The Compression Cache:

Virtual Memory Compression for Handheld Computers

Michael J. Freedman
Recitation: Rivest TR1

March 16, 2000

Abstract

Power consumption and speed are the largest costs for a virtual memory system in handheld computers. This paper describes a method of trading off computation and useable physical memory to reduce disk I/O. The design uses a *compression cache*, keeping some virtual memory pages in compressed form rather than sending them to the backing store. Efficiency is managed by a log-structured circular buffer, supporting dynamic memory partitioning, diskless operation, and disk spin-down.

Contents

1 Introduction

The handheld computer industry has witnessed significant growth in the past few years. Users have begun to use personal data assistants (PDAs) and other mobile computers in great numbers. The applications and features provided by these systems have expanded to match this interest. Newer models of the Palm or Windows CE PDAs provide increased storage capacity, applications such as spreadsheets, databases, and document viewers, and wireless communication to read email and news. Users desire even greater functionality: the ability to surf the Web, to communicate with audio and video, to play music with the advent of mp3 files, and to use speech-based interfaces. This list is far from all-inclusive, but many of these operations share the nature of being highly memory-intensive.

The greater demands placed on mobile computers are difficult to resolve in the face of technological trends. While the processor power and physical memory size of workstations have increased dramatically in the past decade, handheld computers have significantly less memory. The development of memory-intensive applications and faster processors for handheld systems only compounds this problem. Software designers are forced to create programs with smaller memory footprints, but the available physical memory still might not be sufficient. As with all modern computers, virtual memory and paging are necessary to support a variety of applications.

The difficulty in paging on handheld computers follows similar technological trends. While processor speed has increased greatly, I/O speed and battery life has not witnessed similar growth. Handheld computers may communicate over slower wireless networks, run diskless, or use smaller, slower local disks. An on-board battery is generally used for power consumption.

The dominant cost of a virtual memory system is backing stores to disk. VM performance on a mobile computer can be improved by decreasing traffic to and from the backing store. This problem is often handled by implementing page replacement policies that swap out pages in memory that will not be touched for the longest time. An optimal replacement policy cannot be performed on-line, therefore techniques such as the least-recently-used (LRU) heuristic and working set model only approximate optimal page selection based on previous behavior. These algorithms may perform poorly for specific referencing patterns. An obvious alternative to reduce dependence on the backing store is to increase the number of pages that can be stored in physical memory.

This paper describes a system that uses compression to increase the number of pages that can be stored in physical memory. A section of the physical memory that would normally be used directly by the application is used instead to hold pages in compressed form. This area is known as the *compression cache*. With more pages in physical memory, fewer page faults are issued by the virtual memory manager. Fewer slow, energy-intensive I/O accesses are necessary. An important consequence of decreased disk use is the associated savings in power consumption.

This remainder of this paper is organized as follows: section 2 details the design requirements and various options for a compressed virtual memory scheme; section 3 presents a detailed description of the selected compressed virtual memory design; and section 4 reports calculations and discusses the design choices made.

2 Design Criteria and Considerations

The virtual memory system for our handheld computer must support several criteria. The design must be able to compress pages in memory. It requires a method by which compressed pages may be referenced and extracted from compressed store. Furthermore, this process needs to be both relatively quick and should save considerable battery power as compared to disk accesses.

Compression algorithms result in variable-sized encoding. In a typical linked data structure, many words point to nearby objects, are nil, or contain small integers or zero. Thus, algorithms take advantage of this redundancy, or low information-theoretic entropy, to reduce the necessary encoding to represent a word. Using Lempel-Ziv coding, a block of memory is compressible to one-quarter of its size on average [1, 11].

Variable-sized compressed pages add complexity to virtual memory systems. Paging has been generally adopted as a simpler, more efficient storage technique, yet requires fixed-size storage. Several possible means for storing compressed pages are considered.

2.1 Compression to Segmented Memory

Segmentation allows variable-sized storage at any location in memory. As memory operations occur, however, the storage becomes "fragmented" as free chunks of memory appear between sections being used. At some stage, the memory manager needs to shuffle these free memory segments around – known as "burping the memory" – to compact the storage. This process to coalesce free memory into contiguous buffers is computationally dependent upon memory size. To reduce this complexity and improve speed, one can imagine partitioning memory into two, four, or even more chunks on which burping is performed independently. This partitioning, however, leads to the model of fixed-size paging.

2.2 Reversed Compression in Fixed-Size Pages

Compressed pages are stored within the system's fixed-size 8 KByte pages, and are removed from compressed storage when paged. The memory manager could locate available space for a compressed page and copy it to that free space. To fetch the compressed page, the normal hierarchical page table walk is used to located the compressed page. The data is uncompressed back to a 8 KByte page in uncompressed space, and the region is marked as "free." However, a system process is still needed to burp each fixed page to recover the fragmented space.

2.3 Non-Reversed Compression in Fixed-Sized Pages

With non-reversed compression, pages are only copied, not removed, when paged from the compressed store. This scheme can lead to several copies of data in memory. Memory management can prove more difficult, approaching something similar to garbage collection. A compressed page is only evicted if its corresponding uncompressed page is dirty, or the compressed page is no longer referenced. If an uncompressed page does not change, it can merely reference an existing compressed store to move into compressed space.

This design has a major problem associated with compressing new pages. The system would encounter difficulty when attempting to allocating new memory in which to store

compressed pages. Any given fixed-page in compressed space might be storing a mix of both current and outdated pages. The entire page could not therefore be thrown away, but this design seeks specifically to not require memory compacting. Deadlocks to evict pages are foreseeable, as both possible evictees could hold current compressed pages that could not be flushed. Processes could quickly run out of space for compressed storage; using a resizable RAM partition only postpones this problem.

Both techniques for compressing data into fixed-sized pages have management difficulties. Pages might not be compressible – a stream of completely random bits has no redundant information – and thus compression may yield full 8 KByte pages. To maintain the correctness of our pseudo-LRU algorithm, these pages would still be moved into the compressed store. However, there is no room on that fixed page to store any extra information such as the compressed page's back reference, information bits, or page offset. An external data structure not referenced by the page table would add complexity to the design.

2.4 Compression to Log-Structured Buffers

A log-structured circular buffer [9] maps physical pages into the kernel's virtual address space, one by one, eventually wrapping around to the start of the cache. Compressed pages are stored directly in the first free unit within the compression cache. Pages are reclaimed from the compression cache by evicting the oldest compressed page. Using this technique, compacting fragmented memory is not required as often, as the circular buffer mimics a FIFO queue. Data is placed near the top of the buffer, the oldest pages are removed from near the bottom. Log-structured systems are also well-suited for dynamically resizing available physical memory. The next section shall discuss in depth the implementation of a compression cache based on this design.

3 Design

This section details the design and implementation of a compressed virtual memory system.

3.1 Virtual Memory

An optimal system would be able to handle compressed pages with standard memory management techniques. When a page is compressed or uncompressed, the virtual memory manager can find or reclaim unused pages of memory. However, the compression algorithm yields compressed pages of variable size. Therefore, we cannot use a normal fixed-page technique to store compressed pages. A compression cache system is a more relevant approach to our requirements. This design requires some modifications and additions to the virtual memory system.

3.1.1 Basic Hierarchical Paging

Unlike previously implementations of compression caches [2], references to the location of compressed pages will be stored in the page table. For simplicity of access, this limits the number of page faults – page faults will all go to backing stores on disk if available – though increases the size of our page table.

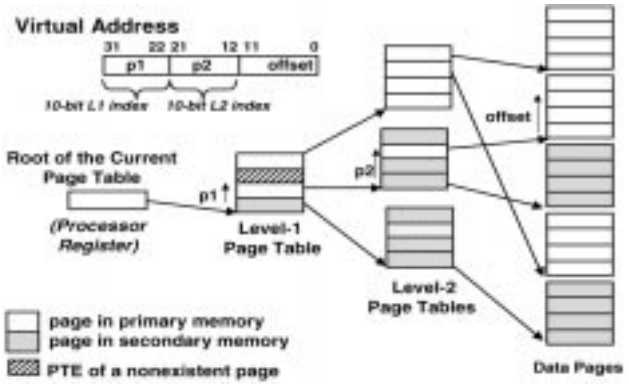

Figure 1: Hierarchical page table

A basic hierarchical page table is used to reference pages in memory. The process supplies a virtual address to the VM manager, the virtual address maps several layers of the table to the proper physical page address, and the supplied offset selects the specific address on the physical page. Figure 1 shows the process by which a virtual address is mapped to a physical page by "walking" the page table.[1]

This system attempts to use similar page recovery techniques for uncompressed and compressed stores. The page table includes references for both types of stores, mapping the entries to physical addresses via the normal means. Compressed stores, however, require some extra information and handling.

3.1.2 Modified Page Tables

The page table requires an additional word of data per entry to reference compressed pages in the circular cache. A `compressed` bit is set high if the page is compressed and removed from uncompressed memory. A `present` bit is set high when a page is moved to cache, and set low when the page is removed (which differs from being merely uncompressed) from the cache. The compressed page's location in memory should also be stored in this data word, similar to indexing in a regular cache or virtual memory system. The high-order bits can determine the page's slot in the cache descriptor table; low-order bits map the explicit offset within its physical block.

3.1.3 Cache Descriptor Table

A cache descriptor table is necessary for the page table to locate compressed pages within the cache. This table stores a mapping of slots in the compression cache to physical pages. It also maintains the current state – new, free, clean, dirty – of the physical block. This status block tells the memory manager which space it should assign in the compression cache, as well as whether a write-back is disk may be needed.

[1]Figure from Krste Asanovic. 6.823 Lecture 12 Notes.

3.1.4 Circular Compression Cache

The compression cache is a log-structured circular buffer in main memory. The handheld computer's operating system allocates a number of physical pages into the kernel's virtual address space for the compression cache. The choice of this allocated size will be discussed later in sections 3.3 and 4.2.1. These pages are used solely for the compression cache. When a compressed page does not fit within a physical page, the remaining bytes are carried through to be written at the top of the next physical page. Likewise, the compressed page at the top of the compression cache carries through to the physical page at the bottom of the buffer, explaining the "circular" description of the cache.[2] The memory manager inserts a small header before each page, describing the page, its compressed size, a back reference to the compressed page's virtual address, and a link to the next physical page in the cache.

The VMM requires two pointers into the cache for efficient writing. The `bottom` pointer references the first compressed page, the bottom of the cache. The `top` pointer references the first available address for writing.

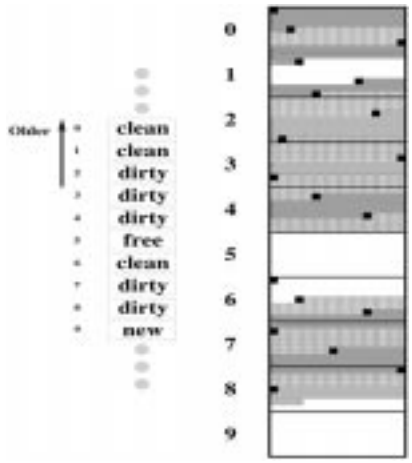

Figure 2: *Compression Cache:* Pages are compressed in the circular buffer, begun with header information, and described in a separate array that maps slots to physical page addresses and maintains current page state. The different patters refer to dirty, clean, and free compressed pages in memory.

3.2 Paging the Compressed Store

The virtual memory manager is charged with maintaining both uncompressed and compressed pages in main memory. This section describes the process by which pages are added to and recovered from the compressed store.

3.2.1 Storing Compressed Pages

Upon initialization of some process, allocated physical pages are filled with the proper uncompressed information. As usual, these physical addresses may be referenced through virtual addresses in the page table. The compressed and present bits are both set low.

When the process runs out of memory for uncompressed pages, the VMM begins to utilize the compression cache by using a page-replacement heuristic. The LRU heuristic is an efficient and practical technique for real systems. While virtual addresses to both compressed and uncompressed pages are present in the page table, the VMM will only consider uncompressed pages when determining the LRU page.

Once selected, the LRU physical page is evicted from uncompressed memory. The first available cache slot and its corresponding physical address are determined from the cache descriptor table. The page is compressed and written to the cache's free location. The page's compressed and present bits in the page table are set high, and the additional word in the

```
Store Page:
    If uncompressed space is not full,
        Store page in uncompressed memory.
    Else uncompressed space is full,
        Choose LRU page to compress.
        If cache is full,
            Select LRU element in cache.
            Flush it and possible write back to disk.
        If page already in cache,
            If page near top of cache,
                Do nothing.
            Else page near bottom of cache,
                Copy compressed page to top of cache.
        Else page not in cache,
            Compress page and store at top of cache.
```

page table entry is set to reflect this new slot:offset location.

The cache may run out of space to store compressed pages. If this occurs, the cache generally flushes the bottom page. We gain efficiency by checking the bottom of the cache descriptor table. If the page is clean, the buffer is merely flushed and no further action is required. If the compressed page is dirty, it is first uncompressed and then backup stored to disk as normal. The page's present bit is set low in its page table entry. As this flushed page corresponds to one of the first pages, the cache is a FIFO queue, which mimics the LRU heuristic.

Several compressed copies of the same page may exist within the circular cache at once. We allow this to simplify memory management. When variable-sized memory is normally freed from a system, areas of unused space start to accumulate throughout the system. To recover this fragmented memory, the system needs to "burp the memory" and coalesce the unused memory into sequential blocks. However, by not actually freeing the memory when pages are uncompressed, the cache handles memory deallocation only when it needs the space. The general practice is to evict the oldest compressed page from the bottom of the cache. For example, during memory-intensive operations like streaming audio or video, we do not desire the complexity of compacting memory. However, if the processor has extra cycles to spend, it may remove other compressed pages in the cache and burp the memory. This method, therefore, minimizes complexity and transparently implements a LRU page replacement policy.

Cache efficiency is improved by checking prior to a compression whether the page already exists within the cache, as suggested in [2]. If the page is `present` in the page table, its location is extracted. This is the key reason why the compressed location is stored as an additional word in the page table, as opposed to replacing the normal physical page address. If the compressed page is near the top of the cache, thus fairly recent, no action occurs. However, if it is present but near the bottom of the cache, the compressed page is copied to the top of the cache, and the page table is updated to reflect this new offset. This copying

```
FETCH PAGE:
    If page in physical memory,
        If page uncompressed,
            Return page via general paging means.
        Else page compressed,
            Look up page number in page table.
            Using number, look up cache offset in reference table.
            Extract compressed page from cache address.
            Uncompress page.
    Else page not in physical memory,
        Issue page fault and recover page from disk.
```

is necessary to maintain the LRU replacement nature and log-structure of our compression cache.

3.2.2 Recovering Compressed Pages

The compression cache and descriptor tables provide an easy means with which to recover compressed pages. When the process accesses a virtual memory address that has its compressed and present bits high, the location value is accessed in the page table entry. Clearly, this differs from the uncompressed case, in which the physical page address is directly used. Using the high-order bits of this location in the descriptor table, the fixed-page address of the compressed page is returned. Accessing this page, offset by the low-order bits, returns the proper compressed page. From the header information, we determine the compressed page's size. The page is uncompressed and moved into normal physical memory, remaining also in compressed form in the cache. The page's compressed bit in the page table is set low, but the present flag remains high to handle the "multiple copies in the cache" optimization just previously described. If no space in physical memory is available, the system chooses a new LRU page to compress.

3.3 Variable Memory Allocation

The optimal partitioning of RAM between uncompressed and compressed memory is highly application dependent. The characteristics of the memory requirements, such as size and reference pattern, differ among applications. If the program is large or accesses pages randomly, the cache will not significantly reduce traffic to the backing store. If the information-theoretic entropy of the average word is large, compression will not yield a significant decrease in size. For these cases, the cache should be smaller. A larger cache would improve performance if the opposite is true: the program is smaller, compression is more successful, and the process touches pages with strong spatial and temporal locality.

Implementations of a compression cache using a fixed-size RAM partition have found this technique only useful for applications that paged heavily even without the compression cache. These applications also fit within the compression cache without excessive traffic to

the backing store [3].

The log-structured circular buffer of the cache dynamically changes size depending on the current process. If the program performs in a way to minimize the effectiveness of the compression cache, the cache should free up its allocated physical pages so that they may be used for uncompressed stores. Likewise, if the program displays behavior that a larger number of pages are accessed often (i.e., a larger working set), then a larger compression cache improves performance. Therefore, instead of flushing the bottom compressed page when the cache runs out of space, the kernel can allocate another physical page for the compression cache.

The size of the compression cache also depends on the availability of an adequate backing store, as we shall now discuss.

3.4 Optimized Disk Accesses

Disk usage may be optimized to access several pages at once. Minimizing the number of disk accesses reduces power dissipation, which is relatively independent of the actual amount of data transferred up a few megabytes of data.

Using the standard virtual memory model, pages need to be fetched one-at-a-time from disk. When a page is not located within the physical memory in either uncompressed or compressed state, the VM manager issues a page fault exception that interrupts the process's execution. The page needs to be fetched from disk to the uncompressed memory, and execution resumes once the data is available.

However, the cache can write several pages to disk in one operation. Instead of removing only the bottom compressed page from the cache when space is required, several pages can be evicted from the cache at once. Only one larger disk write would therefore be necessary, instead of writing each page individually. Between these rarer backing stores and fetches, the disk can be spun down. Power is saved by allowing the disk to sleep, as discussed later in section 4.4.

3.5 Diskless Operation

Disk stores on a handheld computer are both slow and power-intensive. One of the integral reasons for implementing compressed paging to memory is to reduce disk accesses and this associated energy consumption. Measures estimate that writing a 8KBytes page to disk requires 10ms and consumes 2.3W of power [12].

Handheld computers might not have or use disks for a backing store. A diskless system is completely viable using our compression cache implementation. As a page fault would practically have infinite cost, all instruction and data code would need to reside in memory at all times. For a compressed virtual memory system, this means that a copy of every page must be kept in memory. There are two methods to ensure this functionality.

- **Dynamically-Resized Buffer:** The variable-sized compression buffer could grow in time and become quite large. A slight optimization can be made to normal operation by actively destroying outdated pages near the bottom of the compressed cache. If the VMM searches through a larger number of the least-recent compressed cache pages without freeing sufficient space for a new compressed page, another physical page is allocated for the cache's use.

- **Limited Virtual Memory:** The memory manager can limit the amount of virtual memory available to the process. However, limiting virtual memory does not necessarily work well with dynamically-resized compression caches. If the process of resizing partitions is taken to completion, all of the physical memory could be allocated for the compression cache and execution would halt. Therefore, a maximum virtual memory size should be computed that allows for a growing compressed store, while keeping sufficient uncompressed space.

A diskless implementation allows for an enlargement of the compressed memory upon demand, while placing an eventual size limit on the cache. The dynamic condition of the cache allows the memory manager to tailor the size accordingly to application-specific parameters. The important point to note is that the cache and its associated power dissipation are not over-utilized. An error signal is returned when the available memory reaches its configured limit. The user can then change usage patterns or choose instead to perform other operations.

4 Results and Discussion

This section considers the energy and storage efficiency of the compressed VM design.

4.1 Energy Efficiency

We shall detail the energy consumption of our system, and discuss the choices and results of our design.

4.1.1 Constant Clock Speed

The system requires processor cycles, and thus energy, in order to compress pages in physical memory. Our system takes 4 instructions on average to compress one byte of memory. The number of processor cycles requires to compress one 8 KByte page is be calculated:

$$8Kbytes * \frac{1024bytes}{KByte} * \frac{4instr}{byte} * \frac{1cycle}{instr} = 32768cycles$$

The processor can be clocked as fast as 200 MHz, but also may run at 100 MHz and 33 Mhz clock speeds. The peak power consumption of 400mW occurs when the processor runs at 200 MHz, and is directly proportional to the processor clock speed. The energy consumption and speed to compress one page of data at the different clock speeds is shown in figure 3.

Clock speed	Energy	Speed
200 MHz	$6.55 * 10^{-5}$ J	0.164 ms
100 Mhz	$6.55 * 10^{-5}$ J	0.328 ms
33 MHz	$6.55 * 10^{-5}$ J	0.993 ms

Figure 3: Compression statistics

The amount of energy dissipated when compressing a page remains constant across different clock speeds. For the greatest performance in terms of speed, the handheld computer should be clocked as its fastest clock speed of 200 MHz. Obviously, the 0.164 ms required to compress a page to memory is much quicker than the 10 ms seek and write time of a disk.

4.1.2 Disk Stores versus In-Memory Compression

Disk stores are significantly more energy intensive than compressing pages to physical memory. All pages are stored on disk in their uncompressed state, therefore written as fixed-size stores to disk. Pages are also read from disk when page faults occur in the virtual memory, at which point uncompressed pages are wanted. Pages stored in compressed form on disk would require extra handling for their variable-sized nature, and would still need to be uncompressed when brought back into memory. Therefore, all backing stores are performed on uncompressed pages.

Writing a block to disk consumes 2.3 W on average and takes 10 ms [12], taking into account rotational seek, rotational latency, and transmission time. This power usage accounts for an energy loss of 23 mJ for every separate disk access. As described, this number is relatively independent of the actual amount of data transferred.

Assuming an initial compression cache size of 16 MBytes [12], 134 mJ of energy is required to fill the entire cache with compressed pages. As previously calculated, only 0.065 mJ of energy is required to compress one page. If a page is uncompressed, another page generally needs to be compressed and moved to the cache. Assuming that uncompression requires a similar number of instructions per byte, 0.13 mJ of energy is used for the entire process of fetching a page from the compressed store after making available room.

Let us consider the energy efficiency of a 16 MByte RAM partition for a compressed store. If all 32 MBytes of the physical memory was uncompressed, we would experience a 1% miss as shown in figure 4. If we assume that accesses to uncompressed memory are "free" (i.e., consume insignificant amounts of energy in comparison to compressed and disk paging), the cost of this access may be calculated as follows:

$$\begin{aligned} Cost &= Cost_{Uncompressed} + MissPenalty_{Disk} \\ &= Cost_{Uncompressed} + MissRate * Cost_{Disk} \\ &= 0 + (0.01)(23mJ) \end{aligned}$$

Therefore, if we allocate the entire physical memory to the uncompressed store, the cost of touching one page in memory is 0.23 mJ.

If we assume a 16 MB compressed partition, the cost of paging may be calculated differently. The miss rate to disk is 0.0156%, given 80 MB of physical memory storage. The miss rate to compressed store is 4%.

$$\begin{aligned} Cost &= Cost_{Uncompressed} + MissPenalty_{Compressed} + MissPenalty_{Disk} \\ &= Cost_U + MissRate_C * Cost_C + MissRate_D * Cost_D \\ &= 0 + (0.04)(0.066mJ) + (0.000156)(23mJ) \end{aligned}$$

This cost of this paging is 0.0062 mJ, a large improvement resulting from the addition of a compressed store. In the next section we calculate an optimal RAM partitioning given these cost heuristics.

Both single pages and a block of evicted pages may be written to disk, given our disk-write optimization. Depending on the size of the process's working set in relation to the size of the available physical memory, backing stores may occur often or rare. The memory-intensive nature of our handheld computer, however, suggests the former might often be true.

Therefore, 23mJ backing stores deplete battery life much more than compressing pages to memory.

4.2 Memory Efficiency

This section describes the memory use of our virtual memory implementation.

4.2.1 Initial RAM Partitioning

The compression cache design uses a circular buffer that is dynamically resized depending on the behavior and referencing pattern of the process. However, an optimal initial RAM partition can be determined to help minimize energy consumption.

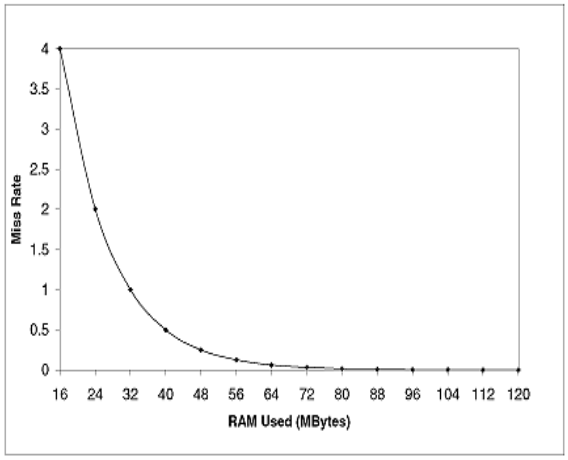

Figure 4: Miss rate for LRU replacement

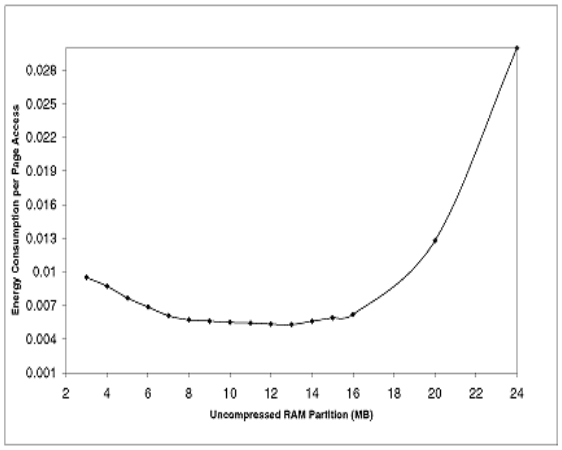

Figure 5: Energy consumption for various RAM partition configurations

The miss rate for the LRU replacement policy can be expressed as a function of the RAM used, shown in figure 4. These miss rates were used to calculate the cost of a single page fault given a fully uncompressed memory and a 16 MB compressed partition. Using the same cost equation as before, figure 5 shows energy consumption as a function of the RAM partition. Uncompressed partitions in the range of 8 to 16 MB appear sufficient; the function reaches an approximated minimum of 0.0053 mJ with a 13 MB partition.

We can conclude that a more optimal initial partitioning of the RAM would be 19 MB for the compressed store, and 13 MB for the uncompressed store. This obviously will change over the life of a process due to the dynamic resizing we support. The cost heuristic also does not account for the delay cost of a compressed access. A user might prefer speed over energy consumption, and desire a different initial partitioning.

4.2.2 Hierarchical Page Table Size

A hierarchical page table is better suited for our handheld system than simpler linear page tables. Assuming that the 32 MB physical memory is initially partitioned as we calculated, the in-memory virtual memory capacity is $13 + 4 * 19 = 89MB$. The disk capacity is a surprisingly large $3.2GB$. All user address spaces have their own page tables to the virtual memory. With both the memory-intensive nature of newer handheld applications and users running multiple applications at once, the total size of all page tables becomes excessive.

Linear page tables are ill-suited for the handheld virtual memory environment. While the virtual address space is very large, only a small fraction of pages are generally populated. Therefore, a sparse representation such as a hierarchical page table is better suited for our system.

Hierarchical tables decrease memory usage by keeping unused sublevels of the page table in the compression or backing store. While the root and L1 tables would explicitly remain in uncompressed memory, we may have to uncompress pages or issue page faults when walking the page table. This is an inherent trade-off of hierarchical page tables. The design is based on the premise that the additional available memory outweighs the miss penalty, which is minimized due to replacement policies based on locality of reference.

4.3 Prefetching Pages

Next-generation applications for handheld computers often share the nature of being highly memory-intensive. These features may include streaming audio and video, accessing large graphic files from the web, or performing speech and character recognition in the user interface. Most of these types of applications display very strong spatial locality. Many types of media files, streaming or otherwise, are fetched sequentially from I/O.

Prefetching may improve performance for our virtual memory system. The processor can speculate which pages will be fetched in the near pages, and use otherwise idle cycles to preemptively fetch these pages. Prefetching remains an area of continued research, but current techniques are well suited [8] for the type of applications we expect the handheld computer to run. Our virtual memory design and its log-structured compression cache should not preclude existing prefetching techniques.

4.4 Disk Spin Policy

Disks consume a significant portion of system power in mobile and handheld computers, shown to be more than 20-30% of total power [5]. Battery life may be extended by performing useful spin-up and spin-down operations to minimize unnecessary battery use. However, this technique has an underlying tradeoff between reduced idle power consumption by spinning the disk down after each use, and reduced interactivity by delaying response time during spin-up. Also, the spin-up process itself consumes energy.

Several different algorithms for spinning up and down a disk have been described in [4]. They are based on off-line optimal algorithms and on-line threshold-demand and predictive algorithms. Threshold-demand spin policies have been measured to reduce power consumption in mobile computers and disks by about 53%, given a threshold of 10 seconds.

Usage patterns may change over time, thus a fixed threshold for demand spin-up may be unsuited for different users. Techniques have been developed [5] for adaptive spin policy. The idle-time threshold for spinning down the disk is varied based on the user's tolerance for undesirable delays.

The handheld computer described in this paper would benefit from spinning down its disk when it is not being used. The handheld computer has been measured to consume 4.7 Watts during disk startup from sleep, which takes 20 ms. An idle disk consumes 0.95 W on average, while a spun-down disk consumes only 0.1 W. We are assuming that spinning-down a disk does not require any extra power. Thus, a spun-down disks saves 0.85 Joules per

second. Given an initial expenditure of 0.094 J (4.7 W * 20ms), spinning down a disk for periods even less than one second saves power.

Our system will incorporate adaptive threshold spin policy. In laptop computers, this policy consumes only slightly more energy while reducing undesirable delays.

Compressed stores compound the benefit of spinning down the disk. The disk will actually receive fewer accesses on average, as more pages can be stored directly in physical memory. The advantages in spinning down the disk in terms of reduced power dissipation are even improved with the nature of certain log-structure file system. Sprite LFS, for example, collects large amounts of new data in the file cache before writing to disk as a single large I/O [9]. Less demand is placed on the disk, thus it may remain in sleep mode for longer periods of time.

4.5 Technology Trends

The compression cache outlined in this paper seeks to reduce virtual memory backing stores, in order to reduce slow and energy-consuming I/O. Technological trends suggest that iterative refinement of hardware will not sufficient match application requirements. Processor speed has seen exponential growth in comparison to disk access time: processor power has doubled approximately every year, while disk seek rates double only every ten years [6]. While workstations use increasingly fast local-area networks, handheld and other mobile computers are mostly limited by slower wireless communications. The inherent propagation delay of these signals impede possible speed-up.

A compression cache should become even more beneficial with future technological developments. The performance of physical memory has improved faster than the backing store. Likewise, battery technology has not improved significantly in the past several years. On the other hand, memory has grown both cheaper and denser, allowing architects to place more physical memory in computers. With larger amounts of physical memory, our disk requirements decrease even further, improving both speed and energy consumption.

In-memory page compression is only one method to reduce power consumption. Chipmakers such as Intel and Transmeta are developing the StrongARM [7] and Crusoe [10] chips, respectively. These low-power chips run much cooler than traditional processor chips, and greatly extend battery life. They are specifically engineered for palm-size devices and other emerging portable computing applications.

5 Conclusion

A compression cache may be utilized to reduce traffic to the backing store by compressing lesser-used pages to physical memory. Another layer of memory hierarchy has been introduced: the most recent pages are stored in an uncompressed state, lesser used pages are kept in the compression cache, and rarely touched pages may be stored on disk. We estimate that compressing pages instead of performing a backing store improves both speed and energy efficiency by 1-2 orders of magnitude. The design is also compatible with other optimizations such as disk spin-down and prefetching. The compression cache is therefore well-suited as a virtual memory system for handheld computers.

References

[1] Andrew Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Operating Systems*, pages 96-107, April 1991.

[2] Doug Banks and Mark Stemm. Investigating virtual memory compression on portable architectures. January 19, 1995.

[3] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *USENIX Proceedings*, pages 519-529, January 1993.

[4] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the Power-Hungry Disk. In *USENIX Winter 1994 Techical Conference*, pages 293-306, January 1994.

[5] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *USENIX Symposium on Mobile and Location-Independent Computing*, April 1995.

[6] M. Frans Kaashoek. 6.033 Lecture 2 Notes, page 7.

[7] Intel StrongARM. http://developer.intel.com/design/strong/

[8] Dan Revel, Dylan McNamee, David Steere, and Jonathan Walpole. Adaptive prefetching for device-independent file I/O. 1997.

[9] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, 10(1): 26-52, Feburary 1992.

[10] Transmeta Crusoe. http://www.transmeta.com/crusoe/

[11] Ross N. Williams. An extremely fast ZIV-Lempel data compression algorithm. *Data Compression Conference*, pages 362-371, April 1991.

[12] 6.033 Design Project #1. http://web.mit.edu/6.033/www/handouts/h13-dp1.html