Department of Informatics

Adaptive Compressed Caching: Embracing and extending for the Linux 2.6 kernel

Master Thesis

Asbjørn Sannes
asbjorsa@ifi.uio.no

May 15, 2008

Abstract

Applications needs to have their working set in main memory to work efficiently. When there is memory contention, data has to be read from and written to slow backing store, such as disk. Compressed caching is a way to keep more pages in main memory by compressing the oldest pages. This reduces disk I/O because more of the needed data is available in main memory. Applications that can gain most from compressed caching have low *entropy* in their data and reuse recently data as a rule. Entropy quantifies the information contained in a message, in our case a page, usually in bits or bits/symbol. This gives an absolute limit on the best possible lossless compression for a page. The opposite charactristics apply to applications that perform worse with compressed caching than without it. They do not reuse recently used data and have high entropy causing the compression to have a bad ratio and the cache to have a low hit rate on the compressed pages.

In this master thesis, we design, implement and evaluate compressed caching for Linux (version 2.6.22). In our implementation, we modify the PFRA (page frame reclaiming algorithm) to convert pages to compressed pages instead of evicting them, and convert them back when they are accessed in the page cache. The compressed pages are kept in a list in the same order as they are put in by the PFRA. We adapt the size of the compressed cache by looking at how it is used, if we need to shrink the compressed cache, the oldest compressed page is evicted.

For compressed caching unfriendly applications we extend an earlier approach of disabling compressed caching globally when it is not useful, with a more fine-grained cache disabling algorithm. We do this by defining ”memory areas”, and disabling compressed caching for them based on their recent behavior. We extend upon earlier approaches of compressed caching and measure the impact on performance.

We define workloads and run tests to measure the performance of compressed caching compared to an unmodified Linux kernel. We change the amount of main memory available and the number of processes running simultaneously and observe the impact on the performance. We then evaluate the results and find more than 40% reduction in running time for some tests.

We discuss our findings and conclude that compressed caching reduces disk I/O when there is memory contention, and can therefore increase performance of applications that can not keep their complete working set in memory uncompressed, but have low enough entropy to keep it in main memory in compressed form.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and motivation

Typical computers have less memory than many applications require, which is very apparent when running multiple memory intensive applications at the same time on a typical desktop. An example of this is that average computers have about two gigabytes of RAM while simultaneously running typical applications such as Mozilla Firefox, OpenOffice, Mozilla Songbird and Eclipse. All of these applications use hundreds of megabytes, and caches large amounts of data to have small latencies when the user requests information.

When the memory usage of the running processes in a system is higher than the physical amount of main memory, the operating system needs to use secondary memory to meet the demand. This introduces a considerable performance decrease for two reasons: (1) a single instruction could result in the execution of multiple functions to locate the needed memory when it is not already present; (2) secondary memory is most often a hard disk which is many orders of magnitude slower to access than main memory. Every access to memory located on secondary memory also means that the kernel needs to make room for the data to be copied into main memory, and this in turn means choosing what data to swap out. Denning et al. [3] describe a working set to be the smallest amount of application data that have to be in main memory for the application to work efficiently. Memory is divided into basic units called pages, therefore the working set is made out of the pages that contain the data needed to make the application work efficiently. This working set is what we want to keep in main memory, or as much of as possible when deciding what pages should be removed from main memory.

Performance of the policy that choose what pages are to be paged out are measured by how many pages are read into main memory and how many pages are paged out to disk. This is called page-traffic by Denning et al.[3] and page fault rate in more recent[10] literature. The lower the page fault rate is, the better the policy. Most policies are based on the principal of temporal locality, which states that recently accessed pages are most likely to be accessed in the near future. In other words we want to remove the page that is least recently used (LRU) of the pages currently in main memory.

If the working set is bigger than main memory, even with a perfect LRU algorithm, this will result in a decrease in performance of the application. Most of this decrease is due to the time spent on retrieving a needed page from disk. With the factor of 10^6 difference in speed between main memory and disk (typically a memory access takes 2-10 μs and a disk access from $5 - 10$ms) it is interesting to see if we could reduce disk I/O. From an application's viewpoint there is only one option when trying to reduce the working set, and that is to store the information the application needs and uses in a more compact way. This is not always possible, practically because there are so many applications out there that we can not easily change, and even if it were possible, it would result in less optimal operations when enough main memory is available for the working set. To reduce memory usage for an application from the kernel's viewpoint can be done in several ways. The most common form for this is to have as little overhead as possible, keeping only the necessary data for a process and keep those internal data structures small and compact. Another way, and the topic of this thesis, is to compress the data of an application in main memory. Since this is done transparently to the application, it will still not make the complete working set available uncompressed in main memory and it will not reduce the page fault rate. In the case that the compression has succeeded in compressing the application data to a size that fits in main memory, the disk I/O will be reduced. The main obstacle to succeeding in compressing application data into a workable size is the entropy of that data. Shannon entropy is the minimum number of bits to represent a symbol in the data. For example, if the data consist of only 0 and 1, only one bit is needed to encode those two symbols. In other words if the randomness of the data is low so is the entropy, this means that the compressibility of the data is high. Since main memory is managed in units of pages, the entropy we look at is per page. This means that even if the application has 100 completely equal pages this will, for per-page compression, not make the entropy of the application lower.

In most cases, caches in a system have close to zero cost associated with them in terms of performance. The worst scenario for using those caches is to not obtain any performance enhancements compared to not having a cache. Compressed cache on the other hand has two costs: the resource cost of main memory used and the CPU used for compression and decompression.

To summerize, the ideal application for compressed caching responds well to LRU page replacement policies and has low entropy on a per page basis.

1.2 Problem description

Under high memory demand, applications suffer considerable performance decrease due to swapping. Most of these decreases comes from disk I/O and its related operations such as seek time and rotational delay, which means that other resources such as CPU is idling doing nothing.

There is a possibility for performance increase by reducing page faults, i.e. number of pages that need to be read from secondary memory. Applications that have an ideal behavior for compressed caching are most likely to take advantage of this. For non-ideal applications there is a real chance for performance decrease, and for these applications it is important to detect them and work out a solution to avoid substantial performance decrease. Behavior that is bad for compressed caching includes applications that work on high entropy data, such as highly compressed data, where trying to compress them could actually end up taking more or the same amount of space wasting CPU and even more main memory. Another example of bad behavior is when the application reuses none or very few pages, which will result in a very bad hit ratio for the compressed cache where CPU and memory is wasted.

The size of the compressed cache is important, because the larger it is, the more page faults occur due to less non-compressed main memory being available. At the same time, the larger the compressed cache is, the less disk I/O will occur if pages are being re-accessed.

Considering the evolution of hard disk speeds and CPU speed [6] we know that CPU has had a much faster growth rate in terms of speed compared with conventional hard disks. This means that as we end up swapping, more and more CPU resources will be wasted. It is these resources we want to take advantage of when using compressed caching.

There are several goals we wish to achieve with this thesis; first of all, we want to implement compressed caching in an up-to-date modern operating system, Linux in this case. While doing so, we want to improve upon earlier work done by others. Previous works have often limited themselves to testing one application at a time, while real world applications usually run in parallel. We want to do some performance evaluation on this subject.

1.3 Outline

This master thesis is structured as follows. Chapter 2 gives an introduction into concepts and problems with compressed caching. It looks at what a cache is and how compressed caching differs from other caching. It also briefly describes how virtual memory and paging works, which is a requirement for compressed caching. It also introduces earlier work on compressed caching as well as some new concepts. Caches and types of memory already present in the operating system is described.

Chapter 3 describes various subsystems in the Linux kernel. The idea here is to highlight traits of these subsystems that impact how compressed caching should be designed and implemented. It goes through the basics of how pages are managed in Linux, how pages are reclaimed, the file cache and how swapping is done. It also touches upon memory management and related weaknesses and strengths with the current approach. The read-ahead mechanisms are described for both anonymous and file-backed pages. It also takes a look at the radix tree data structure in Linux, since this is used throughout memory management. Synchronization of dirty pages connected to specific files and whole file systems are also explained.

In Chapter 4 we describe the design of how Linux should be modified to accommodate compressed caching. We start out with a general overview to base the design on and set the level of abstraction the design should adhere to. We then go into the reasoning around how a compressed page should be represented. We also describe what constitutes a good size for the compressed cache and how to calculate it. We briefly talk about in-memory storage of the content of compressed pages before we examine how the compressed cache will grow and shrink. When and where to do compression and decompression and possible cache policies are discussed. Badly behaving memory areas and corner cases are described, as well as a design of the cell memory allocator and its inner workings.

Chapter 5 contains thorough explanations of the modifications and additions needed to implement compressed caching in the Linux kernel. We go into details on the interception of pages that are being looked up and how the profit lists are maintained. We also describe changes necessary for policy based decisions. The representation of a compressed page is explained in detail. The cell memory allocators layout, functions and algorithms are described in detail.

In Chapter 6 we go through what kind of workloads we want to test and why we choose those. The actual setup is described and explained. Some problems that we have met during testing are explored briefly before we go into the actual runs and their parameters. We show, explain and evaluate our results.

Chapter 7 presents the conclusions where we go through contributions, critical assessment of how this thesis has transpired and suggest future work around compressed caching.

Chapter 2

Compressed caching

In this chapter, we give an introduction into what compressed caching is and how it works. We start with explaining what caching is and how performance is measured. Next, we give a brief introduction of virtual memory since this is one of the fundamental techniques needed to implement compressed caching. We then go into the two types of pages we are interested in storing in the compressed cache: anonymous and file backed pages. Another important aspect of compressed caching we look at is how much memory is used by the compressed cache. We visit those topics in the static cache and adaptive cache size sections. We then describe some earlier results on compression algorithms and what are important aspects of choosing an algorithm for use with compressed caching. The topic of cache unfriendly applications is visited and some previous solutions to this problem are described. How compressed pages are stored in main memory has a significant impact on compressed caching and two previous works on this topic are discussed. In the last section, we also present related work such as compressed swapping.

2.1 Caching

The main purpose of caching is to mitigate the speed difference between different types of storage. It uses the fact that recently used data is most likely to be used again in the near future, and thus should be the data that is kept available.

The larger the difference in speed between two storage media, the greater the potential benefit from a cache is. This is the reason we see caches between CPU registers and main memory and main memory and disk, in which both cases there is an order of magnitude difference in speed.

The L1-L3 memory cache helps performance by keeping recently accessed main memory in faster and more expensive memory. This is to level out the impact of accessing memory instead of registers on the CPU. Another commonly known cache is the one located on hard disks, which keeps recently used blocks on the disk in memory. This is typically 8 to 32 MB of RAM in todays disks.

Both caches mentioned so far are hardware based, but there are also caches that are created by the operating system: namely the file cache. The file cache is an extension of the disk cache in many ways, it caches recently used files and directories in memory so that we do not have to access a slow disk the second time we read the same file. It can also delay writes for some time to optimize writing to disk.

2.2 Cache performance

Depending on what our goals are we decide what constitutes a good cache. The common goal is to reduce the cost of accessing data stored on another storage medium. The cost could be measured by different parameters, e.g.; if you already have two ways of accessing the Internet, one which pays per megabyte and is really fast, and a slow one with a flat rate, the answer to which one has the least cost is not a straight cut answer: If the cost is time, the fast one is the obvious choice. If the cost is money, the slow one. Inside a computer however, the performance is most often measured in time and the caches are there exclusively to reduce the amount of time spent on data stored on different and slower storage media.

2.3 Virtual memory and paging

In most of today's architectures memory is divided into units called pages. The size of a page can vary between a few fixed sizes, where the commonly used size are just referred to as pages and the larger ones, in Linux, are called ”huge” pages. The most common page size is 4096 bytes and is used for multiple architectures such as the IA32. The ”huge” pages can often have sizes of 2 or 4 MB. These pages, however, are not evicted from main memory, and are not focused upon in this thesis.

The idea behind virtual memory is that a memory address in virtual address space can map into any page. This also includes pages that are not present in main memory. Since each process has its own address space, it is protected from being modified by other processes.

A computer can not have enough physical memory to let two or more applications use the amount of memory needed to fill the address space. This is not needed, as most applications use less memory and the virtual pages used do not need to have a physical page equivalent available at all times. If there is insufficient main memory to hold all pages of the application, some can be marked as non-present and stored in alternative storage. On the next

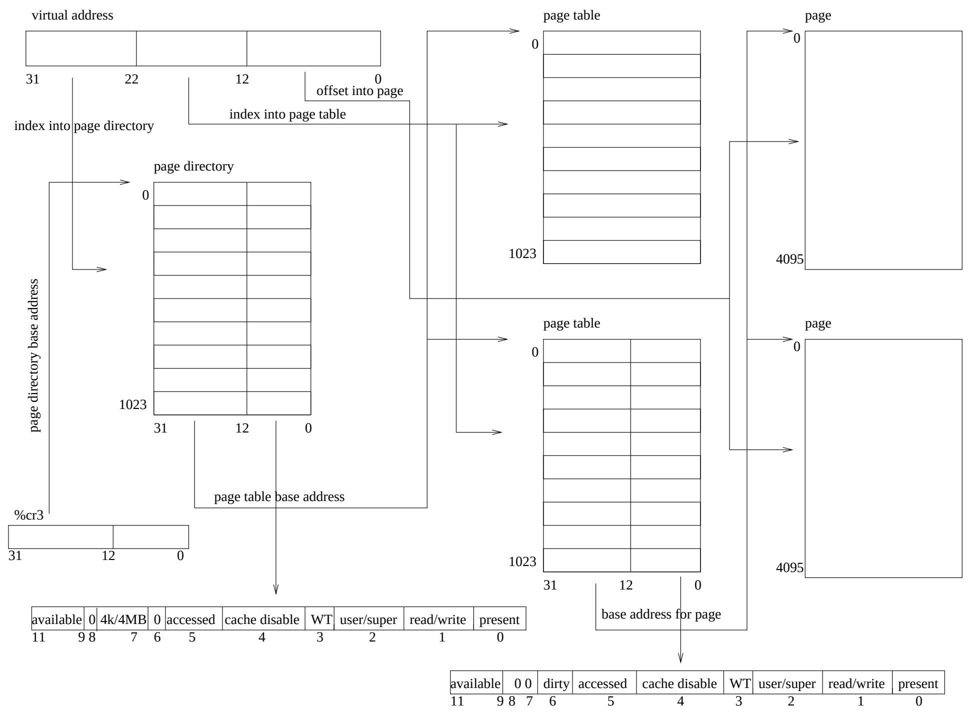

Figure 2.1: Page tables in the IA32 architecture

subsequent page access the operating system will read the page back into main memory and mark it as present.

To be able to implement virtual memory there has to be support in both the hardware and the operating system. The memory management unit (MMU) is responsible of translating memory accesses to their physical pages and causing an interrupt if there is an error condition.

On most available architectures the translation is done by looking up the virtual address in page tables to find the physical page. The page tables can consist of multiple levels.

A virtual address lookup in the IA32 architecture is done as follows (See Figure 2.1: First the MMU (memory management unit) have to find the page directory (first level), the CPU has a register dedicated for this. Then it will use the 10 most significant bits of the address to look up what page table (second level) contains more information. Then the next 10 bits are used to lookup, in the page table, what physical page the virtual address refers to. Error conditions that can happen during a lookup are access violations or pages that are not present, this error code is available when servicing the interrupt.

Segmentation is found on several CPUs, but is not used much in Linux, which is the focus of this thesis. Linux sets up the global descriptor table in such a way that both kernel code (run in ring 0 on IA32) and user space code (run in ring 3 on IA32) have the same view and maps the complete address space. If the application wants to use segmentation there is a system call, called *modify_ldt()*, provided to alter the local descriptor table. Most applications uses a flat memory model that does not directly modify segment registers.

2.4 Anonymous memory

Anonymous memory is memory that is private to the process(es) and not related to any file, or, defined in another way: memory that would be lost if the process was terminated. These are the most obvious pages to put into the compressed cache, and are those pages that usually end up in the swap area when there is memory contention. Clean anonymous pages are pages that have already been swapped out to disk, and have not been modified since, when such a page is evicted from memory it is just discarded as it would not make sense to rewrite the same data to the same place in the swap area again.

2.5 File backed pages

File backed pages are pages that belong to the file cache. They are used to cache the content of recently used files. The pages that belong to the file cache are perhaps not so easily connected with pages that should be put into the compressed cache, however Castro et al.[8] discovered that there are large benefits to be gained by compressing the file backed pages as well. Like anonymous pages, file backed pages can also be considered clean and unmodified. Most file backed pages are clean because we read files more often than we write them, although this is workload dependent.

Clean file backed pages can be discarded on eviction because they are already on disk. Compressing them will keep them longer in the file cache, making it more likely that they will be reused.

2.6 Static cache size

When creating a cache by using main memory, the question of how large the cache should be, is the first to be asked. The answer depends on several trade-offs such as how much the workload running is reusing earlier accessed pages, and how much of the working set could have been in uncompressed main memory if the cache would have been smaller.

Two cases that demonstrate what must be known about workload behavior to be able to set a static cache size: if the workload running is not reusing the pages, no cache will work efficiently, and the optimal static cache size is 0. If the workload could be running with the complete working set in memory when none of the memory is used for the cache, then the optimal size would again be 0.

The first approaches[4] tried with a static size and experimentally found the optimal size, but quickly came to realize that the optimal size varied from workload to workload. This meant that an alternative method had to be found, namely adaptive compressed caching.

2.7 Adaptive cache size

Adjusting the size of the compressed cache by looking at how the cache is being used is an area of great importance. Adapting the cache size for the current workload was proposed by Douglis et al.[4]. Later a scheme to that works was proposed and simulated by Wilson et al.[12] and then adapted and implemented by Castro et al.[8] for Linux 2.4. The adaptive scheme has been shown to work more optimally than any static allocation (with the exception of specific corner-cases) for a lot of applications.

To decide if the cache should be larger or smaller, a cost analysis is done. If the current workload would benefit from a larger cache it is enlarged, if it does not it is reduced in size. Wilson et al. [12] online cost/benefit analysis uses recent program behavior to predict near future behavior. They do this by splitting up pages in memory in three categories (see Figure 2.2): (1) the uncompressed pages available to processes, (2) the compressed pages that could have been uncompressed in memory had it not been for the pages used for the cache also called the expense list in later work[1], and (3) the compressed pages that would not have been in memory if there was no compressed cache, called the profit list.

They then collect statistics on what categories have been recently accessed and perform an analysis based on this. They compare the cost of compression and decompression of the recently accessed pages in the expense list and the potential cost of performing I/O to retrieve the recently accessed pages in the profit list. If the cost of the expense list is larger than the cost of the potential I/O of the profit list, then it would obviously be a benefit to reduce the size of the cache. If the saved cost of accessing the profit list is larger than the cost of accessing the expense list, it could be a benefit to allow the cache to be larger. They do a ”what if” analysis and try out different target sizes for the cache, choosing the one with the smallest cost.

Castro et al.[8] implementation considers the order of which the compressed cache expense and profit lists are accessed and have come up with the following heuristic: if a page has been read from the profit list, allow the cache to grow. The compressed cache will only grow when the VM tries to evict a page from main memory. After two consecutive pages accessing the

[1]Castro et al.[8] coins the term expense and profit lists

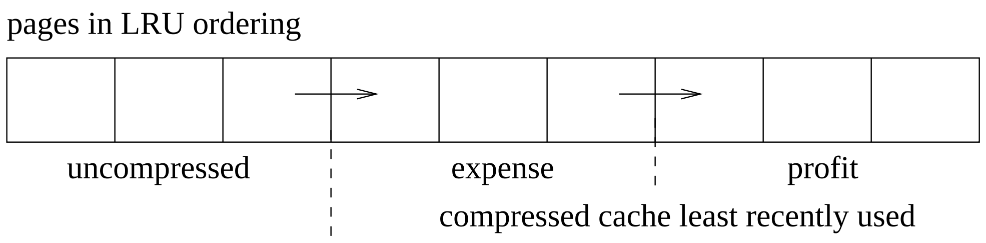

Figure 2.2: LRU list including compressed pages

expense list, the cache is not allowed to grow. When the compressed cache is not allowed to grow and there is not enough space in the cache to store a newly evicted page, one or more compressed pages in the cache have to be evicted to make room for a new one. After three consecutive accesses to the expense list, the cache is actively reduced in size by compacting wasted space in the storage, and if that is not sufficient by evicting the oldest compressed pages first.

To understand what is happening we look closer at what behavior makes the compressed cache grow and what makes the compressed cache shrink. When the profit list is often accessed, new pages are allowed in the cache. If the profit list is accessed often compared to the expense list, new pages are allowed in the cache. If the expense list is accessed more, the cache is shrunk one compressed page at a time. If there is a mixture of accesses the size of the cache is kept constant. This means that the decisions are based on very recent behavior and can quickly jump from a growing trend to a shrinking trend. However, the decisions made only affect a limited number of pages, and so a temporarily wrong decision does not affect the global trend substantially. If we compare this to the cost analysis we are doing, much of the same reasoning applies: when the current size adds to the cost it is reduced and when it is beneficial it is grown.

Tuduce et al.[11], which focuses on large data sets, uses a different approach to adapting the size. Their main concern with earlier approaches is that calculating if a page is in the profit list is a O(n) operation which will not work well with very large memory sizes. In their approach ”zones” of main memory is allocated or freed from their compressed cache in order to grow or shrink it. Their zone size is 4 MB as default, and shrinking and growing is done in such bulks. They grow and shrink the cache based on how much free space is available within the cache, if there is more than four ”zones” worth of free space a zone is freed, if there is less than one ”zone” worth of free space left a new zone is allocated and added to the cache.

2.8 Compression algorithms

To make compressed caching worthwhile, it is obvious that the cost of doing so must be less than the gain. The cost consists of meta-data overhead to describe compressed pages, CPU time to compress and decompress pages and extra number of page faults. The gain consist of less time spent waiting for data on disk. There are multiple algorithms to do compression, which one to choose for compressed caching is a balance between multiple factors: compression time, compression ratio and decompression time. Compression time plus decompression time must obviously be less than read/write I/O time, or else the use of compressed pages would be slower than just swapping them in and out. The compression ratio must be high enough so that the meta-data overhead of storing a compressed page and the content of the compressed page together is smaller than the original page, so that it actually uses less main memory.

Castro et al.[8] tested two compression algorithms: LZO and WKdm. They came to the conclusion that compression rate is more important than the speed of the algorithms. This is true for two reasons: (1) all the compression algorithms are used compress and decompress faster than accessing the data on disk, and (2) the hit ratio of the compressed cache is be higher, and as a direct consequence the profit list is larger and thus getting more of the hits.

2.9 Cache unfriendly applications

There will always be workloads that do not benefit from compressed caching, and even get a detrimental effect on their performance. What these applications have in common is that they do not reuse the pages that are put into the cache often enough to make up for the cost of compressing the pages.

Unlike most other caches there is a cost associated with putting pages into a compressed cache, namely compression and main memory usage. This means that pages that get evicted from the compressed cache without being reused only gives us a cost and no benefit. Castro et al.[8] discovered that the most common scenario where this happens is with clean pages; they limit the impact of such applications by detecting when too many clean pages are evicted from the cache versus how many are put into the cache. When such an event happens, they disable compressed caching for clean pages. To re-enable compression of clean pages they track the clean pages that are evicted without being stored in the compressed cache and enables compressed caching again when they see that enough of the recently evicted ones are restored. Castro et al.[8] performance evaluated applications that were 40.9% slower with compressed caching to 0.06% slower when disabling compression with their heuristic.Unfortunately, in their implementation this is done system-wide, which means that parallel processes that could be benefiting from compressed caching is getting affected negatively by this. Turning off compressed caching on a finer grained scale is therefore one of our goals in this thesis' implementation.

2.10 Storing compressed pages

Multiple ways of storing compressed cache has been tested in earlier works; Castro et al.[8] with their cell structure and Tuduce et al.[11] with their "zones". Their main concern is fragmentation when continually storing and freeing compressed pages of variable size.

Tuduce et al.[11] has an additional concern with systems with large amounts of memory. The "zones" data structure allocates large memory areas (4 MB) called *zones*, each *zone* is divided into *blocks*. When a compressed page is stored in the "zones" data structure it is only stored within one *zone* to make it easy to free a *zone* without consulting multiple zones. The data for the compressed page is stored in multiple blocks that are linked together. This makes sure that the most internal fragmentation that can happen is one block per compressed page.

The cell structure is described in detail in Section 4.15 and Section 5.11 and is the data structure used for this thesis implementation of compressed caching.

2.11 Compressed swapping

Compressed caching is not to be mistaken with compressed swapping even tough they share some characteristics: They both compress and decompress pages, and in some cases compressed swapping also stores them to memory. When compressed swapping is used to store pages in memory the main difference between it and a cache is that a compressed swapping does not evict pages from the swap to disk, while a compressed cache continually purges least recently used pages to disk when needed. Compressed caching can be seen as an extension of the kernels LRU lists, while compressed swapping can be seen as a fixed size area in memory.

Chapter 3

Linux kernel page handling

We are using Linux 2.6.22 which was the newest available kernel from www.kernel.org when the final implementation of the master thesis was started. In this chapter, we will look at how the Linux kernel handles pages, especially how swapping and caching is performed. In the next chapter, we alter these methods and structures to enable us to perform compressed caching.

We start out describing the basic structure of how pages are represented in the Linux kernel. We then present, on a high level, how pages are reclaimed in Linux. We then talk about what pages are part of the page cache, such as file-backed pages and anonymous pages. Next, swap and the swap cache is discussed and how it used to swap pages in and out of main memory. Virtual memory areas is then explained followed by how pages points back to them with reverse mapping. We then visit the topic of read-ahead of pages and how this is used to limit disk seeks. Next, the radix-tree data structure is discussed in detail followed by how it is used to synchronize dirty pages to disk. We then talk about what happens when a process is terminated and its related pages must be dereferenced. This is followed by a large section of how memory management is done in Linux. It starts by talking about how pages are allocated and then how smaller allocations are done on top of allocated pages in the SLAB allocator.

3.1 Page descriptor

Each page frame is described by a page descriptor; this serves multiple purposes, first of all it tells us which page it is, what it is used for and it protects against simultaneous access. The number of page descriptors to allocate is fixed and dependent on how much main memory is in the computer.

3.2 Page frame reclaiming

The PFRA[1] (Page Frame Reclaiming Algorithm) is run to free up pages to the system when there is memory pressure. It is a global page frame reclaiming algorithm, in other words it looks at whole state of the system when choosing what pages. In comparison local PFRAs only look at the pages of the current process to decide which pages to evict. Linux implements a least recently used PFRA which is implemented, by looking at non-frequently used pages and freeing them. How they are freed depends on what they represent: for anonymous user mode pages (pages not mapped to any file or block device) it will swap out the pages to a swap area; for dirty pages (pages that have been altered) containing mapping of files, block devices or inodes, it will write it back to the device. Pages that have not been altered (not dirty) can just be discarded, because they can be read in again. The PFRA is run from two contexts: (1) the PFRA is run regularly from the ”kswap” kernel thread to preempt running out of memory. (2) on-demand when trying to allocate pages when there is little or no free pages left.

3.3 File-backed pages

To achieve good performance, part of often used files are kept in memory in Linux. These pages can, if they have not been modified, be removed from main memory without being written back to the device holding them. When a page is part of a file it has an *address_space* object and an index associated with it, where the index is the position within the *address_space*. All pages that are part of the same file share the same *address_space* object, but with a different index. When reading from a file, the *address_space* object of that file's inode is consulted; it first checks if the page is already in memory by trying to look up a page using the index, if it is not then a new page is allocated and the page is read from the file system. A radix tree, which will be described in detail in a later section, is used to store and efficiently lookup files in an *address_space* object.

3.4 Swap entries

Linux manages available swap storage in swap areas, which are divided into slots where pages can be stored. Each swap area is mapped onto a block device. A swap entry consists of two parts: a swap area and the slot number within that area. A swap entry uniquely identifies a page slot and is therefore stored in the page table entries for pages which are not present in memory.

[1]This term is coined by UTLK[2] and describes the complete system involved in freeing of pages in Linux.

This information can then be used to retrieve the correct page during a fault.

The same way a page can be pointed to by several PTEs (page table entries), a swap entry can also be pointed to by several PTEs. This makes it necessary to keep track of the number of references to the entry. This is provided by *swap_duplicate()* and *swap_free()*: *swap_duplicate()* increases the reference count for the swap entry. *swap_free()* decreases the reference count for the swap entry, and if it is the last reference it also frees the swap entry. Any page which is part of the swap cache also keeps a reference to the swap entry. The reason for this is to avoid situations where a swap entry is reused before it is removed from the swap cache.

3.5 Swap cache

To avoid race conditions where a page is retrieved multiple times from swap in parallel, Linux implements a cache which lets other processes see if a swap entry is already in main memory, or is being read back into memory. To make this work all processes have to check the cache and grab a lock on the page before swapping it in or out. If the page is not in the cache, which could be the case when swapping in a page, a new page is allocated and put into the cache before doing the actual read, so that other processes can find the page. The cache is an *address_space* object and the storage, manipulation and lookup of swap entries are implemented as a radix tree in the swap cache. All pages in main memory that have a non-present swap entry in any page table is part of the swap cache. More about the radix-tree data structure can be found in later sections.

3.6 Swapping in a page

After a page fault the kernel identifies a page as a swapped out page by looking at the PTE of the faulting address. The PTE will contain a swap area identifier and a page slot number within that swap area. The kernel will then invoke *do_swap_page()* which tries to look up the page in the swap cache (by using the swap area and slot number) and if necessary read the page back from swap. If a page was in the cache it is marked as active and a reference is taken for it. It then updates the PTE before returning to user space. It is worth noting that when a swap cache hit is detected (another process is swapping in the page) it is considered a MINOR fault, while having to swap in the page itself is considered a MAJOR fault.

3.7 Swapping out a page

The page frame reclaiming algorithm chooses which pages to be reclaimed and calls the swapping subsystem to swap them out. The swap subsystem first adds the page to the swap cache to keep other processes from swapping it in from an uncompleted write. It then updates all PTEs mapping the page with the identifier of where the page can be found (swap area and slot number). It finally writes the page out to the swap area, removes the page from the swap cache and frees the page.

3.8 Virtual memory areas

VMAs (Virtual memory areas) are non-overlapping linear memory regions in a process. The *vm_area_structure* is used to describe what a region contains, the attributes and access permissions for the page frames and the extent of the region. For VMAs containing anonymous memory the behavior of the region is also described, for example it may be known that it will be accessed in a sequential or random pattern or that it is allowed to expand. For VMAs mapping in files or part of files, each VMA describes what file object is used, the offset into it and the start address and end address of the region. This information is available during page faults, which is needed to satisfy the request.

3.9 Reverse mapping

An anonymous page can be part of several virtual memory areas if a memory area is shared by multiple processes or if the same memory area is mapped into the same processes at several locations. When removing such pages from the page cache all the references to a page must be found and altered. To do this, Linux uses reverse mapping: since VMAs can come and go, there is a separate structure called *anon_vma* which keeps a list of VMAs that can share pages. Each shared anonymous page will point to this structure so that only the VMAs in the list need to be checked to see if they contain present references in their page tables. The Figure 3.1 shows the relation between pages, virtual memory areas and the *anon_vma* structure.

3.10 Read-ahead

Linux supports read-ahead for both files and pages stored in swap. The idea is to utilize sequential reads from hard disks, which is known to be more efficient than multiple single I/O requests. To do this Linux guesses which pages are going to be read in next. This is implemented by appending

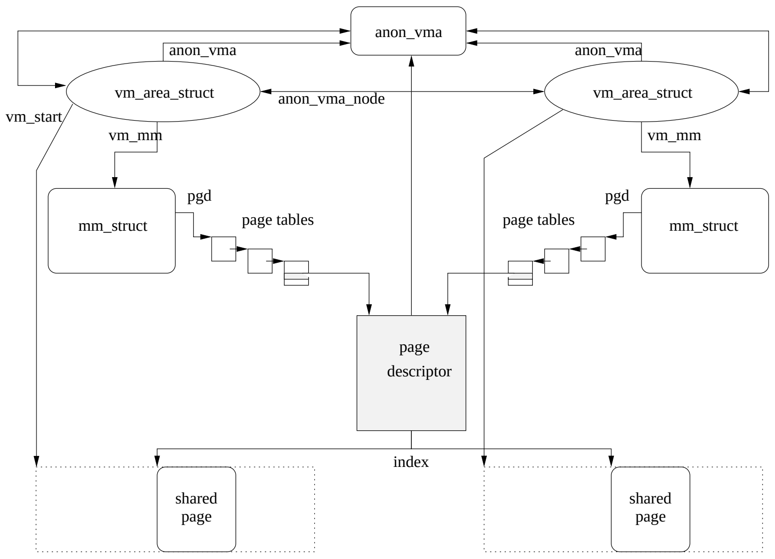

Figure 3.1: Reverse mapping for anonymous pages[2]

adjacent pages to the original request so that they can be read from disk at the same time, saving precious seek time.

For swapped out pages this means unrelated pages can be swapped in, which is why it is limited to pages that do not introduce any seeking. When a swap entry is requested to be swapped in, other nearby pages are added to the request by increasing the slot number of the original swap entry to create a new one. This new swap entry is then checked to see if it is valid. If it is and no corresponding page is in memory the entry is added to the original request. New swap entries are generated until an invalid entry is generated or the generated entry is already available in the swap cache.

For file-backed pages the read-ahead scheme is a bit more extensive; pages read ahead can include pages that will need a seek depending on how large the read-ahead window for that particular file is. The size of the read-ahead window is decided by inspecting earlier behavior of that file descriptor, if the file is used randomly the read-ahead can be disabled completely.

3.11 Radix tree

A radix tree is a commonly used search tree in Linux. Given an index, the radix tree can return a pointer. Or given a tag it can return groups of pointers belonging to that tag. Each node in a tree contains an array of 64 pointers, called slots, and a bitmap of length 64 for each possible tag. In other words each node has 64 pointers coupled with bits describing if each pointer belongs to a tag or not. The height of the tree is dependent on the largest value of an existent index and $2^{height*base}$ gives the largest possible index the tree can accommodate without making it taller. The base is the number of bits used as an offset into the slots and also decides how many slots per node there is. The base is 6 as default for the Linux implementation.

To do a lookup in the tree we take the 6 least significant bits of the index and use these as the offset into the array of the root node and follow that pointer to the next level of nodes. We then look at the next 6 bits and use that to find the next node and so on. To terminate the look up, we check if the least significant bit of the pointer is set, if it is, it is a direct pointer and instead of looking at the next node, the pointer itself is returned with that bit reset. This gives us the limitation of only being able to store pointers with the last bit set to 0, in other words aligned to a two byte boundary.

If we follow Figure 3.2 trying to lookup a single pointer we start out with an index of 32 bits and a tree with height 3. The height of the tree tells us that the largest index in the tree is at-least smaller than 2^{18} or else the tree would need to be higher. It uses the bits 0-5 to lookup the slots of the root node. It finds a pointer that is not a direct pointer and follows that to the next node. It then uses the bits 6-11 as the offset into this new nodes slots. It again finds that is an indirect pointer and follows it to the next node, which has height 1, uses the bits 12-18 and finds the corresponding slot containing a direct pointer. Finally it resets the direct pointer bit and the result is returned.

We can also follow a lookup for dirty pages with Figure 3.2: we look through the dirty tag bitmap and find that the first entry in the bitmap is set. Since it is not a direct pointer we follow the pointer to the next node. We look at the new dirty tag bitmap, finding the third entry set. Again the pointer is followed to the last node where one slot is found with the dirty tag set. If we are doing a range lookup, called a gang lookup, one would continue looking through the bitmap for more slots with the dirty bit set (none in Figure 3.2 on the facing page). Then go up to the previous node and continue looking there and so on until enough slots have been found, or the root node dirty tag bitmap has been fully examined.

3.12 Synchronization of dirty pages

To improve system performance dirty pages are generally not written directly to disk. This is to take advantage of other modifications to the same, or closely related pages happening within a short time-frame. This results in there being a considerable amount of dirty pages spread across files and file systems on the system. For file-backed pages that are dirty it is important to have a mechanism that finds them and synchronizes them to disk. The reason for this is to keep data from being too out of sync on disk compared to in memory. There are two instances where this happens: the explicit

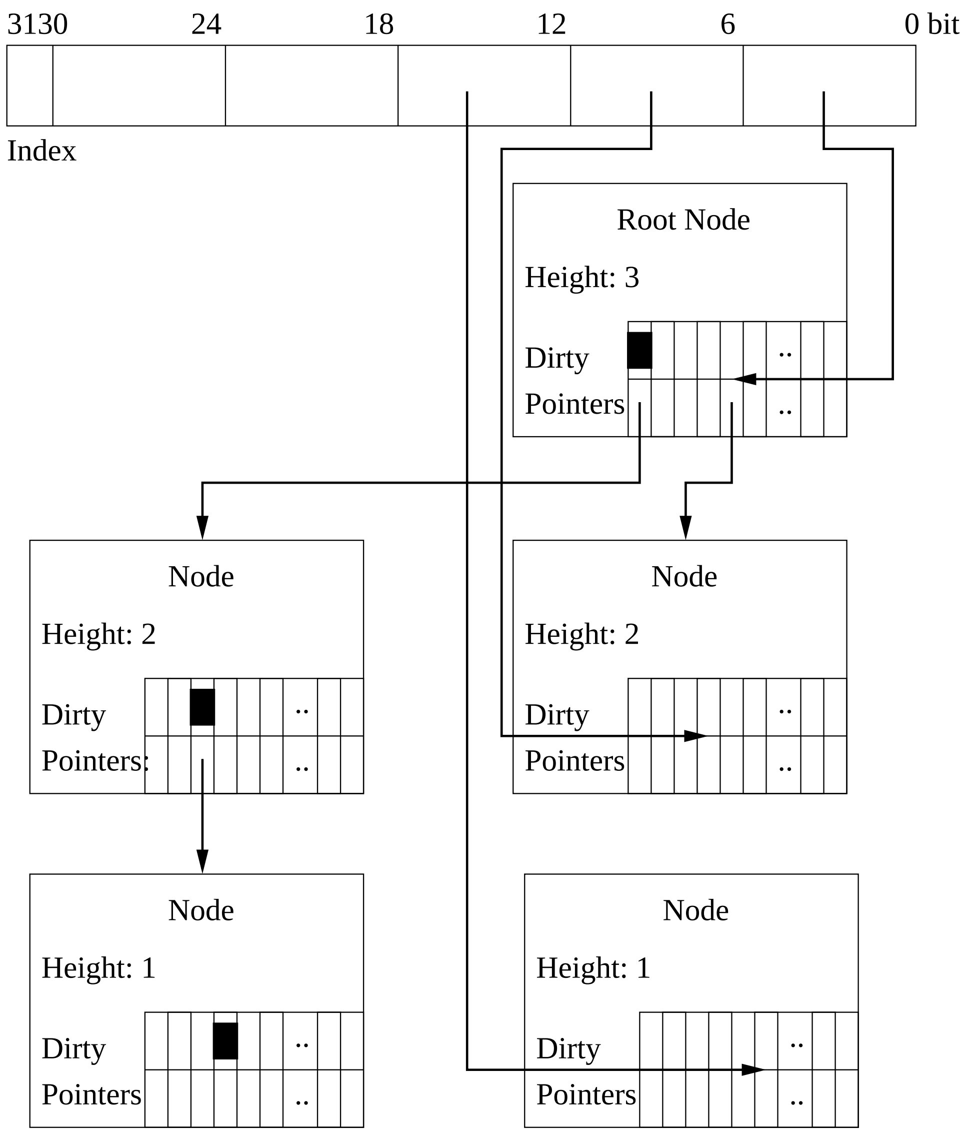

Figure 3.2: A radix-tree with height 3 and tag support.

synchronization where an application asks the kernel to make sure the data so far has been written to disk, and the implicit synchronizing which happens regularly to avoid loosing too much work on hardware failure or power outages.

For file specific explicit synchronizing where only one file is asked to be synchronized, the *address_space* object of that file is consulted and the tag feature of the radix tree (see Section 3.11) is used to easily find every dirty page.

To synchronize the complete system, every super-block must be consulted and all of their dirty inodes must be inspected. If it is an implicit synchronization, how long since the inode was dirtied is checked and very young dirty nodes are skipped. For each inode that is going to be synchronized, the tag feature of the radix tree is used to efficiently to find those pages. Implicit synchronizing happens at a configurable interval, and is typically set to a few seconds. The default limit of how old a dirty page can be before it is forced to be synchronized is 30 seconds.

3.13 Page tables

Linux provides a unified view of page tables by dividing it up into four levels: page global directory, page upper directory, page middle directory and page tables. They all contain pointers to the level below; the PGD (page global directory) contains pointers to the PUD (page upper directory), and PUD to the PMD (page middle directory) which finally points to the page table. For each level of directory a number of bits used per level is set, this is the number of bits used as an index into the corresponding directories. For 32 bit IA32 architecture, which only has a two level page table, the number of bits for PUD and PMD are set to 0. This effectively disables the PUD and PMD and the compiler will in most cases efficiently compile the corresponding code away completely.

3.14 Cleaning up after a process

When a process terminates, all resources referenced by it must be dereferenced and sometimes freed. Virtual memory areas and their pages and swap entries are dereferenced. This is done by going through every virtual memory area and all of their pages by exhaustively following page upper directory through page middle directory and finally the page tables looking for both present and non-present pages.

When non-present pages are found that have a swap entry, the reference of that swap entry must be freed. This is done in *free_swap_and_cache()* where it is checked if the swap entry is the last reference and if the page it represents is in the swap cache. If it is the last entry, it is made available

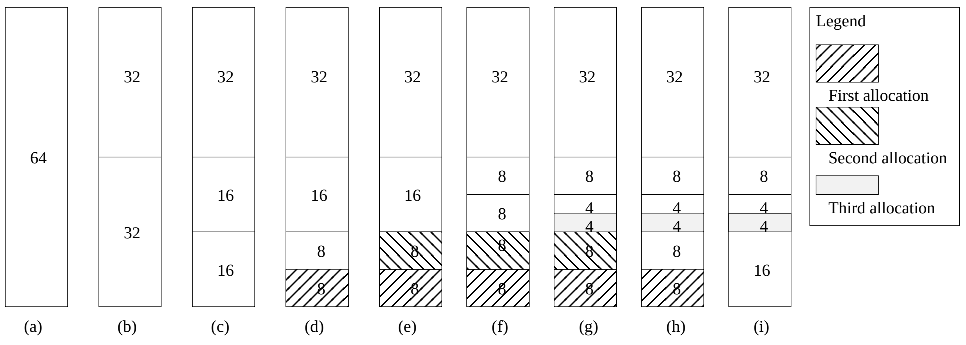

Figure 3.3: Shows different stages of page allocations, [10]

for allocation. If the page is in main memory it is removed from the swap cache and then freed.

While doing this, locks are taken for the MM (memory management) structures and the local TLB (the look-aside buffer) lock is taken. To avoid all the locking the virtual memory areas are freed in bulk after MM and TLB locks are unlocked.

3.15 Memory management

Linux uses the buddy algorithm to manage physical pages of memory. The buddy allocation algorithm works accordingly: round up the requested allocation size to the next power of two, then find the smallest free piece which is large enough for the allocation request. If that piece is not equal in size to the rounded up allocation, divide the free piece into two. If it still is not the same size, continue to divide until this is true. When freeing a piece check if the neighboring piece, a buddy, is free also. If it is, merge with it to create a free piece with twice the size.

An example of how this works, taken from Tanenbaum et al.[10] goes as follows (See Figure 3.3): say you have 64 contiguous free pages making out the initial free piece, and you get a request for a 6 page allocation; you first round up that allocation to the nearest power of two, in this case we get 8. The first and only free piece we find is 64 pages long (a). It is larger than the allocation of 8 we need, so we divide it into two 32 pages long (b) pieces and look at one of them. 32 is still larger than 8, so we divide it again to 16 (c), and then to 8 where we now have the size we need and allocate it (d). When a second allocation is requested, that is rounded up to 8 pages, it is clear that it can be provided directly (e). On a third allocation of 4 pages, the smallest piece we find is 8 pages, so we need to divide it again resulting in two 4 page pieces where we can use one for our allocation (g). Next we see what happens when we free an earlier allocated piece: first the second allocated piece is freed (h), it looks at its buddy, but sees that it is still in use. Then the first allocated piece is freed, when it looks at its buddy and sees that it is free it merges the two buddies resulting in a free 16 page piece (i).

This scheme however results in internal fragmentation for allocations that are not the power of two, wasting memory. To avoid this Linux makes the unused part of an allocation, called a slab, available to the SLAB allocator. For example an allocation request for 5 pages, rounded up the actual size of the allocated piece would be 8 pages, where there would be a slab of 3 pages.

The SLAB allocator is based on the observation that some fixed size data structures are allocated, initialized and deallocated repeatedly. In other words when we have the same type of fixed size data structure, such actions could be optimized in two orthogonal ways: (1) most importantly we can keep the freed allocated area intact until an allocation is requested for the exact same size. This lets us pack such data structures tight without any internal fragmentation. (2) only initialize a data structure if the memory area allocated has not been used for such a data structure before. For this to work the data structure must alaways be in a consistent state on free.

The memory areas allocated by the SLAB allocator are called objects and are grouped into caches as seen in Figure 3.4. Each cache descriptor describes what kind of object is in the cache with attributes such as the size of the objects, the needed alignment of the objects and so on. Additionally the cache descriptor keeps track of slabs belonging to the cache and free and allocated objects.

Each cache is private to its object type: reusing the same cache descriptor for different types of objects of the same size is not recommended. It is therefore important to know when to use it and when not to. For fixed non-dynamic sized data structures that are rapidly freed and allocated it is a good solution to make a private cache. The interface for such allocation and deallocation is *kmem_cache_alloc()* and *kmem_cache_free()* where the cache must be specified to return the correct type of object. Note that we do not need to specify the size or the alignment requirements of the object, because this is already described by the cache descriptor.

To handle requests that are rarely asked for, a set of fixed size caches have been made available. Such requests need to be rounded up to the nearest size available. In Linux these sizes are part of the power of two series starting from 32 (2^5) and ending with 4194304 (2^{22}). This however often results in internal fragmentation, but always less than 50%. The 50% guarantee is a natural result of the power series; we will never round up an allocation request to be more than twice the size since this would mean that the request was already a power of two, and no rounding up should be needed. The interface for such allocations are *kmalloc()* and *kfree()* without

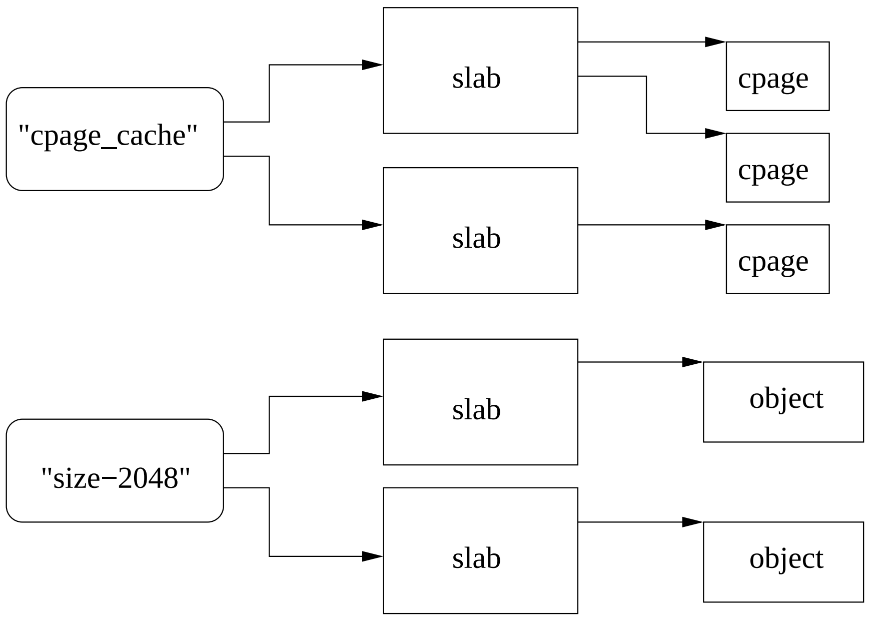

Figure 3.4: Cache, slab and object interaction. Shows two caches, the cpage_cache is a cache for the struct cpage, which is fixed size. The cpages are allocated using kmem_cache_alloc(). The other cache "size-2048" is one of the caches used for rare non-fixed size allocations and includes objects of sizes between 1024 and 2048 bytes long, allocated with *kmalloc()*.

a reference to the cache to be used, as this is inferred by the size of the allocation.

Chapter 4

Design

We looked at how Linux works in the previous chapter, and focus now on how we can alter and make a design that will achieve our goals. In the next chapter we will look at the implementation details of this design. The intention is to use an adaptive cache size in the same way as Castro et al. [8] with an expense and profit list. The policies for turning on and off the compressed cache follows the same reasoning, but is extended to be more fine-grained.

First we set a level of abstraction for how the compressed cache shall be implemented. Since we are going to compress pages, we describe what is needed to represent them in memory. Next step is how to intercept page accesses and what Linux subsystems we can use for this purpose. We then go into how pages are inserted into the compressed cache followed by how the size of the cache size is decided and manipulated. We then describe in what circumstances compressed pages are decompressed. In the next section, we give an account of when we want to disable compressed caching, why and what kind of information we need to do so. We continue with known corner cases of the design followed by some notes on what needs to be changed to make read-ahead still work. We briefly note that memory usage of the compressed cache must be accounted for to accurately update the profit list. Next, we go through problems with storing compressed pages and what behavior is expected of them, followed by a detailed design of the cell data structure.

4.1 Level of abstraction

To avoid detailed architecture dependent code, choosing an appropriate level of abstraction to implement our goals is important. If we work directly with page tables and their entries (IA32 architecture) or similar low-level architectural features, we would end up implementing the same thing for each and every architecture which supports virtual memory in Linux. Fortunately, by looking at how page handling is done, we found that by intercepting swap-outs (for anonymous memory) and page removals (for mapped pages) at their cache lookup, we can have a natural architectural-neutral design. OLPC's[1] compressed cache uses this and is the inspiration for doing it this way.

4.2 Compressed page representation

Linux already has a page descriptor for in-memory pages, which has a lot of attributes and features that are not needed when a page is compressed. The page descriptors are also preallocated to the number of pages present in the system, where the number of compressed pages can vary. For these reasons, we need to make our own lightweight page structure for compressed pages which only include the needed information (struct *cpage*).

The *cpage* structure should have an LRU order and provide features enabling us to do the calculations needed to evaluate the size of the compressed cache. It should also tell us where and how the compressed page data is stored.

4.3 Interception of compressed pages

We introduce the compressed cache as an extension of the swap cache and file cache. By looking at the mechanism of how those caches are used, we want to alter them in such a way that compressed pages can be stored in the cache. This means that we have to alter the code paths used to access the page caches in such a way that when we encounter a compressed page, it is transparently uncompressed and replaced by the uncompressed page in the page cache.

The main data structure used by the page caches is the radix tree (Section 3.11) and the page descriptor. In our implementation of compressed caching the compressed cache is an exclusive cache, so that there can logically only exist either a compressed page or an uncompressed page in the cache for a particular entry.

We notice two things about the data structures: (1) all the page descriptors are located on at least four byte boundaries. (2) the radix tree uses the last bit of the stored pointers internally. This means that at least the two last bits of a page descriptor pointer will be 0 and that we can not use the last bit. If we make sure the compressed page descriptors are also aligned on four byte boundaries we can use the second bit to differentiate between compressed and uncompressed pages. This makes it possible to store both

[1]Nitin Gupta has implemented a compressed swap area, called compressed cache sponsored partly by OLPC.

compressed and uncompressed pages mixed in the cache, and uncompress transparently if we alter all code paths accessing the caches.

The swap cache is used to identify wheter a swap entry has the corresponding page in memory or not. Usually, when a cache hit occurs a page descriptor is returned. A modified version needs to check if it is a compressed or uncompressed page and then decide what to do. In the straightforward case, we want to return a page that can be used. In other words, if we find a compressed page, we allocate a new page, decompress the compressed page into it and return an uncompressed page. Under other circumstances, the lookup is only done to see if a page is in the cache. When doing such lookups, it could be a waste to decompress a compressed page that will not be used.

Both the file cache and the swap cache uses the *address_space* objects to lookup if a page is already in memory. The swap cache uses one object for all the swap areas with the swap entry as the unique identification of pages. The file cache uses one *address_space* object per file and the offset into the file modulo page size as the unique page identifier. The same reasoning altering the cache lookup functions apply to the file cache and swap cache.

4.4 Adding compressed pages

Compressed caching is an extension of the normal LRU implementation of the Linux kernel, and tries not to affect the order of which pages are removed from the page caches. We could fetch pages back from swap to grow the cache, but to get a clear benefit from this we would need to track pages not residing in memory and it could be a quite expensive waste of resources with meta-data overhead. Instead we alter the PFRA to give us pages to compress where it would normally remove the oldest pages from the cache. We let the PFRA add more and more pages until the compressed cache can not or is not allowed to grow anymore, we then start paging out the oldest compressed pages to leave it at a constant size. If the PFRA fails to insert pages into the compressed cache, the pages are paged out in the same way as usual.

4.5 Adaptive cache size

We divide pages in the compressed cache into two lists. The expense list consists of pages that would have been uncompressed in memory if there were no compressed cache. The profit list consists of pages that would not be in memory if there were no compressed cache.

Accessing pages in the expense list is a cost (not only because of the name) because they would be available without any cost if there were no compressed cache. Now they represent two costs: compression and decompression. Accesses to profit list is beneficial as long as I/O is slower than compression and decompression combined. This is a requirement for compressed caching to work.

When a page is put into the compressed cache LRU list, it should be marked as an expense page. When the position of a page is larger than the pages used for compressed caching, it is moved to the profit list and marked as such. This means we have to keep track of how many compressed pages are in the expense list when adding and removing compressed pages from it.

In an early attempt to implement adaptive cache size in this thesis, we tried to follow Wilson et al.[12] closer, but ran into problems with how to time the block I/O subsystem in the kernel. More on this can be read in Appendix A.

Working from Castro et al.'s[8] adaptive compressed cache resizing we want to implement the same scheme: a lot of accesses to the profit list should result in a larger compressed cache, and a lot of accesses to the expense list should result in a smaller cache. This is done by allowing the cache to grow when there are concurrent accesses to the profit list, and shrinking it by evicting the oldest page on each access to the expense list after two consecutive accesses. Accesses to the lists should only be counted when the pages are going to be used. In other words, pages accessed because they are removed from the compressed cache should not count as an access to either of the lists.

4.6 Shrinking the compressed cache

If the compressed cache size is larger than the size we want, we need to discard compressed pages from our cache. We remove pages from the compressed cache in LRU order, removing the oldest compressed page first. If we need to make the compressed cache smaller to make room for a new compressed page, we continue to remove the least recently used page until there is enough room. When removing a compressed page to shrink the cache, we note that we are removing an unused page from the cache belonging to a specific memory area, which may be a file or some anonymous region. Shrinking also means that the size of the expense list becomes smaller, so we have to mark the pages earlier marked as expense ones to be profitable ones.

4.7 Decompressing pages

There are two places where an application can cause decompression of a page: (1) when accessing an anonymous memory area where the page is compressed, and (2) when reading from a part of a file that has compressed pages. If an application tries to access an anonymous page which is compressed, it will cause a page fault, which in turn will try to swap in the non-present page. The swap-in routine will consult the swap cache, which will contain the compressed page (*struct cpage*) instead of a normal page. When a compressed page is found, a new page is allocated and the compressed page is decompressed into that. The swap cache is then updated to include the real page instead of the compressed one. The compressed page is then freed from the compressed cache.

If a compressed page is discovered when accessing a file, similar action to that of anonymous memory happens, except that instead of a swap cache's *address_space* object we are accessing the file's *address_space* object.

Decompression will also occur when removing dirty compressed pages from the compressed cache LRU. This happens while shrinking the cache to make room for newly compressed pages. Only dirty compressed pages result in decompression since the clean pages can be freed without being written back to disk. To avoid handling all the I/O directly in the compressed cache subsystem, we instead mark pages that need to be written back as ex-compressed pages and reinsert them at the end of the kernel LRU list; This leaves the I/O to the standard PFRA functions.

4.8 Cleaning up after a process

When implementing compressed caching there are several issues we need to consider regarding the termination of a process. The most notable is that when we are freeing compressed pages, the process could end up sleeping. Unfortunately, all the code paths for cleaning up after a process termination disables preemption before running the code paths touched upon by compressed caching. To solve this, the arguments in the call path should be changed to include a list, where compressed pages can be added. The compressed pages in the list can later, when preemption is no longer disabled, be freed.

Another issue is the fact that if we do not add a special case for the cache lookups of non-present pages, we could end up decompressing a compressed page that we will be removed without being used. To avoid this, we make alternative cache lookup functions that just return the compressed page struct to be freed. This should be implemented by altering *free_swap_and_cache()*.

4.9 Truncating and deleting files

When a file is truncated or deleted all the pages containing contents from the file must be found and freed. This is done by using the page cache interface function *find_get_pages()* which in turn do a range lookup against in the radix-tree belonging to the *address_space object* of the file. If we continue to use *find_get_pages()* all the compressed pages will be decompressed before being freed, which is a waste of resources. To get around this limitation we introduce a new function, called *find_get_pages_cc()*, that deletes all compressed pages it finds in a range, and returns the rest.

4.10 Disabling of compressed caching

There are several cases where compressed caching does not work very well, and where disabling the compressed cache would be beneficial. In earlier work, such as Castro et al.[8], it was found that disabling the compressed cache when a lot of unused pages where removed from the cache without being used, reduced the overhead of compressed caching considerable for some workloads.

Consider two applications running at the same time, one being a perfect candidate for compressed caching and the other being the worst case application. If we only look at the output from the cache, we could end up disabling the compressed cache for both applications. In this thesis, we try to avoid that by dividing the memory into different areas and observe and adapt the behavior of those independently.

We look at one anonymous memory region as one memory area and a file as another, and accumulate statistics on recent behavior per memory area, instead of per process or globally. This makes sense in several ways. First of all, files are hard to account to any one process because they can be used by several processes. In addition, if one process uses several files, some files' access patterns may match compressed caching perfectly while the others have a worst-case pattern. Mixing their statistics will make it harder to notice one behavior over the other. The same applies to anonymous memory, where memory can be shared among several processes, or, where one area of the application is accessed in a compressed caching friendly way and another is not.

We try to detect areas that are not benefiting from compressed caching because they are reusing too few pages compared to what is being put into the compressed cache. We call this bad ratio and we will go into more detail on that in the next section.

Another problem we could run into is uncompressible data that will only waste CPU resources. If a memory area has high entropy we want to detect this to stop wasting time trying to compress it. We talk more about that in Section 4.10.3.

4.10.1 Bad ratio

Design of fine-grained clean page disabling for file pages is an extension of global clean page disabling by Castro et al.[8]. Instead of doing it on a global scale for the complete system, we do it for each memory area.

We store recent history of used and unused pages that have gone through the cache for the specific area. Used pages are those that are decompressed and accessed, while unused are those that are not used (other than to be written to disk) after exiting the compressed cache. When the ratio of unused pages is higher than a specified threshold, the compression is disabled, and later evicted pages from that area will only be present in the compressed cache as a marker, used to continuously reevaluate a potential ratio of used pages. When this ratio is above a certain threshold, the compressed cache is again enabled for that area.

An issue that could arise when compressed caching is disabled for an area, is that the area keeps getting markers in the compressed cache at such a high rate that it pushes out actual compressed pages. This means that we have only reduced the problem of cache unfriendly areas. To remedy this, we will limit how many markers for a particular memory area are allowed to be in the compressed cache at the same time. This means that when we are removing pages from the compressed cache, we need to account this information into the statistics for the memory area.

Another corner case that can happen with bad hit ratio detection, is when only one memory area is getting pages into the cache, and this area is accessing the pages in a compressed caching unfriendly way. Consider a process that continuously reads from a file that is larger than the compressed cache can store and that the process starts to read from the beginning of the file when it reaches the end of the file. First, the memory area would go into an "early phase" (described in the next section), and a limited number of pages from the file will be put into the compressed cache. Since no other processes are pushing pages through the compressed cache, those pages will stay in the cache. After a while, the process starts reading the file from the beginning again, finding pages in the compressed cache. Now the cache hit ratio is good enough to put more pages into the compressed cache, and more pages are put into the cache until they are removed from the end of the cache. This means that the cache will compress a lot of pages that it will not reuse. We already have a solution ready for this called the "early phase", so instead of going back to the normal state of a memory area when a good enough ratio is detected, it will go back to the "early phase".

4.10.2 Early bad ratio

Waiting for clean pages to be pumped through the LRU before deciding to disable compressed caching for one area can prove fatal to performance, and may in turn falsely detect other areas as compression cache unfriendly.

When a new area gets pages into the compressed cache for the first time, we will have an unspecified ratio of used and unused pages. We can either go to a default pessimistic state, where compressed caching is disabled right away, or a very optimistic one, where we say the ratio is good until we have some compressed pages that have been accessed or evicted from the compressed cache. The pessimistic version could end up always delaying the enabling of compressed caching for small files, resulting in lost potential performance gains. The optimistic version could result in a complete trashing of the compressed cache, before it is determined that compressed caching should be disabled. The middle way is to limit how many pages a memory area can put into the cache before the first removal by either an access or eviction of a compressed page. We call this the *"early bad ratio"* detection and the "early phase".

This could give a penalty to processes in their early stages, but will help the overall system performance. It does not, however, protect the compressed cache against memory areas first being accessed in a certain way, and later another.

4.10.3 Bad compression

Bad compression for certain files with high entropy is unavoidable, but it could result in trying to compress pages that can not be compressed enough to be put into the compressed cache. To avoid this scenario, we could stop compression for areas that are known to give us bad compression ratios. This would cause memory areas that change their behavior throughout their lifetimes to never recover from a limited period of bad compression. Instead we do two things: (1) we do not go into bad compression state before two consecutive pages have been tried and found uncompressible, and (2) when compressed caching is disabled due to bad compression we skip the next couple of pages for that area before retrying compression. If all pages are uncompressable for an area it will skip more and more pages. This gives a moderately conservative approach to disabling compressed caching while the memory area still gets chances to test the compression ratio to re-enable compressed caching. A memory area that repeatably have bad compression will be tested less and less and will not waste CPU resources.

4.10.4 Keeping statistics

To calculate the hit ratio of the compressed cache for a certain area, we need to monitor the accesses to the cache. If a compressed page is accessed to be evicted from main memory, we note this as an unused page. If it is accessed to be reused by a process, we note this as a used page. When we have disabled compressed caching for an area, we need to note the reason and how many markers we have put into the compressed cache. This is to know how to handle further requests to store pages in the compressed cache. Since this information is identical between anonymous memory and file-backed pages, we use a common structure for this called *cc_areastats*.

We add these both to the *address_space* and *anon_vma* structure.

Bad ratio, early bad ratio and bad compression are all mutually exclusive, in other words, only one of them is active for any given memory area at any time. The order of preference is bad compression disabling followed by early warning and bad ratio disabling. For early warning and clean page compression disabling, it is important to note that statistics such as used and unused pages are always maintained. When the compressed cache is disabled for an area, it may not be completely exact, but it is representable. While in a bad compression phase, the *used* and *unused* attributes of a memory area are used to calculate when the next attempt at compression should occur for the mapping.

4.10.5 address_space objects

We turn off compressed caching for files with consistently bad compression rates and for files having a bad hit ratio in the compressed cache. A typical example that would exhibit such behavior is large media files being played or recorded. When we disable compressed caching for such areas we free up space and CPU resources for other contents.

For file-backed pages, differentiation between applications does not really make any sense because multiple applications can have the same file open. What we can do is to differentiate on a file-to-file basis.

To detect those scenarios some statistics must be stored per file, which means that we have to extend the *address_space* structure with the *cc_areastats* structure.

4.10.6 anon_vma

The reverse *mapping* provides us with means we need to find out what processes a page is used in. Since multiple processes may share anonymous memory, keeping track for each application would be a O(n) operation. Doing similar for each anonymous area (*anon_vma*) would not only be O(1), but also give us finer granularity, enabling us to have a policy (turn off or on compressed caching) per anonymous memory area.

4.11 Boarderline cases

A problem that can occur with the current design is when we measure a lot of really bad cache performance. What would happen then is that we would have no reason to reassess the benefit of compressed caching because the max cache size would obviously be near zero, resulting in no pages being compressed. To get around this, we set a minimum cache size to at least enable us to get some data on the performance.

Few accesses to the compressed cache can make the adaptive resizing of the cache fail in ways that makes the cache grow or shrink unbounded. If the compressed cache is accessed in a way that allows it to grow, and then not accessed again for a long time, it will continue to grow until it can not allocate more memory. A similar scenario can happen when the compressed cache is locked at a small size and then not accessed for a while. This could result in very few pages being part of the cache. These scenarios do not happen very often because recently used pages are often reused again.

4.12 Read-ahead

Read-ahead for anonymous pages in Linux accesses the swap cache through the cache functions which are used to intercept compressed pages. This would end up in decompressing compressed pages that are not going to be used. To avoid this, the cache should only be consulted. We achieve this by making a distinct function that will be used when we do not want to uncompress pages. However, the cache may contain marker pages which is not to be counted as present in the cache, so the function must also be aware of this and remove the marker if needed.

A similar problem exists with read-ahead for file-backed pages. Since the lookup is optimized and uses the radix-tree interface directly, it has to be modified to be aware of marker pages.

4.13 Compression algorithm

We copy the Lempel-Ziv-Oberhumer (LZO) implementation made available in newer kernels into our own for compressing and decompressing pages. We also added Lempel-Ziv-Free (LZF), which is said to be faster for compression, to be able to compare them in a compressed caching scenario.

4.14 Storing compressed pages

How compressed pages are stored is a very important issue with compressed caching. We have two types of data that we want to store, the fixed size of the compressed page descriptor (*cpage*) and the variably sized compressed data. There are different approaches for allocating fixed sized allocations and variable sized allocations. For frequent fixed sized and rare variable sized allocations, the SLAB allocator is used, which is a good match for the compressed page descriptors. For the compressed data that can vary much in size, using the SLAB allocator would result in severe internal fragmentation. Internal fragmentation can cause the compression itself to be a wasted effort. A better solution would be the Cell allocator by Castro et al.[8], which can compact the allocations when there is internal fragmentation.

An issue related to the storage of compressed pages is the size of the underlying data structures. If a data structure requires more than one page, and those pages need to be physically allocated next to each other, there is a high probability that such free pages do not exist. The larger the allocation needed, the less likely it is that such an allocation request can be fulfilled. As Castro et al.[8] points out, when storing compressed pages, the best tradeoff is two consecutive pages. This makes it possible to store three pages compressed to two thirds of a page size.

Every time we allocate or deallocate memory for compressed caching, we should note how many pages we are using. This is to keep the expense and profit lists correct for adaptive caching. Part of this can be implemented in the storage data structure, where number of pages used and number of allocations should be counted.

4.15 The cell memory allocator

The impact of internal fragmentation is much higher for compressed caching than other allocations for two reasons. The first one is that it directly impacts the performance of the cache. Leaving up to half of an allocated area unused could result in the compression practically becoming zero. The second reason has to do with how frequently such non-fixed size allocations happen. Where normal allocations tend to be infrequent, while compressed caching continually allocate and deallocate memory areas for compressed pages.

The SLAB allocator used in Linux today focuses heavily on the optimization of fixed sized allocations, and reverts to a power of two scheme for dynamic allocations. This is because the allocator only has a fixed number of different caches, that keeps ranges based on a geometrical distribution. This is in order to keep approximately the same amount of allocations in each cache and keep internal fragmentation low. The most interesting available sizes to look at are the sizes smaller than and equal to the page size, because this is what we are going to compress to. The sizes available in Linux are 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192[2]. This means that if we compress a page of 4096 bytes to 600 bytes, we would need to use an allocation of 1024 bytes, a 41.4 % internal fragmentation, and reduces the compression ratio from 6.83 to 4.0. In another example where we compress a page of 4096 bytes to 2100 bytes, the compression ratio would effectively be 1, which would actually be a net loss considering the meta-data overhead that results from representing the compressed page.

When we look closely at our usage of allocations and deallocations in compressed caching we find that we always write once after compression

[2]The size of a page can be different on other architectures, on IA32 the most common is 4096 bytes.

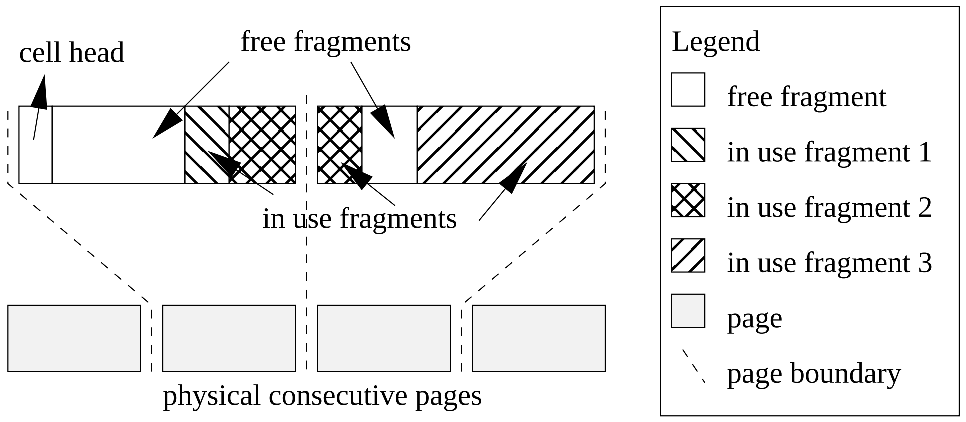

Figure 4.1: When used with compressed caching, each fragment would be a compressed page. Even free space (the blank squares) are represented by fragments. The meta-data of a cell is in the beginning of the first page allocated for the cell.

and read once for decompression. This gives us the opportunity to use a more strict special purpose interface compared to the general *kmalloc()* and *kfree()* interface of the SLAB allocator.

The cell data structure in our implementation is based loosely on the description of Castro et al.[8] cell data structure. The main problems the cell data structure is trying to solve is the waste of space that occurs with the buddy allocator and the fragmentation issue.

The main design goals of the cell data structure are to be space efficient when storing, and let us move allocations around to enable us to allocate all the space available. A third, and obvious objective, is to make it usable for compressed caching. Since we are not going to use it for anything else, we can allow ourselves shortcuts to make the implementation easier.

A cell consists of a specific number of contiguous pages in memory. This is the basic unit the memory allocator can add or remove from the allocators control. It also gives the name to the structure. A fragment is both the meta-data, and data of an allocation or free space within a cell as shown in Figure 4.1.

A brief explanation of how a fragment is allocated: first, a cell with enough space is found, then a fragment is made available within it. In the case where a cell with enough free space does not have enough contiguous space because of fragmentation, a compaction will occur where all the fragments are moved to a new cell.

To easily find a cell with enough free space for an allocation we have multiple lists for different ranges of available free space. If we get a request for an allocation of 200 bytes, we know exactly which list we should be

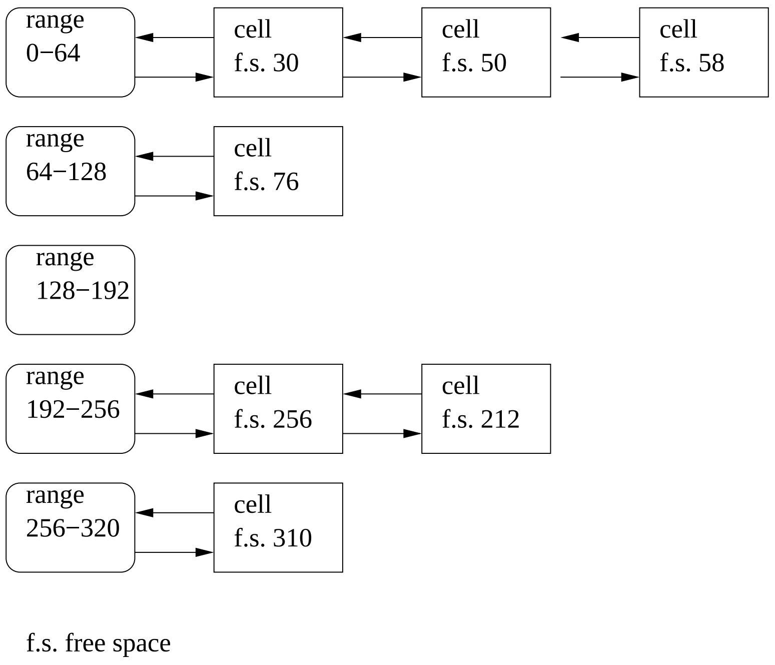

Figure 4.2: When allocating 150 bytes, we see that we end up in a list that is empty, so we go to the next list where we find a cell with 256 bytes of free space. After the allocation, the cell now only has 106 bytes of free space available, and should be moved to the appropriate free list with the range 64-128 (this is not done in the illustration).

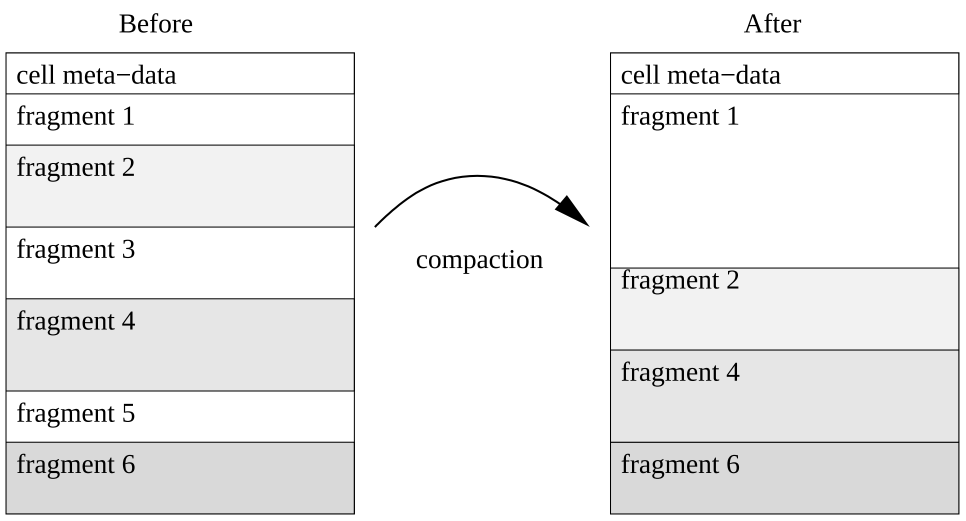

Figure 4.3: After a cell compaction, all the free fragments have been merged into one big fragment, and other fragments have been copied into a contiguous form.

looking at, as shown in Figure 4.2. If we do not find any available cells in the original list, we go to the next list that can contain cells with more free space than the previous. In the case where none of the lists contains a cell with enough free space to accommodate the requested allocation, a new cell is allocated.

The next step in an allocation is to carve out a fragment of the cell we have just located. We do this by going through all the fragments in the cell. If a fragment is free and large enough for our allocation, we split the fragment into two fragments: one for the allocation and one for the rest of the free space. The interesting part happens if we look through all the fragments and do not find a large enough fragment: this means that the cell has enough free space, but not contiguous. To make use of the free space, we have to compact the cell. After the cell has been compacted, we try to carve out a new fragment again. This is guaranteed to work because the free space of the cell has already told us what to expect as a minimum of free space after compaction. The freed space can actually be larger after compaction because we could end up with fewer free fragments, and therefore less fragment overhead as we can see in Figure 4.3. Another reason is that fragments can be freed while compaction occurs.

Compacting is done as follows: Walk through all the fragments in the fragmented cell and copy the used fragments into a new cell. When this is done, all the free space of the cell should be contained within one fragment, ready to be allocated. Because we move not only the data of the fragment, but also the fragment meta-data around, we need to be able to find who is pointing to the fragment. We solve this by having a pointer back to the user of the fragment, and update this after we have copied the data of a fragment from one cell to the next. This also limits us to one user as we only keep one pointer. To avoid internal fragmentation, a compaction on a global scale can be done, and works by taking a cell of few and small allocations and moving the fragments to other cells with sufficient space.

For compressed caching, global compaction should be done when there are few allocations per cell, because we expect a cell to store is minimum two compressed pages. This is done by taking the most unused cell, and reallocating all the fragments within it to other cells.

By limiting our allocator to what is needed for compressed caching we make a lot of locking issues a lot easier to handle. For compressed caching we only need to write the compressed version of a page to a memory location once. The same goes for reading, we only restore the page once from the allocation into a page. By doing it this way, we have effectively limited us to only need to know where the memory is at two specific times: once while storing and once while reading it. This means we can move an allocation around in-between and always free after a read. This is necessary to react to the fragmentation problem.

The public interface for the memory allocator is basic, and consists of three functions: *fragment_write()*, *fragment_readstart()* and *fragment_readend()*. The interface is meant to enforce the limitations, making it easier to use it correctly. It is also worth noting that the reading operation should be as short as possible, to avoid keeping cells unavailable for compaction.

Synchronization of such an allocator is a trade-off between two factors: keeping the concurrency high and the overhead low. The first synchronization issue we meet is when trying to locate a cell with enough free space. If some other context tries to take the exact same cell for allocation, a lot of failure scenarios could happen. The solution is to have a lock per list of free cells, and remove the cell from the list before we allocate a fragment from it. This allows us to allocate the same size at the same time. It also makes it logically impossible to access the cell for allocation from another context. The next problem that can occur, is when a fragment is freed from another context. This would usually result in the cell being reinserted into a free list. The solution is to have a cell lock, that protects the meta-data of a cell.

Fragments that are being read back or being compacted need some form of protection, but at the same time we want to keep the overhead low. To make this rarely needed lock as cheap as possible, we use one bit of the pointer for the locking in a busy-wait spin-lock. The usage of the bit in the fragment pointer means that we must always make sure that the fragment is placed on a 2 byte boundary. This is achieved by always making sure that an allocation, within a cell, is a multiple of two.

The use of a busy-wait spin-lock could give us a very unfair locking when compacting a cell. The unfairness would occur because users of the fragments would continually take the lock on the fragments, making the compaction (and the allocation of another user) wait for a long time. As a direct consequence of the usage limitations mentioned earlier, this situation will not happen because an in-use fragment will only be used once, and then the fragment is freed, and thus unlocked. The fragment can not be part of a new allocation, because the cell is not a part of the free lists while being compacted, so there is no way the compaction will spin forever.

A reverse issue is when the reading of a fragment is blocked by the compaction operation. This could occur when a cell with the fragment would be compacted over and over again in such a way to keep the fragment locked each time the user is trying to read it. This however is quite unlikely, given that it would mean there are no other cells to service the allocations, which is unlikely because the cell would get fuller and fuller and it would get more and more unlikely that we would use that cell instead of a newly allocated one. We have never experienced this situation.

To avoid deadlocks when locking, we need to follow a locking order. The locking order is as follows: first the free list locks, then the cell, then the fragment user-lock.

The locking order incurs some difficulties. For instance, we are unable to insert a cell back into a list without releasing the lock for the cell before taking the lock of the list. To compensate for this, we mark a cell as managed, such that freeing a fragment within such a cell will not result in the removal (and later insertion) of that cell into the cell free-lists. The context that marks a cell as managed is responsible to put such a page back into the free lists.

Chapter 5

Implementation

In this chapter we describe how we need to alter the Linux kernel to implement the design. More specifically, details and explanations that are not obvious, but needed to understand the implementation. In the next chapter we evaluate the performance of the implementation.

We start by describing what is needed to be stored in the compressed page descriptor, and how this is tied to the implementation. We then describe what functions need to be modified to intercept page cache accesses, in addition to how swap entries and tags are handled while doing so. We then go into the details of the memory area statistics structure (*cc_areastats*) and how it is synchronized. Next, we describe how the *address_space* structure and *anon_vma* structure is altered to store the *cc_areastats* structure. After that, we define what the compressed cache LRU list is, followed by an explanation of how the profit and expense lists are updated. In the following section we discuss the locking order within the implementation. We then go into detail on how functions that touch the page cache currently works and how they must be modified to work with compressed caching. We then go into the compressed cache subsystem. Starting with the two most important interface functions, *cc_store_page()* and *cc_restore()*, then continue describing, in detail, the rest of the helper functions. Following the compressed cache subsystem the cell memory allocator implementation is described.

5.1 Compressed page descriptor

We need the *cpage* structure to support all the functions that we are going to perform. Because the ordering of compressed pages are important we need to have a list available to us.

```
struct cpage {
    union {
        struct {
            void *compressed;
            int length;
        };
        struct list_head marker_queue;
    };
    struct address_space *mapping;
    unsigned long index;
    unsigned long flags;
    union {
        struct list_head list;
        struct page *page;
    };
};
```

The double linked list *list* will be used when compressed pages are: inserted into the beginning of the compressed cache LRU list, removed as the oldest pages from the end of the LRU list, and remove compressed pages at the middle of the list during accesses and miscellaneous cleanups. We consider the list, when used as the LRU order of compressed pages, to be the compressed cache.

The *flags* attribute stores flags such as whether a compressed page is dirty, is part of swap cache, or is part of the profit list or expense list. The dirty flag and swap cache flag are both meta-data taken from the original page when they are stored in the compressed cache, and are not altered while storing a page.

We need to know in which file or anonymous area, a compressed page is a part of. This is to know what is pointing to the compressed page so that we can update such pointers when we evict pages from the compressed cache. This information is called a *mapping* in the Linux kernel and is stored in the page descriptor. Compressed caching stores this in the *cpage* attribute *mapping*. The *mapping* can point to both an *anon_vma* area or an *address_space* object. This is differentiated by looking at the least significant bit. Together with the *mapping* attribute we store an index into that area.

We also need to keep a pointer to where the compressed content of the page is. The cell structure must be able to update this pointer so that compressed data can be moved around. When a page is a marker page we do not use the *compressed* and *length* attributes, so we have a union to not waste space for the marker queue.

Under certain circumstances, when the compressed page is not part of the LRU, it is sometimes a need to temporarily store a pointer to a page. To avoid making the *cpage* structure larger than strictly necessary we put the pointer in a union with the list pointers. The users of the *page* attribute is *find_get_pages()* and *find_get_tag_pages()*. We define this structure in include/linux/cc.h.

5.2 Intercepting compressed pages

There are several places where we could intercept file-backed and swap-backed pages, but we try to limit us as much as possible to the general page cache access functions.

These general accessing functions are wrappers for using radix-tree functions on the page cache, and are used to check if pages are already in memory. These functions include *find_get_page()*, *find_lock_page()*, *find_get_pages()* and *find_get_pages_tag()*. All of these have to be modified for compressed caching to work. We use the second bit, discussed in the design, to differentiate between compressed pages and non-compressed pages in the radix-tree. All of the functions return uncompressed pages, and is altered to allocate, decompress into and return them.

There are other places we also need to alter to make the compressed cache work optimally such as in the read-ahead, file truncation and process termination cleanup routines. We have to be careful not to break depended upon behavior of the cache interface functions. Such behavior include what a negative answers mean. For example, if a range of pages were looked up with *find_get_pages()* and the lookup only found marker pages, we could end up giving an empty answer, although we may find pages if we redo the same lookup.

5.2.1 Swap entries

Every page in the swap cache has a reference on a swap entry. When replacing an uncompressed page with a compressed page, this reference is handed from the uncompressed page to the compressed page. This is important to notice, so that we do not leave swap entries with the wrong reference count when cleaning up after terminated processes. The final clean up of swap entries is done in *free_swap_and_cache()*.

5.2.2 Dirty pages and tags

Tagging is a feature of the radix tree that make it easy to find contents with a specific property. For the page cache, the only tag used is the dirty tag, which is used to find dirty pages. When we replace an uncompressed page with a compressed page in the page cache, it is important that in addition to storing the dirty flag in the *cpage*, we need to make sure that the dirty tag is set for the new entry in the radix tree. If we are not doing this, pages that have been modified may not be written to disk resulting in corruption of both files and file systems.

5.3 Area statistics structure

For each memory area, we keep statistics in a structure called *cc_areastats*. The statistics include how many pages leaves the compressed cache used, and unused for each area. We also keep track of how many marker pages there are in each area.

```
struct cc_stats {
    unsigned char   flags;   /* state for area */
    unsigned char   used;    /* page-hit compressed
        cache */
    unsigned char   unused;  /* unused removed
        compressed cache */
    unsigned char   markers;/* marker pages present in
         c.c. */
    struct list_head marker_queue;
} __attribute__((packed));
```

The state of the area is also stored in the area statistics structure to easily know what phase the area is in. This makes it easy to make decisions on what to do with new pages trying to be inserted into the compressed cache.

Most of the accesses to the area statistics structure are done while altering the *address_space* object the compressed pages are a part of. For this reason we reuse the lock for the radix-tree in the *address_space* structure to also be the serialization point for the area statistics structure.

5.4 Address space structure changes

The *address_space* structure is used to look up pages that are part of the page cache. We alter this to include statistics on hit ratio of the compressed pages in the cache and whether compressed caching is turned on for that *address_space* object. We also store how many marker pages are in that cache, so that we can limit the number. We do this by inserting struct *cc_areastats* into the *address_space* structure found in include/linux/fs.h to hold this information.

5.5 Anonymous memory area structure changes

Anonymous pages are all part of the same *address_space* object, and are therefore not a suitable place to differentiate between memory areas when it comes to ratio calculation for anonymous memory.

Anonymous virtual memory areas are pointed to indirectly by the pages via the *anon_vma* structure. This is good because the memory area could have been mapped by several processes or multiple times within one process.

To store the statistics we want, we extend the *anon_vma* structure by adding the *cc_areastats* structure.

5.6 Profit and expense list

The order of the pages put into the compressed cache is kept as intact as possible. This is done with a logical list which consists of two lists as part of the implementation of compressed caching. This means that when we are referring to the LRU lists of compressed caching we are referring the expense and profit lists.

To implement the tracking of the expense and profit lists the following algorithm is used. When a new compressed page is allocated and inserted into the cache it is a part of the expense list. We also increment a counter telling us how many pages are in the expense list. We compare how many pages are in the expense list versus how many are used for storing the compressed pages, if there are more pages in the expense list than we are using for storage in the cell structure we push the oldest out of the expense list and into the profit list.

Once a compressed page has been recognized as part of the profit list, it will never be put back into the expense list. This is because of the very definition of what the profit list contains; the profit list contains compressed pages that would not have been in memory without a compressed cache. In other words, without compressed caching, a compressed page in the profit list would never have been able to be in memory and therefore not in the expense list. By this realization we also address the concerns of Tuduce et al.[11] for O(n) behavior, since we only do this once per page and never have to reiterate over the same pages over and over to calculate whether they are part of the expense or profit list. In fact, we do it only once per page and this is O(1).

The algorithm is mainly implemented in *cc_push_profit()*, which is called every time a page is inserted into the LRU of the compressed cache. When removing a page from the LRU that happens to be in the expense list, the counter *cc_expense* must be decremented.

Each compressed page has a flag so that we can easily check if it is part of the profit list or the expense list. This is implemented by the helper functions: *cpage_profit()*, *cpage_set_profit()* and *cpage_clear_profit()* which tests, sets and clears the profit flag for the *cpage*.

5.7 Locking order

To avoid deadlocks, locks that are held simultaneously must always be taken in the same order. If we do not do this we could end up in a situation where two processes are waiting for each other.

For compressed caching, we have a lock protecting the LRU lists from being altered from different contexts at the same time. When removing or replacing a compressed page the lock for both the *address_space* and the LRU lists must be taken to assure that the compressed page structure can not be removed from another context. This is guaranteed by the fact that the only way to have a reference to a compressed page is through the cache or the LRU lists.

The locking order is *address_space* first then the LRU lists. This means that if the LRU list lock is held, it must first be released before acquiring the lock on the *address_space*.

5.8 Page cache wrappers

Most of the code that touches the page cache uses a small number of functions that wraps around the data structure that keeps track of pages. This makes it possible for us to alter functions that are used everywhere and make it work without changing a lot of code. A problem with this approach is apparent however, since the interface used does not say what it is used for, we may end up doing unnecessary work. For example, some functions such as *find_get_page()* is used to both lookup up a page and use it and lookup up a page and discard it. This makes it necessary for us to make additional functions that can discard a compressed page directly. Other problems that we encountered are the restrictions on what you can do within these functions, for instance taking specific locks may result in deadlocks.

find_get_page() is used to see if there is a page with a certain index in the page cache, and if there is, get a reference on that page and return it. With no compressed caching this only requires read access to the radix-tree, because no modification is necessary. With our compressed cache however, a write lock is necessary when a compressed page is encountered. This is because replacing a compressed page with an uncompressed page needs to modify the radix-tree.

The altered version of *find_get_page()* is as follows. Take a read lock for the radix-tree, and do lookup for a page. If the found page is not a compressed page, we take a reference on that page, release the read lock on the radix-tree and return the page. If it is a compressed page, release read lock for the radix-tree and allocate a new page. The new page is allocated without any unnecessary locks taken to avoid contention on the radix-tree lock. The page lock is taken on the newly allocated page, so that other accesses to the radix-tree that would find it will wait until the decompression of the page is done before accessing the contents of the page.

The write lock for the radix-tree is taken, since it is going to be altered. Since the radix-tree was left unprotected, there is a possibility of two race-conditions that can occur: (1) The compressed page can have been decompressed by some other context, or, (2) the page can have been removed from the radix-tree. If the page has been decompressed and replaced in the radix-tree, a reference on the page is taken and returned. If the page has been removed, a negative lookup is returned. In both of the cases the previously allocated page should be freed.

The check for the two race conditions is done by redoing the lookup in the radix-tree. If a compressed page is found, the compressed page is removed from the LRU list of the compressed cache. An additional reference is taken on the allocated page, which represents the reference the cache has on it, this is done to make sure the page is not freed when the lock on the radix-tree is released.

The compressed page pointer in the radix-tree is replaced with a pointer to the allocated uncompressed page using *radix_tree_replace_slot()*. The write lock on the radix-tree is then released. The compressed page is then decompressed into the allocated page by calling *cc_restore()*. The now uncompressed page is then set to up-to-date and is unlocked. The compressed page is freed. The uncompressed page is inserted into the standard LRU list of the kernel, so that it can be removed from main memory later and is then returned.

find_get_pages() is used in the same way as *find_get_page()* described earlier, except that it is used for ranges of pages instead of only one page.

The lookup in the radix-tree is done using *radix_tree_gang_lookup()* which fills an array with the pages it found. When used without compressed caching, all that is needed to be done is to take a reference on each page and return the number of pages found. As in *find_get_page()*, *find_get_pages()* only needs a read lock for the radix-tree when there are no compressed pages involved.

The idea of the altered function is as follows. If there are compressed pages, allocate enough pages for them and take their locks, replace all of the compressed pages with the uncompressed pages, decompress the compressed pages into the newly allocated pages and return.

A *radix_tree_gang_lookup()* lookup could give compressed pages, therefore each entry in the result array has to be checked to see if it is a compressed page. If it is not, a reference for the page is taken. If it is, the index is stored in a separate array called *cc_pages*. The read lock on the radix-tree is then released. If there were no compressed pages in the result, the function is done and returns. If there were found compressed pages in the result of the lookup, the same number of pages are allocated and their locks taken. The newly allocated pages are stored into the result, and their index into the radix-tree is stored in the page. Next, the index into the result array is stored back into the *cc_pages* array.

The write lock for the radix-tree is then taken and the same two race conditions that could occur in *find_get_page()* can occur here, only for more than one page. Each index into the result is cycled looking up one and one compressed page. If the lookup fails, the corresponding allocated page is unlocked and freed, and a *NULL* is inserted into the result array. If the lookup returns a page that is not compressed, a reference is taken on it, and the uncompressed page is inserted into the result. The allocated page corresponding to the found page is unlocked and freed.

If the page found is a compressed page, *cc_accessed_cpage()* is called to update some statistics about the area, it also removes the *cpage* from the compressed cache LRU list. If the compressed page found is a marker compressed page, the corresponding allocated page is unlocked and freed, and a *NULL* is inserted into the result array. The marker page itself is removed from the radix-tree. The marker *cpage* is added to an local list that is freed later, when the lock on the radix-tree is not held.

For compressed pages that are not marker pages a page cache reference is taken on the corresponding allocated page. The compressed page is replaced in the radix-tree by the page. A pointer to the allocated page is stored into the compressed page so that we have a pointer to it and a pointer to the compressed page is stored into the result array. The write lock for the radix-tree is then released. The list of marker *cpages* is then freed by calling *cc_free_swap_list()*.

Finally, the result array is cycled through again, compacting the result by removing *NULL* entries, when a compressed page is encountered it is decompressed into the page pointed to by the *cpage*. That page replaces the compressed page in the result array. The page is set up to date and is unlocked, and should be usable by any context accessing the cache. The page is also inserted into the LRU list of the kernel. The compressed page is then freed, and the number of pages in the result is returned.

find_get_page_cc() is a replacement for *find_get_page()* when the only reason for the lookup is to free the page. We made this special version to avoid decompressing a page only to free it. The only place it is currently used is in the modified *free_swap_and_cache()* which frees a swap entry if it is the last user of it.

The implementation is similar to the modified version of *find_get_page()*; First a lookup is performed with a read lock for the radix-tree, if an uncompressed page is found, a reference is taken and the page is returned. If a compressed page is found, the read lock for the radix-tree is released and a write lock is acquired instead. Since the lock was released for the radix-tree, the lookup has to be performed again. If it is an uncompressed page, a reference is taken and the page is returned. If it is a compressed page, it is deleted from the radix-tree and removed from the compressed caching LRU list. The compressed page is then returned so that the caller can free the compressed page without any locks taken.

free_swap_and_cache() is used to free swap entries so that they can be reused. There are two reasons to modify this function: (1) we do not want to decompress a page only to evict it again. (2) Because of the locking the caller could have when running *free_swap_and_cache()*, we can not allow the function to sleep.

The second reason became apparent when implementing the cell structure and ran into a locking problem: we could need to sleep to get the cell lock. What this means is that we can not take a lock on a cell while having a spin lock.

This led us to redesign how cleanups after a process exit occur. Instead of trying to free a compressed page right away, which could result in the process sleeping, we postpone it by returning it throughout the call chain, and then keep it in a list, until the spin lock is released. This however, means that any caller of this function must explicitly check for compressed pages and free them. The callers of this function are *zap_pte()*, *zap_pte_range()* and *shmem_free_swp()*.

find_get_page_nocc() is a function introduced by us to consult the cache without decompressing any pages. This is needed by the read-ahead code so that it does not decompress pages that will not be used in the near future. *read_swap_cache_async()* is the only user of this function and has to be modified to use it.

The implementation of the function is as follows. Take a read lock for the radix-tree and do the lookup. If it is an uncompressed page, take a reference on it and return it. If it is a compressed page and it is not a marker page, return that there exists a compressed page. If it is a marker page, it is not guaranteed that the page would have been accessed and should therefore be discarded without altering the area statistics. The write lock for the radix-tree is acquired and the lookup repeated, if it is an uncompressed page it should take a reference and return. If it is a compressed page and not a marker page, return that there exists a compressed page. If it is a marker delete it from the compressed cache LRU list and return a negative lookup.

read_swap_cache_async() is a function that implements read-ahead for swap pages. It adds pages to the swap cache that are adjacent, on disk, to an on-demand request. It originally does this by looking up the adjacent pages in cache by using *find_get_page()* and adding swap entries to the cache until it hits a page in the cache. By altering this function to use *find_get_page_nocc()* when it does not want to decompress compressed pages and *find_get_page()* when it needs an uncompressed page the unnecessary decompression is avoided.

__do_page_cache_readahead() implements read-ahead for file cache pages. *__do_page_cache_readahead()* is needed to be altered so that it also read in pages for compressed marker pages. The function takes an *address_space*, starting position and a window size as parameters. First, it loops through the indexes in the window and looks them up in the cache, if they are not present in the cache a page is allocated and added to a list. When all the allocations are done, all the pages are added to the I/O queue at the same time. The modification needed to support marker pages is in the checking part, if a compressed page is found no allocation is needed, but if a marker page is found, a cache hit for that memory area should be recorded and a page should be allocated. To do this a check to see if the compressed page is a marker page is done, the read lock on the radix-tree is released and *find_get_page()* is called.

5.9 Changes to the PFRA

pageout_cache() is called by the PFRA to put pages into the compressed cache. This is currently the only way to put pages into the compressed cache. The implementation is straightforward, if the page should be compressed *cc_store_page()* is called to do so. If there are allocation problems when trying to compress the page, it is posponed and retried later. If the page is denied entry into the compressed cache, because of recent memory area behavior, nothing is done and the page will be removed from the page cache shortly. When a page has been compressed it must be inserted into the page cache. The lock for the radix-tree it is stored in is taken. We must then check if others have taken references on the page since we took the lock on it, but before we took radix-tree lock. If a reference has been taken the compressed page is freed and *pageout_cache()* returns that the page should be kept. We then replace the page descriptor with a compressed page descriptor in the radix-tree. We then insert the *cpage* into the compressed cache LRU list so that it will be removed at a later time. The uncompressed page is marked as a compressed page so that when it continues through the PFRA it will not remove the *cpage* from the page cache. If the page was stored in the compressed cache, the dirty bit of the page is cleared so that it is not written to disk.

5.10 Compressed cache subsystem

The compressed cache subsystem is located in mm/cc.c and handles all of the internal logic of compressed caching. The high level functions such as storing and restoring a page into and from the compressed cache is exported to be used in other parts of the kernel.

cc_store_page()'s main purpose is compress an uncompressed page and store it in the compressed cache.

The area statistics of the uncompressed page are examined by running *cc_area_state()*. This can give three answers: *CC_ENABLED*, which means the uncompressed page should be compressed and inserted into the compressed cache, *CC_DISABLED*, which means that it should not be compressed and *CC_MARKER*, which means that the page should not be compressed, but a marker page should be inserted into the compressed cache.

If *cc_area_state()* returns *CC_DISABLED*, *cc_store_page()* immediately returns a recognizable error code that means that the page will not be added to the compressed cache and that it must be swapped out in the usual manner.

To avoid having an empty compressed cache, the size of the compressed cache is checked to see that it is at least the minimum size. If it is not the minimum size, the growth lock on the compressed cache is reset.

A *cpage* is allocated from the memory cache, and meta-data about the uncompressed page is stored into it. If the state of the area is such that a marker should be inserted instead of a compressed page, the *cpage* is marked as a marker and is then returned.

A temporary page is allocated to be used as a work area for the compression. If the allocation fails we free the *cpage* allocated earlier and return. The page is then compressed into the work area and the length is stored into the *cpage*. If the page was not compressed enough to put into the compressed cache, a bad compression is noted and an error code is returned. If the page was compressed a good compression is noted. Both the bad and good compression noting is part of the bad compression detection.

A fragment is allocated to store the compressed page, this is done by calling *fragment_write()*. If it fails we retry after shrinking the compressed cache with one page. We retry this until we are able to store the page into the cell structure.

To avoid deadlocks when shrinking the cache, there is a requirement that no locks for any *address_space* object are held before calling *cc_shrink_lru()*. This requirement is then transfered to the caller of *cc_store_page()* which has to avoid having acquired any *address_space* object locks. All temporary data is then freed before returning the *cpage*.

cc_restore() is used to restore pages that are accessed and is a thin wrapper around *cc_restore_helper()*. The main objective of the *cc_restore()* function is to implement part of the adaptive cache size heuristic. It looks at whether the compressed page to be restored is part of the expense or the

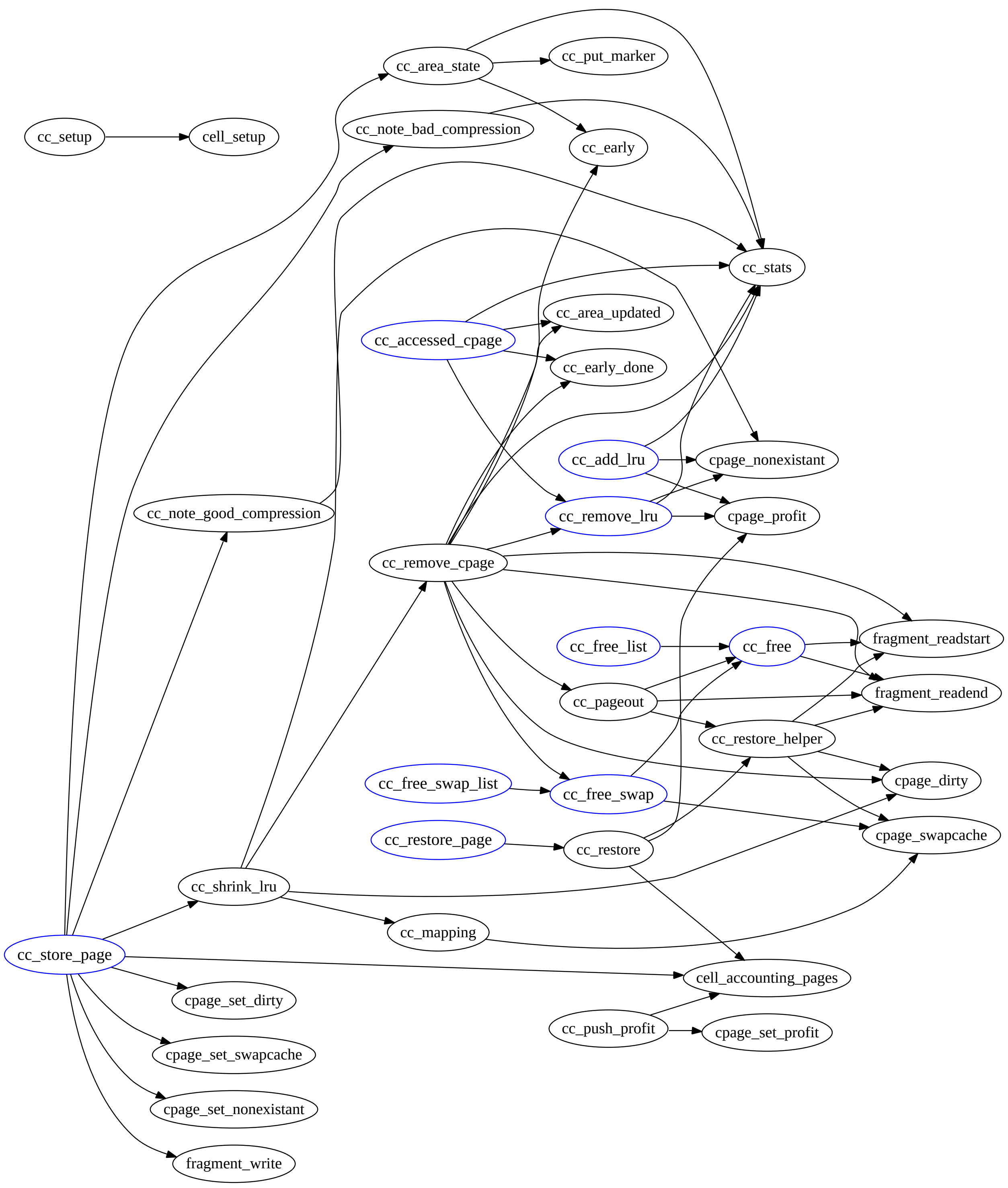

Figure 5.1: Internal call-graph of the compressed cache.

profit list. If it is part of the profit list the size of the cache is unlocked by resetting the *cc_locked* counter, and is therefore allowed to grow. If the compressed page is part of the expense list the counter is increased, if it is the third or later expense list access, the cache is shrunk with one compressed page. To make sure the cache has a minimum size, we unlock it in the case where the cache is smaller than a specified value. It then calls *cc_restore_helper()* to do the actual decompression and restoration of the meta-data for the page.

cc_accessed_page() is used to update statistics about compressed cache hits per area. It looks up the statistics struct of the area. Resets the early bad ratio detection and notes a cache hit for that area. The lock for the radix-tree in the *address_space* object is used to serialize access to the area statistics, and must be taken when this function is called.

cc_note_good_compression() is used to note when a good compression occurs. It gets a page as a parameter, takes the lock for the radix-tree in the *address_space* object of that page and then resets the bad compressions status of that area. In other words, if an area has one or two earlier noted bad compression, both of those flags are reset to normal.

cc_note_bad_compression() is a helper function to note bad compression for an area. It checks if the *BAD1* flag is set, if it is not, it is set and the function returns. Next it checks if *BAD2* flag is set, if it is not, it is set, and the *used* and *unused* attributes of the statistics structure is set to 2 and 0 respectively. In the case where two consecutively pages have been noted before and *BAD1* and *BAD2* flags are already set for the area, we increase the unused statistic and return. When the *unused* attribute is no longer smaller than the *used* attribute we increase the *used* attribute by one and reset unused to 0. By doing this we are increasing the number of pages we will not try to compressed before we try the next time.

```
static void cc_push_profit(void)
{
    unsigned long expenses = atomic_read(&cc_expense);
    unsigned long pages_used =
        cell_accounting_pages();
    unsigned long pushable;
    struct cpage *cpage;

    if (expenses < pages_used)
        return;
    pushable = expenses - pages_used;
    while ((pushable--) &&
        (!list_empty(&cell_list_expense))) {
        cpage = list_entry(cell_list_expense.next,
            struct cpage, list);
        list_del(&(cpage->list));
        cpage_set_profit(cpage);
        list_add_tail(&(cpage->list),
            &cell_list_profit);
        /* Marker pages does not take any space */
        if (cpage_marker(cpage)) {
            pushable++;
        } else {
            atomic_dec(&cc_expense);
        }
    }
}
```

cc_push_profit() pushes pages from the expense list into the profit list. It chooses how many pages to push by looking at how many pages are a part of the expense list compared to how much space is used to store the compressed pages. In other words it pushes pages that would not be in memory without the compressed cache from the expense list into the profit list.

The implementation is as follows. It first finds the number of pages in the expense list and how many pages the cell structure is using. If the number of compressed pages in the expense list is smaller than the pages used, no pages should be pushed to the profit list and it returns without doing anything. If the number of compressed pages in the expense list is larger than the number of pages used by the cell structure the number of how many pages can be pushed is calculated. Then one and one *cpage* is moved from the expense list to the profit list until enough compressed pages have been pushed or no compressed pages are left in the expense list. For each move the expense list counter is updated and the profit flag is set on the moved *cpage*. Pages that are only markers, and do not take up any space in the cell structure is not accounted as part of the pages that were moved, although it is still moved between the lists.

cc_remove_lru() is a helper function to remove pages from the compressed cache LRU list. Because of the locking order the lock for the radix-tree in the *address_space* object must be taken before this function is run.

First it takes the lock for the compressed cache LRU lists. If the *cpage* is a marker page the marker statistic for that area is decreased. If the *cpage* is a part of the expense list then the number of expense pages are decreased.

cc_add_lru() is a helper function to add pages to the compressed cache LRU. If the *cpage* is a marker, the statistics of the area it belongs is updated to present the increase of marker pages. By default, newly added *cpage*s are added to the expense list, and are marked as such. After adding a page to the expense list, and increasing the *cc_expense* counter, it is checked if any of the pages in the expense list belongs to the profit list, this is done by calling *cc_push_profit()*.

cc_mapping() is a small helper function that returns the *address_space* object containing the compressed page. The *mapping* attribute of the *cpage* structure is not adequate to use directly because it could contain an *anon_vma* pointer instead of an *address_space* pointer. If the *cpage* is part of the swap cache the *swapper_space address_space* object should be returned, else the *address_space* object of the file containing the compressed page should be returned.

cc_stats() takes a *cpage mapping* pointer as a parameter and returns the stats structure for that area. If that *mapping* pointer is a *anon_vma* pointer, a pointer to the stats for that *anon_vma* is returned. If the mapping is a file *address_space* object, the stats for that *address_space* object is returned. If it is not part of an *anon_vma* or a file *address_space* object, a pointer to the statistics structure of the swap cache *address_space* object is returned.

cc_pageout() moves dirty compressed pages into backing store. It does this by decompressing compressed pages into an uncompressed page, replaces the compressed page in the *address_space* object with the new page, flags it as a page that comes from the compressed cache and reintroduces the page into the LRU list of the normal page cache. The page is flagged as a previously compressed caching page, so that the PFRA will not try to put the page back into the compressed cache.

When *cc_pageout()* is called, the *address_space* write lock must have already been taken, this is to avoid race conditions. The page is marked as a swap cache page if the compressed page is part of the swap cache. The lock for the page is taken to make sure that no other context can use the uncompressed page before we are finished restoring it.

We also take a reference to the page to account for the reference for the cache, this means we have two references to it when we insert it into the radix-tree. If we are not doing this, the page could be accessed through the *address_space* object, and the last reference removed before we are finished with the page. To keep the tags, such as the dirty tag, intact after replacing a compressed page with an uncompressed page, the slot is updated directly using *radix_tree_replace_slot()*.

The write lock for the radix-tree is released, so that other processes can access the *address_space* object while decompressing the page. We then decompress and restore as much information into the page as possible, this is done by calling *cc_restore_helper()*. The *cpage* has no more uses after this, and is freed.

The page is flagged as *up-to-date* to signal that the page contents is up-to-date and ready for use. We flag the page as a previously compressed page to signal the PFRA that it should not be reinserted into the cache. The page is unlocked and other processes may map the page into their address space. It is inserted into the inactive LRU list of the standard page cache, so that it will be removed from the cache as soon as possible. We also remove our own reference to the uncompressed page.

cc_shrink_lru() shrinks the compressed cache by removing the least recently used compressed page. Pages part of the compressed cache can be a member of any *address_space* object, for this reason no locks for any *address_space* object must be taken by the caller while calling *cc_shrink_lru()*.

During this function the locks must be taken in the order of: *inode_lock* and then *cc_list_lock*. The reason for this locking order is two other locking orders: (1) The *inode_lock* must be taken before the lock in the *address_space* object. (2) The *address_space* lock before the *cc_list_lock*. If the new locking order is not followed a deadlock could occur. An example of this follows. If we have three processes: process 1 has the *cc_list_lock* and is waiting for the inode_lock, process 2 has the *address_space* lock and is waiting for the *cc_list_lock* and process 3 has the *inode_lock* and is waiting for the *address_space* lock. In other words, process 1 is waiting for process 3, process 3 is waiting for process 2 and process 2 is waiting for process 1. The solution is to take the *inode_lock* before the *cc_list_lock* in process 1.

After both the *inode_lock* and *cc_list_lock* has been taken the LRU lists are consulted to find the oldest compressed page. This is done by looking first in the profit list, which holds the oldest pages. If the profit list is empty the expense list is consulted.

If the compressed page belongs to a file, the *address_space* object could disappear, if the compressed page was deleted from other context. The only reason that guarantees that the *address_space* object is currently available is that the lock on the compressed caching LRU lists has been acquired, and that lock is necessary to remove any compressed pages from the *address_space* objects.

To remove a compressed page, the locks for the *address_space* object referencing the compressed page and the *cc_list_lock* need to be taken, in that order. To make sure that the *address_space* object is available after releasing the *cc_list_lock* a reference to the inode is taken, if the *address_space* object is part of an inode. Since the swap cache *address_space* object is never freed, no such measures need to be taken for compressed pages that are part of the swap cache.

If the page is dirty it need to be written to backing store, dirty compressed pages needs to written out, to do this a page is allocated so that the compressed page can be decompressed into it so that it can be written in uncompressed form back to disk.

cc_remove_cpage() is called to do the actual removal from the *address_space* and compressed cache LRU lists, and, if needed, paged out. After *cc_remove_cpage()* returns, the *inode* reference can be removed. If the *cpage* representing the compressed page was a marker, the process is repeated until the LRU lists are empty or a real compressed page is removed from the compressed cache.

cc_remove_cpage() gets a pointer to an *address_space* object, an offset and a page as parameters. The main purpose of this function is to remove a compressed page from the compressed cache. The requirement to call *cc_remove_cpage()* are that the *address_space* object is guaranteed to exist and that no locks are taken for it.

Two race conditions can happen before this function takes the needed locks: (1) the compressed page has been removed from another process context, (2) the compressed page has been decompressed and there is now an uncompressed page in the cache. For both of these cases there is basically nothing to do, if any resources have been allocated they are freed and the function returns.

The write lock is taken on the radix-tree, and the existence of a compressed page at the offset given in the parameters is checked. It first takes the lock on the mapping and uses the offset to look up a slot in the mapping. If the lookup fails or if the page found is not a compressed page, the compressed page must have already been removed. In that case the mapping lock is released and the page freed before returning.

The area statistics for the compressed page are retrieved to look at what state the area is in and to update the usage statistics. Since *cc_remove_cpage()* is only run on pages that are being removed without being accessed, it is counted as a page that exited the compressed cache without being used, an unused compressed page. If the area the compressed page is a part of is in the "early phase", this is considered an access that will end it. The compressed page is removed from the compressed cache LRU list. Compressed pages that are not dirty do not need to be written back to disk and can safely be deleted from the mapping and removed from the compressed cache LRU list and freed. For dirty compressed pages *cc_pageout()* is called to do the decompression and final pageout.

cc_restore_helper() restores a compressed page into an uncompressed page. In other words, it does the following: it copies the meta-data from the compressed page into the uncompressed page, and then decompresses the compressed data into the uncompressed page. To do this, it copies the *mapping* and private data from the *cpage* to the page descriptor directly. If the compressed page is dirty, the restored uncompressed page should be flagged as a dirty page.

To be able to decompress the page, the compressed contents of the page must be located and locked to that location while being decompressed. This is achieved by running *fragment_readstart()*, which is part of the interface for the cell data structure. It will return a pointer to where the compressed data is, we then run the decompression algorithm with that area as the source. When it is done we call *fragment_readend()* to unlock and free the compressed contents, and return.

cc_free() is a helper function to free compressed pages. It will free the allocation of the compressed data only if there is any data allocated. There are two reasons there could be allocated compressed data pointed to by the *cpage*: (1) when clean compressed pages are evicted from the compressed cache their compressed data is not used (2) when compressed pages being deleted, because they will not be accessed again, nothing is done with their compressed data before freeing them.

Next it frees the *cpage* structure back to the memory cache, most often the SLAB cache. This means that the allocation can be reused when allocating a *cpage* in the near future.

cc_free_list() is a helper function that is used when freeing several compressed pages at once. This is used at several locations throughout the code when collecting compressed pages that can not be freed because of locking requirements, and the freeing is deferred until they can be satisfied.

cc_free_swap() frees swap entries associated with a compressed page before freeing the compressed page. This is especially important when freeing compressed pages that are part of the swap cache and the compressed page is not replaced with an uncompressed page in the swap cache.

cc_free_swap_list() frees a list of compressed pages including their swap entry references. This is useful when locking requirements prevents freeing of compressed pages directly until later.

cc_setup() is the initialization function of the compressed cache subsystem. It sets up all the data structures needed by the compressed cache. It resets the *cc_locked* state to 0 so that the cache can grow by default. This is used by the adaptive cache size heuristics. It sets up a memory cache called "cc_cpage" for allocation of the *cpage* structure such that the alignments two last bits will always be 0. It initializes the locks for the compressed cache LRU lists and initialize the list heads. The cell structure is initialized by calling *cell_setup()*. Finally, the shrinker is set up, this is infrastructure from the vanilla kernel to tell compressed caching to free up some space preemptively, before running out of memory. It is set up so that it runs the function *cc_shrinker()* when such a request comes.

cc_early() and cc_early_done() checks and resets the early phase bit respectively. *cc_early_done()* should be called if any compressed pages that are part of that area have been accessed or evicted from the compressed cache.

cc_put_marker() returns true if there can be put more marker pages into the cache for that area. It does this by checking how many marker pages there are and comparing this to the maximum allowed. Because the lock for the statistics structure is released between the run of this function and the usage of the information it returns, there is a possibility that more than the maximum value of marker pages could exist. This is very unlikely since the gap is very small, and the number should not be very much higher than the max set.

cc_area_state() takes a page as an argument and returns what should be done with the page. This is the main implementation of per area bad ratio, early and bad compression phases. First we look up the statistics of the area the page is contained in. We take the lock of the *address_space* object, so that our reading of the statistics will be atomic. We check if it is in the bad compression state, if it is we return that it is in bad compression state. If the area is in the early phase, we check to see how many pages we have already put into the compressed cache. If it has hit the limit, we return that the page should not be put into the compressed cache.

If it has not hit the early phase limit, the write lock is taken for the *address_space* object. This means that we have to release the already taken read lock before taking the write lock. If a race with another instance of *cc_area_state()* occurs, while the lock is not taken, the area may not be in an early phase anymore. To avoid that we have to recheck if it is still in the early phase before increasing the number of pages that are part of the compressed cache. If everything works out it returns that the page may be put into the compressed cache.

If the area is not in the early phase, a bit in the flags for the statistics that tells us whether the ratio of unused pages is high is checked. If it has been set, compressed caching is disabled for that area. To be able to recover from a disabled compressed cache, we need to have some statistics on how well the compressed cache would have worked if we would have put the pages into the compressed cache. To implement this we put marker pages in place of real compressed pages into the cache, but we limit the number of such pages. This is done by calling *cc_put_marker()* that checks to see if we should insert a marker *cpage*. If we should, we return that a marker page should be inserted into the cache.

cc_area_updated() is a function called every time the statistics of an area are updated. This is to keep as much as possible of the logic of the state of an area in one place. No statistics should be updated if the area is in the bad compression state. If the hit ratio of the area is less than 20% the compressed caching is disabled for the area. If the area already has disabled compression, but the hit ratio has increased to 50% the compressed cache is enabled again by resetting the statistics for the area, and restarting the early phase. If any of the statistics is above a certain threshold we divide all the statistics by two to make sure we are mostly affected by recent statistics. Since the statistics structure *cc_areastats* is made as small as possible, the size of numbers it can store is limited. To avoid overflowing the counters of *used* and *unused* pages we must make sure the numbers are never above the highest number that can be stored. For the *used* and *unused* attributes, this is achieved by dividing both by two when either of them reaches above a certain threshold. This solves both the problem of recency and overflowing the statistics attributes. For the *markers* attribute, it is already limited by how many markers we want to have in a memory area, so it will not overflow.

5.11 Cell implementation

The cell implementation is located in mm/cell.c and include/linux/cell.h. We first explain how the main interface functions *fragment_write()*, *fragment_readstart()* and *fragment_readend()* works followed by the interface functions to query number of allocations in the cell allocation and page usage. We try to follow a logical order of how the functions are used when allocating a fragment. Starting with *cell_get()* is called to find a cell for the allocation, next *cell_fragment()* is called to find a fragment for the allocation and then *fragment_split()* to to make a new fragment from that. After a fragment is found the cell is put back with *cell_put()*. For all this to work some data structures need to be initialized, this is done in *cell_setup()*. During an allocation of a fragment within a cell there may not be any large enough fragments available, this is solved by the compaction routine *cell_compact()*. Because of external fragmentation of the fragments there may be many cell with few allocations, this is handled by *cell_global_compact* which relocates the fragments in the cell with fewest allocations. We then present helper functions that deal with the handling of cells in different cases, followed by helper macros.

fragment_write() is used to allocate a fragment and copy data in to the data part of the fragment. The first thing it does is to round up the needed allocation size to the nearest number divisible by two. Then it uses *cell_get()* to find a cell with the appropriate amount of space. If it fails it may allocate a new cell, or return an error. When it has a cell with enough free space it tries to find a contiguous space within the cell by using *cell_fragment()*. If this fails, the cell is compacted with *cell_compact()* which returns a compacted cell, now *cell_fragment()* is guaranteed to succeed. It then puts the cell, with *cell_put()*, back into the free lists.

fragment_readstart() and *fragment_readend()* is used to tell the memory allocator to back off while reading the memory and later to say that the fragment can be freed. *fragment_readstart()* takes the lock on the fragment and returns a pointer to the allocation.

fragment_readend() takes the lock of the cell, and checks the state of the cell. If it is marked as a managed cell or is a left over cell after a compaction, clear out the fragment and update the free space of the cell and put the cell back. In the case that the cell has no special state, it figures out which free list it is in (if it is in a list), then because of locking order, we need to give away the lock of the cell, take the possible list lock and then take the lock of the cell again. Now we have to recheck that we actually took the correct list lock, as there could have been other process contexts changing the cell. If the incorrect list lock was taken, then it is released and the same process is tried again. If everything went well so far, we are ready to remove the cell from the old free list, mark the fragment as free, update the free space of the cell and put the cell back into the free lists. If there is a consecutive fragment to this cell that is also a free cell, the two cells are merged and the free space accounting incremented with the size of a fragment header. If this freeing of fragments results in there being only one allocation per cell, run global compaction before returning.

cell_accounting_pages() and cell_accounting_fragments() returns the number of pages used by the cell structure and number of allocated fragments. This is part of the external interface, giving the ability to compressed caching to know how much memory is used.

cell_get() is used to find a cell with a minimum of free space, this is achieved by using the free lists. The lookup is essentially an index into an array of lists with the index calculated as *space_needed / CELL_GRANULARITY*. If no cells with enough space are found, in this list, the index is increased and the next list is tried until either a cell is found or we hit the largest index possible, which means that we have no such cell. For each list we look in, we take the semaphore of the list. If we find a cell, we take the lock on the cell and remove it from the list. We also release the semaphore of the list before we return the cell with the lock taken.

cell_fragment() finds free fragments in a cell of appropriate size, and takes a chunk of them to make up a new fragment. It does this by going through the fragments of a cell, if it is free and large enough, split the fragment into a new free one and return it. To avoid looking through all the fragments in a cell, we keep track of how much space we have gone through already, if it is impossible to find a large enough free fragment later we return early. We also merge adjacent free fragments while going through them to optimize later allocations.

fragment_split() tries to split a fragment into two, and return the new one. The first issue that can occur when splitting a fragment is that there is not enough space in the already existing fragment to include both the needed allocation size and the fragment header. If this is the case, the original fragment is returned. If there is enough space to split it, it is split in such a way that the new fragment is located at the end of the original fragment. This way other allocations will find the free part of the original fragment before the newly allocated (and not free) fragment.

cell_put() puts a cell back into the free lists. There are several things that have to be done before inserting into a list. First it needs to check if a cell is marked as managed, if it is there is really nothing to be done for the cell as another *cell_put()* is running or will be running shortly, so the cell is only unlocked. Next the function checks if there are any allocations in the cell, if there are not, then the cell should be freed in one of the two following ways. (1) If the cell was part of a recent compaction that did not want to wait for a fragment to finish, it should be put back into the list of dedicated pages for compaction with *cell_keep()*. (2) In the more usual case, we just free the page with *cell_free()*. The last special case is if there are more allocations, but the cell is still marked as a page to be put into the dedicated pages for compaction. For that scenario, it just unlocks the cell and returns, since there will be a *cell_put()* when the last fragment is freed from the cell. Since the locking order dictates that we need to take the list lock before we take the cell lock, we must unlock the cell, take the list lock and then reacquire the lock on the cell. This means that the cell could be altered while we do not have the lock. If another context changes the amount of free space, it would change what free list the cell is supposed to be in. To handle this situation, we first do the *space_needed* / *CELL_GRANULARITY* calculation to find the index of the free list then we recheck that our previous calculation is still applicable by redoing it after we have re-acquired the lock. If it is not, we simply unlock everything and retry.

Another race condition that could happen while the lock is not held is the insertion of a cell into a free list. To avoid this, we mark the cell as managed which will make other *cell_put()* instances let us do the job. When we are done, we remove the marking before unlocking the cell. After re-acquiring the cell lock, we have to recheck some of our earlier assumptions, namely that there are still allocations in the cell. If there are no more allocations we free the cell with *cell_free()* after unlocking the free list.

cell_setup() is used to initialize the data structures needed by the memory allocator. This includes free lists and their semaphores, it also includes preallocation of cells to perform compaction. The reasoning for preallocation of cells for compacting is to avoid a situation where memory contention can stop us from allocating a fragment, that then would need to be handled by allocation of a new cell that is guaranteed to fail as well. Avoiding such situations means that we limit ourselves to how many compaction can occur at the same time, but this is not a very important limitation as the compaction in progress would compete against each other for CPU time. We can set the number of cells dedicated to compaction to a number relative to the number of CPUs in the system for instance.

cell_lock() and cell_unlock() is used to acquire and release the lock of a cell. Since a cell consists of pages, we reuse the *lock_page()* and *unlock_page()* functions already provided by the Linux kernel. We do this by locating the first allocated page in the cell and taking the lock for that.

cell_compact() moves all of the fragments in a cell into a new cell and returns it. First it grabs one of the dedicated pages for compacting by calling on *cell_temporary()*. Next it walks through the fragments one by one doing the following. Check if the fragment is in use, if it is not skip to the next fragment. Try to take the lock atomically on the fragment, if this fails, skip to the next fragment. It then determines the size of the fragment and allocates such a fragment in the new cell, it then copies the content of the fragment from the old fragment to the new one. It then updates the new fragments user pointer and the users pointer to point to the new fragment while simultaneously unlocking the fragment. It also decrements the allocation counter in the cell. It then continues on to the next fragment. When all the fragments have been iterated over, *cell_compact()* determines whether any of the fragments were busy, and thus are not moved, by inspecting how many fragment allocations are left. If there are still allocations left, mark the cell as keepable, so that it will be inserted into the list of dedicated cells for compacting when that fragment is later freed and returned. If there are no allocations left, it puts the cell directly into the dedicated list by calling *cell_keep()*.

cell_global_compact() takes the most unused cell and cycles through its fragments. It takes the lock on the fragment and locates a cell that can accommodate the fragment. If there is no such cell, then global compaction will return without doing anything. If it has located a cell, a fragment is allocated from the cell. If this is not possible, the cell is compacted and the fragment allocation retried. Next the contents of the old fragment is copied to the newly allocated fragment. All meta-data such as free space and number of allocations are updated both in the most unused cell and the cell containing the new fragment.

The reason we also keep the meta-data of the most unused cell up to date is to avoid leaving a cell inconsistent if we have to stop the compaction mid-way. This can happen if it is impossible to relocate a fragment. In that case the global compact is canceled, reinserting the most unused cell back into the free lists.

cell_most_unused() is a helper function for *cell_global_compact()*. It finds one of the cells with most free space and returns this. It cycles through the free cell lists starting at the list representing cells with most free space. It then finds the first cell with two or less allocations, removes it from the free list and returns that. In the case that no such cell exists for that free list the next free list is tried. In some cases there are no available cells, and no cell is returned.

cell_allocate() allocates the contiguous pages. After the allocation, we set all the page descriptors' *private* attribute to point to the first page in the allocation. This way we can easily find the first page in the allocation. We also increase the number of pages allocated by the cell structure. This number made available for users of the cell data structure, such as compressed caching. It is also used to decide if a global compaction should be done.

cell_free() puts all the pages belonging to the cell back into the buddy allocator. Atomically subtract the number of pages used by the cell structure by the number of pages a cell uses.

cell_keep() puts a cell into the list of dedicated compaction cells. The reason we have this list is to avoid having to allocate and deallocate cells every time we do a compaction. First the cell is reinitialized, then the lock for the list is taken and the cell inserted. Next the list lock is unlocked and the semaphore of the temporary cells is upped.

cell_temporary() first waits for the semaphore of the temporary cells, this way we will never access an empty list. Next we take the lock for the list, remove the cell, and unlock the list. The release of the semaphore for temporary cells is left to *cell_keep*, which is called when an empty temporary cell is tried to be put into the free lists. The cell is then returned.

CELL_PAGE() helps us find the first page in the cell. It uses *virt_to_page()* to find out what page structure belongs to the cell structure. Because the cell structure is in the beginning of the cell, we know that this is the first page.

CELL_MANAGED(), CELL_MANAGE()
and CELL_UNMANAGE() is used to check, set, and clear whether a cell is managed or not.

CELL_KEEP(), CELL_SET_KEEP()
and CELL_CLEAR_KEEP() is used to check, set, and clear whether a cell is to be kept in the dedicated list of cells for compaction on freeing of the last fragment allocated in the cell.

FRAGMENT_CELL() takes a fragments as its argument and returns a pointer to a cell structure. It finds out what cell the fragment is a member of by first finding out what page it is located in by using *virt_to_page()*, it then looks at the private member of that page which makes us end up with the first page, which is then converted back into the address by using *page_address()* to give us the cell structure.

FRAGMENT_LENGTH() takes a fragment as its argument and gives us the length of the data in the fragment. It calculates this by looking at the difference between where the data starts and where the next fragments starts. In the special case where the fragment is the last fragment in the cell, it finds out where the end of the cell is and calculates the difference between where the data starts and the cell ends.

Chapter 6

Performance evaluation

In this chapter we look at how to measure the performance of compressed caching. We describe in detail the setup of the workloads, present the results and evaluate them.

6.1 Workloads

We choose workloads where compressed caching can make a difference, for better or worse. Since compressed caching is a method to reduce I/O, the minimum requirement is that the workload make extensive use of the page cache. This means that a CPU intensive workload with light memory usage would not be an interesting test-case, because the performance would not be affected by compressed caching. This is important when choosing parameters for a particular workload. The most important parameter is how much memory is available. If we choose a too high amount, compressed caching will not have an effect, if we choose a too low amount, the application will not run.

A Linux kernel build is a workload that has a fair amount of I/O and CPU usage.This has been used by Castro et al.[8] for testing compressed caching and is in the class of workloads that is known to benefit from compressed caching. The reason it benefits from compressed caching is that it reuses data quite often. First consider the compiler, which is run for each source file. The same application text together with the same libraries is mapped in each time it is run. Next, consider the data that it works on: headers and source files. Headers are often reused in many source files and will often be reused with the next source file. The entropy of the header and source files are very low and can have a very good compression ratio [1]. This means that headers that will be reused stay longer in the compressed

[1]Linux 2.6.22.19 stored as a compressed source archive takes 44MB, uncompressed it takes 294MB.

cache, and longer in the profit list. This makes the build of the kernel a good candidate for compressed caching.

The number of parallel compilations can be set for the kernel compilation workload, we call these ”make -j1”, ”make -j2” and ”make -j3” tests, taken from the command used to start the compilation processes. As can be seen in Appendix B.1, the workload includes uncompressing the source from an archive and then compiling it. We vary the amount of memory available at boot time by giving the ”mem” parameter to the kernel. We run this test with four different kernels: compressed caching, an unmodified kernel, compressed caching using LZF as the compression algorithm and one where the compressed caching is never disabled denoted ”cc”, ”vanilla”, ”cc-lzf” and ”cc-nodisable” respectively (see Table 6.1).

Another workload we use is GNU Utils' *sort*. The purpose of this test is to expose a workload which can be heavily penalized by compressed caching. GNU sort accesses its working set in least recently used order, which is the opposite of what the Linux memory management has as its prerequisite to work well. In other words, when the working set is larger than available main memory, the workload is accessing pages that have recently been evicted from main memory. This is the worst case scenario of LRU page frame reclaiming. For the unmodified kernel the worst-case means reading more from disk, while for kernels with compressed caching it means compressing and decompressing pages in addition to reading more from disk. However, the worst-case happens sooner on unmodified kernels, because less of the working set can be stored in main memory than with compressed caching. The script, found in Appendix B.2, used to perform the sort workload runs the sort command five times in a row marking the first one, to know where the test started. The result is the sum of all five runs. We run this test with three different kernels: compressed caching, an unmodified kernel and compressed caching with disabling logic disabled called ”cc”, ”vanilla” and ”cc-nodisable” respectively in Table 6.1.

We also run a test which is a mixture of a ”make -j1” and a sort test. ”make -j1” is ran, and after 120 seconds the sort test is started in parallel. We run this test, called ”sortcomp” in Table 6.1, with the same kernels as in the sort test.

6.2 Setup

To have high confidence in the test results, we run each test multiple times, and make sure that each test is run under the same conditions. To achieve as equal conditions as possible, the workloads are run alone without any unnecessary processes running in parallel, and the environment is reset as much as possible between each run. To reset the environment, we format the volume containing the touched data and reboot. To implement this scheme,

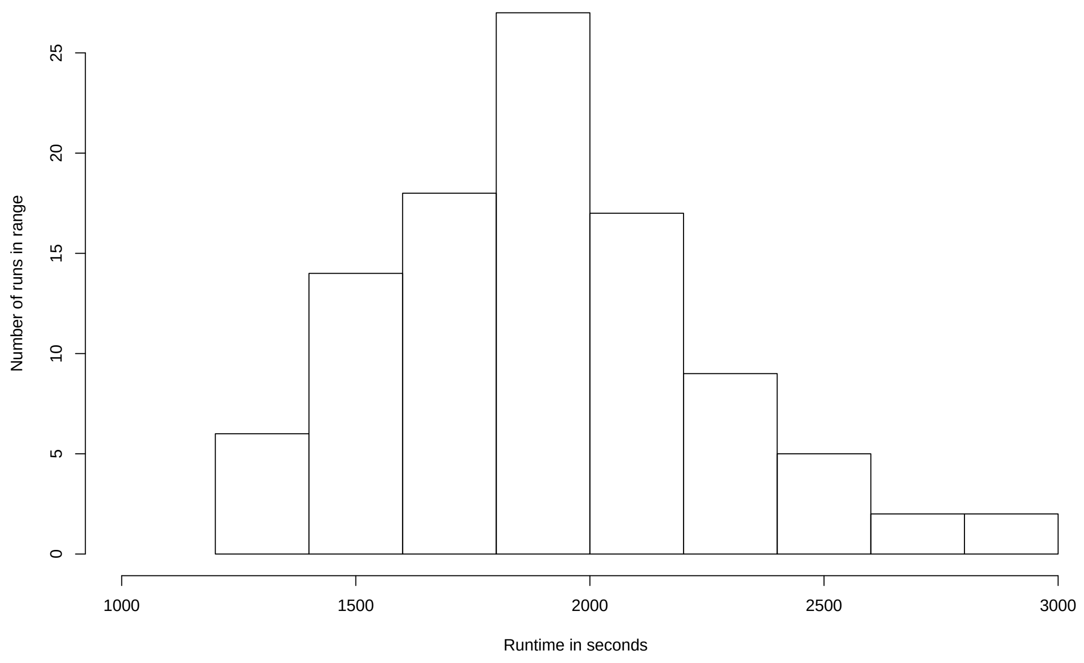

Figure 6.1: Histogram of 2.6.22 vanilla make -j3 test with 80MB of memory

we have made our own run-level which runs one test after a successful boot, records the result and resets the environment and then reboots again. Using a re-configuration feature of the boot loader we change what kernel and what test to run each time we boot the system.

6.3 Boxplots

When running the tests we observed great variance in the test results for all the kernels we tested with. A histogram of the results from running the "make -j3" workload on an unmodified Linux kernel can be seen in Figure 6.1 and the summary can be found in Table 6.1. The great variance, as seen in the histogram, can make it hard to compare the effect of changes when altering the kernel. The reason for the great variance in the results is probably[2] due to timing, where a slight change in the order of the processes could result in a very different order in which pages are being evicted from the page cache. To make it easier to compare the results we decided to

[2]Since it is hard to test this to figure out what really went wrong, and that it is outside the scope of this thesis, we do not discuss this further. A discussion[9] took place on the Linux kernel mailing list.

present them as boxplots[7] in addition to scatter plots. If we follow Figure 6.2: the main box shows the interquartile range (IQR) or in other words the middle fifty percent of the data set. The bold line going horizontally through this box is the median of the data set. The stippled lines represent the range between the maximum and minimum. The range used for this thesis is limited to 1.5 of the IQR to show extreme values outside of the range. The outliers are the samples that are outside the limited range. If

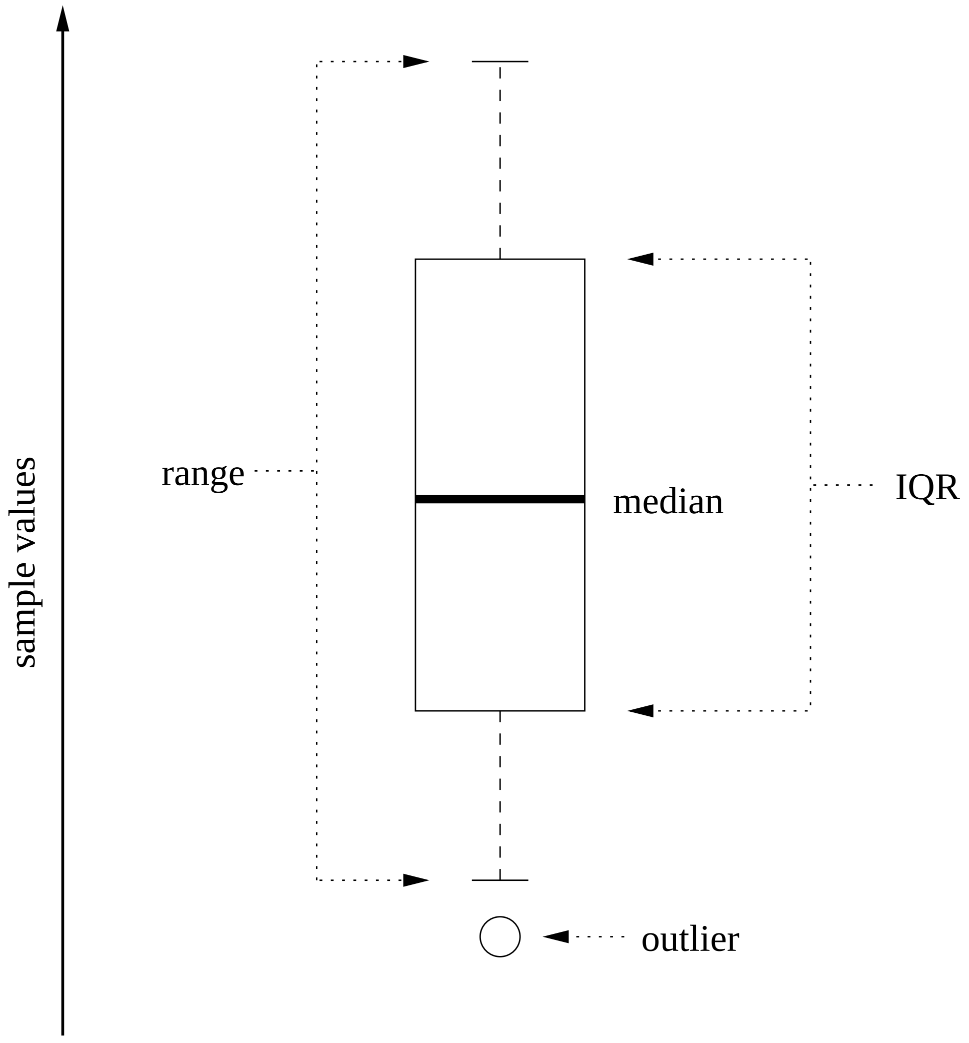

Figure 6.2: Boxplot introduction

we study a data set of the numbers 0 to 100, the IQR would be from 25 to 75, a 50 interval, this means that the stippled lines could extend from $25 - (50 * 1.5) = -50$ to $75 + (50 * 1.5)$. In other words: if we removed the numbers 0 and 100 from the data set and added -51 and 150, -51 would

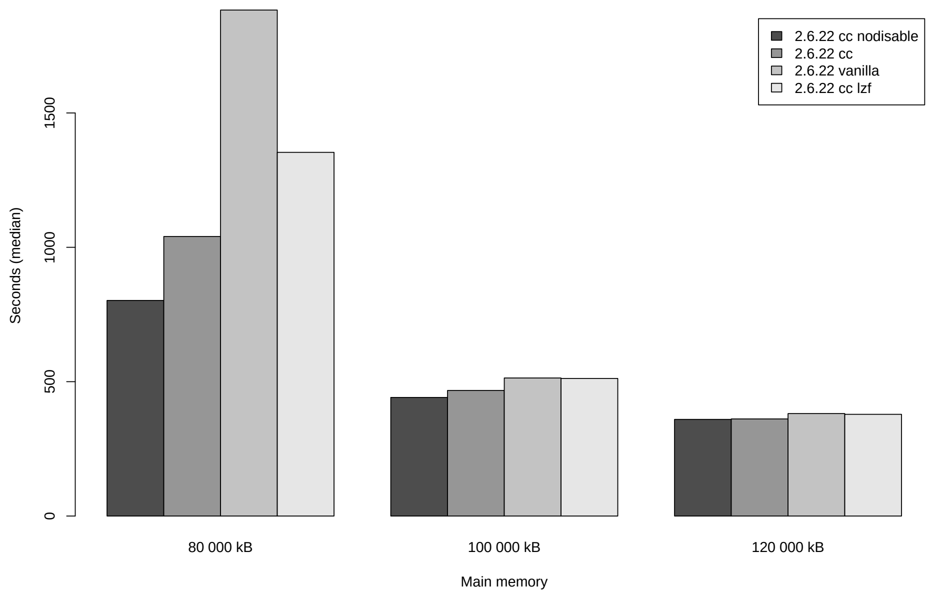

Figure 6.3: make -j3 with varying available memory

be an outlier and 150 would be part of the range.

6.4 Results

We have put a summary of all the results in Table 6.1 with one row for each workload-memory-kernel combination we have tested. Each row includes the name of the workload, how much memory was available, what kernel was used, the median, the mean and the standard deviation within the results and how many times it was run.

We have run the "make -j3" workload with different amounts of memory, as can be seen in Figure 6.3 the performance increase is larger the less main memory is available.

We have run three different workloads, "make -j3", "make -j2" and "make -j1" with 80MB amount of main memory available. These workloads do the same work with the difference being how much is done in parallel. The more processes that run at the same time, the more memory is used. As we can see from Figure 6.4 the highest performance increase provided by compressed caching is found when the highest number of processes are run. Even if compressed caching is performing better than the unmodified kernel with the same amount of memory, it is not performing better than an unmodified kernel with enough memory to avoid evicting pages all together. This is expected and shows that compressed caching is not a replacement for not

test	memory	kernel	median	mean	s.d.	runs
make -j3	80000kB	cc-nodisable	802.30s	830.14s	109.34s	141
	80000kB	cc	1040.21s	1064.30s	140.97s	73
	80000kB	cc-lzf	1353.38s	1387.53s	235.09s	100
	80000kB	vanilla	1882.82s	1909.59s	355.22s	100
	100000kB	cc-nodisable	441.30s	447.99s	27.66s	117
	100000kB	cc	467.41s	482.79s	47.26s	100
	100000kB	cc-lzf	511.99s	530.70s	56.13s	100
	100000kB	vanilla	513.77s	560.36s	116.34s	100
	120000kB	cc-nodisable	359.67s	363.79s	20.36s	117
	120000kB	cc	361.46s	367.88s	23.95s	100
	120000kB	cc-lzf	378.49s	380.57s	27.52s	100
	120000kB	vanilla	381.33s	385.00s	28.85s	100
make -j2	80000kB	cc	641.14s	650.36s	61.84s	100
	80000kB	cc-nodisable	587.40s	605.34s	61.91s	89
	80000kB	vanilla	844.90s	871.69s	97.32s	100
make -j1	80000kB	cc	424.24s	428.04s	19.69s	100
	80000kB	cc-nodisable	407.30s	416.32s	21.72s	89
	80000kB	vanilla	474.42s	474.45s	5.26s	100
sort	40000kB	cc	261.82s	261.80s	4.90s	91
	40000kB	vanilla	249.72s	250.32s	3.57s	100
	40000kB	cc-nodisable	278.22s	278.34s	7.82s	100
	60000kB	cc	235.23s	235.41s	2.72s	34
	60000kB	vanilla	235.99s	237.00s	3.35s	52
	60000kB	cc-nodisable	284.70s	284.55s	10.14s	43
	80000kB	cc	240.94s	241.17s	2.92s	195
	80000kB	vanilla	244.39s	244.88s	3.36s	55
	80000kB	cc-nodisable	279.69s	282.09s	10.14s	55
sortcomp	80000kB	cc	594.00s	597.42s	23.50s	100
	80000kB	vanilla	639.00s	639.22s	4.64s	100
	80000kB	cc-nodisable	583.50s	588.89s	23.08s	100

Table 6.1: All the results from the test runs.

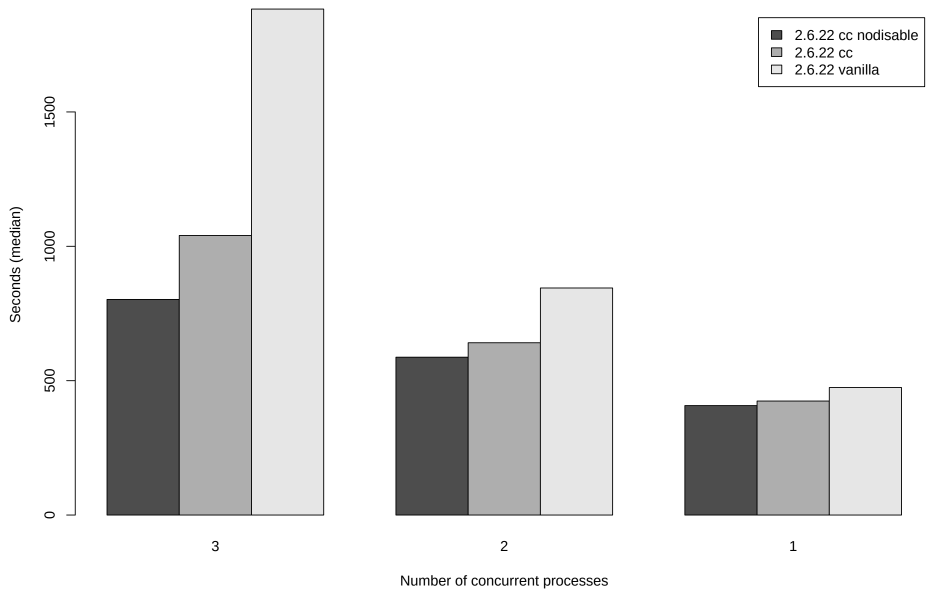

Figure 6.4: 80MB with varying number of concurrent processes

having enough main memory, but is a measure to postpone trashing and heavy disk I/O.

In the scatter plots for the "make -j3" with 80MB of main memory tests we see that the unmodified "vanilla" Linux kernel get results that vary from 1220 seconds to over 2930 seconds (see Figure 6.5). For the compressed caching Linux kernel the results vary from about 820 seconds to just below 1500 seconds (see Figure 6.6) and with a similar kernel with no disabling logic we see the results vary from about 685 seconds to about 1292 seconds (see Figure 6.7). If we change the compression algorithm from LZO to LZF we see the results varying from about 970 seconds to about 2230 seconds (see Figure 6.8).

When we compare the result of running the different kernels in Figure 6.9, we see clearly that the IQR of the compressed caching results is much denser than that of the unmodified kernel. This is probably due to the randomness disk I/O may introduce, which could accumulate into the worst or the best results. The unmodified kernel does more disk I/O than the two other kernels. The difference in performance between LZO and LZF is due to LZF not achieving the same compression rates as that of LZO. Compressed caching without the disabling logic is 20% faster than compressed caching with disabling of compressed cache, this is probably due to the heuristic disabling compressed caching for an area being too aggressive for this workload. Comparing the median of "vanilla" kernel with the other modified

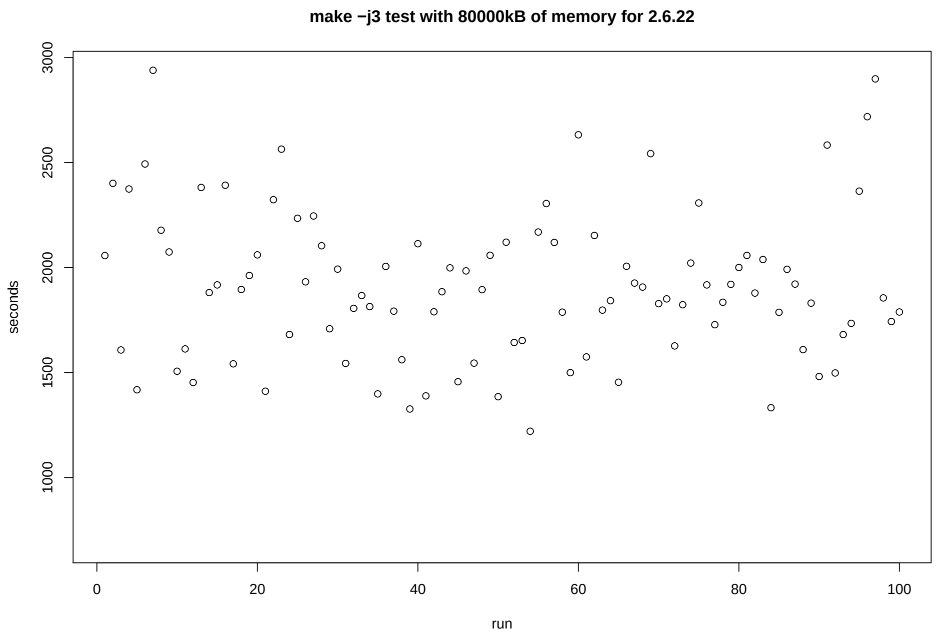

Figure 6.5: Vanilla Linux 2.6.22.17 running make -j3 test with 80MB of available memory.

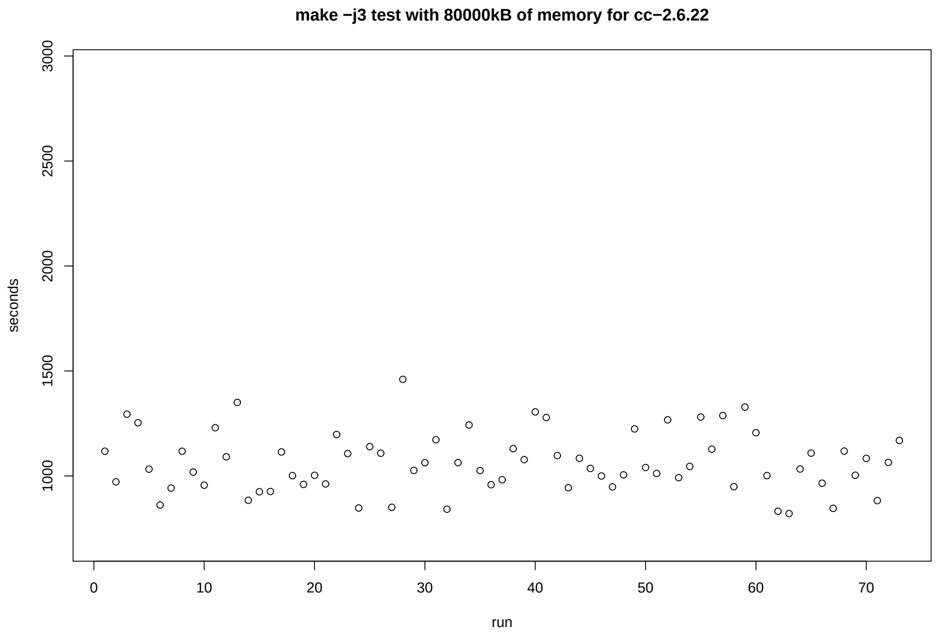

Figure 6.6: Linux 2.6.22.17 with compressed caching running make -j3 test with 80MB of available memory.

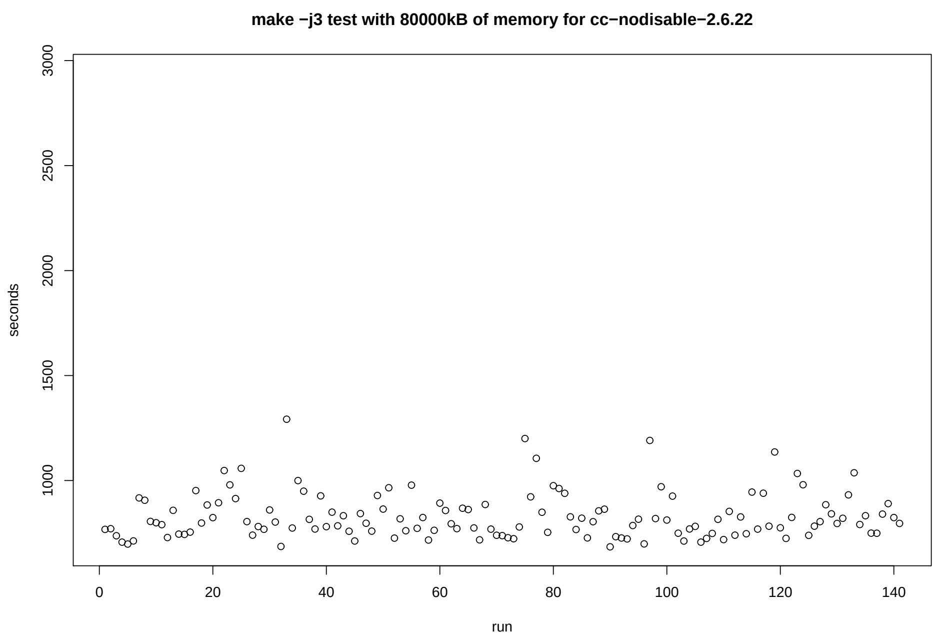

Figure 6.7: Linux 2.6.22.17 with compressed caching, no c.c. disabling, running make -j3 test with 80MB of available memory.

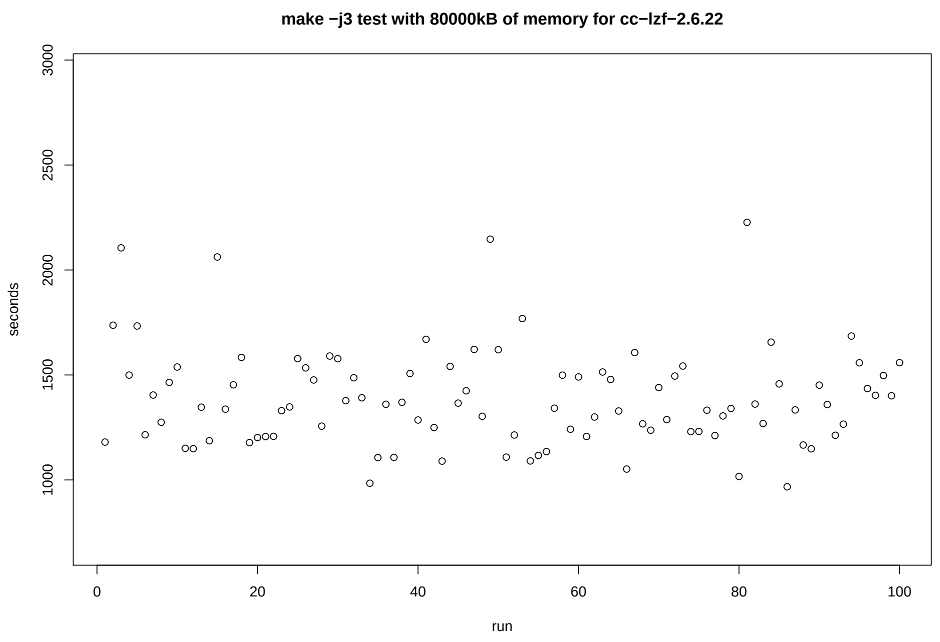

Figure 6.8: Linux 2.6.22.17 with compressed caching, using LZF compression algorithm, running make -j3 test with 80MB of available memory.

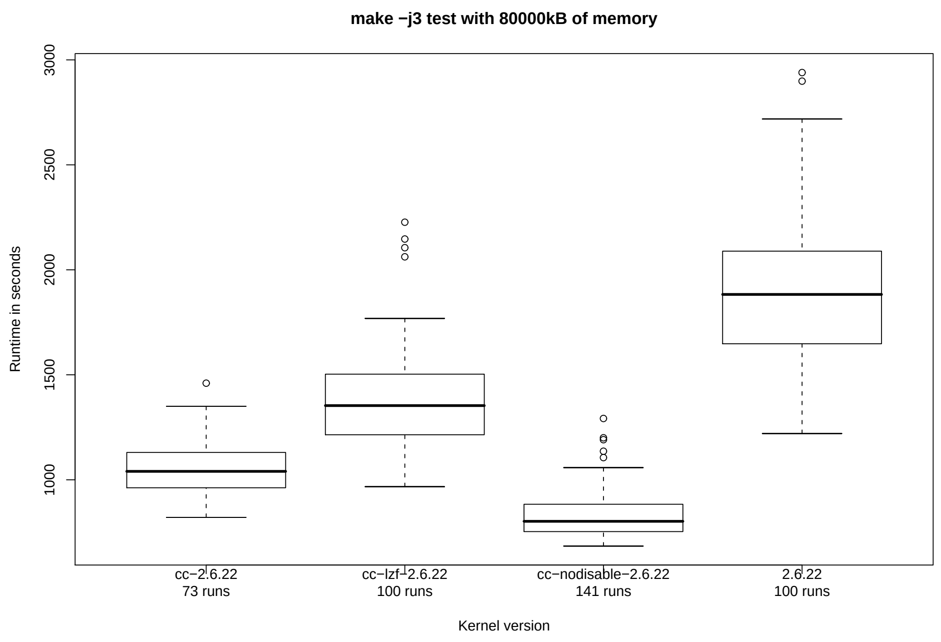

Figure 6.9: Boxplot comparison of make -j3 80MB test

kernels, there is a performance increase of 57% with ”cc-nodisable”, 44% with ”cc” and 28% with ”cc-lzf”. This means that there is a potential of 13% performance increase in tweaking or altering the disabling logic for the compressed cache.

We can see the same pattern for results of the ”make -j3” test with 100MB of available memory (as seen in Figures 6.10, 6.11 and 6.12) as we saw with 80MB (see Figures 6.5, 6.6 and 6.8 and 6.7). Compressed caching has a more narrow range than both the vanilla kernel and the altered compressed caching with LZF compression algorithm. Compressed caching without the disabling logic is again performing better than the other kernels. The reduction in running time compared to the ”vanilla” kernel is 14% for ”cc-nodisable”, 9% for ”cc” and 0.3% for ”cc-lzf”. When more memory is available, the effect of compressed caching is less. This is because the number of potentially saved disk accesses is smaller.

For the ”make -j3” test with 120MB, (see Figure 6.19) the results overlap a lot more, and the effect of compressed caching is even less than with 110MB main memory.

If we compare Figure 6.9, Figure 6.14 and Figure 6.19 we can see that the trend between the different kernels are the same, but that the effect of compressed caching is less the more main memory is available.

The performance increase for this test, which has more main memorpy, is less than the previous. The increase in performance compared to the

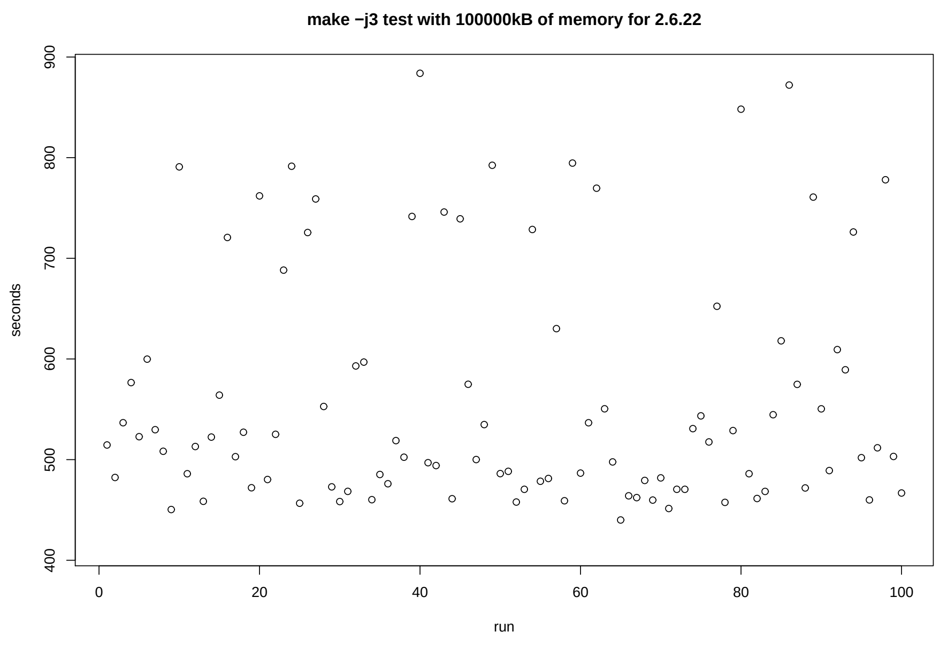

Figure 6.10: Vanilla Linux 2.6.22.17 running make -j3 test with 100MB of available memory.

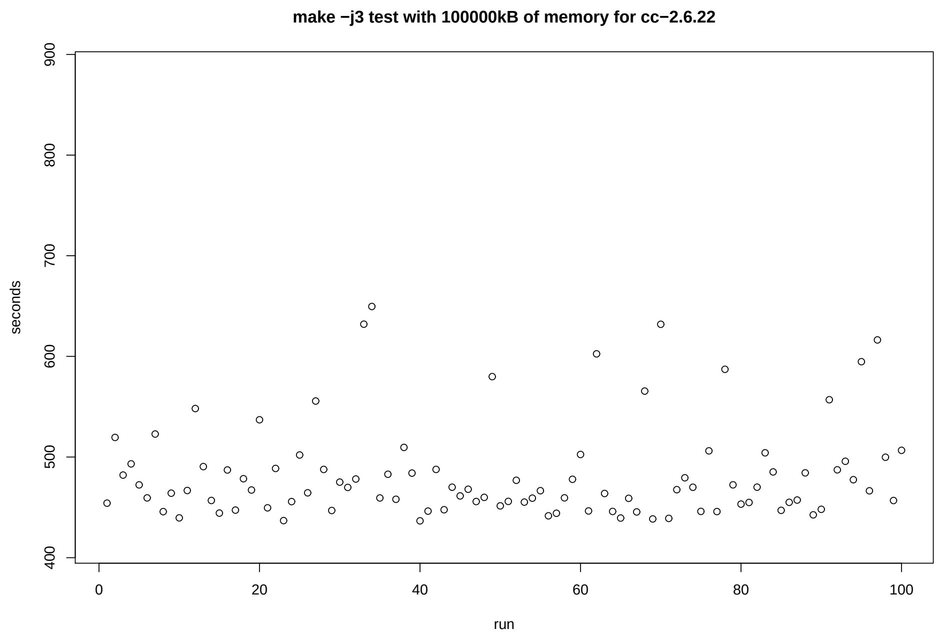

Figure 6.11: Linux 2.6.22.17 with compressed caching running make -j3 test with 100MB of available memory.

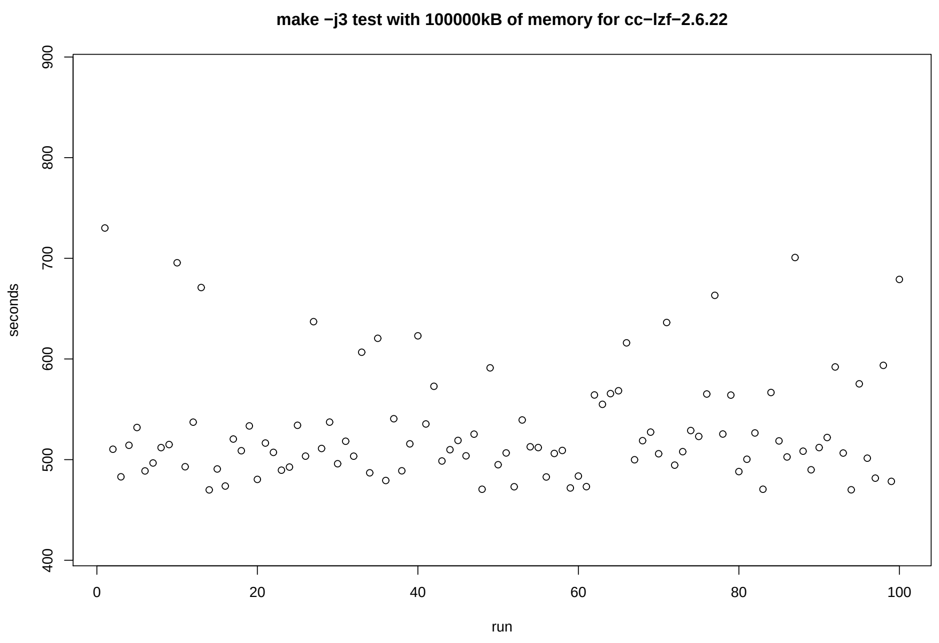

Figure 6.12: Linux 2.6.22.17 with compressed caching, using LZF compression algorithm, running make -j3 test with 100MB of available memory.

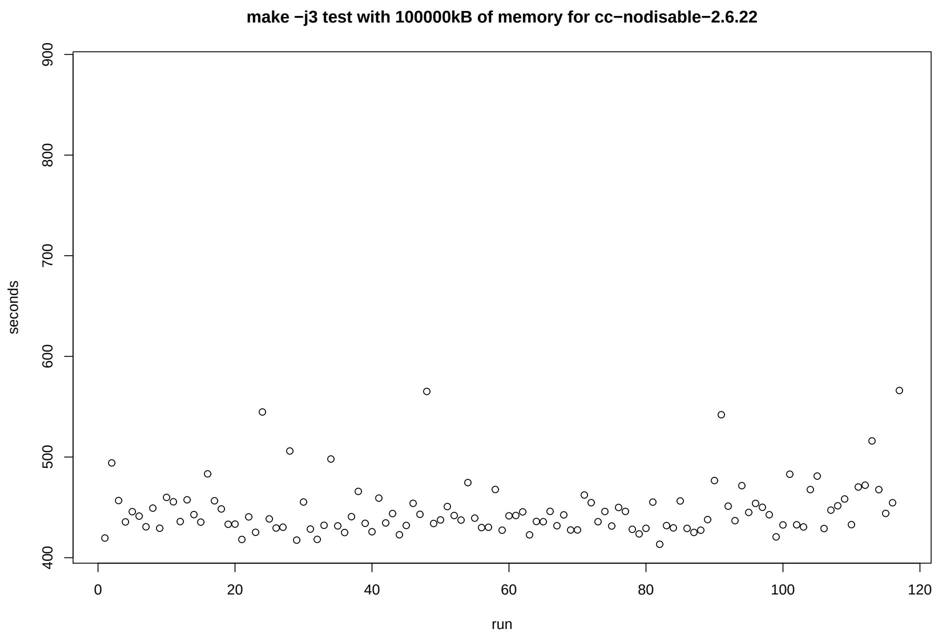

Figure 6.13: Linux 2.6.22.17 with compressed caching, no c.c. disabling, running make -j3 test with 100MB of available memory.

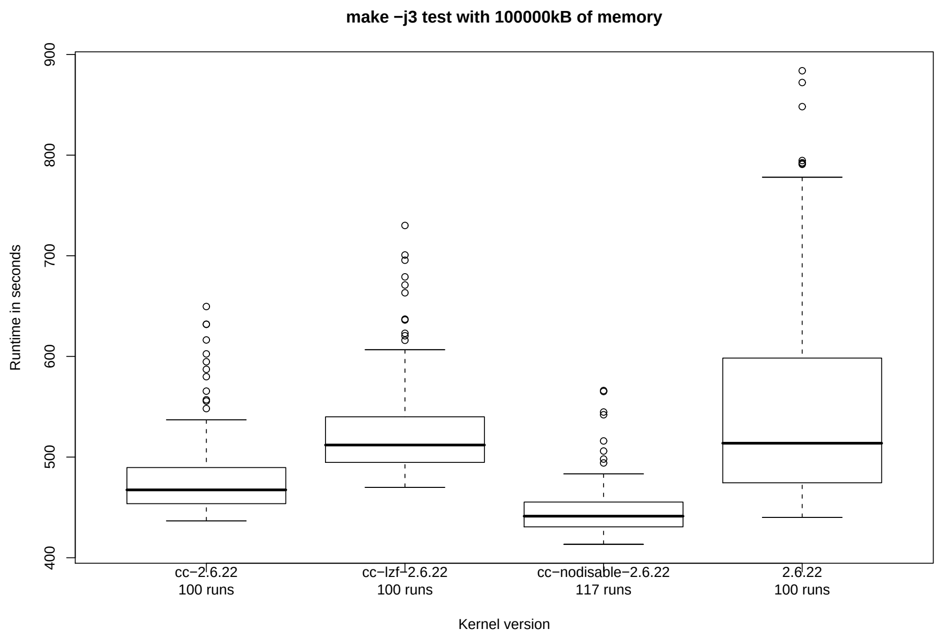

Figure 6.14: Boxplot comparison of make -j3 100MB test

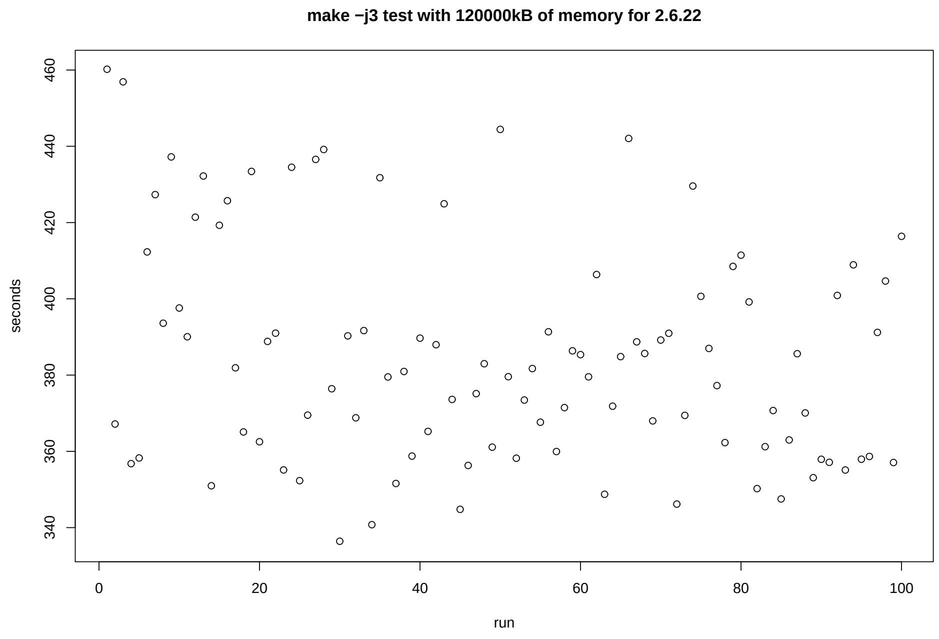

Figure 6.15: Vanilla Linux 2.6.22.17 running make -j3 test with 120MB of available memory.

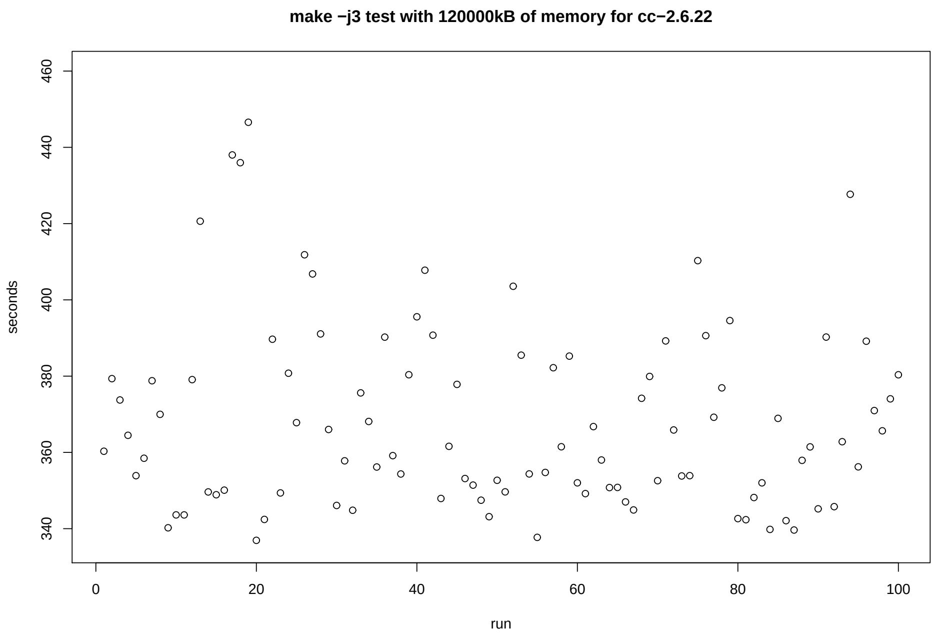

Figure 6.16: Linux 2.6.22.17 with compressed caching running make -j3 test with 120MB of available memory.

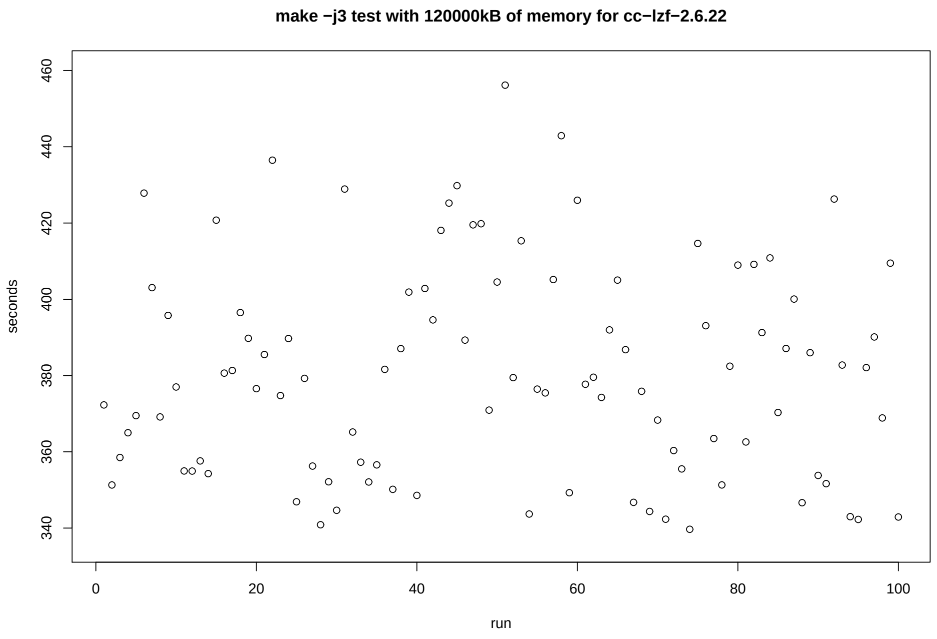

Figure 6.17: Linux 2.6.22.17 with compressed caching, using LZF compression algorithm, running make -j3 test with 120MB of available memory.

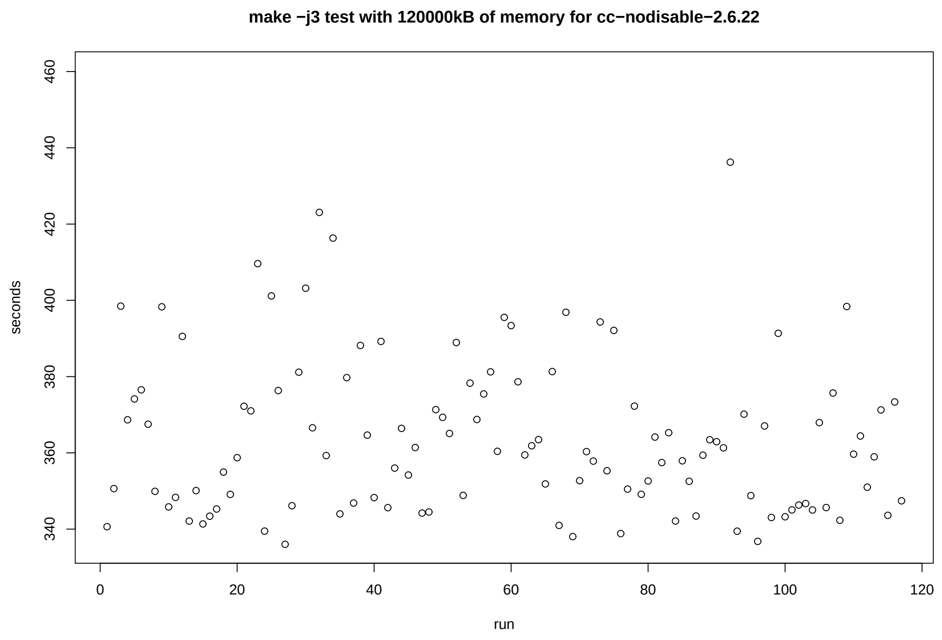

Figure 6.18: Linux 2.6.22.17 with compressed caching, no c.c. disabling, running make -j3 test with 120MB of available memory.

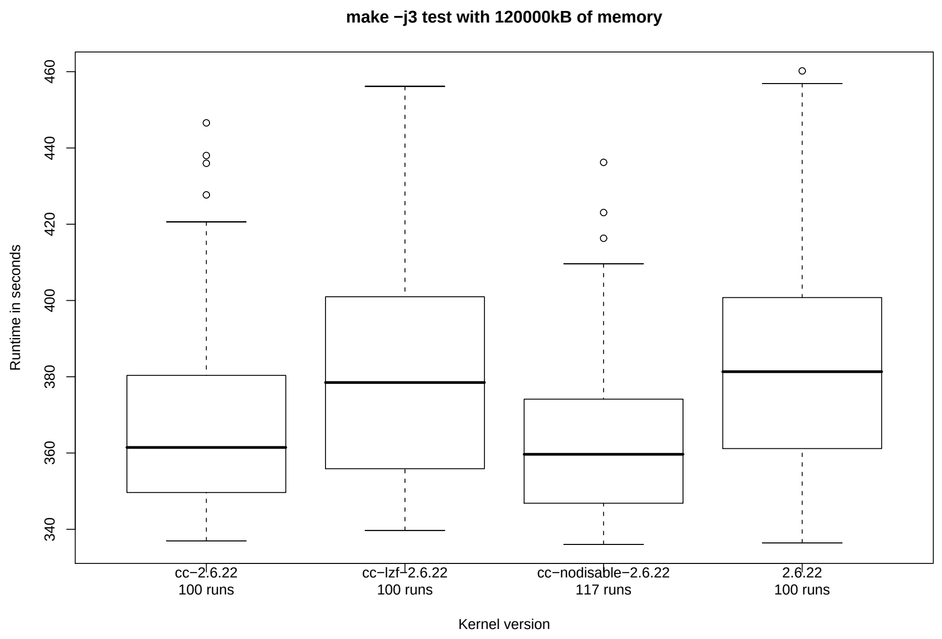

Figure 6.19: Boxplot comparison of make -j3 120MB test

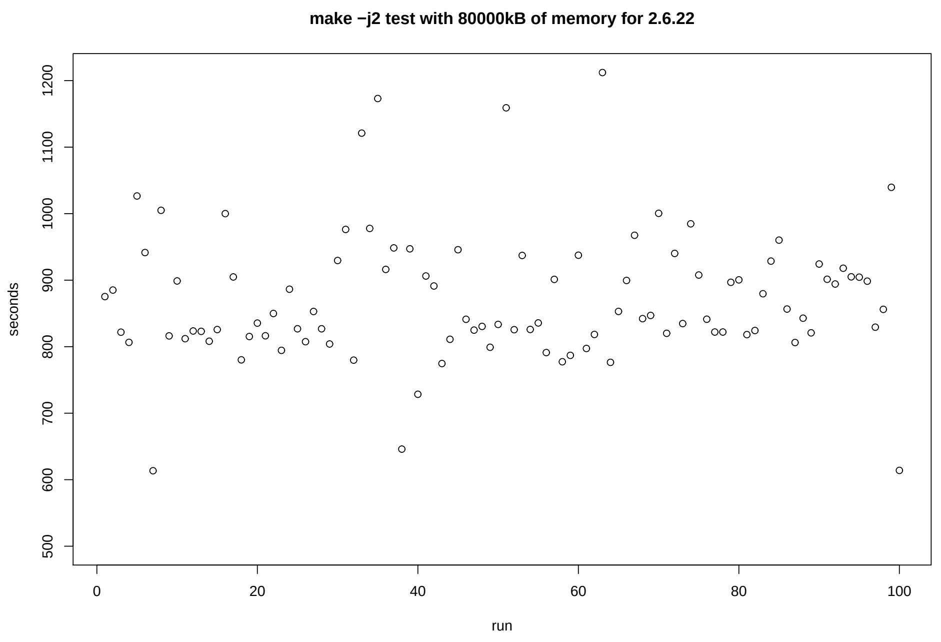

Figure 6.20: Vanilla Linux 2.6.22.17 running make -j2 test with 80MB of available memory.

"vanilla" kernel is 5% for "cc-nodisable", 5% for "cc" and 0.7% for "cc-lzf".

Studying the scatter plots from the "make -j2" test with 80MB of available main memory in Figures 6.20, 6.21 and 6.22, we find that the compressed caching kernel has less variance in the results than the vanilla kernel and that the median is significantly lower. Reduction in median running time compared to "vanilla" kernel is 30% for "cc-nodisable" and 24% for "cc".

Studying the result of the "make -j1" test with 80MB of main memory in Figures 6.20, 6.21 and 6.22, we find that the compressed caching kernel has a larger variance in the results than the "vanilla" kernel and that the median is significantly lower. The reason for this is not obvious, but a plausible explanation is that the order of accesses to the compressed cache between the routinely run PFRA and application can make the process have variance in how much disk I/O is needed. Consider that case where the workload is about to access the oldest compressed page, if the PFRA shrinks the cache first the process ends up needing to do disk I/O. This in turn changes the timing between the application and the PFRA, which adds to the variance. The variance is smaller than for both "make -j2" and "make -j3" tests with the same kernels and amount of main memory. Compared to the "vanilla" kernel the reduction in run time is 14% for "cc-nodisable" and 11% for "cc".

Comparing the results from "make -j1", "make -j2" and "make -j3 (see

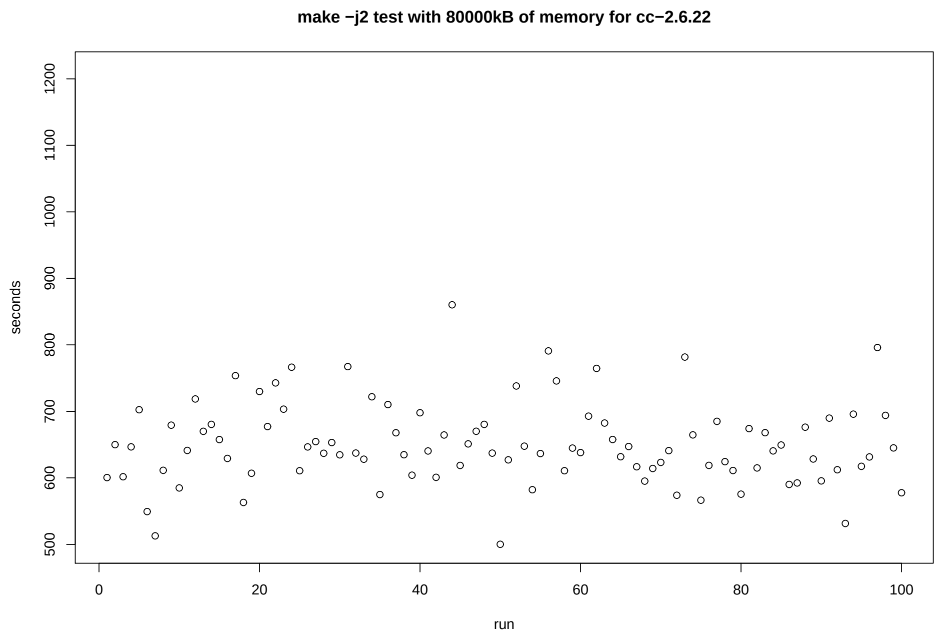

Figure 6.21: Linux 2.6.22.17 with compressed caching running make -j2 test with 80MB of available memory.

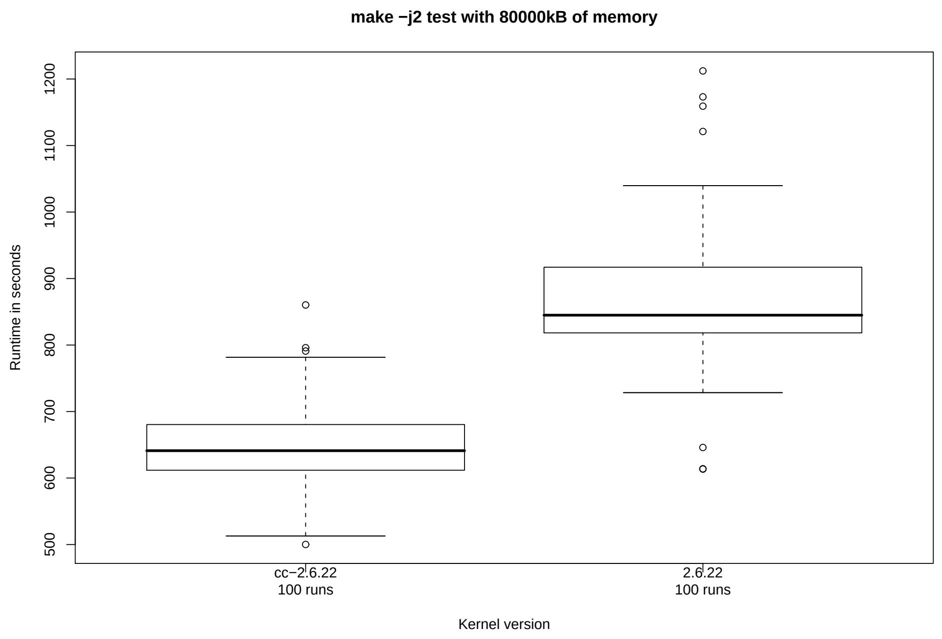

Figure 6.22: Boxplot comparison of make -j2 80MB test

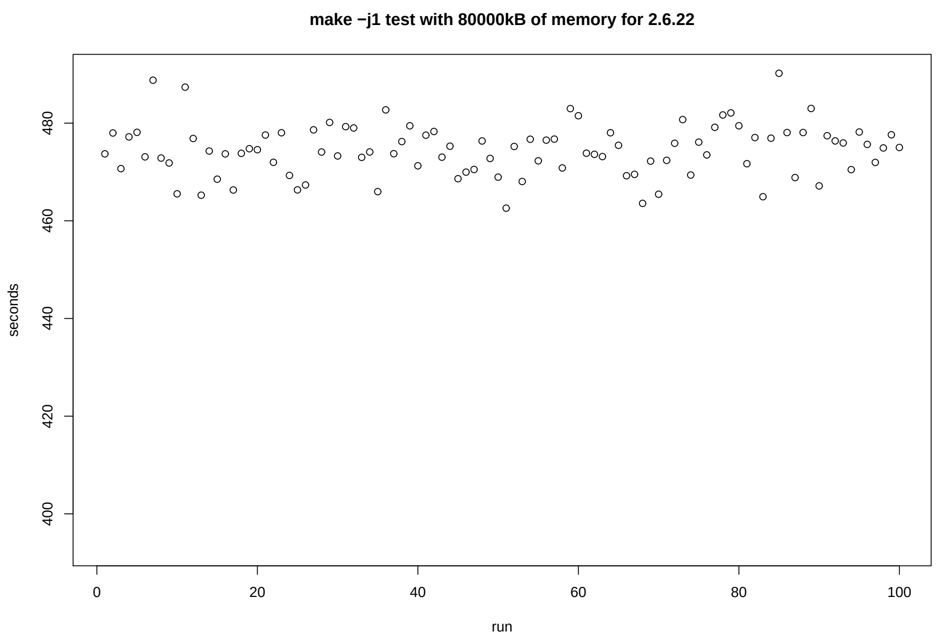

Figure 6.23: Vanilla Linux 2.6.22.17 running make -j1 test with 80MB of available memory.

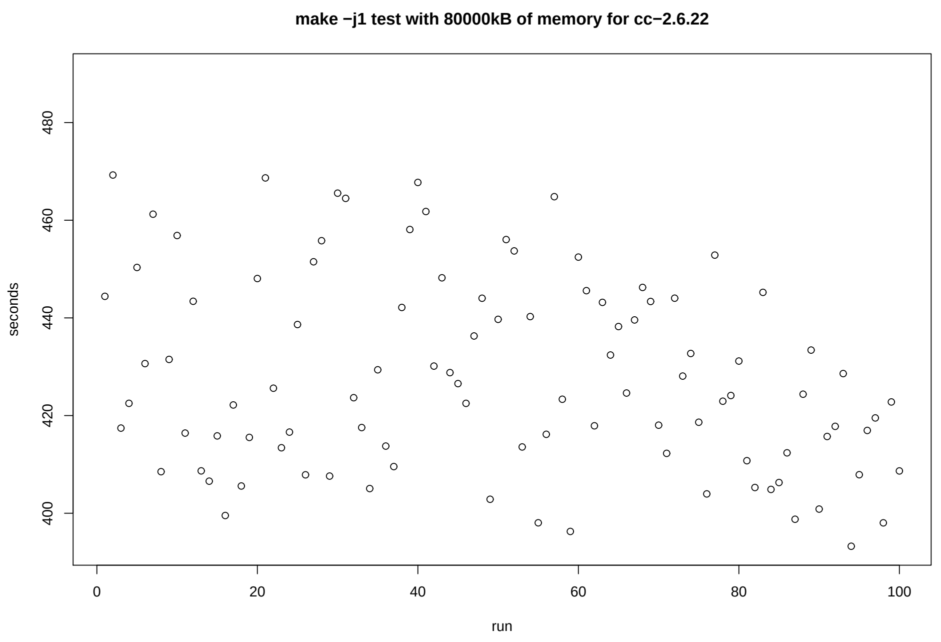

Figure 6.24: Linux 2.6.22.17 with compressed caching running make -j1 test with 80MB of available memory.

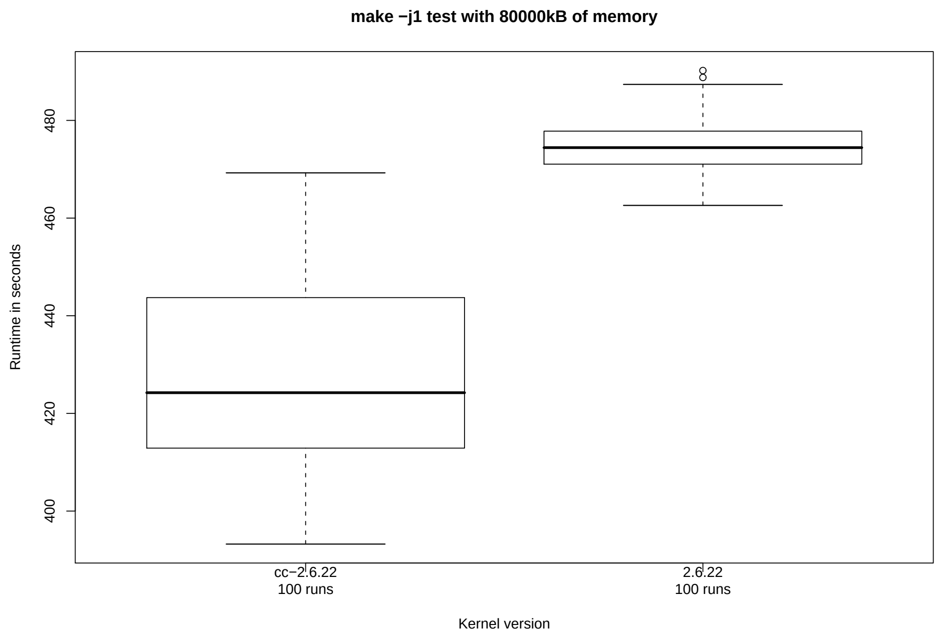

Figure 6.25: Boxplot comparison of make -j1 80MB test

Figures 6.25, 6.22 and 6.9) with 80MB of main memory available, we see that the relative effect of compressed caching is less when less processes are running. We also see an increase in running time between the workloads: for the ”vanilla” kernel that ”make -j2” is 78% slower than ”make -j1”, and ”make -j3” is 123% slower than ”make -j2”. In comparison the same values for ”cc-nodisable” are 44% and 37% and for ”cc” they are 51% and 62%.

The sort workload is meant to be a worst-case of compressed caching, and as expected the vanilla kernel has, as can be seen in Figure 6.26 and Figure 6.27, a lower median and lower variance in the results than compressed caching. Compared to the ”vanilla” kernel the ”cc-nodisable” kernel is 11.4% slower and ”cc” is 6.57% slower.

As we can see, the compressed caching kernel with disabling of the compressed cache has better performance than without any disabling logic, this is what we aimed for. When more main memory is available for the sort workload the difference between compressed caching and the unmodified kernel becomes less. An exception to this is for the kernel with no disabling logic where the time to run the workload becomes larger when increasing the available main memory from 40MB to 60MB. It is hard to explain this, an educated guess is that the size of the cache is sized more optimal when the expense list is larger compared to the profit list. We did not have time to look into this at any detail.

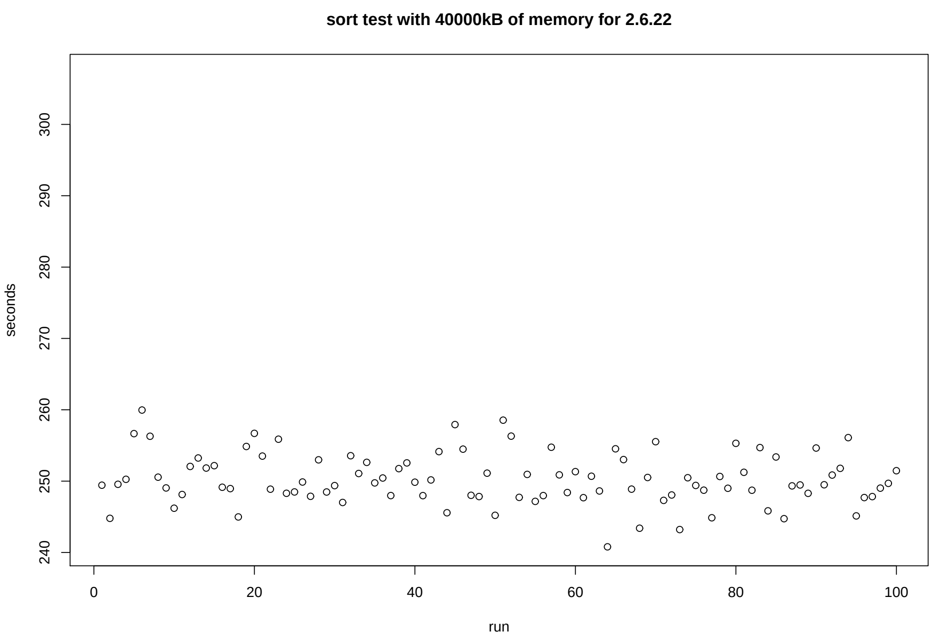

Figure 6.26: Vanilla Linux 2.6.22.17 running sort test with 40MB of available memory.

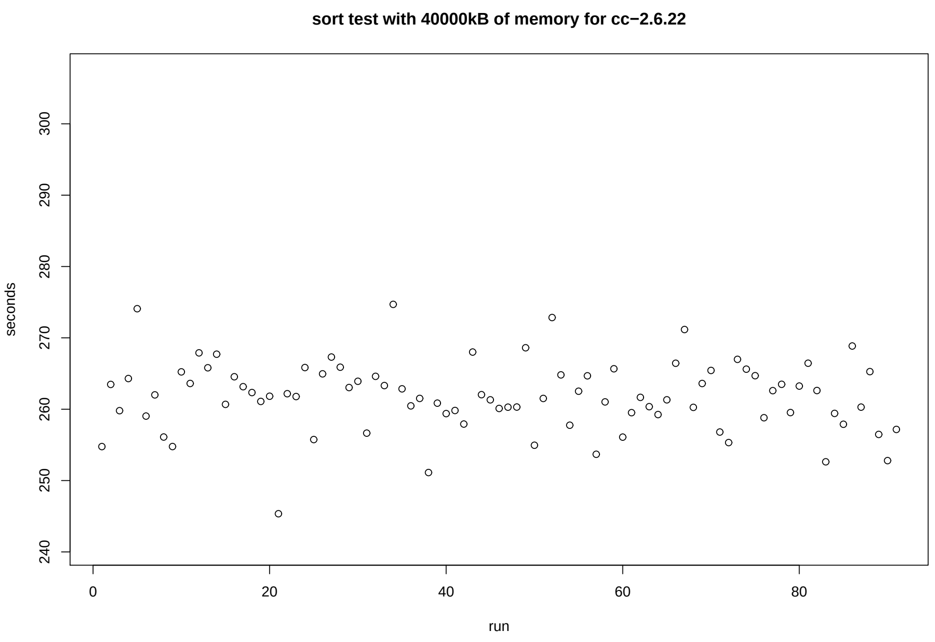

Figure 6.27: Linux 2.6.22.17 with compressed caching running sort test with 40MB of available memory.

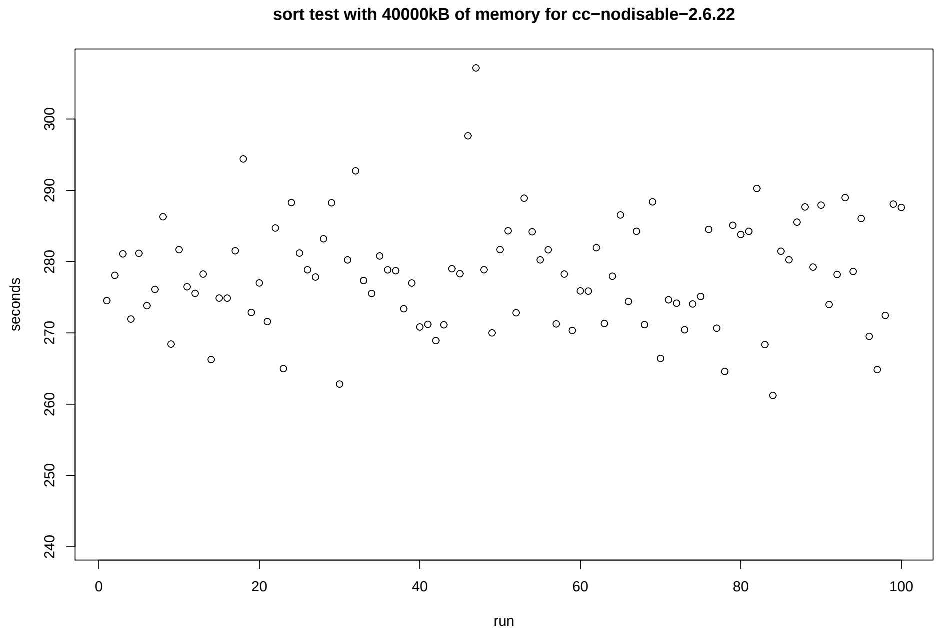

Figure 6.28: Linux 2.6.22.17 with compressed caching, without disabling cache, running sort test with 40MB of available memory.

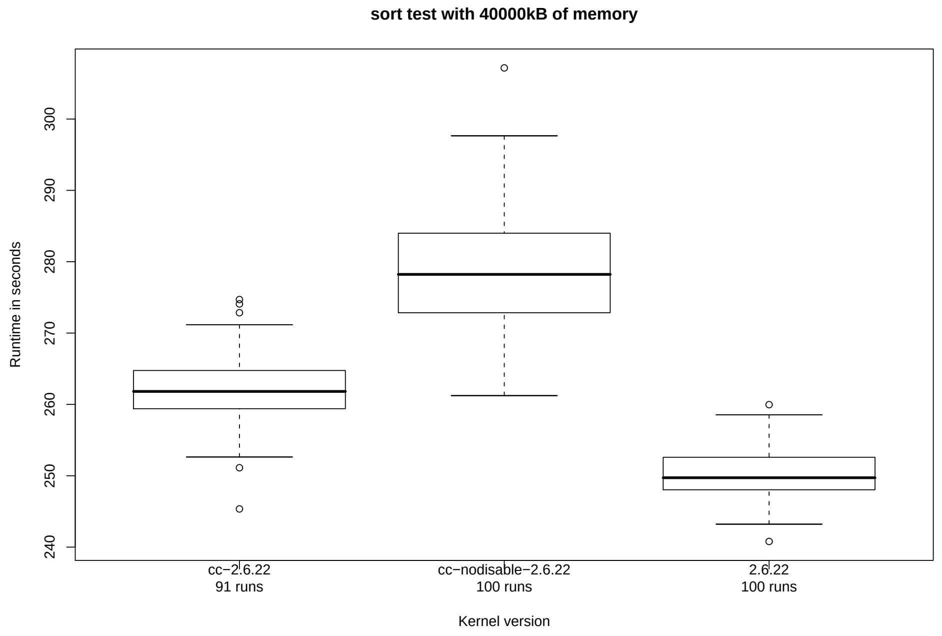

Figure 6.29: Boxplot comparison of sort 40MB test

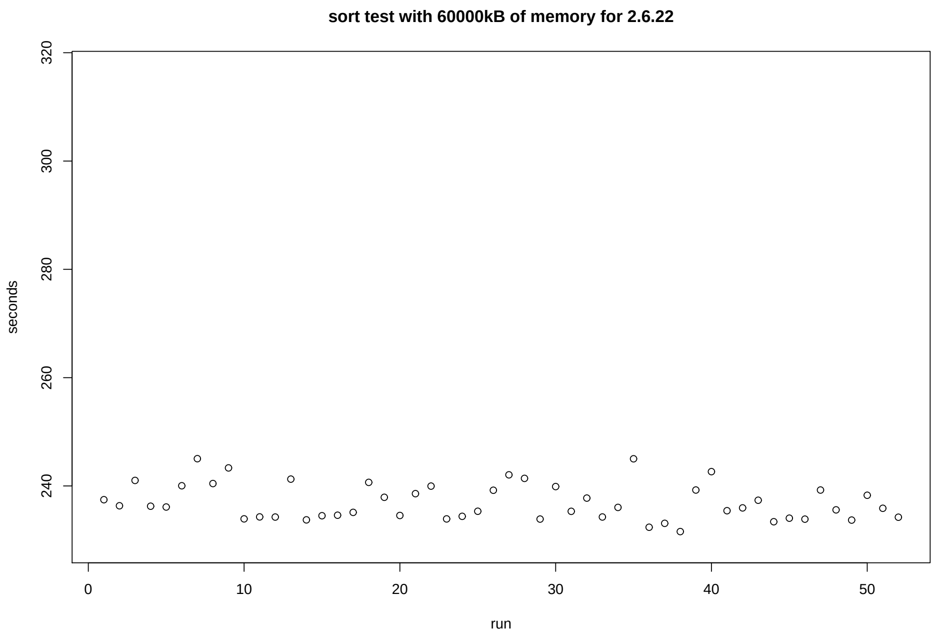

Figure 6.30: Vanilla Linux 2.6.22.17 running sort test with 60MB of available memory.

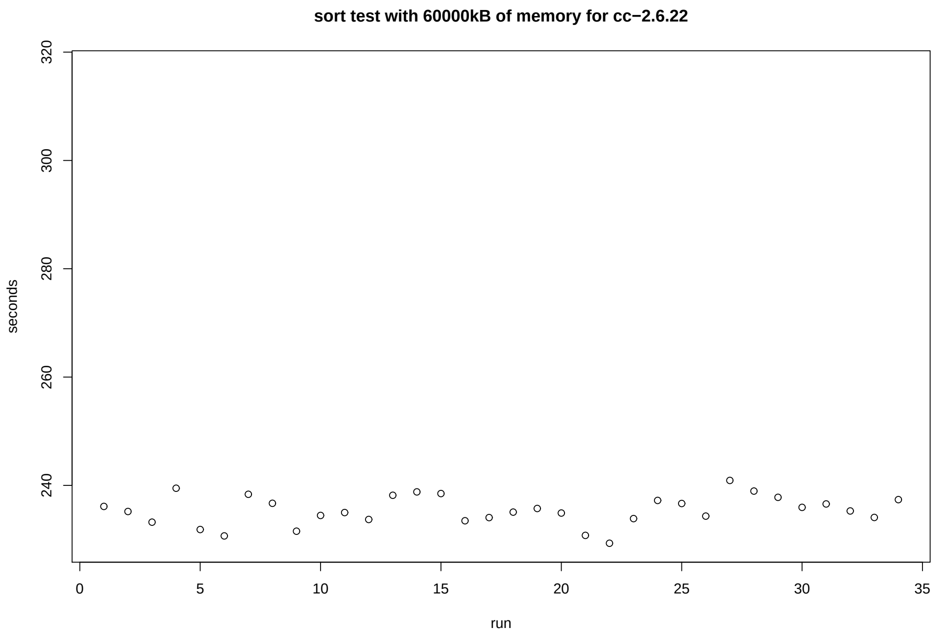

Figure 6.31: Linux 2.6.22.17 with compressed caching running sort test with 60MB of available memory.

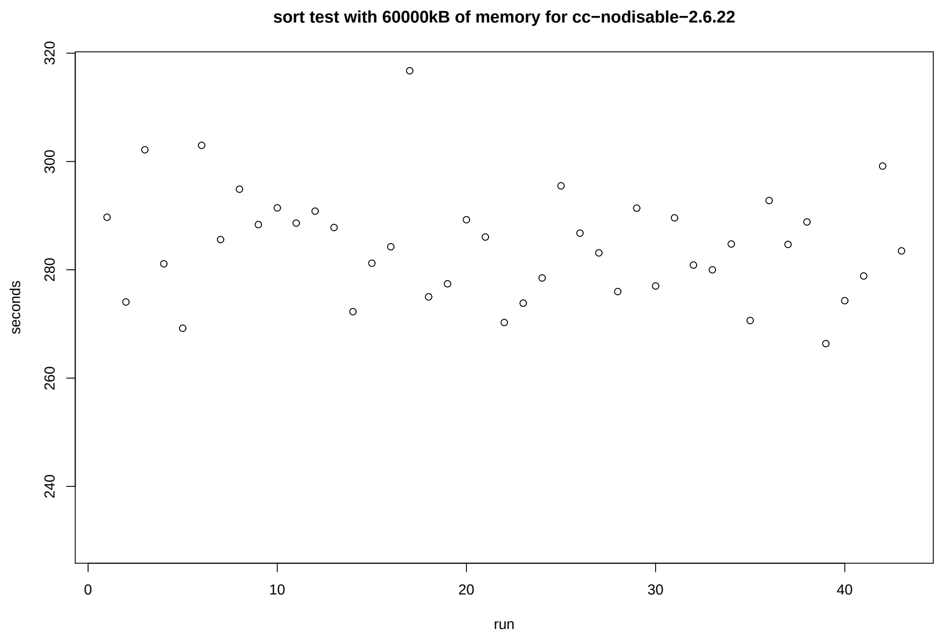

Figure 6.32: Linux 2.6.22.17 with compressed caching, without disabling cache, running sort test with 60MB of available memory.

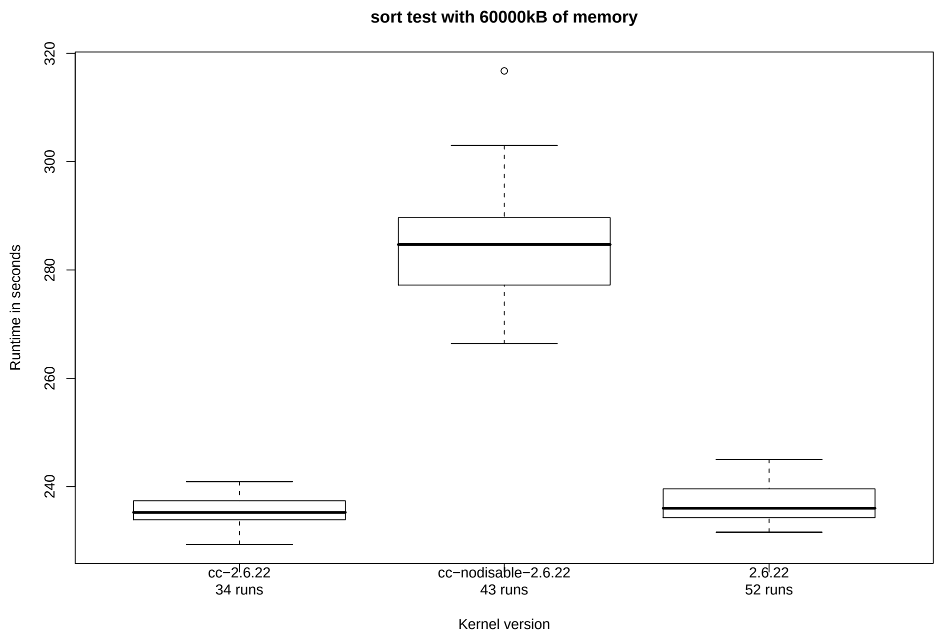

Figure 6.33: Boxplot comparison of sort 60MB test

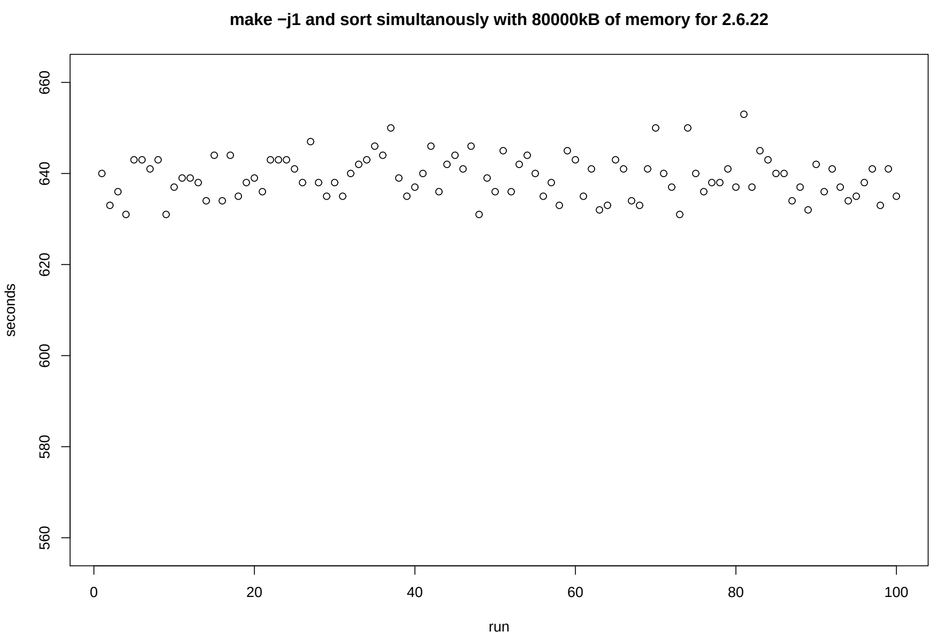

Figure 6.34: Vanilla Linux 2.6.22.17 running sort test with 80MB of available memory.

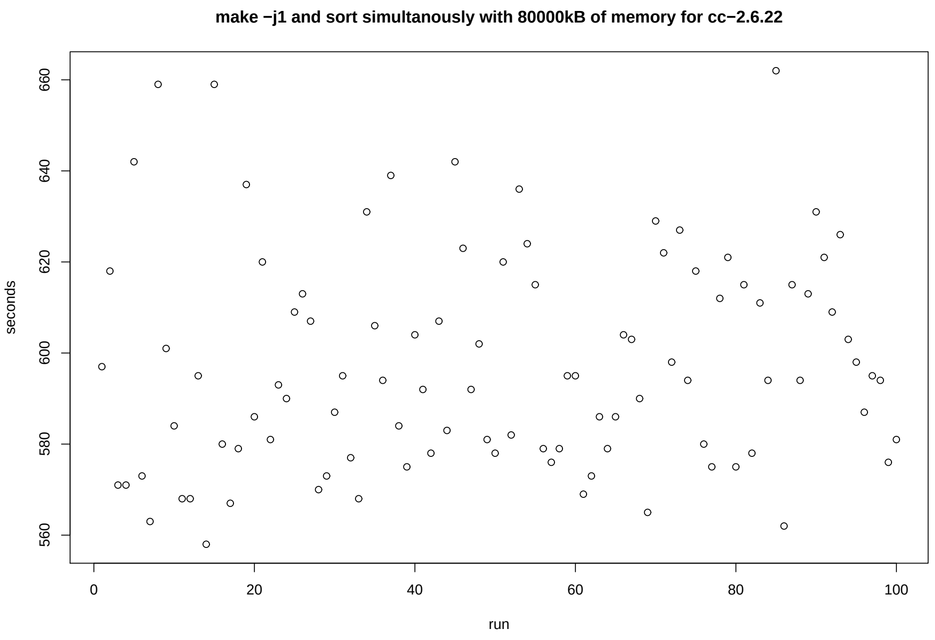

Figure 6.35: Linux 2.6.22.17 with compressed caching running sort test with 80MB of available memory.

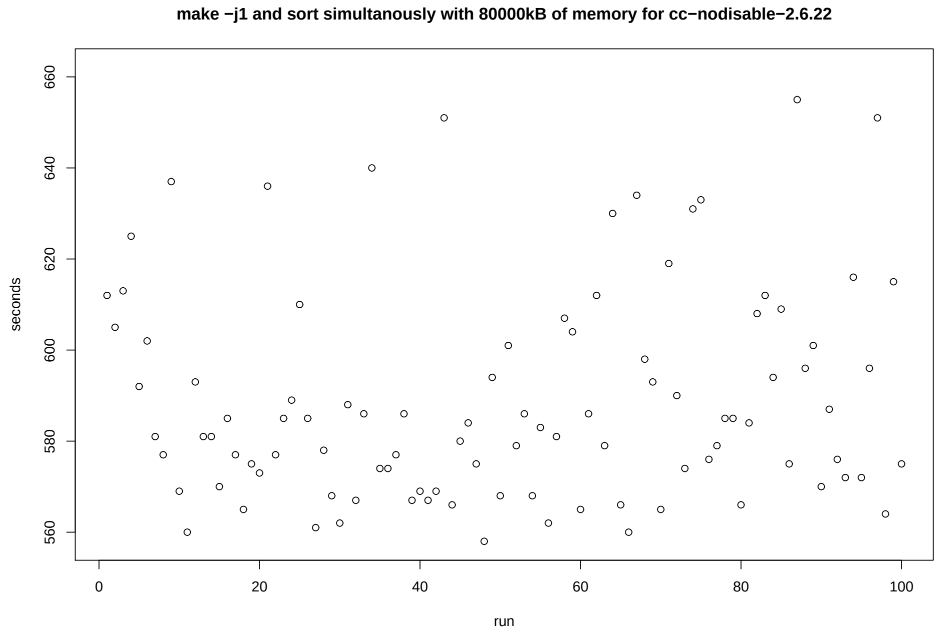

Figure 6.36: Linux 2.6.22.17 with compressed caching, without disabling cache, running sort test with 80MB of available memory.

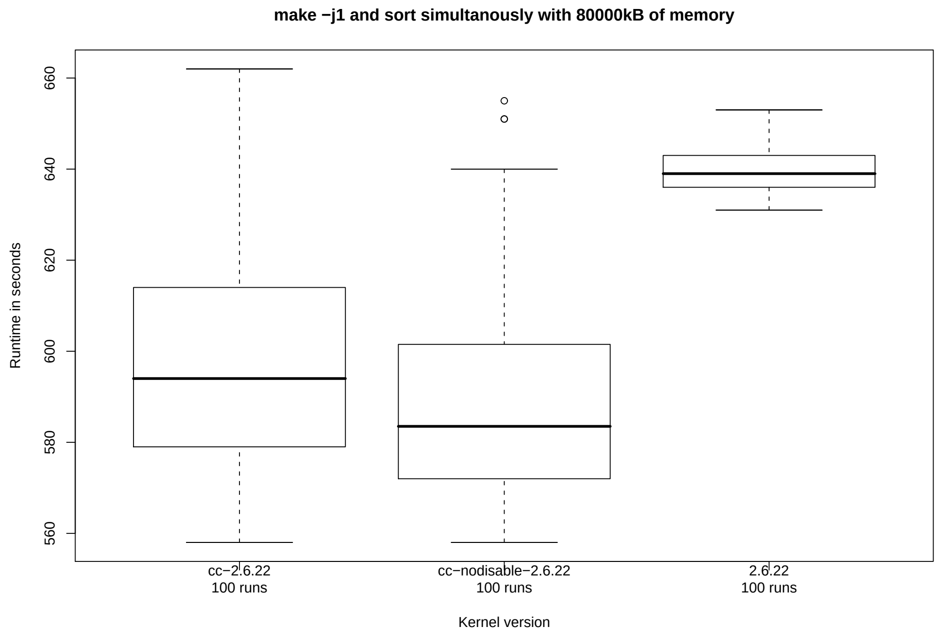

Figure 6.37: Boxplot comparison of sort and compile test with

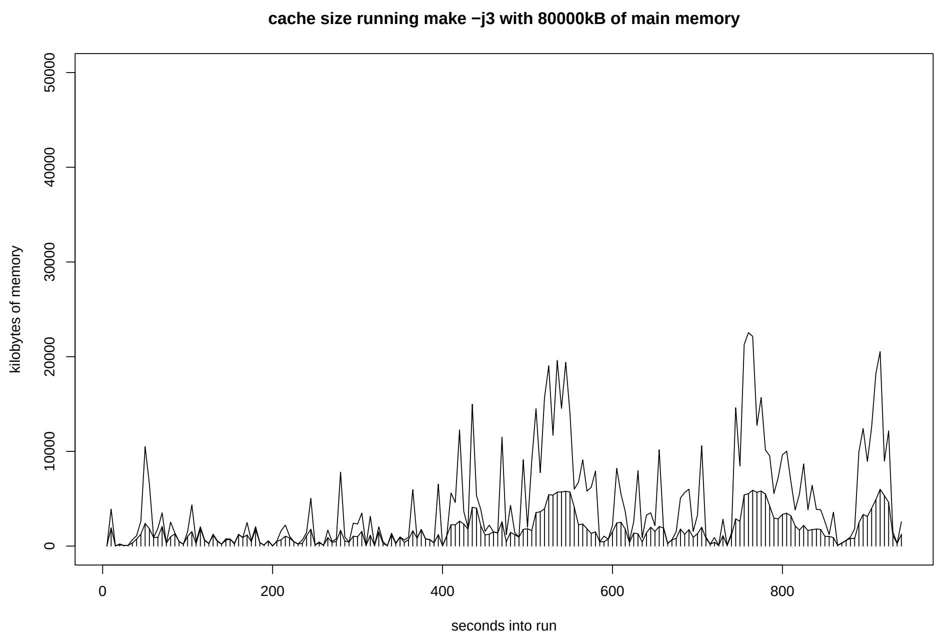

Figure 6.38: Following the cache size through a make -j3 test with 80MB of memory

6.5 Evaluation

Looking at Figures 6.38, 6.39, 6.40 and 6.41 which track how much memory is stored in the compressed cache and how much main memory is used to store them, we can see a few things. Most importantly, the compressed cache never uses more main memory than it stores, which is by design. The global compaction explained in Section 4.15 makes sure that we do not have fewer than one compressed page in each cell. We also see that under the run of the test we find spikes where a lot of data is compressed with a high ratio.

Looking at Figure 6.40 and 6.41, which are both running the sort workload with 40MB of available memory, we would expect to see five spikes since in the sort workload, sort is run five times in a row. For "nodisable" (see Figure 6.41 we see three large spikes and two smaller ones, all about 60 seconds. The 60 second interval is caused by the sort processes using about 60 seconds and writing the result to file. The reason the second and fourth is smaller is probably a result of the access pattern: if the last access to the cache was in the profit list we get a spike, if the last access to the cache was in the expense list we do not get such a spike. If we look at the "cc" (see Figure 6.40) cache size through a sort it has no spikes, this is the behavior we want, because the cache is never reused so most memory areas

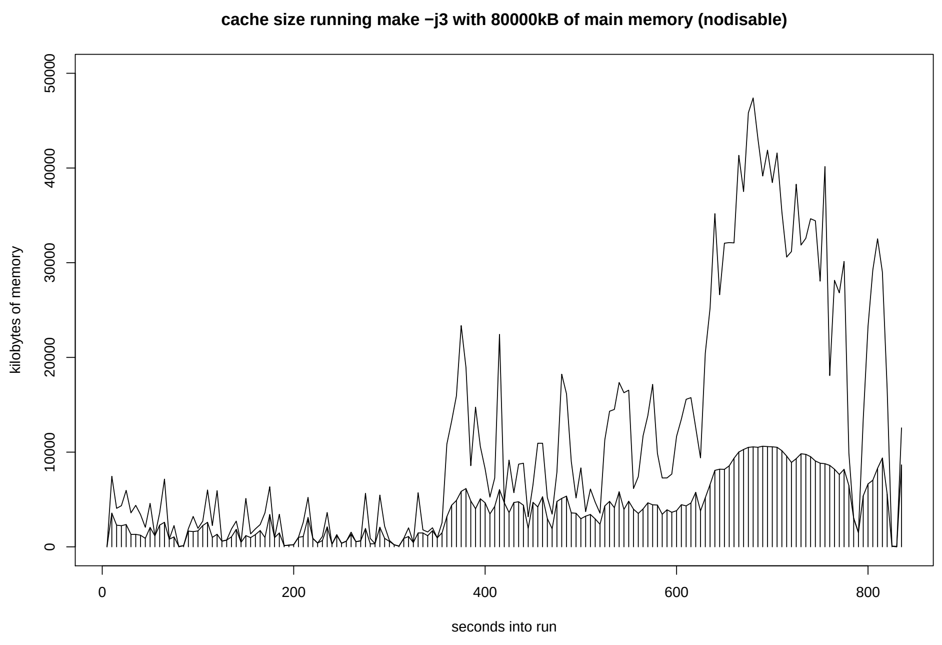

Figure 6.39: Following the cache size through a make -j3 test with 80MB of memory and no disabling logic.

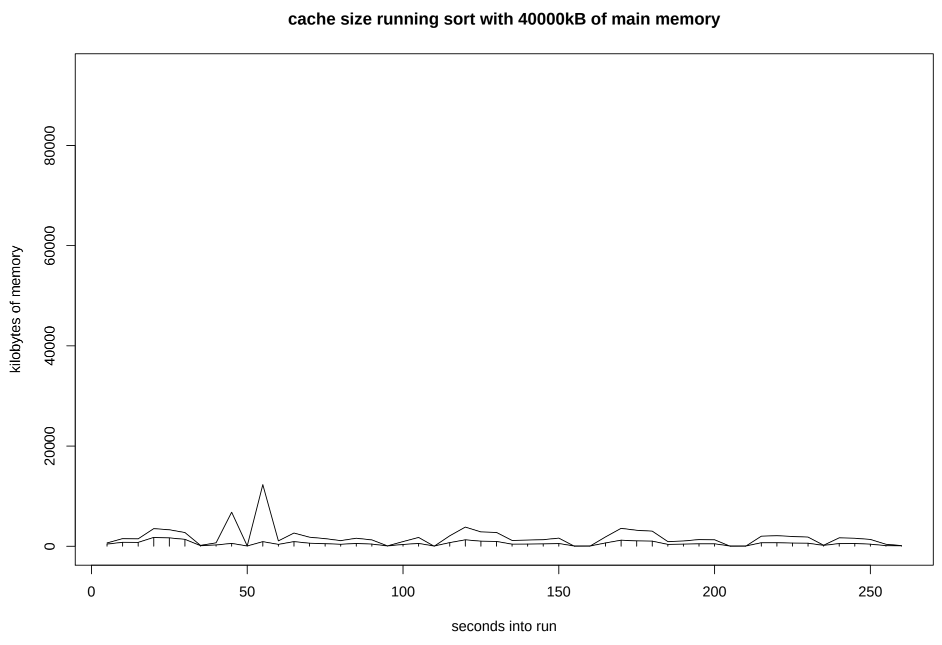

Figure 6.40: Following the cache size through a sort of a 110MB large file with 40MB of memory available

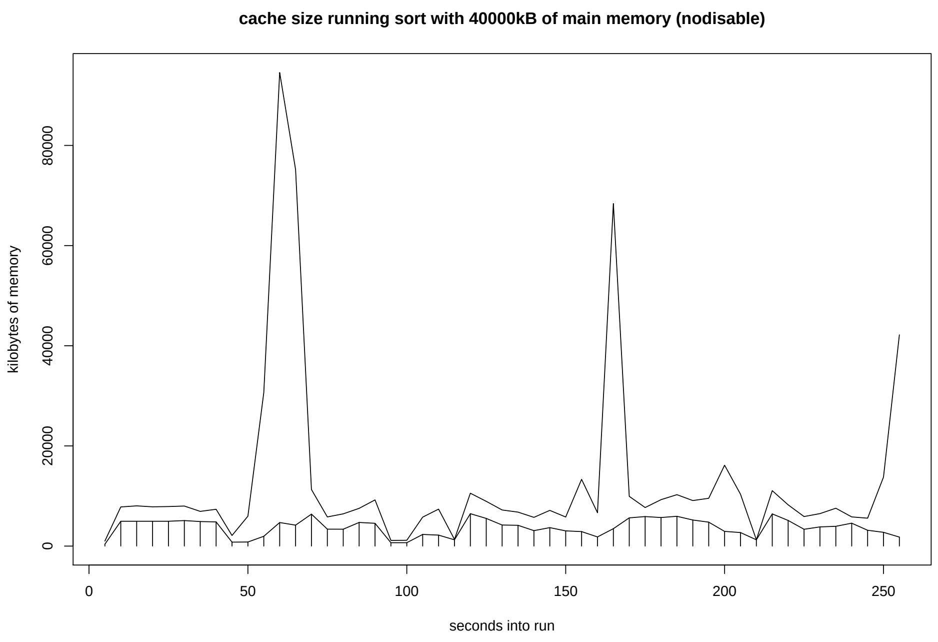

Figure 6.41: Following the cache size in the same scenario as in Figure 6.40, but with no disabling of compressed caching.

are disabled.

The file being sorted is a file consisting of the ascii string representation of a number between 0 and 32767 and it has a size of 109MB. This gives a low entropy, because only the symbols from 0 to 9 and newline are used. This is reflected in Figure 6.41 where we can see more pages in the compressed cache than can be stored uncompressed in main memory.

One of the drawbacks of compressed caching is that the worst-case is slower than the worst-case of an unmodified kernel, this is because of the useless compression and memory usage under such circumstances. The worst-case can always be made to occur by continually changing how a memory area is behaving to avoid detection. It is, however, less likely to occur the more detection logic is put in, because it would take less time to detect such changes.

We see from the results that never turning off compressed caching is advantagous in all the test except the sort test. As we can see when comparing the cache size (see Figure 6.38 and 6.39) the compressed caching with cache disabling logic has a much smaller cache size than the kernel without the disabling logic. This happens because the disabling logic is disabling too aggressively, adjusting the parameters to be less aggressive can make it better, but then it could also make the sort test slower.

Chapter 7

Conclusion

This thesis presents the design, implementation and evaluation of compressed caching in Linux. We describe scenarios that earlier implementations have not discussed and propose solutions to those. Based on these solutions we design and implement compressed caching for Linux. We then test and evaluate these solutions.

7.1 Summary of contributions

The VM of the Linux kernel is a large and complex system which is in constant flux. This makes it hard to find relevant documentation that is up to date. It is also a large task in itself to get a good overview of how the VM works and interacts with other subsystems.

The design and implementation went through several phases: We started with an architecture specific implementation altering the page tables and page fault handling. We took some bits from the swap entry identifier to mark it as a compressed page and used this to look it up in a radix tree only for the purpose of compressed caching. This has three major drawbacks: (1) it only works for anonymous pages, (2) supports only one architecture and (3) limits us to store pages into a cache, but not be able to evict compressed pages to replace them with newer ones on demand.

In the next iteration, we changed the design to be architecture independent as much as possible and made the compressed cache have its own LRU list, so that we could evict the oldest page if we needed to. We still do allocate storage for the compressed pages with *kmalloc()*. From this point, storing a file cached page into the cache is a very natural extension, because the data structure for the page cache is shared between both file backed pages and swap backed pages.

The next step is to implement the cell allocator. It became clear that we need to change the locking of the compressed cache to avoid invalid locking. To accomplish this, we separate out freeing of compressed pages in its own step, so that we can free compressed pages later.

We started by implementing an adaptive resizing heuristics on the cache, trying to use more of the concepts developed by Kaplan et al. [12] directly, but it turned out to be far more difficult and time consuming than we originally hoped for, and we abandoned it. Instead, we implemented the simple and working algorithm of Castro et al.[8], which is an approximation to the same theory.

Next, we focus on the inclusion and handling of marker pages. We implemented this in such a way that we can use it for disabling of compressed caching later. We build on top of the marker pages to implement disabling of the compressed caching for memory areas when their use of the cache is low.

Some of these steps could have been skipped, if we had a deeper understanding of the VM. Unfortunately, gaining such information would have probably taken just as much time.

7.2 Critical Assessment

This section describes what we would have done differently, if we had the insights we have today when we started the thesis. Since our master thesis is limited in scope and time it opens more questions than it answers.

One of the experiments we wanted to do, but did not have time for, is a comparison of global disabling of compressed caching and our disabling per memory area. To do this we must design and implement global compressed caching. This would have been very interesting considering the results Castro et al.[8] achieved implementing it.

Most of the time spent on this master thesis was spent on the first fully working design and implementation, and unfortunately there was not enough time to do enough testing and using that feedback into enhancing the results further. There are several reasons for this, the most pronounced one is the time it takes to get reliable and comparable results. The reason it takes so much time is the high variation in the results, which means that we can not see a trend with only a few runs and thus have to repeat runs until we have a high enough confidence in the results. This limited us to use only a handful of workloads. If we had more time testing other interesting real life workloads, such as web or mail servers, we certainly would have done so. Tweaking of parameters such as when to enable and disable compressed caching, how many marker pages should be allowed in the compressed cache and how much history should be kept is something that we want to look more into.

To better handle the time constraint, a splitting of this master thesis into two or more parts would have allowed for more focus on the issues. A good split would have been the storage of compressed pages and the implementation of compressed caching. The reason we think that this is important is that we did not have any time to do any tests on how design decisions for the cell allocator, such as look up of free cells and compacting strategies, affects performance and fragmentation. The cell allocator was designed and implemented without any iterative feedback from any test results. Some of the things we did not have time for are mentioned were in the future work section.

7.3 Future work

There are several issues and possible optimizations to be explored for compressed caching. To implement, test and evaluate extensions using this thesis design and implementation as a base, can be achieved with considerable less effort than implementing compressed caching from scratch. Here are some extensions that we believe are worth looking into:

- Testing more compression algorithms and perhaps adapt special compression algorithms when specific data is detected to achieve better compression. We only try two compression algorithms in this master thesis and we only use one at a time for the complete system.

- Changing the memory allocator: the interface used is clearly defined and changing the allocator should be straight forward. There are probably enhancements that could be done to the cell allocator such as locating free cells and global compacting.

- The algorithm to adapt the size of the cache can easily be changed and evaluated. We tried to alter the block layer to measure how long a request is in the I/O queue before being sent to the actual device. This was meant to be a way to measure how busy the I/O device is, but it would be a too large change to the block layer subsystem, so we decided to go with another approach.

- Evaluate compressed caching for small devices that uses swapping or file systems over the network, where sending page traffic over the air is expensive considering power usage. In this thesis, we only run tests on commercial-off-the-shelf desktop hardware, which is not limited by battery capacity. This is briefly discussed by Briglia et al.[1].

- Try to compress several pages at a time, this could reduce entropy considerably and increase the compression ratio. This is done per page in our implementation, but it is doable by changing a few functions. A suggestion on how to do this is to wait until a number of pages has been added to the compressed cache before compressing them.

- Storing compressed pages without decompressing them can achieve better performance by not decompressing pages unnecessarily. This includes changes to the swap subsystem, which is why we did not do this. This means adding information for each swap slot if it is compressed or not. If multiple compression algorithms are to be used, which one used in a particular slot must also be stored.

- Evaluating hit ratio disabling of cache in a non-compressed caching setting. Thrashing the file cache is something that occurs not only in the compressed cache scenario when accessing large files, but also in the vanilla Linux kernel. Throwing these out midway through the cache could potentially increase performance. This would be an extension of recent behavior, where the heuristic for removing pages also looks at the pattern and not only the order of accesses to the pages. An idea here would be to evict pages at different intervals through the cache, in other words, let it be less and less likely that a page stays in the cache if similar pages have not ”proven” themselves to be useful before. This will let applications that had a very good cache hit ratio earlier, but are currently idling, be kept longer in memory.

- Look at the possibility to reuse the compressed cache subsystem to also be a dynamic swapping area when no swapping area is available. Our implementation requires a swap area to store anonymous pages in the compressed cache, since it shares the page cache structure with swapping to identify pages. A suggestion on how to implement this would be to have a separate data structure when no swap space is available. Issues that would need to be resolved would be how pages are migrated when a real swap device is added and if file backed pages and anonymous pages should be separated in the LRU list. A motivation to look at this aspect of compressed caching is to avoid wearing down media that can be worn out when it is written to repeatedly.

Most of the points above have not been implemented because of the lack of time, and to limit the scope of this thesis.

References

[1] Leonid Moiseichuk Anderson Briglia, Allan Bezerra and Nitin Gupta. Evaluating effects of cache memory compression on embedded systems. In *Proceedings of the Linux Symposium*, pages 53–65, Ottawa, Ontario, Canada, 2007.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3. edition, 2005.

[3] Peter J. Denning. The working set model for program behavior. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12, New York, NY, USA, 1967. ACM Press.

[4] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the Winter 1993 USENIX Conference*, pages 519–529, 1993.

[5] M. Tim Jones. Anatomy of the linux slab allocator. IBM developerWorks website, 2007. `http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/`.

[6] Seagate Global Product Marketing. Economies of capacity and speed: Choosing the most cost-effective disc drive size and rpm to meet it requirements. Technical report, 2004. `http://www.seagate.com/docs/pdf/whitepaper/economies_capacity_spd_tp.pdf`.

[7] David S. Moore and George P. Mccabe. *Introduction to the Practice of Statistics*. Freeman and Company, New York, 4. edition, 2003.

[8] Alair Pereira do Lago Rodrigo S. de Castro and Dilma Da Silva. Adaptive compressed caching: Design and implementation. *Symposium on Computer Architecture and High Performance Computing*, 15:10–18, 2003.

[9] Asbjorn Sannes. Lkml discussion: Unpredictable performance. Website, 2008. `http://lkml.org/lkml/2008/1/25/326`.

[10] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2001.

[11] Irina C. Tuduce and Thomas Gross. Adaptive main memory compression. In *Proceedings of The 2005 USENIX Annual Technical Conference*, pages 237–250, 2005. `http://www.usenix.org/events/usenix05/tech/general/tuduce.html`.

[12] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, 1999.

Appendix A

Early attempt

In an early attempt at adaptive size in this thesis we tried to follow Wilson et al. closer, but ran into problems with how to time the block I/O subsystem in the kernel.

We divide the pages up into the same lists as the current design with a expense and a profit list.

To decide if the cache should increase or decrease in size we do the following considerations: Let A_e, A_p, T_c, and T_{io} be the number of accesses to the expense list, the number of accesses to the profit list, the time to do a compress and decompress a page, and the time to retrieve a page from disk respectively. As long as equation A.1

$$A_e * T_c \leq A_p * (T_{io} - T_c) \tag{A.1}$$

holds maintaining compressed caching benefits the system and it would probably benefit even more from a larger one. If it doesn't hold, we are better off with a smaller compressed cache.

To keep track of A_e and A_p we need to take a note every time a page is retrieved from the compressed cache, of course it would not be wise to keep such information forever. The reason is twofold; If we get big enough numbers for A_e and A_p, recent information is not going to count much, although recent behavior often is expected behavior in the future. The second reason is that earlier hits for the two lists may have been in the other before the change in the size of the cache and so would force the list to become bigger or smaller than what we consider optimal. We could get around this in various ways, one could be to use a history of N number of pages to calculate the averages over, or we could reset it every Nth page.

Doing the measurement on T_{io} and T_c can be done once, perhaps on the initialization of the compressed cache system, however doing the measurement more often would give us the advantage of getting feedback from the system. Say we have some very I/O intensive load that would slow down swapping out even more, then the time of an I/O would be higher and the hits on the profit list should have a higher weight. The same can be said when the CPU is highly loaded and compression/decompression times would take more time, the weight for the expense list should be lower. The main problem doing this is that the timing will most likely vary little: For measuring compressing and decompressing it is unlikely that we will preempt very often because this operation takes less time than that of the timer interrupt used to change processes. Where an I/O starts and where it ends can vary the timing results quite a lot; if we go to close to where the I/O is submitted, we will get very consistent results with little to no variation. If we do it to far away from what we are really trying to do we get a lot of noise that has nothing to do with how much I/O saturation we have.

The time unit used in A.1 is not important, as long as it is the same for both T_c and T_{io}. This means that we can choose the time unit used for measurement without affecting the result. Therefore I will use the cheapest timing source available in Linux: jiffies. A question is how a measurement should affect the use of T_c and T_{io}. If we say they should be equal to the latest measurement we have done it would be vulnerable to extreme single values. We could of course do an average over N last measurements to avoid that.

Unfortunately, we were unable, without excessive changes, to change the Linux block layer to give us the information we wanted. Instead we implemented the same approach as Castro et. al.

Appendix B

Workloads

These are the scripts used for the different workloads. The important data to be recorded for each run is outputted to *STDERR* and is recorded to a file.

B.1 make -jX workloads

```
#!/bin/sh
rm -rf linux-2.6.22.12
echo "Starting test"
cat /proc/vmstat 1>&2
iostat 1>&2
#sh /usr/src/tests/monitorsize.sh 1>&2 &
time (
        tar -jxf ../linux-2.6.22.12.tar.bz2
        cp ../config-2.6.22 linux-2.6.22.12
        cd linux-2.6.22.12
        yes n | make oldconfig > /dev/null
        make -j${1} > /dev/null)
cat /proc/vmstat 1>&2
iostat 1>&2
echo "Test done"
```

B.2 sort workload

```
echo "Boot" 1>&2
#sh /usr/src/tests/monitorsize.sh 1>&2 &
time sort -n /root/randomfil.txt > /dev/null
time sort -n /root/randomfil.txt > /dev/null
time sort -n /root/randomfil.txt > /dev/null
time sort -n /root/randomfil.txt > /dev/null
time sort -n /root/randomfil.txt > /dev/null
```

B.3 sortcomp workload

```
#!/bin/sh
date +"combined start: %s" 1>&2
sh /usr/src/tests/run-jXtest.sh 1 1>&2
sleep 120
time sort -n /root/randomfil.txt > /dev/null &
wait
date +"combined stop: %s" 1>&2
sh /usr/src/tests/clean-jXtest.sh 1
```

Appendix C

Results

These are raw data results.

C.1 make -j3 test results

C.1.1 cc 2.6.22 with 80MB

Runtime in seconds	1117.732, 972.049, 1294.29, 1253.213, 1032.466, 861.539, 942.275, 1117.636, 1018.522, 956.121, 1229.795, 1090.947, 1350.263, 884.048, 924.832, 925.96, 1114.342, 1001.56, 959.861, 1003.227, 962.085, 1197.297, 1106.528, 847.441, 1139.562, 1108.179, 850.704, 1460.159, 1026.339, 1063.167, 1172.692, 841.671, 1063.561, 1242.8, 1025.701, 958.159, 981.917, 1130.239, 1077.741, 1305.246, 1278.187, 1097.082, 943.9, 1084.011, 1035.87, 999.731, 947.874, 1005.162, 1223.975, 1040.206, 1011.96, 1267.271, 991.939, 1045.596, 1280.566, 1127.809, 1287.736, 948.949, 1328.062, 1205.893, 1001.782, 832.077, 820.999, 1033.079, 1108.804, 965.443, 845.707, 1118.511, 1003.46, 1083.646, 882.723, 1064.495, 1169.237

C.1.2 cc-lzf 2.6.22 with 80MB

Runtime in seconds	1180.322, 1736.956, 2105.391, 1499.364, 1733.325, 1215.125, 1404.389, 1274.694, 1464.517, 1537.931, 1150.38, 1148.969, 1346.485, 1186.645, 2061.916, 1337.123, 1453.029, 1583.624, 1177.802, 1201.908, 1206.682, 1207.64, 1329.938, 1347.647, 1577.903, 1533.789, 1476.121, 1256.183, 1590.186, 1577.489, 1377.541, 1486.634, 1391.366, 984.188, 1106.59, 1360.35, 1107.402, 1369.779, 1506.909, 1285.443, 1669.684, 1249.615, 1089.605, 1540.856, 1365.763, 1424.544, 1621.554, 1302.923, 2146.733, 1620.182, 1108.652, 1213.981, 1768.513, 1090.478, 1116.784, 1135.112, 1341.864, 1499.139, 1241.407, 1490.542, 1207.109, 1299.709, 1514.115, 1478.629, 1328.316, 1051.783, 1606.562, 1267.185, 1236.7, 1440.077, 1287.358, 1494.703, 1542.549, 1229.965, 1230.98, 1331.924, 1211.538, 1304.91, 1340.526, 1016.717, 2226.81, 1361.359, 1268.626, 1656.361, 1457.464, 967.337, 1333.555, 1166.345, 1148.293, 1451.486, 1359.117, 1212.726, 1265.741, 1685.338, 1557.94, 1435.26, 1403.01, 1497.328, 1400.906, 1558.561

C.1.3 cc-nodisable 2.6.22 with 80MB

Runtime in seconds	767.413, 770.603, 737.053, 706.648, 697.546, 712.428, 917.342, 906.133, 805.302, 798.997, 789.894, 728.588, 858.061, 744.891, 743.429, 754.391, 952.222, 797.629, 883.623, 823.436, 894.177, 1047.604, 979.372, 914.061, 1058.001, 804.417, 739.939, 780.909, 767.727, 859.909, 802.298, 686.072, 1292.027, 773.898, 999.817, 949.354, 815.109, 769.139, 926.965, 780.401, 849.487, 784.248, 832.423, 758.433, 712.055, 842.698, 796.539, 759.067, 928.794, 864.138, 965.647, 725.832, 817.7, 760.892, 978.05, 772.348, 823.855, 716.652, 762.965, 892.513, 857.34, 794.053, 770.914, 868.107, 861.963, 774.316, 717.54, 885.817, 768.588, 739.391, 737.644, 727.639, 722.85, 778.865, 1200.068, 922.361, 1105.826, 849.012, 753.309, 975.434, 962.007, 939.189, 827.064, 766.853, 820.765, 726.799, 803.898, 855.642, 863.641, 684.234, 732.65, 726.365, 721.946, 785.862, 815.927, 698.306, 1190.974, 818.96, 970.412, 811.909, 925.865, 749.163, 711.862, 769.399, 781.721, 706.678, 724.368, 748.084, 815.395, 718.827, 852.946, 739.892, 826.63, 746.582, 945.075, 768.729, 939.574, 782.546, 1136.092, 774.987, 724.496, 824.383, 1033.621, 979.917, 738.998, 782.725, 804.352, 884.716, 841.383, 795.516, 820.265, 931.755, 1036.95, 790.644, 832.503, 749.471, 748.937, 840.223, 890.15, 823.663, 795.737

C.1.4 vanilla 2.6.22 with 80MB

Runtime in seconds	2057.068, 2401.157, 1607.54, 2374.293, 1417.935, 2493.354, 2939.55, 2177.822, 2074.28, 1506.185, 1612.583, 1452.569, 2381.53, 1880.831, 1917.417, 2392.15, 1541.52, 1895.535, 1962.123, 2060.51, 1411.052, 2323.326, 2563.958, 1681.11, 2234.74, 1932.092, 2245.636, 2103.943, 1708.545, 1992.592, 1543.542, 1805.8, 1866.657, 1814.16, 1398.314, 2005.467, 1792.397, 1560.972, 1326.4, 2113.661, 1388.814, 1789.443, 1884.818, 1998.691, 1456.303, 1984.174, 1544.805, 1894.735, 2058.468, 1384.899, 2120.802, 1643.117, 1652.56, 1220.304, 2169.277, 2304.873, 2119.618, 1787.734, 1499.515, 2632.369, 1574.41, 2152.844, 1797.407, 1842.16, 1453.856, 2006.133, 1926.16, 1907.079, 2542.602, 1828.013, 1850.879, 1626.826, 1822.989, 2021.493, 2307.664, 1917.423, 1727.825, 1834.717, 1920.11, 2000.354, 2057.894, 1878.583, 2038.39, 1332.434, 1786.891, 1991.614, 1921.322, 1609.108, 1830.641, 1481.273, 2583.429, 1497.95, 1680.828, 1734.412, 2363.936, 2718.597, 2898.658, 1855.494, 1742.922, 1788.406

C.1.5 cc 2.6.22 with 100MB

Runtime in seconds	454.186, 519.486, 482.078, 493.211, 472.337, 459.4, 522.854, 445.754, 464.097, 439.563, 466.772, 548.185, 490.515, 456.859, 444.307, 487.15, 447.278, 478.464, 467.238, 537.024, 449.544, 488.677, 436.854, 455.733, 502.039, 464.444, 555.646, 487.702, 446.892, 475.075, 469.829, 478.167, 632.005, 649.579, 459.351, 482.908, 458.031, 509.58, 484.029, 436.614, 446.277, 487.699, 447.641, 470.033, 461.37, 467.992, 455.799, 459.95, 579.89, 451.484, 455.951, 476.907, 455.223, 459.155, 466.559, 441.596, 444.127, 459.463, 477.888, 502.547, 446.436, 602.514, 463.794, 445.953, 439.381, 458.982, 445.465, 565.531, 438.5, 631.898, 439.048, 467.582, 479.385, 469.981, 445.997, 506.156, 445.837, 587.166, 472.386, 453.25, 454.871, 470.065, 504.206, 485.217, 447.047, 455.024, 457.281, 484.286, 442.561, 448.071, 556.936, 487.295, 495.782, 477.483, 594.709, 466.453, 616.36, 499.813, 456.81, 506.66

C.1.6 cc-lzf 2.6.22 with 100MB

Runtime in seconds	730.124, 510.317, 482.914, 514.248, 531.921, 488.76, 496.664, 511.932, 514.976, 695.585, 492.922, 537.221, 670.979, 469.92, 490.705, 473.749, 520.456, 508.835, 533.465, 480.359, 516.466, 507.206, 489.498, 492.597, 534.107, 503.466, 637.058, 511.072, 537.26, 495.914, 518.261, 503.414, 606.7, 486.894, 620.542, 479.264, 540.623, 488.893, 515.669, 622.994, 535.42, 572.844, 498.629, 509.839, 519.073, 503.746, 525.308, 470.56, 591.162, 494.93, 506.632, 473.05, 539.382, 512.747, 511.988, 482.763, 506.128, 509.101, 471.834, 483.66, 473.148, 564.234, 554.93, 565.577, 568.409, 616.022, 499.873, 518.811, 527.317, 505.824, 636.239, 494.525, 507.899, 528.894, 523.047, 565.18, 663.239, 525.424, 564.101, 488.114, 500.371, 526.487, 470.57, 566.699, 518.58, 502.639, 700.803, 508.386, 489.896, 511.988, 521.975, 592.07, 506.53, 470.026, 575.286, 501.43, 481.59, 593.656, 478.394, 679.047

C.1.7 cc-nodisable 2.6.22 with 100MB

Runtime in seconds	419.539, 494.159, 456.809, 435.528, 445.718, 441.299, 430.694, 449.245, 429.258, 459.912, 455.52, 435.885, 457.49, 442.823, 435.33, 483.295, 456.529, 448.425, 433.201, 433.365, 418.109, 440.536, 425.225, 544.753, 438.532, 429.461, 430.279, 505.996, 417.452, 455.318, 428.432, 418.179, 432.172, 498.034, 431.591, 425.003, 440.741, 465.851, 434.148, 425.763, 459.182, 434.525, 443.869, 422.808, 432.034, 454.015, 443.113, 565.173, 433.983, 437.536, 450.816, 441.901, 437.371, 474.606, 439.3, 429.933, 430.234, 467.762, 427.298, 441.582, 441.932, 445.513, 422.689, 436.002, 435.744, 446.031, 431.726, 442.533, 427.503, 427.572, 462.298, 454.653, 435.749, 445.904, 431.494, 449.894, 445.979, 428.185, 423.639, 429.166, 455.225, 413.346, 431.901, 429.627, 456.36, 429.077, 425.111, 427.301, 437.864, 476.647, 542.042, 451.197, 436.79, 471.559, 444.971, 453.939, 450.065, 442.674, 420.7, 432.553, 482.955, 432.684, 430.557, 467.656, 481.063, 428.971, 447.249, 451.556, 458.323, 432.833, 470.24, 472.04, 516.022, 467.61, 443.996, 454.583, 566.116

C.1.8 vanilla 2.6.22 with 100MB

Runtime in seconds	514.53, 482.273, 536.66, 576.528, 522.771, 599.84, 529.719, 508.3, 450.411, 790.877, 485.969, 513.015, 458.573, 522.382, 564.072, 720.725, 502.919, 527.202, 472.041, 762.031, 480.245, 525.138, 688.275, 791.502, 456.667, 725.63, 758.978, 552.806, 472.93, 458.342, 468.459, 593.049, 596.899, 460.202, 485.239, 476.023, 518.883, 502.397, 741.604, 883.835, 496.942, 494.108, 745.978, 461.144, 739.26, 574.879, 500.07, 534.799, 792.462, 486.105, 488.369, 457.846, 470.497, 728.644, 478.498, 481.267, 630.142, 459.121, 794.639, 486.61, 536.615, 769.668, 550.488, 497.704, 440.021, 464.032, 462.247, 479.317, 459.798, 481.847, 451.441, 470.505, 470.495, 530.781, 543.469, 517.546, 652.357, 457.525, 528.843, 848.182, 485.98, 461.373, 468.411, 544.605, 618.016, 872.148, 574.773, 471.837, 760.826, 550.438, 489.132, 609.306, 589.246, 726.156, 501.921, 459.943, 511.743, 778.041, 503.142, 466.814

C.1.9 cc 2.6.22 with 120MB

Runtime in seconds	360.303, 379.34, 373.746, 364.484, 353.9, 358.469, 378.791, 369.983, 340.23, 343.61, 343.609, 379.085, 420.623, 349.631, 348.896, 350.1, 438.006, 435.977, 446.577, 336.94, 342.423, 389.678, 349.372, 380.763, 367.801, 411.829, 406.801, 391.07, 366.011, 346.086, 357.79, 344.828, 375.608, 368.112, 356.168, 390.226, 359.154, 354.339, 380.364, 395.563, 407.778, 390.742, 347.92, 361.591, 377.837, 353.147, 351.424, 347.465, 343.144, 352.687, 349.652, 403.565, 385.501, 354.354, 337.72, 354.744, 382.205, 361.479, 385.249, 352.021, 349.195, 366.768, 358.006, 350.779, 350.815, 347.013, 344.91, 374.189, 379.919, 352.579, 389.24, 365.881, 353.818, 353.899, 410.288, 390.616, 369.216, 376.926, 394.57, 342.639, 342.35, 348.172, 352.018, 339.804, 368.928, 342.096, 339.645, 357.914, 361.447, 345.2, 390.241, 345.772, 362.801, 427.69, 356.212, 389.161, 370.969, 365.659, 374.046, 380.355

C.1.10 cc-lzf 2.6.22 with 120MB

Runtime in seconds	372.297, 351.306, 358.512, 365.003, 369.49, 427.844, 403.057, 369.15, 395.772, 376.999, 354.978, 354.951, 357.614, 354.277, 420.748, 380.636, 381.326, 396.512, 389.765, 376.563, 385.512, 436.478, 374.748, 389.712, 346.894, 379.263, 356.261, 340.872, 352.138, 344.666, 428.91, 365.189, 357.275, 352.088, 356.578, 381.623, 350.159, 387.06, 401.885, 348.576, 402.831, 394.593, 418.072, 425.225, 429.796, 389.295, 419.524, 419.814, 370.93, 404.524, 456.16, 379.461, 415.334, 343.688, 376.442, 375.45, 405.176, 442.902, 349.274, 425.973, 377.715, 379.557, 374.27, 391.971, 405.029, 386.785, 346.75, 375.87, 344.366, 368.324, 342.33, 360.332, 355.518, 339.684, 414.64, 393.07, 363.486, 351.317, 382.45, 408.967, 362.584, 409.161, 391.263, 410.85, 370.314, 387.097, 400.06, 346.653, 385.993, 353.81, 351.65, 426.289, 382.753, 342.991, 342.273, 382.08, 390.12, 368.881, 409.475, 342.905

C.1.11 cc-nodisable 2.6.22 with 120MB

Runtime in seconds	340.639, 350.625, 398.479, 368.671, 374.13, 376.529, 367.515, 349.908, 398.294, 345.825, 348.322, 390.548, 342.114, 350.106, 341.361, 343.392, 345.272, 354.958, 349.118, 358.733, 372.216, 371.006, 409.62, 339.476, 401.148, 376.347, 336.021, 346.134, 381.151, 403.179, 366.573, 423.063, 359.292, 416.319, 343.998, 379.725, 346.839, 388.167, 364.649, 348.258, 389.223, 345.626, 356.007, 366.421, 354.186, 361.403, 344.21, 344.494, 371.323, 369.309, 365.085, 388.935, 348.843, 378.284, 368.754, 375.453, 381.242, 360.407, 395.533, 393.369, 378.634, 359.454, 361.86, 363.47, 351.853, 381.321, 340.974, 396.871, 338.045, 352.704, 360.326, 357.837, 394.326, 355.304, 392.111, 338.836, 350.497, 372.254, 349.139, 352.627, 364.14, 357.454, 365.28, 342.123, 357.919, 352.539, 343.425, 359.379, 363.432, 362.92, 361.317, 436.225, 339.46, 370.146, 348.786, 336.788, 367.053, 343.05, 391.348, 343.239, 345.04, 346.312, 346.704, 345.016, 367.937, 345.649, 375.7, 342.308, 398.375, 359.665, 364.418, 350.992, 358.958, 371.258, 343.61, 373.336, 347.427

C.1.12 vanilla 2.6.22 with 120MB

Runtime in seconds	460.205, 367.17, 456.901, 356.778, 358.264, 412.309, 427.327, 393.603, 437.216, 397.591, 390.051, 421.413, 432.206, 350.976, 419.292, 425.727, 381.909, 365.093, 433.409, 362.515, 388.834, 390.992, 355.127, 434.511, 352.316, 369.499, 436.566, 439.165, 376.414, 336.427, 390.289, 368.8, 391.671, 340.766, 431.76, 379.509, 351.575, 380.949, 358.756, 389.689, 365.237, 387.972, 424.927, 373.626, 344.791, 356.301, 375.136, 382.982, 361.106, 444.427, 379.577, 358.199, 373.456, 381.712, 367.645, 391.354, 359.958, 371.479, 386.362, 385.364, 379.546, 406.369, 348.745, 371.842, 384.84, 442.051, 388.721, 385.661, 367.989, 389.192, 390.965, 346.169, 369.435, 429.58, 400.649, 386.992, 377.242, 362.292, 408.503, 411.451, 399.181, 350.231, 361.227, 370.704, 347.522, 362.989, 385.602, 370.077, 353.094, 357.922, 357.126, 400.889, 355.109, 408.916, 357.938, 358.662, 391.197, 404.665, 357.073, 416.391

C.2 make -j2 test

C.2.1 cc 2.6.22 with 80MB

Runtime in seconds	600.469, 649.943, 601.703, 646.641, 702.469, 549.355, 512.826, 611.367, 679.261, 584.884, 641.288, 718.668, 670.024, 680.469, 657.613, 629.263, 753.666, 563.069, 606.955, 729.853, 677.18, 742.811, 703.33, 766.428, 610.737, 646.551, 654.568, 636.988, 653.165, 634.662, 767.286, 637.313, 628.089, 721.937, 574.967, 710.256, 667.832, 634.807, 604.066, 697.928, 640.476, 600.835, 664.499, 860.24, 618.746, 651.127, 670.098, 680.452, 637.237, 500.152, 627.102, 738.175, 647.697, 582.207, 636.603, 790.917, 745.787, 610.755, 644.794, 638.039, 692.816, 764.618, 682.456, 657.737, 631.796, 647.176, 616.699, 595.166, 614.146, 623.29, 640.985, 573.875, 781.665, 664.643, 566.452, 618.792, 685.004, 624.41, 611.091, 575.548, 674.117, 614.953, 667.959, 640.584, 649.427, 590.162, 592.387, 676.23, 628.399, 595.49, 689.852, 612.228, 531.487, 695.841, 617.388, 631.481, 795.981, 694.009, 645.074, 577.624

C.2.2 cc-nodisable 2.6.22 with 80MB

Runtime in seconds	587.404, 576.395, 480.962, 558.643, 583.817, 577.896, 590.26, 695.027, 638.675, 586.145, 604.683, 587.993, 575.353, 564.899, 504.448, 567.283, 769.442, 572.027, 560.939, 575.721, 587.807, 740.545, 587.112, 612.56, 563.244, 578.023, 588.617, 562.564, 623.243, 582.727, 679.98, 610.037, 558.785, 558.39, 577.708, 570.518, 638.08, 567.022, 670.99, 586.138, 758.416, 625.612, 637.06, 629.329, 650.183, 588.333, 566.374, 620.948, 593.462, 574.139, 605.436, 563.057, 628.199, 627.388, 565.442, 661.42, 566.354, 567.191, 603.368, 690.087, 581.392, 506.442, 600.82, 566.617, 799.162, 556.96, 570.519, 593.607, 604.687, 611.827, 569.759, 596.874, 762.488, 558.82, 753.768, 706.856, 612.512, 597.695, 587.081, 582.566, 553.811, 576.013, 581.291, 589.92, 565.699, 557.253, 822.139, 588.29, 596.554

C.2.3 vanilla 2.6.22 with 80MB

Runtime in seconds	875.284, 885.112, 821.795, 806.501, 1026.533, 941.497, 613.443, 1004.99, 816.209, 898.795, 811.99, 823.597, 823.01, 808.163, 825.7, 1000.027, 904.819, 780.185, 815.25, 835.436, 816.357, 849.888, 794.445, 886.376, 826.87, 807.484, 852.915, 826.893, 804.038, 929.558, 976.273, 779.736, 1121.077, 977.765, 1173.086, 916.173, 948.486, 645.981, 947.12, 728.424, 906.245, 891.207, 774.619, 811.111, 945.768, 841.203, 824.886, 830.475, 799.099, 833.459, 1159.131, 825.53, 937.154, 825.878, 835.678, 791.138, 901.174, 777.323, 786.953, 937.504, 797.299, 818.367, 1212.144, 776.422, 853.022, 899.547, 967.492, 842.13, 846.97, 1000.473, 820.069, 940.184, 834.74, 984.72, 907.736, 841.26, 821.997, 822.01, 896.759, 900.46, 818.126, 824.288, 879.595, 928.638, 960.178, 856.556, 806.226, 842.821, 820.786, 924.269, 901.354, 894.199, 917.927, 904.99, 904.495, 898.5, 829.308, 856.094, 1039.617, 613.96

C.3 make -j1 test

C.3.1 cc 2.6.22 with 80MB

Runtime in seconds	444.418, 469.259, 417.434, 422.505, 450.329, 430.643, 461.239, 408.513, 431.5, 456.871, 416.414, 443.404, 408.676, 406.566, 415.836, 399.536, 422.172, 405.577, 415.528, 448.061, 468.667, 425.602, 413.411, 416.615, 438.645, 407.869, 451.493, 455.809, 407.608, 465.546, 464.477, 423.659, 417.547, 405.054, 429.373, 413.751, 409.558, 442.139, 458.1, 467.739, 461.786, 430.13, 448.212, 428.795, 426.547, 422.502, 436.296, 444.029, 402.862, 439.706, 456.043, 453.715, 413.571, 440.269, 398.043, 416.163, 464.823, 423.342, 396.273, 452.45, 445.578, 417.906, 443.181, 432.397, 438.23, 424.613, 439.583, 446.242, 443.364, 418.032, 412.261, 444.037, 428.083, 432.716, 418.634, 403.96, 452.859, 422.94, 424.109, 431.163, 410.763, 405.27, 445.23, 404.878, 406.291, 412.372, 398.771, 424.375, 433.407, 400.855, 415.72, 417.793, 428.602, 393.239, 407.891, 416.956, 419.517, 398.035, 422.789, 408.665

C.3.2 cc-nodisable 2.6.22 with 80MB

Runtime in seconds	395.102, 440.658, 405.899, 406.172, 423.558, 430.19, 411.718, 450.072, 401.62, 399.838, 403.734, 412.894, 393.478, 405.721, 400.521, 395.025, 404.872, 402.962, 404.452, 410.284, 393.037, 434.184, 450.299, 419.397, 390.495, 431.039, 396.094, 402.471, 418.649, 399.305, 450.768, 420.353, 404.365, 466.252, 397.923, 389.603, 404.305, 396.482, 401.546, 413.205, 454.034, 421.584, 394.41, 440.415, 470.71, 426.216, 444.942, 406.578, 390.203, 396.561, 437.648, 395.459, 390.533, 407.302, 461.877, 405.158, 398.473, 422.525, 400.763, 418.783, 415.796, 432.977, 466.039, 402.943, 444.374, 418.476, 431.703, 410.765, 401.438, 393.379, 387.872, 405.266, 450.84, 429.835, 390.389, 395.216, 411.218, 461.8, 432.934, 444.42, 385.886, 406.309, 449.571, 405.389, 421.993, 409.734, 404.055, 442.308, 437.111

C.3.3 vanilla 2.6.22 with 80MB

Runtime in seconds	473.711, 477.985, 470.686, 477.192, 478.122, 473.099, 488.793, 472.85, 471.847, 465.526, 487.375, 476.854, 465.247, 474.287, 468.524, 473.707, 466.314, 473.818, 474.766, 474.563, 477.57, 471.979, 478.038, 469.293, 466.334, 467.338, 478.625, 474.086, 480.146, 473.284, 479.281, 479.011, 472.998, 474.098, 465.978, 482.719, 473.739, 476.225, 479.454, 471.262, 477.536, 478.294, 473.031, 475.284, 468.621, 469.969, 470.516, 476.35, 472.77, 468.941, 462.593, 475.224, 468.051, 476.705, 472.289, 476.518, 476.732, 470.825, 482.98, 481.518, 473.846, 473.617, 473.154, 478.034, 475.461, 469.233, 469.514, 463.574, 472.221, 465.443, 472.384, 475.879, 480.728, 469.375, 476.112, 473.496, 479.146, 481.672, 482.096, 479.458, 471.691, 477.046, 464.938, 476.922, 490.208, 478.077, 468.853, 478.078, 483.005, 467.15, 477.421, 476.362, 475.942, 470.479, 478.18, 475.649, 471.957, 474.926, 477.618, 475.014

C.4 sort test

C.4.1 cc 2.6.22 with 40MB

Runtime in seconds	254.761, 263.475, 259.794, 264.298, 274.086, 259.037, 262.008, 256.101, 254.768, 265.233, 263.61, 267.898, 265.807, 267.712, 260.679, 264.548, 263.155, 262.335, 261.09, 261.819, 245.35, 262.172, 261.774, 265.839, 255.746, 264.956, 267.31, 265.888, 263.038, 263.915, 256.647, 264.608, 263.315, 274.696, 262.854, 260.468, 261.514, 251.125, 260.851, 259.384, 259.821, 257.92, 268.011, 262.037, 261.312, 260.106, 260.285, 260.303, 268.612, 254.951, 261.51, 272.848, 264.811, 257.757, 262.522, 264.68, 253.685, 261.023, 265.668, 256.088, 259.516, 261.662, 260.355, 259.249, 261.316, 266.441, 271.168, 260.26, 263.605, 265.434, 256.802, 255.324, 266.991, 265.614, 264.697, 258.805, 262.6, 263.493, 259.531, 263.242, 266.447, 262.619, 252.628, 259.413, 257.892, 268.857, 260.291, 265.268, 256.467, 252.796, 257.166

C.4.2 cc-nodisable 2.6.22 with 40MB

Runtime in seconds	274.527, 278.076, 281.087, 271.942, 281.162, 273.819, 276.1, 286.295, 268.434, 281.675, 276.47, 275.544, 278.253, 266.266, 274.884, 274.877, 281.521, 294.4, 272.868, 277.007, 271.588, 284.715, 264.987, 288.274, 281.198, 278.853, 277.837, 283.201, 288.251, 262.82, 280.239, 292.728, 277.346, 275.529, 280.791, 278.842, 278.722, 273.389, 276.998, 270.832, 271.2, 268.921, 271.135, 279, 278.304, 297.647, 307.162, 278.859, 270.002, 281.683, 284.325, 272.823, 288.897, 284.185, 280.245, 281.659, 271.262, 278.253, 270.34, 275.892, 275.863, 281.951, 271.324, 277.952, 286.544, 274.406, 284.254, 271.155, 288.381, 266.421, 274.641, 274.167, 270.438, 274.058, 275.115, 284.517, 270.66, 264.598, 285.095, 283.806, 284.248, 290.263, 268.359, 261.231, 281.454, 280.257, 285.538, 287.668, 279.233, 287.92, 273.979, 278.196, 288.975, 278.62, 286.047, 269.511, 264.852, 272.452, 288.066, 287.599

C.4.3 vanilla 2.6.22 with 40MB

Runtime in seconds	249.431, 244.787, 249.558, 250.257, 256.659, 259.963, 256.291, 250.555, 249.045, 246.204, 248.126, 252.06, 253.244, 251.833, 252.167, 249.143, 248.955, 244.981, 254.858, 256.697, 253.509, 248.873, 255.877, 248.297, 248.485, 249.876, 247.873, 252.985, 248.477, 249.373, 247.012, 253.561, 251.071, 252.632, 249.752, 250.439, 247.976, 251.752, 252.555, 249.851, 247.969, 250.166, 254.138, 245.569, 257.925, 254.485, 248.014, 247.842, 251.118, 245.21, 258.546, 256.307, 247.726, 250.945, 247.164, 247.974, 254.75, 250.879, 248.403, 251.329, 247.681, 250.687, 248.623, 240.792, 254.531, 253.021, 248.879, 243.401, 250.513, 255.53, 247.303, 248.06, 243.215, 250.477, 249.398, 248.732, 244.866, 250.654, 249, 255.3, 251.239, 248.725, 254.702, 245.836, 253.386, 244.737, 249.334, 249.465, 248.292, 254.639, 249.493, 250.853, 251.794, 256.098, 245.127, 247.698, 247.835, 249.021, 249.691, 251.462

C.4.4 cc 2.6.22 with 60MB

Runtime in seconds	236.114, 235.179, 233.206, 239.474, 231.848, 230.667, 238.356, 236.689, 231.539, 234.448, 234.985, 233.706, 238.172, 238.797, 238.492, 233.465, 234.055, 235.064, 235.733, 234.89, 230.766, 229.312, 233.867, 237.222, 236.655, 234.325, 240.914, 238.949, 237.793, 235.94, 236.57, 235.276, 234.079, 237.369

C.4.5 cc-nodisable 2.6.22 with 60MB

Runtime in seconds	289.705, 274.057, 302.167, 281.102, 269.211, 302.983, 285.579, 294.884, 288.346, 291.436, 288.625, 290.832, 287.807, 272.27, 281.216, 284.255, 316.756, 275.006, 277.419, 289.249, 286.053, 270.265, 273.843, 278.492, 295.504, 286.771, 283.126, 275.978, 291.39, 277.016, 289.59, 280.876, 280.001, 284.767, 270.635, 292.79, 284.696, 288.836, 266.37, 274.298, 278.862, 299.155, 283.493

C.4.6 vanilla 2.6.22 with 60MB

Runtime in seconds	237.462, 236.34, 241.022, 236.261, 236.113, 240.036, 245.036, 240.448, 243.333, 233.923, 234.284, 234.259, 241.263, 233.738, 234.501, 234.597, 235.123, 240.665, 237.906, 234.547, 238.579, 239.974, 233.915, 234.39, 235.329, 239.208, 242.053, 241.401, 233.886, 239.888, 235.316, 237.762, 234.27, 236.039, 245.017, 232.374, 233.098, 231.576, 239.25, 242.644, 235.428, 235.949, 237.362, 233.392, 234.05, 233.874, 239.247, 235.585, 233.699, 238.284, 235.878, 234.228

C.5 sortcomp test

C.5.1 cc 2.6.22 with 80MB

Runtime in seconds	597, 618, 571, 571, 642, 573, 563, 659, 601, 584, 568, 568, 595, 558, 659, 580, 567, 579, 637, 586, 620, 581, 593, 590, 609, 613, 607, 570, 573, 587, 595, 577, 568, 631, 606, 594, 639, 584, 575, 604, 592, 578, 607, 583, 642, 623, 592, 602, 581, 578, 620, 582, 636, 624, 615, 579, 576, 579, 595, 595, 569, 573, 586, 579, 586, 604, 603, 590, 565, 629, 622, 598, 627, 594, 618, 580, 575, 612, 621, 575, 615, 578, 611, 594, 662, 562, 615, 594, 613, 631, 621, 609, 626, 603, 598, 587, 595, 594, 576, 581

C.5.2 cc-nodisable 2.6.22 with 80MB

Runtime in seconds	612, 605, 613, 625, 592, 602, 581, 577, 637, 569, 560, 593, 581, 581, 570, 585, 577, 565, 575, 573, 636, 577, 585, 589, 610, 585, 561, 578, 568, 562, 588, 567, 586, 640, 574, 574, 577, 586, 567, 569, 567, 569, 651, 566, 580, 584, 575, 558, 594, 568, 601, 579, 586, 568, 583, 562, 581, 607, 604, 565, 586, 612, 579, 630, 566, 560, 634, 598, 593, 565, 619, 590, 574, 631, 633, 576, 579, 585, 585, 566, 584, 608, 612, 594, 609, 575, 655, 596, 601, 570, 587, 576, 572, 616, 572, 596, 651, 564, 615, 575

C.5.3 vanilla 2.6.22 with 80MB

Runtime in seconds	640, 633, 636, 631, 643, 643, 641, 643, 631, 637, 639, 639, 638, 634, 644, 634, 644, 635, 638, 639, 636, 643, 643, 643, 641, 638, 647, 638, 635, 638, 635, 640, 642, 643, 646, 644, 650, 639, 635, 637, 640, 646, 636, 642, 644, 641, 646, 631, 639, 636, 645, 636, 642, 644, 640, 635, 638, 633, 645, 643, 635, 641, 632, 633, 643, 641, 634, 633, 641, 650, 640, 637, 631, 650, 640, 636, 638, 638, 641, 637, 653, 637, 645, 643, 640, 640, 634, 637, 632, 642, 636, 641, 637, 634, 635, 638, 641, 633, 641, 635

Appendix D

Code

D.1 CC subsystem

D.1.1 *include/linux/cc.h*

```c

#ifndef _LINUX_CC_H
#define _LINUX_CC_H

#include <linux/list.h>
#include <linux/bitops.h>

#define CC_COMPRESSED_PAGE_BIT 1
#define CC_COMPRESSED_PAGE_ALIGN 4

#define PageCompressed(addr) (((unsigned long)addr & 2) == 2)
#define ClearPageCompress(addr) clear_bit(CC_COMPRESSED_PAGE_BIT, addr)
#define MaskPageCompressed(addr) ((struct cpage *)((unsigned long)addr & ~2))
#define CPagePage(cpage) (struct page *)((void *)\
  ((unsigned long)cpage | (1 << CC_COMPRESSED_PAGE_BIT)))

struct cpage {
    union {
        struct {
            void *compressed;
            int length;
        };
        struct list_head marker_queue;
    };
    struct address_space *mapping;
    unsigned long index;
    unsigned long flags;
    union {
        struct list_head list;
        struct page *page;
    };
};

struct cc_stats {
    unsigned char   flags;  /* state for area */
    unsigned char   used;   /* page-hit compressed
        cache */
    unsigned char   unused; /* unused removed
        compressed cache */
    unsigned char   markers;/* marker pages present in
         c.c. */
    struct list_head marker_queue;
} __attribute__((packed));

/* cpage flags */
enum {
    CC_FLAG_DIRTY = 0,
    CC_FLAG_PROFIT,
    CC_FLAG_MARKER,
    CC_FLAG_SWAPCACHE
};

#define cpage_dirty(cpage) test_bit(CC_FLAG_DIRTY,
   &cpage->flags)
#define cpage_set_dirty(cpage) set_bit(CC_FLAG_DIRTY,
   &cpage->flags)
#define cpage_clear_dirty(cpage)
    clear_bit(CC_FLAG_DIRTY, &cpage->flags)

#define cpage_profit(cpage) test_bit(CC_FLAG_PROFIT,
   &cpage->flags)
#define cpage_set_profit(cpage)
    set_bit(CC_FLAG_PROFIT, &cpage->flags)
#define cpage_clear_profit(cpage)
    clear_bit(CC_FLAG_PROFIT, &cpage->flags)

```

```
#define cpage_marker(cpage) test_bit(CC_FLAG_MARKER, &cpage->flags)
#define cpage_set_marker(cpage) \
    set_bit(CC_FLAG_MARKER, &cpage->flags)
#define cpage_clear_marker(cpage) \
    clear_bit(CC_FLAG_MARKER, &cpage->flags)

#define cpage_swapcache(cpage) test_bit(CC_FLAG_SWAPCACHE, &cpage->flags)
#define cpage_set_swapcache(cpage) set_bit(CC_FLAG_SWAPCACHE, &cpage->flags)
#define cpage_clear_swapcache(cpage) \
    clear_bit(CC_FLAG_SWAPCACHE, &cpage->flags)

/* filemap.c */
extern struct page *find_get_page_cc(struct address_space *mapping,
    unsigned long index);
extern struct page *find_get_page_nocc(struct address_space *mapping,
    unsigned long index);
extern unsigned find_get_pages_cc(struct address_space *mapping, pgoff_t start,
    unsigned int nr_pages, struct page **pages);

/* swapfile.c */
struct swap_info_struct;
extern int swap_entry_free(struct swap_info_struct *p, unsigned long offset);

/* cc.c */
extern void cc_accessed_cpage(struct cpage *cpage);
extern struct page *cc_get_page_from_cache(void);
extern struct cpage *cc_store_page(struct page *page);
extern struct page *cc_restore_page(struct cpage *cpage);
extern struct cpage *cc_add_lru(struct cpage *cpage);
extern void cc_remove_lru(struct cpage *cpage);
extern int cc_restore(struct cpage *cpage, struct page *page);
extern void cc_free(struct cpage *cpage);
extern void cc_free_list(struct list_head *cpage_free_list);
extern void cc_free_swap(struct cpage *cpage);
extern void cc_free_swap_list(struct list_head
    *cpage_free_list);
extern void cc_setup(void);
extern unsigned long cc_start_timing(void);
extern void cc_stop_timing(unsigned long
    start_jiffies);

#endif /* _LINUX_CC_H */
```

D.1.2 *mm/cc.c*

```


#include <linux/mm.h>
#include <linux/mm_inline.h>
#include <linux/rmap.h>
#include <linux/cc.h>

#include <linux/slab.h>
#include <linux/list.h>

#include <linux/swap.h>
#include <linux/writeback.h>

#include <linux/pagemap.h>
#include "lzf.h"
#include "lzo/lzo.h"
#include <linux/cell.h>

#include <linux/jiffies.h>
#include <linux/fs.h>

/* Internal switches:
 * CC_STORE_CELL turns on storing in the cell
     datastructure!
 * CC_LZO makes us compress and decompress with LZO
     instead of LZF
 * CC_NODISABLE tells us never to disable compressed
     caching for any area */
#define CC_STORE_CELL
#define CC_LZO
#define CC_NODISABLE

```

```
#define CC_CACHE_MIN 100
#define CC_CACHE_HISTORY 100
#define CC_MAX_MARKERS 32
#define CC_MAX_EARLY_WARNING 20
#define CC_MAX_BAD_COMPRESSION_SKIP 100

enum disabled_reason { UNUSED, EARLY_WARNING_DONE,
    BAD1, BAD2 };
enum mapping_state { CC_ENABLED, CC_DISABLED,
    CC_MARKER};

static struct kmem_cache *cc_cpage;
static struct list_head cell_list_expense,
    cell_list_profit;
static spinlock_t cc_list_lock;

#ifdef CC_LZO
static struct semaphore cc_lzo_sem;
static void *cc_lzo_workmem;
static void *cc_lzo_tempcomp;
#endif

/* CC State variables */
static atomic_t cc_expense;
static unsigned int cc_locked; /* 1+ - yes, 0 - no */
/* static struct shrinker *cc_shrinker_data; */

/* Prototypes */
static int cc_remove_cpage(struct address_space
    *mapping, pgoff_t offset,
     struct page *page, struct page **freed_page);
static int cc_shrink_lru(struct page **freed_page);
/* static int cc_shrinker(int nr_to_scan, gfp_t
    gfp_mask); */
static int cc_restore_helper(struct cpage *cpage,
    struct page *page);
static struct address_space *cc_mapping(struct cpage
    *cpage);
static inline struct cc_stats *cc_stats(struct
    address_space *mapping);
static enum mapping_state cc_area_state(struct page
    *page);
static void cc_area_updated(struct cc_stats *stat);
static int cc_put_marker(struct cc_stats *stats);
static int cc_early(struct cc_stats *stats);
static void cc_early_done(struct cc_stats *stats);
static void cc_note_bad_compression(struct page
    *page);
static void cc_note_good_compression(struct page
    *page);


/* Last try before the system runs out of memory */
struct page *cc_get_page_from_cache()
{
    return NULL;
/*  struct page *page = NULL;
    unsigned int max = 32;
    while (max > 0 && !page && cc_shrink_lru(&page))
        --max;
    printk(KERN_EMERG "%s: Returning 0x%lx with
        max=%u\n", __FUNCTION__,
         (unsigned long)page, max);
    return page;*/
}

/* This function pushes pages that would not be in
    memory
 * without CC into the profit list, must be called
    with the list locks taken. */
static void cc_push_profit(void)
{
    unsigned long expenses = atomic_read(&cc_expense);
    unsigned long pages_used =
        cell_accounting_pages();
    unsigned long pushable;
    struct cpage *cpage;

    if (expenses < pages_used)
        return;
    pushable = expenses - pages_used;
    while ((pushable--) &&
        (!list_empty(&cell_list_expense))) {
        cpage = list_entry(cell_list_expense.next,
            struct cpage, list);
        list_del(&(cpage->list));
        cpage_set_profit(cpage);
```

```
            list_add_tail(&(cpage->list),
                &cell_list_profit);
            /* Marker pages does not take any space */
            if (cpage_marker(cpage)) {
                pushable++;
            } else {
                atomic_dec(&cc_expense);
            }
    }
}


/* both write lock on mapping and lru lists must be
    taken */
void __cc_remove_lru(struct cpage *cpage)
{
    struct cc_stats *stats = cc_stats(cpage->mapping);
    if (cpage_marker(cpage)) {
        list_del((&cpage->marker_queue));
        stats->markers--;
    } else if (!cpage_profit(cpage))
        atomic_dec(&cc_expense);
    list_del((&cpage->list));
}

/* write lock on mapping must be taken */
void cc_remove_lru(struct cpage *cpage)
{
    unsigned long flags;
    spin_lock_irqsave(&cc_list_lock, flags);
    __cc_remove_lru(cpage);
    spin_unlock_irqrestore(&cc_list_lock, flags);
}

/* write lock on mapping must be taken */
struct cpage *cc_add_lru(struct cpage *cpage)
{
    struct cpage *oldmarker = NULL;
    struct cc_stats *stats = cc_stats(cpage->mapping);
    unsigned long flags;
    spin_lock_irqsave(&cc_list_lock, flags);
    list_add_tail(&(cpage->list), &cell_list_expense);
    if (cpage_marker(cpage)) {
        /* remove oldest marker */
        if (!cc_put_marker(stats)) {
            struct address_space *mapping;
            oldmarker =
                list_entry(stats->marker_queue.next,
                 struct cpage, marker_queue);
            BUG_ON(oldmarker == cpage);
            __cc_remove_lru(oldmarker);
            mapping = cc_mapping(oldmarker);
            radix_tree_delete(&mapping->page_tree,
                oldmarker->index);
            mapping->nrpages--;
        }
        stats->markers++;
        list_add_tail(&(cpage->marker_queue),
           &stats->marker_queue);
    } else {
        atomic_inc(&cc_expense);
        cc_push_profit();
    }
    spin_unlock_irqrestore(&cc_list_lock, flags);
    return oldmarker;
}

/* For our use, this should only be ran on thing that
 * have an actual address_space object like a file or
    swapper_space. */
static struct address_space *cc_mapping(struct cpage
   *cpage)
{
    if (cpage_swapcache(cpage)) {
        return &swapper_space;
    }
    /* Last bit is set, meaning that it is an
        anonymous page
     * should not happen. */
    BUG_ON((unsigned long)cpage->mapping &
        PAGE_MAPPING_ANON);
    return (struct address_space *)cpage->mapping;
}

/* We have three things that can happen:
 * 1. Page is part of PageSwapCache and is an anonmous
      page
 * 2. Page is part of a file and pages[A[Awapcache
     (tmpfs)
 * 3. Page is part of a file
 * 4. hm, swap cache and no anon_vma.. -> process
     anonymous memory?
 */

static inline struct cc_stats *cc_stats(struct
    address_space *mapping)
{
     struct anon_vma *anon_vma;

     if (((unsigned long)mapping & PAGE_MAPPING_ANON)
         != 0) {
          anon_vma = (struct anon_vma *)
               ((unsigned long)mapping &
                   ~PAGE_MAPPING_ANON);
          return &anon_vma->cc_stats;
     }
     if (mapping != NULL) {
          return &mapping->cc_stats;
     }
     /* Pages not part of an inode and not anonymous,
         but part of
      * Swap Cache: TODO: How they occur is unknown...
      * BUG_ON(!PageSwapCache(page)); */
     return &swapper_space.cc_stats;
}

/* mapping write lock must already be taken and
    checked for consistency
 * must not be part of the cc_list anymore
 * this function is only called on cpages that are
     dirty (no need to
 * write clean ones).
 * lock for page must have been taken
 */
static void cc_pageout(struct cpage *cpage, struct
    address_space *mapping,
     void **slot, struct page *page, unsigned long
         flags,
     struct page **freed_page)
{
     struct zone *zone;
    unsigned long lflags;

    BUG_ON(page == NULL);
    #ifdef CC_DEBUG
    printk(KERN_EMERG "%s: index=0x%lx, mapping=0x%lx, %s\n", __FUNCTION__,
        cpage->index, (unsigned long)page_mapping(page),
        &swapper_space == mapping?"swap":"file");
    #endif

    page_cache_get(page); /* swap cache reference */

    if (&swapper_space == mapping) {
        SetPageSwapCache(page);
    }

    BUG_ON(TestSetPageLocked(page));
    /* Replace page in cache */
    radix_tree_replace_slot(slot, page);
    write_unlock_irqrestore(&mapping->tree_lock, flags);
    #ifdef CC_DEBUG
    printk(KERN_EMERG "%s: put page into mapping cache: 0x%lx,"
        " flags=0x%lx, private=0x%lx\n", __FUNCTION__,
        (unsigned long)page, page->flags, page_private(page));
    #endif

    cc_restore_helper(cpage, page);
    if (freed_page != NULL) {
        fragment_readstart((struct fragment **)&cpage->compressed);
        *freed_page = fragment_readend(
            (struct fragment **)&cpage->compressed, 1);
    }

    cc_free(cpage); /* done with cpage, no more uses! */

    /* Only thing we need to do is to add the page to
     * the inactive list of the lru. */

    zone = page_zone(page);

    /* This should really be set when the page has
       been written out,
     * I wonder why this is not done as in pageout(),
         probably because
     * it is already set..? */
    SetPageUptodate(page);
    SetPageCC(page);
    unlock_page(page);

    /* lru_cache_add was recommended by peterz, but
     * did not work out for me.. need to look into
         this.
     * lru_cache_add(page);
     * put_page(page); */

    spin_lock_irqsave(&zone->lru_lock, lflags);
    SetPageLRU(page);
    add_page_to_inactive_list(zone, page);
    spin_unlock_irqrestore(&zone->lru_lock, lflags);
    put_page(page); /* our reference, still part of
       page cache */
}

/* When there is memory pressure, this function is
    called to limit the size
 * of the compressed cache

 * This is temporarily dropped because it shrinks the
      compressed cache too much!
static int cc_shrinker(int nr_to_scan, gfp_t gfp_mask)
{
    unsigned int thrown_cpages = 0;
    if (nr_to_scan == 0)
        return cell_accounting_fragments();
    while (nr_to_scan-- && cc_shrink_lru(NULL))
        thrown_cpages++;
    return thrown_cpages;
}
*/

```

```
/* Drop enough clean pages out of compressed cache to
   free a dirty one */
static int cc_shrink_clean(void)
{
    unsigned long flags;
    struct list_head *pos;
    struct address_space *mapping[16];
    pgoff_t offset[16];
    int count = 0;
    int max = 15;
    int i;
    return 0;
    spin_lock_irqsave(&cc_list_lock, flags);
    list_for_each(pos, &cell_list_profit) {
        struct cpage *cpage = list_entry(pos, struct
           cpage, list);
        if (cpage_dirty(cpage))
            continue;
        mapping[count] = cc_mapping(cpage);
        offset[count] = cpage->index;
        count++;
        if (!(max--))
            break;
    }
    if (max) {
        list_for_each(pos, &cell_list_profit) {
            struct cpage *cpage = list_entry(pos,
                struct cpage, list);
            if (cpage_dirty(cpage))
                continue;
            mapping[count] = cc_mapping(cpage);
            offset[count] = cpage->index;
            count++;
            if (!(max--))
                break;
        }
    }
    spin_unlock_irqrestore(&cc_list_lock, flags);
    /* Actual removing of clean pages */
    for (i = 0; i < count; i++) {
        cc_remove_cpage(mapping[i], offset[i], NULL,
           NULL);
    }
    return 1;
```

```
}

/* no locks for any/the mapping must be taken by the
    caller */
static int cc_shrink_lru(struct page **freed_page)
{
    struct address_space *mapping;
    unsigned long offset, flags;
    struct cpage *cpage;
    struct page *page;
    struct inode *inode;
    int retval, marker;

again:
    page = NULL;
    marker = 0;
    /* find oldest cpage (head of the list), store
       information,
     * about the mapping and offset */
    spin_lock(&inode_lock);
    spin_lock_irqsave(&cc_list_lock, flags);
    if (!list_empty(&cell_list_profit)) {
        cpage = list_entry(cell_list_profit.next,
           struct cpage, list);
    } else if (!list_empty(&cell_list_expense)) {
        cpage = list_entry(cell_list_expense.next,
           struct cpage, list);
    } else {
        #ifdef CC_DEBUG
        printk(KERN_EMERG "%s: Shrink empty cache?\n",
            __FUNCTION__);
        dump_stack();
        #endif
        spin_unlock_irqrestore(&cc_list_lock, flags);
        spin_unlock(&inode_lock);
        return 0;
    }

    mapping = cc_mapping(cpage);
    offset = cpage->index;
    inode = container_of(mapping, struct inode,
       i_data);

    /* We don't want the inode/mapping to disappear
       until we say so */
    if (mapping != &swapper_space)
         __iget(inode);

    /* will we need to allocate a page? */
    if (cpage_dirty(cpage))
         page = (void *)1; /* hackish, but works */
    else
         page = NULL;

    if (cpage_marker(cpage))
         marker = 1;

    spin_unlock_irqrestore(&cc_list_lock, flags);
    spin_unlock(&inode_lock);

    /* <- At this point the cpage could disappear from
         under us. */

    /* Allocating a page outside any locks. */
    if (page) {
         page = alloc_page(GFP_KERNEL|__GFP_NOWARN);
         if (page == NULL) {
              #ifdef CC_DEBUG
              printk(KERN_EMERG "%s: Shrinking failed, 
                  try later?\n",
                   __FUNCTION__);
              #endif
              cc_locked++;
              return 0;
         }
         BUG_ON(page == NULL);
              }

    retval = cc_remove_cpage(mapping, offset, page,
       freed_page);
    if (mapping != &swapper_space)
         iput(inode);
    /* retry, we should remove an actual cpage */
    if (marker)
         goto again;
    return retval;
}
```

```

/* Helper function that removes a cpage given the
   mapping and the offset and
 * if needed; a page to be used to write to
    backingstore. The page should be
 * freed if not needed. */
static int cc_remove_cpage(struct address_space
   *mapping, pgoff_t offset,
    struct page *page, struct page **freed_page)
{
    unsigned long flags;
    void **slot;
    struct cpage *cpage = NULL;
    struct cc_stats *areastats;

    write_lock_irqsave(&mapping->tree_lock, flags);
    slot = radix_tree_lookup_slot(&mapping->page_tree,
         offset);
    if (slot == NULL ||
        !PageCompressed(radix_tree_deref_slot(slot))) {
        /* The page has already removed, we do nothing
             */
        #ifdef CC_DEBUG
        printk(KERN_EMERG "%s: cpage removed from 
            somewhere else! "
             "(probably cache hit) 0x%lx/0x%lx\n",
                __FUNCTION__,
             (unsigned long)mapping, (unsigned
                 long)offest);
        #endif
        write_unlock_irqrestore(&mapping->tree_lock,
            flags);
        if (page != NULL) {
             ClearPageLocked(page);
             put_page(page);
        }
        return 1;
    }

    cpage =
        MaskPageCompressed(radix_tree_deref_slot(slot));
    areastats = cc_stats(cpage->mapping);

    /* Early warning phase is over */
    if (unlikely(cc_early(areastats) && !cpage_marker(cpage))) {
        cc_early_done(areastats);
    }

    /* pages removed with cc_remove_cpage() are counted as an unused page
     * except when in early state that is.. */
    if (likely(!(areastats->flags & (1 << BAD2)) && !cc_early(areastats))) {
        areastats->unused++;
        cc_area_updated(areastats);
    }
    /* removed from cc lru list */
    cc_remove_lru(cpage);
    /* Clean pages can be thrown out without beeing written back */
    if (!cpage_dirty(cpage)) {
        /* do more or less the same as __delete_from_swap_cache()
         * or __remove_from_page_cache() */
        radix_tree_delete(&mapping->page_tree, offset);
        mapping->nrpages--;
        write_unlock_irqrestore(&mapping->tree_lock, flags);
        if (freed_page != NULL && !cpage_marker(cpage)) {
            fragment_readstart(
                (struct fragment **)&cpage->compressed);
            *freed_page = fragment_readend(
                (struct fragment **)&cpage->compressed, 1);
        }
        cc_free_swap(cpage);
        return 1;
    }
    /* Page that were thought to be clean, are now dirty and we don't
     * have a page to uncompress it into */
    if (unlikely(page == NULL)) {
        printk(KERN_EMERG "Oops, page has become dirty!\n");
        dump_stack();
        BUG();
        return 0;
    }
    /* dirty page: should be written back */
    cc_pageout(cpage, mapping, slot, page, flags,
       freed_page);
    return 1;
}

/*
 * no locks for the mapping must be taken by the
     caller
 * return NULL if it runs out of memory,
 * returns 1 on bad compression
 * returns pointer (larger than (void *)1) to a cpage
     on success
 */
struct cpage *cc_store_page(struct page *page)
{
    struct cpage *cpage;
    void *compressed_data;
    #ifdef CC_LZO
    size_t dst_len;
    int retval;
    #else
    struct page *tmppage;
    unsigned char *htab[(1 << HLOG)];
    #endif
    int counter;
    enum mapping_state mapping_state = CC_ENABLED;

    #ifndef CC_NODISABLE
    /* compressed cache disabling only for clean pages
         */
    if (!PageDirty(page))
        mapping_state = cc_area_state(page);
    #endif

    /* compressed cache is disabled */
    if (mapping_state == CC_DISABLED)
        return (void *)1;

    counter = cell_accounting_pages();
    if (CC_CACHE_MIN >= counter)
        cc_locked = 0;

    #ifdef CC_DEBUG
    printk(KERN_EMERG "%s: page=0x%lx, count=%i, 
        private=0x%lx,"
        " index=0x%lx, flags=0x%lx, mapping=0x%lx, 
            mapcount=0x%x\n",
        __FUNCTION__, (unsigned long)page,
            page_mapcount(page),
        page_private(page), page_index(page),
            page->flags,
        (unsigned long)page->mapping,
            page_mapcount(page));
    #endif
    cpage = (struct cpage *)kmem_cache_alloc(cc_cpage,
        GFP_KERNEL|__GFP_NOWARN); /* really
            GFP_KERNEL? */

    if (cpage == NULL) {
        #ifdef CC_DEBUG
        printk(KERN_EMERG "%s could not allocate a 
            struct cpage.\n"
             , __FUNCTION__);
        #endif
        return NULL;
    }

    /* storing metadata about the page */
    cpage->compressed = NULL;
    cpage->flags = 0;
    if (PageDirty(page))
        cpage_set_dirty(cpage);
    if (PageSwapCache(page)) {
        cpage->index = page_private(page);
        cpage_set_swapcache(cpage);
    } else {
        cpage->index = page->index;
    }
    cpage->mapping = page->mapping;
    cpage->list.next = NULL;
    cpage->list.prev = NULL;

    /* Only insert marker */
    if (mapping_state == CC_MARKER) {
        #ifdef CC_DEBUG
        printk(KERN_EMERG "Putting in marker instead 
            for 0x%lx (%i)\n",
             (unsigned long)cc_mapping(cpage),
             cc_mapping(cpage)->cc_stats.markers);
        #endif
        cpage_set_marker(cpage);
        cpage->marker_queue.next = NULL;
        cpage->marker_queue.prev = NULL;
        return cpage;
    }

    /* Temporary storage to compress to */
    #ifdef CC_LZO
    down(&cc_lzo_sem);
    compressed_data = cc_lzo_tempcomp;
    #else
    tmppage = alloc_page(GFP_KERNEL|__GFP_NOWARN);
    compressed_data = page_address(tmppage);
    if (tmppage == NULL) {
        kmem_cache_free(cc_cpage, cpage);
        #ifdef CC_DEBUG
        printk(KERN_EMERG "%s: tmppage allocation 
            failure\n",
             __FUNCTION__);
        #endif
        return NULL;
    }
    #endif

    /* Compression */
    #ifdef CC_LZO
    retval = lzo1x_1_compress(
        page_address(page), PAGE_SIZE,
        cc_lzo_tempcomp, &dst_len, cc_lzo_workmem);
    if (retval != LZO_E_OK) {
        printk(KERN_EMERG "Error while compressing? %i
             \n", retval);
    }
    cpage->length = dst_len;
    #else
    cpage->length = lzf_compress(page_address(page),
       PAGE_SIZE,
            compressed_data, PAGE_SIZE-200, htab);
        #endif

        /* Bad compression ratio */
        if (cpage->length <= 1 || cpage->length >
            PAGE_SIZE-200) {
            #ifdef CC_DEBUG
            printk(KERN_EMERG "%s:␣page=0x%lx␣bad␣
                compression␣0x%lx\n",
                __FUNCTION__, (long unsigned)page,
                    page_private(page));
            #endif
            #ifdef CC_LZO
            up(&cc_lzo_sem);
            #else
            put_page(tmppage);
            #endif
            kmem_cache_free(cc_cpage, cpage);
            cc_note_bad_compression(page);
            return (void *)1;
        }
        cc_note_good_compression(page);

        #ifdef CC_DEBUG
        printk(KERN_EMERG "%s:␣ALLOCATING␣PAGE,␣
            nr_swap_pages=%li,"
            "␣compressedlength=%i\n", __FUNCTION__,
                nr_swap_pages,
            cpage->length);
        #endif

        #ifdef CC_STORE_CELL
        if (!fragment_write(cpage->length, cc_locked ==
            0?1:0,
            (struct fragment **)&cpage->compressed,
                compressed_data)
            ) {
            #ifdef CC_DEBUG
            printk(KERN_EMERG "%s:␣temporary␣fragment␣"
                "allocation␣failure!\n", __FUNCTION__);
            #endif
            if (cc_locked) {
                counter = cell_accounting_pages();

                do {
                    if (fragment_write(cpage->length,
                        cc_locked == 0?1:0,
                        (struct fragment
                            **)&cpage->compressed,
                        compressed_data))
                            break;

                    if (!cc_shrink_lru(NULL))
                        break;

                    if (counter <
                        cell_accounting_pages()) {
                        fragment_write(cpage->length, 1,
                        (struct fragment
                            **)&cpage->compressed,
                        compressed_data);
                        break;
                    }
                } while (1);
            }
            if (cpage->compressed == NULL) {
                #ifdef CC_LZO
                up(&cc_lzo_sem);
                #else
                put_page(tmppage);
                #endif
                kmem_cache_free(cc_cpage, cpage);
                return NULL;
            }
        }
        #else
        cpage->compressed = kmalloc(cpage->length,
            GFP_KERNEL);

        BUG_ON(cpage->compressed == NULL);
        if (IS_ERR(cpage->compressed)) {
            /* Clean up this part and use ERR stuff,
             * this is a leak by the way */
            BUG();
            return (void *)1;
        }
        memcpy(cpage->compressed, page_address(tmppage),
            cpage->length);
```

```
    #endif

    #ifdef CC_LZO
    up(&cc_lzo_sem);
    #else
    put_page(tmppage);
    #endif

    /* We need the mapping lock to insert a page into
     * the list so this is done later. */
    return cpage;
}

/* Used to restore pages that are accessed, also
   allocates a page */
struct page *cc_restore_page(struct cpage *cpage)
{
    struct page *page;

    page = alloc_page(GFP_KERNEL);
    if (page == NULL) {
        printk(KERN_EMERG "%s could not allocate a 
            page.\n",
             __FUNCTION__);
        BUG();
        return NULL;
    }

    cc_restore(cpage, page);
    return page;
}

/* Used to restore pages that are accessed */
int cc_restore(struct cpage *cpage, struct page *page)
{
    /* Adaptive Cache Size Heuristic */
    if (cpage_profit(cpage)) {
        #ifdef CC_DEBUG
        printk(KERN_EMERG "Unlocked at %i\n",
            cell_accounting_pages());
        #endif
        cc_locked = 0;
    } else {
        int cache_size = cell_accounting_pages();
        if (cache_size <= CC_CACHE_MIN) {
            #ifdef CC_DEBUG
            printk(KERN_EMERG "Forced unlock at %i\n",
                cache_size);
            #endif
            cc_locked = 0;
            return cc_restore_helper(cpage, page);
        }
        cc_locked++;
        if (cc_locked >= 3) {
            #ifdef CC_DEBUG
            printk(KERN_EMERG "Shrinking at %i\n",
               cache_size);
            #endif
            cc_shrink_lru(NULL);
        }
    }
    return cc_restore_helper(cpage, page);
}


/* This function is run on every access to a cpage,
 * removing it from the LRU and doing some accounting
 * should only be ran with mapping lock held! */
void cc_accessed_cpage(struct cpage *cpage)
{
    struct cc_stats *areastats =
       cc_stats(cpage->mapping);
    cc_remove_lru(cpage);

    /* Early warning phase is over, only for real
       compressed pages! */
    if (unlikely(cc_early(areastats) &&
       !cpage_marker(cpage))) {
        cc_early_done(areastats);
    }
    /* Only run when in (early) bad ratio phases */
    if (likely(!(areastats->flags & (1 << BAD2))
        && !cpage_dirty(cpage) &&
            !cc_early(areastats))) {
        areastats->used++;
        cc_area_updated(areastats);
    }
}
```

```

static int cc_restore_helper(struct cpage *cpage,
   struct page *page)
{
     int retval = 0;
     void *buffer;
     #ifdef CC_LZO
     size_t dst_len;
     #endif

     BUG_ON(page == NULL);

     /* State of page */
     page->mapping = cpage->mapping;
     inc_zone_page_state(page, NR_FILE_PAGES);
     if (cpage_swapcache(cpage)) {
          set_page_private(page, cpage->index);
          SetPageSwapCache(page);
     } else {
          page->index = cpage->index;
     }
     if (cpage_dirty(cpage))
          SetPageDirty(page);
     #ifdef CC_DEBUG
     if (cpage->compressed == NULL) {
          printk(KERN_EMERG "%s:␣cpage=0x%lx,␣
              index=0x%lx,␣length=%i,"
               "␣retval=%i,\n", __FUNCTION__, (unsigned
                    long)cpage,
               cpage->index, cpage->length, retval);
     }
     #endif
     BUG_ON(cpage->compressed == NULL);

     #ifdef CC_STORE_CELL
     buffer = fragment_readstart((struct fragment
         **)&cpage->compressed);
     #else
     buffer = cpage->compressed;
     #endif
     #ifdef CC_DEBUG
     printk(KERN_EMERG "%s:␣Going␣to␣decompress␣...␣
         fragment=0x%lx,"
          "␣0x%lx/0x%lx\n", __FUNCTION__,
        (unsigned long)cpage->compressed, (unsigned
            long)page->mapping,
        (unsigned long)cpage->index);
    #endif

    /* Decompressing */
    #ifdef CC_LZO
    dst_len = PAGE_SIZE;
    retval = lzo1x_decompress_safe(buffer,
        cpage->length,
        page_address(page), &dst_len);
    if (unlikely(retval != LZO_E_OK)) {
        printk(KERN_EMERG "Corruption? LZO returned 
            code: %i\n", retval);
        printk(KERN_EMERG "length: %i\n",
            cpage->length);
        BUG();
    }
    retval = dst_len;
    #else
    retval = lzf_decompress(buffer, cpage->length,
        page_address(page),
        PAGE_SIZE);
    #endif

    #ifdef CC_STORE_CELL
    fragment_readend((struct fragment
        **)&cpage->compressed, 0);
    BUG_ON(cpage->compressed != NULL);
    #endif

    /* for filemapped pages, is this always uptodate?
     * make sure we don't set this until after we have
     * have restored the page.
     * SetPageUptodate(page); */

    #ifdef CC_DEBUG
    if (unlikely(retval != PAGE_SIZE)) {
        printk(KERN_EMERG "%s: The world has 
            officially "
        "come to an end retval=%i (corrupted page)\n",
        __FUNCTION__, retval);
    }
    #endif
    BUG_ON(retval != PAGE_SIZE);

    return retval;
}

/* Frees all data assosicated with a cpage */
void cc_free(struct cpage *cpage)
{
    #ifdef CC_STORE_CELL
    if (cpage->compressed != NULL &&
        !cpage_marker(cpage)) {
        fragment_readstart((struct fragment
            **)&cpage->compressed);
        fragment_readend((struct fragment
            **)&cpage->compressed, 0);
        BUG_ON(cpage->compressed != NULL);
    }
    #else
    kfree(cpage->compressed);
    cpage->compressed = NULL;
    #endif
    kmem_cache_free(cc_cpage, cpage);
}


/* cc_free() on all cpages in list, should be allowed
    to sleep! */
void cc_free_list(struct list_head *cpage_free_list)
{
    struct list_head *pos, *n = NULL;

    list_for_each_safe(pos, n, cpage_free_list) {
        struct cpage *cpage = list_entry(pos, struct
            cpage, list);
        cc_free(cpage);
    }
}

/* Frees all data assosicated with a cpage, and if it
    is a page in the
 * swap cache, free the swap entry. Remove swap entry,
    lock order says
 * we must do this without mapping lock! */
void cc_free_swap(struct cpage *cpage)
```

```
{
    swp_entry_t entry = { .val = cpage->index };
    if (cpage_swapcache(cpage)) {
        swap_free(entry);
    }
    cc_free(cpage);
}

/* cc_free_swap() on all cpages in list, should be
   allowed to sleep! */
void cc_free_swap_list(struct list_head
   *cpage_free_list)
{
    struct list_head *pos, *n = NULL;

    list_for_each_safe(pos, n, cpage_free_list) {
        struct cpage *cpage = list_entry(pos, struct
           cpage, list);
        cc_free_swap(cpage);
    }
}

void __init cc_setup(void)
{
    /* invariant on alignment of struct page,
     * unfortunatly only valid for x86, important to
        note however
     * BUG_ON( ((unsigned long)mem_map %
        CC_COMPRESSED_PAGE_ALIGN)
     * || (sizeof(mem_map) %
        CC_COMPRESSED_PAGE_ALIGN) );
     */
    cc_locked = 0;
    cc_cpage = kmem_cache_create("cc_cpage", sizeof(
       struct cpage ),
        CC_COMPRESSED_PAGE_ALIGN, SLAB_PANIC, NULL,
           NULL);
    spin_lock_init(&cc_list_lock);
    INIT_LIST_HEAD(&cell_list_expense);
    INIT_LIST_HEAD(&cell_list_profit);
    INIT_LIST_HEAD(&swapper_space.cc_stats.marker_queue);
    #ifdef CC_STORE_CELL
    cell_setup(5);
    #endif
    #ifdef CC_LZO
    sema_init(&cc_lzo_sem, 1);
    cc_lzo_workmem = kmalloc(LZO1X_MEM_COMPRESS,
        GFP_KERNEL);
    cc_lzo_tempcomp = kmalloc(LZO1X_MEM_COMPRESS,
        GFP_KERNEL);
    #endif
    /* cc_shrinker_data  = set_shrinker(1,
        cc_shrinker); */
}

/* Disabling of compressed caching */
static int cc_early(struct cc_stats *stats)
{
    if (stats->flags & (1 << EARLY_WARNING_DONE)) {
        return 0;
    }
    return 1;
}

/* Early warning phase is over */
static void cc_early_done(struct cc_stats *stats)
{
    #ifdef CC_DEBUG
    printk(KERN_EMERG "%s: stats 0x%lx done with early
        warning!\n",
        __func__, (unsigned long)stats);
    #endif
    stats->flags |= (1 << EARLY_WARNING_DONE);
    stats->used = 0;
    /* Bad area detected, would have trashed the cache
        */
    if (stats->unused > 200 || stats->unused >
        cell_accounting_pages()) {
        stats->unused = max(CC_MAX_EARLY_WARNING,
            CC_MAX_MARKERS) + 10;
    }
    cc_area_updated(stats);
}

/* Should we put in a marker instead? */
static int cc_put_marker(struct cc_stats *stats)
{
    if (stats->markers <= CC_MAX_MARKERS)
```

```
        return 1;
    return 0;
}

/* Checks if the mapping is in a state where
   compression is disabled,
 * returns:
 * CC_ENABLED - Go ahead and put page into mapping
 * CC_DISABLED - Do not put anything into the mapping
 * CC_MARKER - Can put a marker into the mapping
 */
static enum mapping_state cc_area_state(struct page
   *page)
{
    int disabled = 0, retval = CC_ENABLED;
    struct address_space *mapping =
       page_mapping(page);
    struct cc_stats *stats = cc_stats(page->mapping);

    BUG_ON(mapping == NULL);
    read_lock_irq(&mapping->tree_lock);

    /* Are we in a bad compression state? */
    if (stats->flags & (1 << BAD2)) {
        if (stats->used == stats->unused)
            goto done;
        read_unlock_irq(&mapping->tree_lock);
        write_lock_irq(&mapping->tree_lock);
        if (stats->flags & (1 << BAD2))
            stats->used++;
        write_unlock_irq(&mapping->tree_lock);
        retval = CC_DISABLED;
        return retval;
    }

    /* lets an unproven mapping to put up to
       CC_MAX_EARLY_WARNING
     * into compressed cache */
    if (cc_early(stats)) {
        read_unlock_irq(&mapping->tree_lock);
        write_lock_irq(&mapping->tree_lock);
        /* changed while unlocked? */
        if (!cc_early(stats)) {
            write_unlock_irq(&mapping->tree_lock);
            return retval;
        }
        /* Note how many we have in the cache and
         * how many we have skipped */
        if (stats->used >= CC_MAX_EARLY_WARNING) {
            retval = CC_DISABLED;
            if (stats->unused < 230)
                stats->unused++; /* skipped */
        } else {
            stats->used++;
        }
        write_unlock_irq(&mapping->tree_lock);
        return retval;
    }
    disabled = stats->flags & (1 << UNUSED);
    if (disabled) {
        retval = CC_MARKER;
    }
done:
    read_unlock_irq(&mapping->tree_lock);
    return retval;
}

/* Updates the stats of bad compression,
 * if two consecutive pages has bad compression
 * then the next two pages will not be compressed.
 * next it will skip 3, then 4 and so on until a good
 * compression ratio is reported */
static void cc_note_bad_compression(struct page *page)
{
    struct address_space *mapping =
        page_mapping(page);
    struct cc_stats *stats = cc_stats(page->mapping);

    write_lock_irq(&mapping->tree_lock);
    if (!(stats->flags & (1 << BAD1))) {
        stats->flags |= (1 << BAD1);
        goto done;
    }
    if (!(stats->flags & (1 << BAD2))) {
        stats->flags |= (1 << BAD2);
        stats->used = 2;
        stats->unused = 0;
        goto done;
    }
    if (stats->unused < stats->used) {
        stats->unused++;
        goto done;
    }
    if (stats->used < CC_MAX_BAD_COMPRESSION_SKIP)
        stats->used++;
    stats->unused = 0;
done:
    write_unlock_irq(&mapping->tree_lock);
}

static void cc_note_good_compression(struct page
   *page)
{
    struct address_space *mapping =
        page_mapping(page);
    struct cc_stats *stats = cc_stats(page->mapping);
    write_lock_irq(&mapping->tree_lock);
    /* Put back into early phase */
    if (stats->flags & (1 << BAD2)) {
        stats->flags &= ~((1 << BAD1) & (1 << BAD2)
            & (1 << EARLY_WARNING_DONE));
    }
    write_unlock_irq(&mapping->tree_lock);
}

/* Adjusts history to recent history,
 * write lock for the mapping must be taken! */
static void cc_area_updated(struct cc_stats *stats)
{
    int disabled_ratio = stats->flags & (1 << UNUSED);

    /* cc_unused and cc_used are meaningless when
     * bad compression is enabled */
    if (stats->flags & (1 << BAD2))
        return;

    if (cc_early(stats)) {
        BUG();
        return;
    }

    /* We disable compressed caching if 80% of the
        pages are unused */
    if (!disabled_ratio && stats->unused >
        stats->used*4) {
        #ifdef CC_DEBUG
        printk(KERN_EMERG "Bad␣ratio␣mapping=0x%lx,␣
            used=%u,␣"
            "unused=%u,␣markers=%u\n", (unsigned
                long)mapping,
            stats->used, stats->unused,
            stats->markers);
        #endif
        stats->flags |= (1 << UNUSED);
    /* Reenable compressed caching if 50% of the pages
        would be reused */
    } else if (disabled_ratio && (stats->used >=
        stats->unused)) {
        #ifdef CC_DEBUG
        printk(KERN_EMERG "Good␣ratio␣mapping=0x%lx,␣
            used=%u,␣"
            "unused=%u,␣markers=%u\n", (unsigned
                long)mapping,
            stats->used, stats->unused,
            stats->markers);
        #endif
        stats->flags &= ~(1 << UNUSED);
        /* we are skeptical about this area, so we
            restart
         * the early phase for it */
        stats->flags &= ~(1 << EARLY_WARNING_DONE);
        stats->used = 0;
        stats->unused = 0;
    }

    /* Only recent history */
    if (stats->used+stats->unused > CC_CACHE_HISTORY)
        {
        stats->used /= 2;
        stats->unused /= 2;
    }
}

/* EOF */
```

D.2 Cell allocator

D.2.1 *include/linux/cell.h*

```

/* Public structures */

struct fragment;

/* interface */
void cell_setup(unsigned int compact_cells);
int cell_accounting_pages(void);
int cell_accounting_fragments(void);
int fragment_write(unsigned int size, unsigned int
    grow,
     struct fragment **user, void *data);
void *fragment_readstart(struct fragment **user);
struct page *fragment_readend(struct fragment **user,
    int returnpage);
```

D.2.2 *mm/cell.c*

```

/*
 * This data structure is loosly based on the
      description by Castro et. al it in
 * their compressed caching paper.
 *
 * Lock ordering is as follows:
   - 1. free_cells[i], 2. cells
 * Invariants:
 * There will never be a user that points to a page
     that is not a cell.
 * The cell structure should always be in a consistant
      state when not locked.
 *
 * Debugging can be turned on by defining CELL_DEBUG
 */

#include <linux/list.h>
#include <asm/page.h>
#include <asm/semaphore.h>
#include <linux/mutex.h>
```

```
#include <linux/pagemap.h>
#include <linux/mm.h>
#include <linux/sched.h>

/* Interface */
#include <linux/cell.h>

/* Implementation */



struct fragment {
    struct fragment *next;
    struct fragment **user;
    char data[0];
};

/* We need a doubly linked list because we need to
   remove stuff from the
 * middle of the list at times */
struct cell {

    struct list_head list;
    unsigned short free_space;
    unsigned short allocations;
    unsigned long flags;
    struct fragment fragment[0];
};

struct cell_accounting {
    atomic_t pages;
    atomic_t cells;
    atomic_t fragments;
};

static struct cell_accounting accounting;

#define FRAGMENT_LOCKBIT 0

/* Cell flags */
#define CELL_FLAG_MANAGED 0
#define CELL_FLAG_KEEP 1
#define CELL_MANAGED(cell) test_bit(CELL_FLAG_MANAGED,
    &cell->flags)
```

```
#define CELL_MANAGE(cell) __set_bit(CELL_FLAG_MANAGED,
     &cell->flags)
#define CELL_UNMANAGE(cell)
    __clear_bit(CELL_FLAG_MANAGED, &cell->flags)
#define CELL_KEEP(cell) test_bit(CELL_FLAG_KEEP,
    &cell->flags)
#define CELL_SET_KEEP(cell) __set_bit(CELL_FLAG_KEEP,
    &cell->flags)
#define CELL_CLEAR_KEEP(cell)
    __clear_bit(CELL_FLAG_KEEP, &cell->flags)


#define CELL_ORDER 1
#define CELL_LENGTH 2
#define CELL_GRANULARITY 256
#define CELL_CATEGORIES
    (PAGE_SIZE*CELL_LENGTH/CELL_GRANULARITY)
#define CELL_MASK (PAGE_SIZE-1)
#define CELL_PAGE(cell) (struct page
    *)page_private(virt_to_page(cell))
#define FRAGMENT_CELL(fragment) (struct cell
    *)page_address(CELL_PAGE(fragment))
/* FRAGMENT_LENGTH gives the length of the data part
    of the fragment. */
#define FRAGMENT_LENGTH(_fragment) (_fragment->next ==
     NULL?\
     ((unsigned long)FRAGMENT_CELL(_fragment)+\
     (PAGE_SIZE*CELL_LENGTH)-(unsigned
        long)_fragment->data) :\
     ((unsigned long)_fragment->next - (unsigned
        long)_fragment->data) )
static struct list_head free_cells[CELL_CATEGORIES];
static struct mutex
    free_cells_list_lock[CELL_CATEGORIES];

#define free_cells_lock(i)
    mutex_lock(&free_cells_list_lock[i])
#define free_cells_unlock(i)
    mutex_unlock(&free_cells_list_lock[i])

/* For debugging locking:
#define free_cells_lock(i) \
     printk(KERN_EMERG "%s: %i waiting for %i\n",\
      __FUNCTION__, __LINE__, i);\
     down(&free_cells_list_lock[i]);\
     printk(KERN_EMERG "%s: Locked free_cells_list[%i]
          %i\n",\
     __FUNCTION__, i, __LINE__)
#define free_cells_unlock(i) \
    up(&free_cells_list_lock[i]); \
    printk(KERN_EMERG "%s: Unlocked
        free_cells_list[%i] %i\n",\
         __FUNCTION__, i, __LINE__)
*/

/* We need some locking: */
static inline void bit_spin_lock(int bitnum, unsigned
   long *addr)
{
    while (test_and_set_bit(bitnum, addr)) {
        yield();
    }
}

static inline int bit_spin_trylock(int bitnum,
   unsigned long *addr)
{
    if (test_and_set_bit(bitnum, addr)) {
        return 0;
    }
    return 1;
}

static struct semaphore temporary_cells_semaphore;
static struct semaphore temporary_cells_lock;
static struct list_head temporary_cells;

/* Prototypes */
static struct cell *cell_temporary(void);
static void cell_global_compact(void);

static inline void fragment_init(struct fragment
   *fragment)
{
    fragment->next = NULL;
    fragment->user = NULL;
}

```

```
static inline void cell_init(struct cell *cell)
{
    fragment_init(cell->fragment);
    /* cell->free_space = CELL_LENGTH*PAGE_SIZE
        - sizeof(struct cell) - sizeof(struct
            fragment); */
    cell->free_space =
        FRAGMENT_LENGTH(cell->fragment);
    cell->allocations = 0;
    CELL_UNMANAGE(cell);
    CELL_CLEAR_KEEP(cell);
}

static void cell_lock(struct cell *cell)
{
    struct page *page;
    page = CELL_PAGE(cell);
    lock_page(page);
}

static inline void cell_unlock(struct cell *cell)
{
    struct page *page;
    page = CELL_PAGE(cell);
    unlock_page(page);
}

static void cell_debug(struct cell *cell)
{
    struct fragment *fragment;
    unsigned int freespace, usedspace;

    printk(KERN_EMERG "%s: Debugging cell=0x%lx,
        free_space=%u\n",
        __FUNCTION__, (unsigned long)cell,
            cell->free_space);
    fragment = cell->fragment;
    freespace = 0;
    usedspace = 0;
    do {
        unsigned int fragment_length =
            FRAGMENT_LENGTH(fragment);
        printk(KERN_EMERG "%s: fragment=0x%lx,
            length=%u, user=0x%lx,"
            " next=0x%lx\n", __FUNCTION__, (unsigned
                long)fragment,
            fragment_length, (unsigned
                long)fragment->user,
            (unsigned long)fragment->next);
        if (fragment->user != NULL) {
            usedspace += FRAGMENT_LENGTH(fragment)
                + sizeof(fragment);
        } else {
            freespace += FRAGMENT_LENGTH(fragment);
            usedspace += sizeof(fragment);
        }
    } while ((fragment = fragment->next));
    printk(KERN_EMERG "%s: End statistics: cell=0x%lx,
        calculated: "
        "freespace=%u, usedspace=%u, managed=%s, 
            keep=%s\n",
        __FUNCTION__, (unsigned long)cell, freespace,
            usedspace,
        CELL_MANAGED(cell)?"true":"false",
        CELL_KEEP(cell)?"true":"false");
}


void cell_debug_all(void)
{
    struct cell *cell;
    unsigned int i = 0;
    struct list_head *pos;

    printk(KERN_EMERG "%s: 
        ----------------------------------------------------------------\n",
        __FUNCTION__);
    for (; i < CELL_CATEGORIES; i++) {
        free_cells_lock(i);
        if (list_empty(&free_cells[i])) {
            free_cells_unlock(i);
            continue;
        }
        /* Find a cell with enough space (if
            possible) */
        cell = NULL;
        list_for_each(pos, &free_cells[i]) {
            struct cell *cell_pos =
                    list_entry(pos, struct cell, list);
                printk(KERN_EMERG "-->␣Looking␣at␣
                    cell_pos->freespace"
                    "=%u,␣cell=0x%lx,␣allocations=%u\n",
                    cell_pos->free_space,
                    (unsigned long)CELL_PAGE(cell_pos),
                    cell_pos->allocations);
                cell_debug(cell_pos);
            }
            free_cells_unlock(i);
        }
        i = 0;
        printk(KERN_EMERG "--␣looking␣at␣temporary␣cells␣
            --\n");
        list_for_each(pos, &temporary_cells) {
            struct cell *cell_pos = list_entry(pos, struct
                cell, list);
            i++;
            printk(KERN_EMERG "-tmp␣%i->␣Looking␣at␣
                cell_pos->freespace=%u,"
                "␣cell=0x%lx,␣allocations=%u\n", i,
                cell_pos->free_space,
                (unsigned long)CELL_PAGE(cell_pos),
                cell_pos->allocations);
            cell_debug(cell_pos);
        }
}

static struct cell *cell_allocate(void)
{
        struct page *page;
        struct cell *cell;
        int i;

        page = alloc_pages(GFP_KERNEL|__GFP_NOWARN,
            CELL_ORDER);

        if (page == NULL)
            return NULL;

        split_page(page, CELL_ORDER);
        for (i = 0; i < (1 << CELL_ORDER); i++)
            set_page_private(page+i, (unsigned long)page);

```

```
    atomic_add(CELL_LENGTH, &accounting.pages);
    atomic_inc(&accounting.cells);

    cell = (struct cell *)page_address(page);
    cell_init(cell);

    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: count it %i page=0x%lx, 
        cell=0x%lx\n",
        __FUNCTION__, atomic_read(&accounting.cells),
        (unsigned long)page, (unsigned long)cell);
    #endif
    return cell;
}

/* Nothing should be referencing this cell when
   freeing it! */
static struct page *cell_free(struct cell *cell, int
   returnpage)
{
    int i = 0;
    struct page *page = CELL_PAGE(cell);
    if (returnpage)
        i = 1;
    for (; i < (1 << CELL_ORDER); i++)
        put_page(page+i);
    atomic_dec(&accounting.cells);
    atomic_sub(CELL_LENGTH, &accounting.pages);
    if (returnpage)
        return page;
    else
        return NULL;
}

/* Returns a cell with enough space, removed from the
   free list */
static struct cell *cell_get(unsigned int size)
{
    struct cell *cell;
    unsigned int i = size/CELL_GRANULARITY;
    struct list_head *pos;

    for (; i < CELL_CATEGORIES; i++) {
        free_cells_lock(i);
        if (list_empty(&free_cells[i])) {
            free_cells_unlock(i);
            continue;
        }
        /* Find a cell with enough space (if
            possible) */
        cell = NULL;
        list_for_each(pos, &free_cells[i]) {
            struct cell *cell_pos =
                list_entry(pos, struct cell, list);
            #ifdef CELL_DEBUG
            printk(KERN_EMERG "Looking␣at␣
                cell_pos->freespace=%u,"
                "␣size=%u,␣page=0x%lx\n",
                    cell_pos->free_space,
                size, (unsigned long)CELL_PAGE(cell));
            #endif
            if (cell_pos->free_space > size) {
                cell = cell_pos;
                #ifdef CELL_DEBUG
                printk(KERN_EMERG "%s:␣Found␣cell␣with
                    ␣"
                    "free_space=%u\n", __FUNCTION__,
                    cell->free_space);
                #endif
                break;
            }
        }

        if (cell == NULL) {
            #ifdef CELL_DEBUG
            printk(KERN_EMERG "%s:␣Going␣to␣the␣next␣
                category\n",
                __FUNCTION__);
            #endif
            free_cells_unlock(i);
            continue;
        }

        cell_lock(cell);
       #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s:␣cell=0x%lx␣deleted␣from
            ␣list\n",
            __FUNCTION__, (unsigned long)cell);
        #endif
        BUG_ON(cell->list.next == NULL &&
            cell->list.prev == NULL);
        list_del(&(cell->list));
        cell->list.next = NULL;
        cell->list.prev = NULL;
        free_cells_unlock(i);
        return cell;
    }
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Returned NULL\n",
        __FUNCTION__);
    #endif
    return NULL;
}

/* Takes an emptied cell and keeps it for temporary
   usage
 * Locks should be taken for the cell */
static void cell_keep(struct cell *cell)
{
    cell_init(cell);
    down(&temporary_cells_lock);
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Putting 0x%lx into list 
        temporary_cells list\n",
        __FUNCTION__, (unsigned long)cell);
    #endif
    list_add(&(cell->list), &temporary_cells);
    cell_unlock(cell);
    up(&temporary_cells_lock);
    up(&temporary_cells_semaphore);
}

/* Inserts a cell back into the free lists, and
   removes the locks
 * Make sure that freeing instances do not put the
     cell back into
 * the free-lists behind our backs. */
static struct page *cell_put(struct cell *cell, int
   returnpage)
{
    unsigned int i;

    /* This is only set when allocating, skip
     * reinserting into a list until later. */
    if (CELL_MANAGED(cell)) {
        #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s: Quick end for..\n", __FUNCTION__);
        #endif
        cell_unlock(cell);
        return NULL;
    }

    if (cell->allocations == 0) {
        if (unlikely(CELL_KEEP(cell))) {
            #ifdef CELL_DEBUG
            printk(KERN_EMERG "%s: Last allocation of a compacted"
            " cell=0x%lx, keep!\n", __FUNCTION__,
            (unsigned long)cell);
            #endif
            cell_keep(cell);
            return NULL;
        } else {
            cell_unlock(cell);
            return cell_free(cell, returnpage);
        }
    }

    /* This cell is waiting to be returned to the temporary storage,
     * but still has allocations that are active */
    if (unlikely(CELL_KEEP(cell))) {
        #ifdef CELL_DEBUG
        printk("%s: More allocations on cell=0x%lx,"
            " will be kept later\n", __FUNCTION__,
            (unsigned long)cell);
        #endif
        cell_unlock(cell);
        return NULL;
    }

    /* Protect us from the race, we have already won!
     * Whatever happens next, we are the ones to put it back! */
    CELL_MANAGE(cell);
```

```

tryagain:
    i = cell->free_space/CELL_GRANULARITY;
    cell_unlock(cell);
    /* <- race covered by MANAGED */
    free_cells_lock(i);
    cell_lock(cell);
    /* Last fragment was freed from somewhere else */
    if (unlikely(cell->allocations == 0)) {
        free_cells_unlock(i);
        cell_unlock(cell);
        return cell_free(cell, 0);
    }
    /* Maybe it now should be put into another list.
       */
    if (i != cell->free_space/CELL_GRANULARITY) {
        free_cells_unlock(i);
        #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s: Something were freed 
            when we wern't"
             " looking, take correct lock\n",
                __FUNCTION__);
        #endif
        goto tryagain;
    }
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Putting 0x%lx into list 
        %i\n", __FUNCTION__,
         (unsigned long)cell, i);
    #endif
    list_add_tail(&(cell->list), &free_cells[i]);
    CELL_UNMANAGE(cell);
    cell_unlock(cell);
    free_cells_unlock(i);
    return NULL;
}

/* Splits a fragment into one length bit and if
   possible the
 * rest into the next fragment. */
static struct fragment *fragment_split(struct cell
   *cell,
    struct fragment *fragment, unsigned int length)
{
    unsigned int fragment_length;
    fragment_length = FRAGMENT_LENGTH(fragment);
    if (length < fragment_length - sizeof(struct
        fragment)) {
        struct fragment *new_fragment;

        new_fragment = (struct fragment
            *)&fragment->data[
            fragment_length-sizeof(struct
                fragment)-length];
        fragment_init(new_fragment);

        new_fragment->next = fragment->next;
        fragment->next = new_fragment;

        cell->free_space -=
            FRAGMENT_LENGTH(new_fragment)
            + sizeof(struct fragment);

        if ((unsigned long)fragment & 1) {
            cell_debug(cell);
            BUG();
        }

        return new_fragment;
    } else {
        cell->free_space -= FRAGMENT_LENGTH(fragment);
        return fragment;
    }
}


/* Must have been removed from other categories first
 * First it looks for a large enough free fragment,
 * If it doesn't find one it returns NULL. */
static struct fragment *cell_fragment(struct cell
    *cell, int length)
{
    unsigned int tmpfree, needed_length;
    struct fragment *fragment, *prev_fragment;

    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s:␣Looking␣at␣a␣cell␣with␣
        free_space=%u,"
        " length=%u\n", __FUNCTION__,
            cell->free_space, length);
    #endif

    if (unlikely(cell->free_space < length)) {
        cell_debug(cell);
        BUG_ON(cell->free_space < length);
    }

    needed_length = (sizeof(struct fragment) +
        length);
    tmpfree = cell->free_space;
    fragment = cell->fragment;
    prev_fragment = NULL;
    do {
        if (fragment->user == NULL) {
            unsigned int fragment_length;

            /* Merging two free fragments */
            if (prev_fragment && prev_fragment->user
                == NULL) {
                tmpfree +=
                    FRAGMENT_LENGTH(prev_fragment)
                    + sizeof(struct fragment);
                cell->free_space += sizeof(struct
                    fragment);
                prev_fragment->next = fragment->next;
                fragment = prev_fragment;
            }

            /* Found large enough fragment */
            fragment_length =
                FRAGMENT_LENGTH(fragment);
            if (fragment_length >= length) {
                return fragment_split(cell, fragment,
                    length);
            }

            /* There can be no large enough free
                fragments
             * left in this cell */
            tmpfree -= fragment_length;
            if (tmpfree < length) {
                #ifdef CELL_DEBUG
```

```
                        printk(KERN_EMERG "%s: tmpfree(%u) < length(%u)"
                            " no space left! (the cell is said to"
                            " have %u left), frglen=%u\n",
                            __FUNCTION__, tmpfree, length,
                            cell->free_space, fragment_length);
                        #endif
                        return NULL;
                }

            }
    } while ((fragment = fragment->next));

    /* should be caught by the cut-off on not enough space left */
    BUG_ON(fragment == NULL);

    return NULL;
}

/* Update the user of the page in a good way :P */
static inline void fragment_update_user(struct fragment *oldfragment,
    struct fragment *newfragment)
{
    newfragment->user = oldfragment->user;
    *oldfragment->user = newfragment;
}


/* Return one of the temporary cells with locks taken */
static struct cell *cell_temporary(void)
{
    struct cell *cell;
    down(&temporary_cells_semaphore);
    down(&temporary_cells_lock);
    BUG_ON(list_empty(&temporary_cells));
    cell = (struct cell *)list_entry(temporary_cells.next,
        struct cell, list);
    cell_lock(cell);
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: cell=0x%lx deleted from list temporary_cells"
        " list\n", __FUNCTION__, (unsigned long)cell);
    #endif
    list_del(&(cell->list));
    cell->list.next = NULL;
    cell->list.prev = NULL;
    up(&temporary_cells_lock);
    return cell;
}

/* Copies all active fragments to new cell. */
static struct cell *cell_compact(struct cell *oldcell)
{
    struct fragment *fragment, *prev_fragment;
    struct cell *newcell = cell_temporary();

    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Compacting oldcell=0x%lx, newcell=0x%lx,"
        " free_space=%u\n", __FUNCTION__, (unsigned long)oldcell,
        (unsigned long)newcell, newcell->free_space);
    #endif
    fragment = oldcell->fragment;
    prev_fragment = NULL;
    do {
        if (fragment->user != NULL) {
            unsigned int fragment_length =
                FRAGMENT_LENGTH(fragment);
            struct fragment *newfragment;
            if (!bit_spin_trylock(FRAGMENT_LOCKBIT,
                (unsigned long *)fragment->user)) {
                #ifdef CELL_DEBUG
                printk(KERN_EMERG "%s: Skipping, fragment is"
                    " in use (bitlock on 0x%lx,"
                    " content=0x%lx)\n", __FUNCTION__,
                    (unsigned long)fragment->user,
                    (unsigned long)*fragment->user);
                #endif
                prev_fragment = fragment;
                continue;
                }

                newfragment = cell_fragment(newcell,
                    fragment_length);

                if (unlikely(newfragment == NULL)) {
                     printk(KERN_EMERG "%s: FAILED 
                         fragment?"
                          "OLDCELL:\n", __FUNCTION__);
                     cell_debug(oldcell);
                     BUG();
                }

                memcpy(newfragment->data, fragment->data,
                     fragment_length);
                newcell->allocations++;
                newfragment->user = fragment->user;
                #ifdef CELL_DEBUG
                printk(KERN_EMERG "%s: *User(0x%lx) set to
                    0x%lx\n",
                     __FUNCTION__, (unsigned
                         long)fragment->user,
                     (unsigned long)newfragment);
                #endif
                /* This also clears the bitlock */
                *fragment->user = newfragment;

                fragment->user = NULL;
                oldcell->allocations--;
                /* keep a cell consistent, not really
                    needed */
                oldcell->free_space += fragment_length;
          }

          if (prev_fragment && prev_fragment->user ==
              NULL) {
                prev_fragment->next = fragment->next;
                fragment = prev_fragment;
          }

          prev_fragment = fragment;
     } while ((fragment = fragment->next));

     if (oldcell->allocations != 0) {
        #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s: Marked old cell 0x%lx as freeable by"
            " readend, cell won't be freed for %i\n", __FUNCTION__,
            (unsigned long)oldcell, oldcell->allocations);
        #endif
        CELL_SET_KEEP(oldcell);
        cell_unlock(oldcell);
    } else {
        cell_keep(oldcell);
    }
    return newcell;
}

int fragment_write(unsigned int size, unsigned int grow,
    struct fragment **user, void *data)
{
    struct cell *cell;
    struct fragment *fragment;
    unsigned int length;

    /* So that we can use bit 0 for FRAGMENT_LOCK_BIT */
    if (size % 2)
        length = size + 1;
    else
        length = size;

    cell = cell_get(size);
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Found a cell=0x%lx\n", __FUNCTION__,
        (unsigned long)cell);
    #endif

    if (cell == NULL && grow == 0)
        return 0;

    if (cell == NULL && grow == 1) {
        #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s: Did not find a cell: cell=0x%lx\n",
            __FUNCTION__, (unsigned long)cell);
        #endif
        cell = cell_allocate();
        if (cell == NULL)
            return 0;
        cell_lock(cell);
    }

    fragment = cell_fragment(cell, length);

    if (fragment == NULL) {
        cell = cell_compact(cell);
        fragment = cell_fragment(cell, length);
    }

    if (unlikely(fragment == NULL))
        cell_debug(cell);

    BUG_ON(fragment == NULL);
    cell->allocations++;
    atomic_inc(&accounting.fragments);
    fragment->user = user;
    *fragment->user = fragment;
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: *User(0x%lx) set to 0x%lx\n", __FUNCTION__,
        (unsigned long)fragment->user, (unsigned long)fragment);
    #endif

    memcpy(fragment->data, data, size);
    cell_put(cell, 0);
    return 1;
}

/*
 * Returns a pointer to the data.
 */
void *fragment_readstart(struct fragment **user)
{
    struct fragment *fragment;
    bit_spin_lock(FRAGMENT_LOCKBIT, (unsigned long
        *)user);
    fragment = (struct fragment *)((unsigned
        long)*user & ~1);
    return (fragment->data);
}

/*
 * Frees the fragment, after a read.
 */
struct page *fragment_readend(struct fragment **user,
    int returnpage)
{
    struct cell *cell;
    struct fragment *fragment;
    struct page *page;
    int i;

    fragment = (struct fragment *)((unsigned
        long)*user & ~1);
    cell = FRAGMENT_CELL(fragment);

    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Bitlock=0x%lx, 
        cell=0x%lx\n",
         __FUNCTION__, (unsigned long)user, (unsigned
            long)cell);
    #endif

    cell_lock(cell);

tryagain:
    if (CELL_MANAGED(cell) || CELL_KEEP(cell)) {
        cell->free_space += FRAGMENT_LENGTH(fragment);
        cell->allocations--;
        *fragment->user = NULL;
        fragment->user = NULL;
        atomic_dec(&accounting.fragments);
        return cell_put(cell, returnpage);
    }

    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: fragment=0x%lx\n",
        __FUNCTION__,
            (unsigned long)fragment);
    #endif

    i = cell->free_space/CELL_GRANULARITY;

    cell_unlock(cell);

    /* <- race could occur here. */

    free_cells_lock(i);
    cell_lock(cell);

    if (CELL_MANAGED(cell) || CELL_KEEP(cell)) {
        free_cells_unlock(i);
        goto tryagain;
    }

    if (i != (cell->free_space/CELL_GRANULARITY) ) {
        #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s: fragment moved from under us\n",
            __FUNCTION__);
        #endif
        free_cells_unlock(i);
        goto tryagain;
    }

    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: cell=0x%lx deleted from list\n",
        __FUNCTION__, (unsigned long)cell);
    #endif
    list_del(&(cell->list));
    cell->list.next = NULL;
    cell->list.prev = NULL;
    free_cells_unlock(i);

    cell->free_space += FRAGMENT_LENGTH(fragment);
    cell->allocations--;

    /* Merge free fragments */
    if (fragment->next != NULL && fragment->next->user
        == NULL) {
        fragment->next = fragment->next->next;
        cell->free_space += sizeof(struct fragment);
    }

    *fragment->user = NULL;
    fragment->user = NULL;
    atomic_dec(&accounting.fragments);
    page = cell_put(cell, returnpage);

    if (atomic_read(&accounting.fragments) <
        atomic_read(&accounting.pages)) {
        cell_global_compact();
    }
    return page;
}

static struct cell *cell_most_unused(void)
{
    struct cell *cell;
    unsigned int i;
    struct list_head *pos;

    for (i = CELL_CATEGORIES-1; i > 0; i--) {
        free_cells_lock(i);

        if (list_empty(&free_cells[i])) {
            free_cells_unlock(i);
            continue;
        }

        /* Find a cell with few allocations and less
           than 50% used */
        cell = NULL;
        list_for_each(pos, &free_cells[i]) {
            struct cell *cell_pos = list_entry(pos,
                struct cell,
                 list);
            if (cell_pos->allocations <= 2) {
                cell = cell_pos;
                break;
            }
        }

        if (cell == NULL) {
            #ifdef CELL_DEBUG
```

```
                printk(KERN_EMERG "%s: Going to the next 
                    category\n",
                    __FUNCTION__);
                #endif
                free_cells_unlock(i);
                continue;
            }

            cell_lock(cell);
            list_del(&(cell->list));
            free_cells_unlock(i);
            return cell;
    }
    #ifdef CELL_DEBUG
    printk(KERN_EMERG "%s: Returned NULL\n",
        __FUNCTION__);
    #endif
    return NULL;
}

/* Global compaction:
 * Reallocates all fragments within a inefficiently
     used cell,
 * and then free that cell. */
static void cell_global_compact(void)
{
    struct cell *cell, *unused_cell =
        cell_most_unused();
    struct fragment *fragment, *new_fragment;

    if (unused_cell == NULL)
        return;

    fragment = unused_cell->fragment;
    do {
        unsigned int fragment_length =
            FRAGMENT_LENGTH(fragment);
        if (fragment->user == NULL)
            continue;
        if (!bit_spin_trylock(FRAGMENT_LOCKBIT,
            (unsigned long *)fragment->user))
            continue;

        /* Actual reallocation of fragment */
        cell = cell_get(fragment_length);
        if (!cell) {
            *fragment->user = fragment;
            break;
        }

        new_fragment = cell_fragment(cell,
            fragment_length);
        if (new_fragment == NULL) {
            cell = cell_compact(cell);
            new_fragment = cell_fragment(cell,
                fragment_length);
        }

        if (new_fragment == NULL) {
            cell_debug(cell);
            BUG();
        }

        memcpy(new_fragment->data, fragment->data,
            fragment_length);
        cell->allocations++;
        new_fragment->user = fragment->user;
        *new_fragment->user = new_fragment; /* resets
            the bit */
        fragment->user = NULL;
        unused_cell->allocations--;
        unused_cell->free_space += fragment_length;
        cell_put(cell, 0);
        cell = NULL;
    } while ((fragment = fragment->next));
    cell_put(unused_cell, 0);
}



/* How many pages are the cell structure consuming? */
int cell_accounting_pages(void)
{
    return atomic_read(&accounting.pages);
}

/* How many fragments do we have? */
int cell_accounting_fragments(void)
```

```
{
    return atomic_read(&accounting.fragments);
}

/* How many compacting operations should we do at the same
   time at the same cpu? */
void __init cell_setup(unsigned int compact_cells)
{
    int i;

    for (i = 0; i < CELL_CATEGORIES; i++) {
        mutex_init(&free_cells_list_lock[i]);
        INIT_LIST_HEAD(&free_cells[i]);
    }

    sema_init(&temporary_cells_semaphore,
        compact_cells);
    sema_init(&temporary_cells_lock, 1);
    INIT_LIST_HEAD(&temporary_cells);

    while (compact_cells--) {
        struct cell *cell = cell_allocate();
        BUG_ON(cell == NULL);
        #ifdef CELL_DEBUG
        printk(KERN_EMERG "%s: Putting 0x%lx into list
            temporary_cells"
             " list\n", __FUNCTION__, (unsigned
                long)cell);
        #endif
        list_add_tail(&(cell->list),
            &temporary_cells);
    }
}

/* EOF */
```

D.3 Page cache

Includes some chosen functions used to access the page cache.

D.3.1 *find_get_page()*

```
/**
 * find_get_page - find and get a page reference
 * @mapping: the address_space to search
 * @offset: the page index
 *
 * Is there a pagecache struct page at the given
    (mapping, offset) tuple?
 * If yes, increment its refcount and return it; if
    no, return NULL.
 */
struct page * find_get_page(struct address_space
   *mapping, unsigned long offset)
{
    struct page *page, *newpage;
    void **slot;

    read_lock_irq(&mapping->tree_lock);
    page = radix_tree_lookup(&mapping->page_tree,
        offset);
    if (!PageCompressed(page)) {
        if (page)
            page_cache_get(page);
        read_unlock_irq(&mapping->tree_lock);
        return page;
    }
    read_unlock_irq(&mapping->tree_lock);
    /* <- This is where we could have a race
        condition..
     * Race is made even more likly by putting an
         allocation of a page here,
     * but by doing this we can even sleep while
         allocating this page.
     */
    newpage = alloc_page(GFP_KERNEL);
    BUG_ON(newpage == NULL);
    lock_page(newpage);

    write_lock_irq(&mapping->tree_lock);
    slot = radix_tree_lookup_slot(&mapping->page_tree,
         offset);

    if (likely(slot != NULL)) {
        page = radix_tree_deref_slot(slot);
    } else {
        page = NULL;
    }

    if (PageCompressed(page)) {
        struct cpage *cpage;
        cpage = MaskPageCompressed(page);

        /* Could have been in cache if the cache was
           not disabled */
        if (cpage_marker(cpage)) {
            unlock_page(newpage);
            put_page(newpage);
            radix_tree_delete(&mapping->page_tree,
               offset);
            mapping->nrpages--;
            cc_accessed_cpage(cpage);
            write_unlock_irq(&mapping->tree_lock);
            cc_free_swap(cpage);
            return NULL;
        }

        get_page(newpage); /* cache also has a
           reference to it */

        /* replace compressed page with real page */
        cc_accessed_cpage(cpage);
        radix_tree_replace_slot(slot, newpage);
        write_unlock_irq(&mapping->tree_lock);

        /* uncompressing page */
        cc_restore(cpage, newpage);
        SetPageUptodate(newpage);
        unlock_page(newpage);
        cc_free(cpage);

        /* add to LRU to be removed in the future */
        lru_cache_add(newpage);

        return newpage;
    } else {
        unlock_page(newpage);
        put_page(newpage);
    }

    if (page)
        page_cache_get(page);
    write_unlock_irq(&mapping->tree_lock);
    return page;
}
```

D.3.2 *find_get_page_cc()*

```
/**
 * find_get_page_cc - find and get a page reference,
     or on cpage remove from mapping and return the
     cpage.
 * @mapping: the address_space to search
 * @offset: the page index
 *
 * Is there a pagecache struct page at the given
     (mapping, offset) tuple?
 * If yes, increment its refcount and return it; if
     no, return NULL.
 * Compressed pages are returned after beeing removed
     from the mapping.
 * All pages found here is part of the swapcache.
 * Only ran from free_swap_and_cache().
 */
struct page *find_get_page_cc(struct address_space
   *mapping,
    unsigned long offset)
{
    struct page *page;
    struct cpage *cpage = NULL;

    read_lock_irq(&mapping->tree_lock);
    page = radix_tree_lookup(&mapping->page_tree,
        offset);

    if (!PageCompressed(page)) {
        if (page)
            page_cache_get(page);
        read_unlock_irq(&mapping->tree_lock);
        return page;
    }

    BUG_ON(mapping != &swapper_space);
    read_unlock_irq(&mapping->tree_lock);
    /* <- this is where we could have a race condition
        */
    write_lock_irq(&mapping->tree_lock);
    page = radix_tree_lookup(&mapping->page_tree,
        offset);

    /* remove the never to be used compressed page */
    if (PageCompressed(page)) {
        cpage = MaskPageCompressed(page);
        mapping->nrpages--;
        cc_remove_lru(cpage);
        radix_tree_delete(&mapping->page_tree,
            offset);
    } else if (page)
        page_cache_get(page);
    write_unlock_irq(&mapping->tree_lock);
    return page;
}
```

D.3.3 *pageout_cache()*

```
/* pageout_cache is called by shrink_page_list() to
   store a page
 * into the compressed cache, it will also implicitly
    evict pages
 * from the cache to make room for the new page. */
static pageout_t pageout_cache(struct page *page,
    struct address_space *mapping, struct scan_control
        *sc)
{
    pageout_t retval = PAGE_CLEAN;
    unsigned long flags;
    void **slot = NULL;
    struct cpage *cpage, *oldmarker;

    if (!is_page_cache_freeable(page))
        return PAGE_KEEP;

    if (!mapping) {
        /*
```

```
         * Some data journaling orphaned pages can
           have
         * page->mapping == NULL while being dirty
           with clean buffers.
         */
        if (PagePrivate(page)) {
            if (try_to_free_buffers(page)) {
                ClearPageDirty(page);
                printk(KERN_EMERG "%s: orphaned 
                    page\n",
                     __FUNCTION__);
                return PAGE_CLEAN;
            }
        }
        return PAGE_KEEP;
    }

    /* Try to get rid of page->private, since we
       cannot store this in cc */
    if (PagePrivate(page) &&
        !try_to_release_page(page, sc->gfp_mask))
        return PAGE_ACTIVATE;

    BUG_ON(PagePrivate(page));
    BUG_ON(mapping->a_ops == NULL);

    if (mapping->a_ops->writepage == NULL)
        return PAGE_ACTIVATE;

    cpage = cc_store_page(page);
    /* not enough memory, retry later */
    if (cpage == NULL) {
        return PAGE_KEEP;
    }

    /* cache disabled or bad compression, will be
       swapped out normally */
    if (cpage == (void *)1) {
        return PAGE_CLEAN;
    }

    if (radix_tree_preload(GFP_ATOMIC))
        return PAGE_KEEP;

    write_lock_irqsave(&mapping->tree_lock, flags);
    /* If someone gets the page, before we end up here,
     * it is in use isn't it?
     * But how can this happen? It should have been unmapped earlier? hm,
     * yes unmapped from processes, but not from the mapping itself. */
    if (unlikely(page_count(page) != 2)) {
        write_unlock_irqrestore(&mapping->tree_lock, flags);
        radix_tree_preload_end();
        cc_free(cpage); /* not in cc lru */
        retval = PAGE_KEEP;
        return retval;
    }

    /* if page has been removed/changed from cache before it reached us?
     * because the lock is taken before getting the mapping, the page
     * can not have been removed from the cache. */
    slot = radix_tree_lookup_slot(&mapping->page_tree, page_index(page));
    if (radix_tree_deref_slot(slot) != page) {
        BUG();
        cc_free(cpage);
        cpage = NULL;
        retval = PAGE_KEEP;
        goto done;
    }

    /* If the page has been changed while/after we compressed it */
    if (0) {
        cc_free(cpage);
        retval = PAGE_ACTIVATE;
        goto done;
    }

    /* replacing page with cpage in corresponding cache */
    radix_tree_replace_slot(slot, CPagePage(cpage));
    oldmarker = cc_add_lru(cpage);

    /* This will make remove_mapping() behave, dirty
       bit
     * is stored in cpage and radix-tree. */
    SetPageCC(page);
    ClearPageDirty(page);
done:
    write_unlock_irqrestore(&mapping->tree_lock,
       flags);
    radix_tree_preload_end();
    if (oldmarker)
        cc_free_swap(oldmarker);
    return retval;
}
```

Appendix E

CD-ROM

E.1 General overview

The CD-ROM contains the source code, workloads and results. We keep README files in each directory to explain the contents. The contents is the results we already have, the source code, tests and this thesis.

E.2 Re-running the experiments

The prerequisites to rerunning the experiments are as follows:

Set up your favorite Linux distribution onto a separate partition, and make sure it makes use of a swap area. Make sure a version of GCC (GNUs compiler collection) able to compile Linux 2.6.22 is installed. Make a separate volume used for testing and mount it at /usr/src/test.

Copy all the tests from the CD-ROM to a new directory called /usr/src/tests, edit *clean-jXtest.sh* to format your volume. Copy the randomfil.txt to /root.

Download Linux 2.6.22 from www.kernel.org (or copy it from the CD-ROM) and apply the compressed caching patch (cc.patch) found on the CD-ROM or downloaded from `http://code.google.com/p/cclinux/downloads/list`. Just copy in your working *.config* and compile the kernel.

Set up your boot-loader to boot with the patched kernel. To limit the amount of ram use the *mem* kernel parameter, for our tests we used *mem=122880000*, *mem=102400000*, *mem=81920000*, *mem=61440000* and *mem=40960000*.

Boot into the kernel you want to run a test on, preferably in single user mode, and execute one of the following scripts: *jXtest.sh* (takes one parameter), *compile_and_sort.sh* and *sort.sh*.

The results you want to look at is output into *STDERR*.

E.3 Getting help

Contact Asbjorn Sannes (asbjorsa@ifi.uio.no) if you need any help to repeat the experiments or encounter bugs. Good luck!