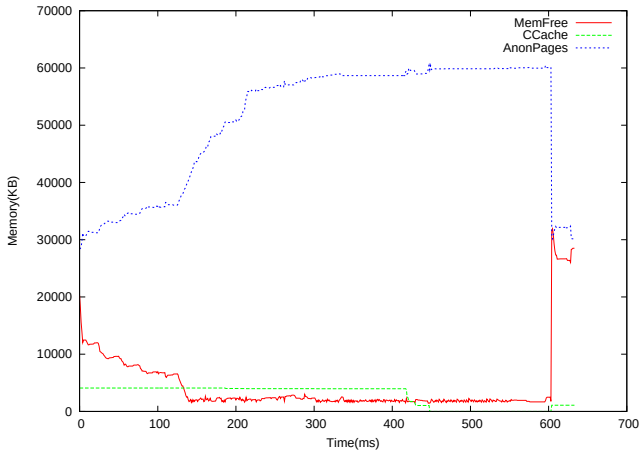

Figure 5: CCache x time: max_anon_cc_size = 1024, Mem = 100M

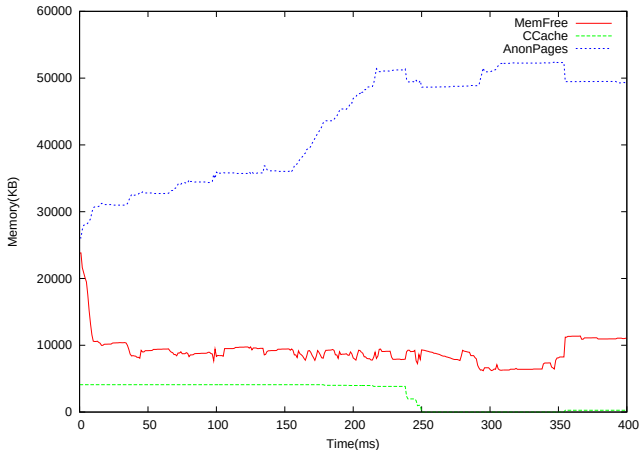

Figure 6: CCache x time: swappiness = 60 and min_free_kbytes = 3072

Using the same test but with `swappiness` and `min_free_kbytes` configured with 60 and 3072 respectively, the OOM killer is not called any more, and we have an increased usage of CCache. See Figure 6.

Figure 6 shows that the CCache size is not enough for the memory load of the test. But we have a problem here: if the CCache size is increased, more pages are taken off the LRU list and cannot be freed. The best solution here is implementing the adaptive [1, 13] CCache resizing. We will cover a bit more about adaptive CCache in Section 4.

In the tests, the new page size was collected for each page that was compressed. With this, we can have measurements using the most common ranges of compressed page size. Figure 7 illustrates the Compression Page Size Distribution in five ranges, pages sized between: 0K–0.5K, 0.5K–1K, 1K–2K, 2K–3K and 3K–4K. About 42% of pages have sizes between 1K and 2K after compression.

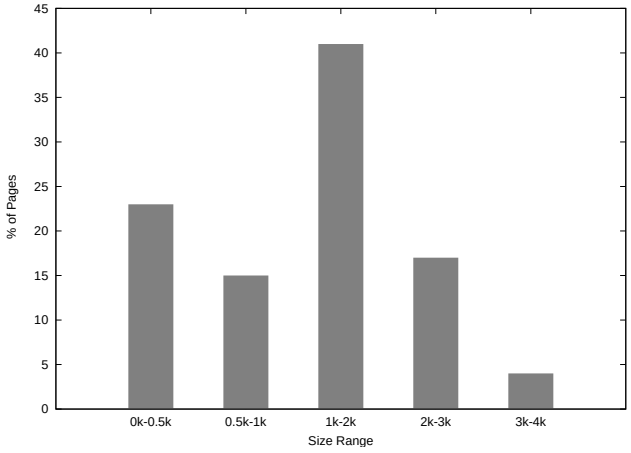

Figure 7: Compression size distribution

These results indicate that we have a general system gain with CCache. Since most of pages sizes are between 1K–2K, and we adopted a chunk-based approach with no fragmentation. In general, we have an increase of 100% of 'system available memory.' It is an important CCache advantage: applications that before, could not be executed on the system, now have more 'visible available memory' and can be executed, allowing an embedded system to support applications that it would otherwise be unable to run. It is important to remember that on CCache, compressed pages with size more than 4KB are discarded.

3.3 Performance Tests

Performance tests aim to analyze the CCache overhead: compression overhead, chunks lists handling overhead, and page recovery overhead. With these tests we expect to prove that CCache, even with all those overheads, is faster than using a swap partition on a block device.

Only anonymous pages are being considered here. As explained in Section 2.2, there are some steps when a page is compressed. Table 1 shows the results when running the fillmem allocating 70MB of memory (`swappiness = 60, min_free_kbytes = 1024KB, max_cc_anon_size = 1280 pages`):

The test using WK4x4 triggered the OOM killer and could not be completed. But taking a look at the other

Test	No-CCache	CCache WKdm	CCache WK4x4	CCache LZO
Time(s)	14.68	4.64	–	4.25

Table 1: fillmem(70): CCache time x Real Swap time

values, we can conclude that the swap operation using CCache is about **3 times** faster than using a swap partition on an MMC card. But how about the OOM-killer?

As discussed before, CCache does not have dynamic resizing. It implies that for some use cases, CCache has no benefits and the memory allocated to the compressed data becomes prejudicial to the system. Since we do not have the adaptativity feature implemented, it is very important to tune CCache according the use case.

3.3.1 Compression Algorithms Comparison

As we wrote above, it is essential to minimize overhead by using an optimal compression algorithm. To compare a number of algorithms, we performed tests on the target N800 device. In addition to those algorithms used in CCache (WKdm, WK4x4, zlib, and miniLZO), we tested methods used in the Linux kernel (rubin, rtime) and LZO compression.

Speed results are shown below relative to `memcpy` function speed, so 2.4 means something works 2.4 times slower than `memcpy`. Reported data is the size of data produced by compiler (`.data` and `.bss` `segments`). The test pattern contained 1000 4K-pages of an ARM ELF file which is the basic test data for CCache. Input data was compressed page-per-page (4 KB slices).

Name	ARM11 code size (bytes)	ARM11 data size (bytes)	Comp time (relative)	Comp ratio (%)	Decomp time (relative)	Speed asymm.
Wkdm	3120	16	2.3	89	1.4	1.8
Wk4x4	4300	4	3.5	87	2.6	0.9
miniLZO	1780	131076	5.6	73	1.6	3.5
zlib	49244	716	73.5	58	5.6	13.1
rubin_mips	180	4	152	98	37	4.2
dyn_rubin	180	4	87.8	92	94.4	0.9
rtime	568	4	7	97	1	7
lzo1	2572	0	8.8	76	1.6	5.5
lzo2	4596	0	62.8	62	2	31.4

Table 2: Compression Algorithms Comparison

From these tests we can draw the following conclusions for the target system:

1. zlib is a very slow method, but it provides the best compression;
2. using WK4x4 makes no practical sense because the number of pages compressed for 50% or better is the same as for WKdm, but in terms of speed, WK4x4 is about 2 times slower;
3. lzo2 has good compression and very good decompression speed, so it will be used to replace zlib everywhere.
4. by using WKdm and miniLZO sequentially, we can obtain good compression levels with small overhead.

3.4 Power Consumption Tests

In order to evaluate the power consumption using CCache, we replaced the device's battery with an Agilent direct current power supply. This Agilent equipment shows the total current in real time and with this, the power consumption was calculated using an output voltage of 4V.

Figure 8 shows the power consumption for the interactive user test used and described in Section 3.2. Steps 1 to 5 can be discarded in an analysis of power consumption since in these steps CCache is active, but it is not called. The important steps are 5 to 8. In these steps we have a stress memory situation and CCache is working. As we can see, we have a gain of about 3% on average when compression is used. It can be explained for smaller I/O data rates. Is important to take a look at the 'Video Play' step: this kind of application needs more energy to process the video stream. These results show that in this situation we have a good gain when CCache is working. It is important to note that some variation is accepted since all interactive tests results are use-case dependent.

4 Related Work

The first implementation of compressed cache was done by Douglis [2] in 1993 and results were not conclusive. He achieved speed up for some applications, and slowdowns for others. Later, in 1999 Kaplan [5] pointed out that the machine Douglis used had a difference between the processor speed and the access times on hard disk

	Tests Step	Power Consumption for Max Corrent (W) Voltage used was 4V	
		W/O CC	With CC
Browser Pages	1	2,332	2,328
	2	2,136	2,184
	3	2,348	2,360
	4	2,392	2,384
	5	2,448	2,372
	6	2,492	2,440
	7	2,484	2,436
	8	2,572	2,444
Video Play	9	2,880	2,804
PDF File Open	10	2,528	2,480
File Manager Filesystem Scan	11	2,356	2,316

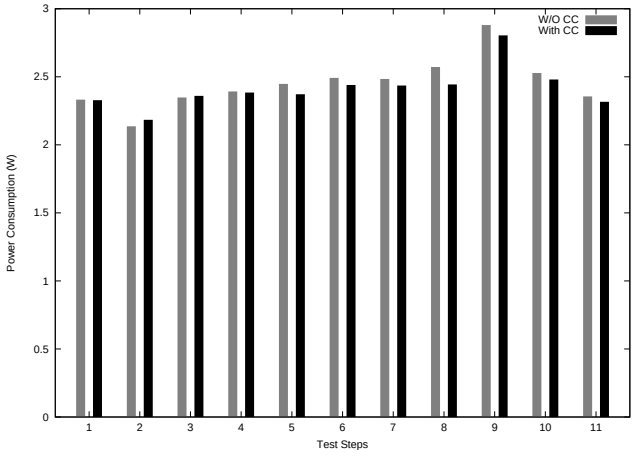

Figure 8: Power Consumption tests using CCache

that was much smaller than encountered nowadays. Kaplan also proposed a new adaptive scheme, since that used by Douglis was not conclusive about the applications' performance.

Following the same scheme, Rodrigo Castro [1] implemented and evaluated compressed cache and compressed swap using the 2.4.x Linux kernel. He proposed a new approach to reduce fragmentation of the compressed pages, based on contiguous memory allocated areas—cells. He also proposed a new adaptability policy that adjusts the compressed cache size on the fly. Rodrigo's adaptive compressed cache approach is based on a tight compressed cache, without allocation of superfluous memory. The cells used by compressed cache are released to the system as soon as they are no longer needed. The compressed cache starts with a minimum size and as soon as the VM system starts to evict pages, the compressed cache increases its memory usage in order to store them.

All those implementations are focused on desktop or server platforms. One implementation which is focused on embedded devices is CRAMES [14]—Compressed RAM for embedded systems. CRAMES was implemented as a loadable module for the Linux kernel and evaluated on a battery-powered embedded system. CRAMES supports in-RAM compressed filesystems of any type and uses a chunks approach (not the same described in this paper) to store compressed pages.

The compressed cache implementation evaluated in this paper still does not have the adaptive feature implemented. From previous work, CCache uses the same compression algorithms used by Rodrigo's implementation, but the storage of compressed pages is quite different. The chunks approach reduces the fragmentation close to zero and speeds up the page recovery. CCache is the first Open Source compressed cache implementation for 2.6.x Linux kernel and it is under development, both for desktop/server platforms and for embedded devices, as presented in this paper.

5 Conclusions and Future Work

Storing memory pages as compressed data decreases the number of access attempts to storage devices such as, for example, hard disks, which typically are accessed much slower than the main memory of a system. As such, we can observe more benefits when the difference between the access time to the main memory and the storage device is considerable. This characteristic is not typical for embedded Linux systems, and the benefits of storing pages as compressed data are much smaller than on an x86 architecture. Storage devices in the embedded Linux systems are typically flash memory, for instance MMC cards, and as we know, access times for these devices are much faster than access times for a hard disk. It allows us to come to the conclusion that wide use of CCache is not justified in the embedded systems.

On the other hand, embedded systems have limitations in the available memory. Thus, the experimental tests results show that CCache can improve not only the input and output performance but the system behavior in general by improving memory management like swapping, allocating big pieces of memory, or out-of-memory handling. Being that as it may, it is also common knowledge that this scenario has been changing along the development of embedded Linux. The best benefit of this CCache implementation is that developers can adjust it to its system characteristics. Implementing a software-based solution to handle the memory limitations is much

better than hardware changes, which can increase the market price of a product.

The power tests show that CCache makes a positive impact on power consumption, but for the N800 architecture the overall benefit is minor due to the high level of system integration which is based on the OMAP 2420.

Finally, we can make some important points which can improve this CCache implementation:

- use only fast compression methods like WKdm and minilzo, maybe with lzo2;
- make performance and memory optimization for compression take into account CPU and platform hardware-accelerating features;
- enable compression according to instant memory load: fast methods when memory consumption is moderated, and slow when high memory consumption is reached.
- compress/decompress pages that go to/from a real swap partition.
- adaptive size of compressed cache: detection of a memory consumption pattern and predict the CCache size to support this load.

6 Acknowledgements

First we have to thank Nitin Gupta for designing and implementing CCache on Linux 2.6.x kernels. We would like to thank Ilias Biris for the paper's revision and suggestions, Mauricio Lin for helping on the Linux VM subsystem study, and Nokia Institute of Technology in partnership with Nokia for providing the testing devices and support to work on this project.

References

[1] Rodrigo Souza de Castro, Alair Pereira Lago, and Dilma da Silva. Adaptive compressed caching: Design and implementation. 2003. http://linuxcompressed.sourceforge.net/docs/files/paper.pdf.

[2] Douglis F. The compression cache: Using on-line compression to extend physical memory. 1993. http://www.lst.inf.ethz.ch/research/publications/publications/USENIX_2005/USENIX_2005.pdf.

[3] Nitin Gupta. Compressed caching for linux project's site, 2006. http://linuxcompressed.sourceforge.net/.

[4] Texas Instruments, 2006. http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?contentId=4671&navigationId=11990&templateId=6123.

[5] S. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, 1999.

[6] LinuxDevices.com, 2007. http://www.linuxdevices.com/Articles/AT9561669149.html.

[7] linux.inet.hr, 2007. http://linux.inet.hr/proc_sys_vm_hierarchy.html.

[8] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005. ISBN 0-672-32720-1.

[9] Markus Franz Xaver Johannes Oberhumer, 2005. http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[10] Linux omap mailing list, 2007. http://linux.omap.com/mailman/listinfo/linux-omap-open-source.

[11] Juan Quintela, 2006. http://carpanta.dc.fi.udc.es/~quintela/memtest/.

[12] Steve Slaven, 2006. http://hoopajoo.net/projects/xautomation.html.

[13] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression. 2005. http://www.lst.inf.ethz.ch/research/publications/publications/USENIX_2005/USENIX_2005.pdf.

[14] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. Crames: Compressed ram for

embedded systems. 2005. http://ziyang.eecs.northwestern.edu/~dickrp/publications/yang05sep.pdf.

[15] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.