The Case for Compressed Caching in Virtual Memory Systems

Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis

Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78751-1182
{wilson|sfkaplan|smaragd}@cs.utexas.edu
http://www.cs.utexas.edu/users/oops/

Abstract

Compressed caching uses part of the available RAM to hold pages in compressed form, effectively adding a new level to the virtual memory hierarchy. This level attempts to bridge the huge performance gap between normal (uncompressed) RAM and disk.

Unfortunately, previous studies did not show a consistent benefit from the use of compressed virtual memory. In this study, we show that technology trends favor compressed virtual memory—it is attractive now, offering reduction of paging costs of several tens of percent, and it will be increasingly attractive as CPU speeds increase faster than disk speeds.

Two of the elements of our approach are innovative. First, we introduce novel compression algorithms suited to compressing in-memory data representations. These algorithms are competitive with more mature Ziv-Lempel compressors, and complement them. Second, we adaptively determine how much memory (if at all) should be compressed by keeping track of recent program behavior. This solves the problem of different programs, or phases within the same program, performing best for different amounts of compressed memory.

1 Introduction

For decades, CPU speeds have continued to double every 18 months to two years, but disk latencies have improved only very slowly. Disk latencies are five to six orders of magnitude greater than main memory access latencies, while other adjacent levels in the memory hierarchy typically differ by less than one order of magnitude. Programs that run entirely in RAM benefit from improvements in CPU speeds, but the runtime of programs that page is likely to be dominated by disk seeks, and may run many times more slowly than CPU-bound programs.

In [Wil90, Wil91b] we proposed compressed caching for virtual memory—storing pages in compressed form in a main memory *compression cache* to reduce disk paging. Appel also promoted this idea [AL91], and it was evaluated empirically by Douglis [Dou93] and by Russinovich and Cogswell [RC96]. Unfortunately Douglis's experiments with Sprite showed speedups for some programs, but no speedup or some slowdown for others. Russinovich and Cogswell's data for a mixed PC workload showed only a slight potential benefit. There is a widespread belief that compressed virtual memory is attractive only for machines without a fast local disk, such as diskless handheld computers or network computers, and laptops with slow disks. As we and Douglis pointed out, however, compressed virtual memory is more attractive as CPUs continue to get faster. This crucial point seems to have been generally overlooked, and no operating system designers have adopted compressed caching.

In this paper, we make a case for the value of compressed caching in modern systems. We aim to show that the discouraging results of former studies were primarily due to the use of machines that were quite slow by current standards. For current, fast, disk-based machines, compressed virtual memory offers substantial performance improvements, and its advantages only increase as processors get faster. We also study future trends in memory and disk bandwidths. As we show, compressed caching will be increasingly attractive, regardless of other OS improvements (like sophisticated prefetching policies, which reduce the average cost of disk seeks, and log-structured file systems, which reduce the cost of writes to disk).

We will also show that the use of better compression algorithms can provide a significant further improvement in the performance of compressed caching. Better Ziv-Lempel variants are now available, and we introduce here a new family of compression algorithms designed for in-memory data representations rather than file data.

The concrete points in our analysis come from simulations of programs covering a variety of memory requirements and locality characteristics. At this stage of our experiments, simulation was our chosen method of evaluation because it allowed us to easily try many ideas in a controlled environment. It should be noted that all our simulation parameters are either relatively conservative or perfectly realistic. For instance, we assume a quite fast disk in our experiments. At the same time, the costs of compressions and decompressions used in our simulations are the actual runtime costs for the exact pages whose compression or decompression is being simulated at any time.

The main value of our simulation results, however, is not in estimating the exact benefit of compressed caching (even though it is clearly substantial). Instead, we demonstrate that it is possible to detect reliably how much memory should be compressed during a phase of program execution. The result is a compressed virtual memory policy that adapts to program behavior. The exact amount of compressed memory crucially affects program performance: compressing too much memory when it is not needed can be detrimental, as is compressing too little memory when slightly more would prevent many memory faults. Unlike any fixed fraction of compressed memory, our adaptive compressed caching scheme yields uniformly high benefits for all test programs and a wide range of memory sizes.

2 Compression Algorithms

In [WLM91] we explained how a compressor with a knowledge of a programming language implementation could exploit that knowledge to achieve high compression ratios for data used by programs. In particular, we explained how pointer data contain very little information on average, and that pointers can often be compressed down to a single bit.

Here we describe algorithms that make much weaker assumptions, primarily exploiting data regularities imposed by hardware architectures and common programming and language-implementation strategies. These algorithms are fast and fairly *symmetrical*—compression is not much slower than decompression. This makes them especially suitable for compressed virtual memory applications, where pages are compressed about as often as they're decompressed. [1]

As we explain below, the results for these algorithms are quite encouraging. A straightforward implementation in C is competitive with the best assembly-coded Ziv-Lempel compressor we could find, and superior to the LZRW1 algorithm (written in C by Ross Williams)[Wil91a] used in previous studies of compressed virtual memory and compressed file caching.

As we will explain, we believe that our results are significant not only because our algorithms are competitive and often superior to advanced Ziv-Lempel algorithms, but because they are *different*. Despite their immaturity, they work well, and they complement other techniques. They also suggest areas for research into significantly more effective algorithms for in-memory data.

(Our algorithms are also interesting in that they could be implemented in a very small amount of hardware, including only a tiny amount of space for dictionaries, providing extraordinarily fast and cheap compression with a small amount of hardware support.)

2.1 Background: Compression

To understand our algorithms and their relationship to other algorithms, it is necessary to understand a few basic ideas about data compression. (We will focus on lossless compression, which allows exact reconstruction of the original data, because lossy compression would generally be a disaster for compressed VM.)

All data compression algorithms are in a deep sense *ad hoc*—they must exploit *expected regularities* in data to achieve any compression at all. All compression algorithms embody expectations about the kinds of regularities that will be encountered in the data being compressed. Depending on the kind of data being compressed, the expectations may be appropriate or inappropriate and compression may work better or worse. The main key to good compression is having the right kinds of expectations for the data at hand.

Compression can be thought of as consisting of two phases, which are typically interleaved in practice: *modeling* and *encoding* [BCW90, Nel95]. Modeling is the process of detecting regularities that allow a more concise representation of the information. Encoding is the construction of that more concise representation.

Ziv-Lempel compression. Most compression algorithms, including the overwhelmingly popular Ziv-Lempel family, are based on detection of *exact* repetitions of *strings* of atomic tokens. The token size is usually one byte, for speed reasons and because much data

[1] A variant of one of our algorithms has been used successfully for several years in the virtual memory system of the Apple Newton, a personal digital assistant with no disk [SW91] (Walter Smith, personal communication 1994, 1997). While we have not previously published this algorithm, we sketched it for Smith and he used it in the Newton, with good results—it achieved slightly less compression than a Ziv-Lempel algorithm Apple had used previously, but was much faster. Unfortunately, we do not have any detailed performance comparisons.

is in some sense byte-oriented (e.g., characters in a text file) or multiple-byte oriented (e.g., some kinds of image data, Intel Architecture machine code, unicode).

A Ziv-Lempel compressor models by reading through the input data token by token, constructing a dictionary of observed sequences, and looking for repetitions as it goes. It encodes by writing strings to its output the first time they are observed, but writing special codes when a repetition is encountered (e.g., the number of the dictionary entry). The output thus consists of appropriately labeled "new" data and references to "old" data (repetitions).

The corresponding LZ decompressor reads through this data much like an interpreter, reconstructing the dictionary created during compression. When it sees a new string, it adds it to the dictionary just as the compressor did, as well as sending it to its (uncompressed) output. When it sees a code for a repetition of a dictionary item, it copies that item to its output. In this way, its dictionary always matches the dictionary that the compressor had at the same point in the data stream, and its output replicates the original input by expanding the repetition codes into the strings they represent.

The main assumption embodied by this kind of compressor is that literal repetitions of multi-token strings will occur in the input—e.g., you'll often see several bytes in a row that are exactly the same bytes in the same order as something you saw before. This is a natural assumption in text, and reasonable in some other kinds of data, but often wrong for in-memory data.

2.2 In-Memory Data Representations

It is commonly thought that LZ-style compression is "general purpose," and that in-memory data are fairly arbitrary—different programs operate on different kinds of data in different ways, so there's not much hope for a better algorithm than LZ for compressing in-memory data. The first assumption is basically false,[2] and the second is hasty, so the conclusion is dubious.

While different programs do different things, there are some common regularities, which is all a compression algorithm needs to work well on average. Rather than consisting of byte strings, the data in memory are often best viewed as records and data structures—the overall array of memory words is typically used to store records, whose fields are mostly one or two words. Note that fields of records are usually *word-aligned* and that the data in those words are frequently numbers or pointers. Pointers can be usefully viewed *as* numbers—they are integer indices into the array of memory itself.

Integer and pointer data often have certain strong regularities. Integer values are usually numerically small (so that only their low-order bytes have significant information content), or else similar to other integers very nearby in memory.

Likewise, pointers are likely to point to other objects nearby in memory, or be similar to other nearby pointers—that is, they may point to another area of memory, but other pointers nearby may point to the same area. These regularities are quite common and strong. One reason is that heap data are often well-clustered; common memory allocators tend to allocate mostly within a small area of memory most of the time; data structures constructed during a particular phase of program execution are often well-clustered and consist of one or a few types of similar objects [WJNB95].

Other kinds of data often show similar regularities. Examples include the hidden headers many allocators put on heap objects, virtual function table pointers in C++ objects, booleans, etc.

These regularities are strong largely because in-memory data representations are designed primarily for speed, not space, and because real programs do not usually use random data or do random things with data. (Even randomized data can be very regular in this way; consider an array of random integers less than 1000—all of them will have zeroes in their upper 22 bits.)

2.3 Exploiting In-Memory Data Regularities

Our goal in this section is to convey the basic flavor of our algorithms (which we call WK algorithms); the actual code is available from our web site and is well-commented for those who wish to explore it or experiment with it.

We note that these algorithms were designed several years ago, when CPU's were much slower than today—they therefore stress simplicity and speed over achieving high compression. We believe that better algorithms can be designed by refining the basic modeling technique, perhaps in combination with more traditional sequence-oriented modeling, and by using more sophisticated encoding strategies. Given their simplicity, however, they are strikingly effective in our experiments.

[2] It is worth stressing this again, because there is widespread confusion about the "optimality" of some compression algorithms. In general, an *encoding* scheme (such as Huffman coding or arithmetic coding) can be provably optimal within some small factor, but a compressor cannot, unless the regularities in the data are known in advance and in detail. Sometimes compression algorithms are proven optimal based on the simplifying assumption that the source is a stochastic (randomized, typically Markov) source, but real data sources in programs are generally *not* stochastic[WJNB95], so the proof does not hold for real data.

Our compression algorithms exploit in-memory data regularities by scanning through the input data a 32-bit *word* at a time, and looking for data that are *numerically* similar—specifically, repetitions of the *high-order* 22-bit pattern of a word, even if the low-order 10 bits are different.[3] They therefore perform *partial* matching of whole-word bit patterns.

To detect repetitions, the encoder maintains a dictionary of just 16 *recently-seen words*. (One of our algorithms manages this dictionary as a direct mapped cache, and another as a 4x4 set-associative cache, with LRU used as the replacement algorithm for each set. These are simple software caching schemes, and could be trivially implemented in very fast hardware. Due to lack of space and because the exact algorithm did not matter for compressed caching performance, we will only discuss the direct-mapped algorithm in this paper.)

For these compression algorithms to work as well as they do, the regularities must be very strong. Where a typical LZ-style compressor uses a dictionary of many kilobytes (e.g., 64 KB), our compressors use only 64 *bytes* and achieve similar compression ratios for in-memory data.

The compressor scans through a page, reading each word, probing its cache (dictionary) for a matching pattern, and emitting a two-bit code classifying the word. A word may

- not match a dictionary entry, or
- match only in the upper 22 bits, or
- match a whole 32-bit pattern.

As a special case, we check first to see if the word is all zeroes, i.e., matches a full-word zero, in which case we use the fourth two-bit pattern.

For the all-zeroes case, only the two-bit tag is written to the compressed output page. For the other three cases, additional information must be emitted as well. In the no-match case, the entire 32-bit pattern that did not match anything is written to the output. For a full (32-bit) match, the dictionary index is written, indicating which dictionary word was repeated. For the partial (22-bit) match case, the dictionary index and the (differing) low 10 bits are written.

The corresponding decompressor reads through the compressed output, examining one two-bit tag at a time and taking the appropriate action. As with more conventional compression schemes, a tag indicating no-match directs it to read an item (one word) from the compressed input, insert it in the dictionary, and echo it to the output. A tag indicating all-zeroes directs it to write a word of zeroes to its output. A tag indicating a full-word match directs it to copy a dictionary item to the output, either whole (in the full match case) or with its low bits replaced by bits consumed from the input (for a partial match).

The encoding can then be performed quickly. Rather than actually writing the result of compressing a word directly to the output, the algorithm writes each kind of information into a different intermediate array as it reads through the input data, and then a separate postprocessing pass "packs" that information into the output page, using a fast packing routine. (The output page is segmented, with each segment containing one kind of data: tags, dictionary indices, low bits, and full words.) For example, the two-bit tags are actually written as bytes into a byte array, and a special routine packs four consecutive words (holding 16 tags) into a single word of output by shifting and XORing them together. During decompression, a prepass unpacks these segments before the main pass reconstructs the original data.

3 Adaptively Adjusting the Compression Cache Size

To perform well, a compressed caching system should adapt to the working set sizes of the programs it caches for. If a program's working set fits comfortably in RAM, few pages (or no pages) should be kept compressed, so that the overwhelming majority of pages can be kept in uncompressed form and accessed with no penalty. If a program's working set is larger than the available RAM, and compressing pages would allow it to be kept in RAM, more pages should be compressed until the working set is "captured". In this case, the reduction in disk faults may greatly outweigh the increase in compression cache accesses, because disk faults are many times more expensive than compression cache faults.

Douglis observed in his experiments that different programs needed compressed caches of different sizes. He implemented an adaptive cache-sizing scheme, which varied the split between uncompressed and compressed RAM dynamically. Even with this adaptive caching system, however, his results were inconsistent; some programs ran faster, but others ran slower. We believe that Douglis's adaptive caching strategy may have been partly at fault. Douglis used a fairly simple scheme in which the two caches competed for RAM on the basis of how

[3]The 22/10 split was arrived at experimentally, using an early data set that partially overlaps the one used in this study. The effectiveness of the algorithm is not very sensitive to this parameter, however, and varying the split by 2 bits does not seem to make much difference—using more high bits means that matches are encoded more compactly, but somewhat fewer things match.

recently their pages were accessed, rather like a normal global replacement policy arbitrating between the needs of multiple processes, keeping the most recently-touched pages in RAM. Given that the uncompressed cache *always* holds more recently-touched pages than the compressed cache, this scheme requires a bias to ensure that the compressed cache has any memory at all. We believe that this biased recency-based caching can be maladaptive, and that a robust adaptive cache-sizing policy *cannot* be based solely on the LRU ordering of pages *within* the caches.

3.1 Online Cost/Benefit Analysis

Our own adaptive cache-sizing mechanism addresses the issue of adaptation by performing an online cost/benefit analysis, based on recent program behavior statistics. Assuming that behavior in the relatively near future will resemble behavior in the relatively recent past, our mechanism actually keeps track of aspects of program behavior that bear directly on the performance of compressed caching for different cache sizes, and compresses more or fewer pages to improve performance.

This system uses the kind of recency information kept by normal replacement policies, i.e., it maintains an approximate ordering of the pages by how recently they have been touched. Our system extends this by retaining the same information for pages which have been recently evicted. This information is discarded by most replacement policies, but can be retained and used to tell *how well* a replacement policy is working, compared to what a *different* replacement policy would do.

We therefore maintain an LRU (or *recency*) ordering of the pages in memory *and* a comparable number of recently-evicted pages. This ordering is not used primarily to model what *is* in the cache, but rather to model *what the program is doing*.

A Simplified Example. To understand how our system works, consider a very simple version which manages a pool of 100 page frames, and only chooses between two compressed cache sizes: 50 frames, and 0 frames. With a compression cache of 50 frames and a compression ratio of 2:1, we can hold the 50 most-recently-accessed pages in uncompressed form in the uncompressed cache, and the next 100 in compressed form. This effectively increases the size of our memory by 50% in terms of its effect on the disk fault rate.

The task of our adaptation mechanism is to decide whether doing this is preferable to keeping 100 pages in uncompressed form and zero in compressed form. (We generally assume that pages are compressed before being evicted to disk, whether or not the compression cache is of significant size. Our experiments show that this cost is very small.)

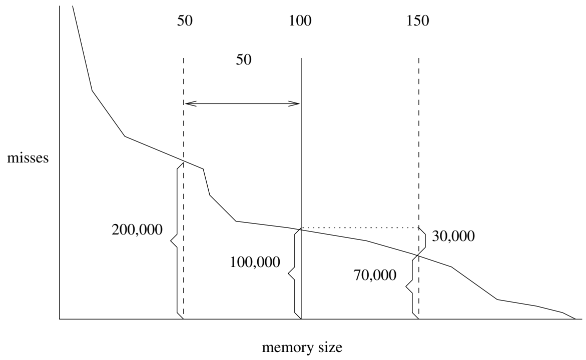

Figure 1: Cost/benefit computation using the miss-rate histogram.

Figure 1 shows an example miss rate histogram decorated with some significant data points. (This is not real data, and not to scale because the actual curve is typically *very* high on the far left, but the data points chosen are reasonable).

The benefit of this 50/50 configuration is the reduction in disk faults in going from a memory of size 100 to a memory of size 150. We can measure this benefit simply by counting the number of times we fault on pages that are between the 101st and 150th positions in the LRU ordering (30,000 in Figure 1), and multiplying that count by the cost of disk service.

The cost of this 50/50 configuration is the cost of compressing and decompressing all pages outside the uncompressed cache region. These are exactly the touches to the pages beyond the 51st position in the LRU ordering (200,000 touches). Thus, in the example of Figure 1, compressed caching is beneficial if compressing and decompressing 200,000 pages is faster than fetching 30,000 pages from disk.

In general, our recency information allows us to estimate the cost and benefit of a compression cache of a given size, regardless of what the current size of the compression cache actually is, and which pages are currently in memory. That is, we can do a “what if analysis” to find out if the current split of memory between caches is a good one, and what might be a better one. We can simply count the number of touches to pages in different regions of the LRU ordering, and interpret those as hits or misses relative to different sizes of uncompressed cache and corresponding sizes of compressed cache and overall effective memory size.

Multiple target sizes. We generalize the above scheme by using several different "target" compression cache sizes, interpreting touches to different ranges of the LRU ordering appropriately for each size. The adaptive component of our system computes the costs and benefits of each of the target sizes, based on recent counts of touches to regions of the LRU ordering, and chooses the target size with the lowest cost. Then the compressed cache size is adjusted in a *demand-driven* way: memory is compressed or uncompressed only when an access to a compressed page (either in compressed RAM or on disk) occurs.

Actually, our system chooses a target *uncompressed* cache size, and the corresponding overall effective cache size is computed based on the number of page frames left for the compressed cache, multiplied by an estimate of the compression ratio for recently compressed pages. This means that the statistics kept by our adaptivity mechanism are not exact (our past information may contain hits that are in a different recency region than that indicated by the current compressibility estimate). Nevertheless, this does not seem to matter much in our simulations; approximate statistics about which pages have been touched how recently are quite sufficient. This indicates that our system will not be sensitive to the details of the replacement policy used for the uncompressed cache; any normal LRU approximation should work fine. (E.g., a clock algorithm using reference bits, a FIFO-LRU segmented queue using kernel page traps, or a RANDOM-LRU segmented queue using TLB miss handlers.)

The overheads of updating the statistics, performing the cost/benefit analyses, and adaptively choosing a target split are low—just a few hundred instructions per uncompressed cache miss, if the LRU list is implemented as a tree with an auxiliary table (a hash table or sparse page-table like structure).

3.2 Adapting to Recent Behavior

To adapt to recent program behavior our statistics are decayed exponentially with time. Time, however, is defined as the number of *interesting events* elapsed. Events that our system considers "interesting" are page touches that could affect our cost benefit analysis (i.e., would have been hits if we had compressed as much memory as any of our target compression sizes currently suggests). Defining time this way has the benefit that touches to very recently used pages are ignored, thus filtering out high-frequency events.

Additionally, the decay factor used is inversely proportional to the size of memory (total number of page frames), so that time typically advances more slowly for larger memories than for small ones—small memories usually have a shorter replacement cycle, and need to decay their statistics at a faster rate than larger ones.

If the decay rate were not inversely proportional to the memory size, time would advance inappropriately slowly for small memories and inappropriately quickly for large ones. The small cache would wait too long to respond to changes, and the large one would twitchily "jump" at brief changes in program behavior, which are likely not to persist long enough to be worth adapting toward.

Extensive simulation results show that this strategy works as intended: our adaptivity ensures that for any memory size, the cache responds to changes in the recent behavior of a program relatively quickly, so that it can benefit from relatively persistent program behavior, but not so quickly that it is continually "distracted" by short duration behaviors.

A single setting of the decay factor (relativized automatically to the memory size) works well across a variety of programs, and across a wide ranges of memory sizes.

4 Detailed Simulations

In this section, we describe the methodology and results of detailed simulations of compressed caching. We captured page image traces, recording the pages touched *and their contents*, for six varied UNIX programs, and used these to simulate compressed caching in detail.

(The code for our applications, tracing and filtering tools, and compressors and simulator are all available from our web site for detailed study and further research.)

Note that our traces do not contain references to executable code pages. We focus on data pages, because our main interest is in compressing in-memory data. As we will explain in Section 5, compressing code equally well is an extra complication but can certainly be done. Several techniques complementary to ours have been proposed for compressing code and the data from [RC96] indicate that references to code pages exhibit the same locality properties as references to data pages.

4.1 Methodology

Test suite. For these simulations, we traced six programs on an Intel x86 architecture under the Linux operating system with a page size of 4KB (we will study the effect of larger page sizes in Section 4.3). The behavior of most of these programs is described in more detail in [WJNB95]. Here is a brief description of each:

- **gnuplot**: A plotting program with a large input producing a scatter plot.

- **rscheme**: A bytecode-based interpreter for a garbage-collected language. Its performance is dominated by the runtime of a generational garbage collector.
- **espresso**: A circuit simulator.
- **gcc**: The component of the GNU C compiler that actually performs C compilation.
- **ghostscript**: A PostScript formatting engine.
- **p2c**: A Pascal to C translator.

These programs constitute a good test selection for locality experiments (as we try to test the adaptivity of our compressed caching policy relative to locality patterns at various memory sizes). Their data footprints vary widely: gnuplot and rscheme are large programs (with over 14,000 and 2,000 pages, respectively), gcc and ghostscript are medium-sized (around 550 pages), while espresso and p2c are small (around 100 pages).

We used the following three processors:

1. **Pentium Pro at 180 MHz**: This processor approximately represents an average desktop computer at this time. Compressed caching is not only for fast machines.
2. **UltraSPARC-10 300 Mhz**: While one of the fastest processors available now, it will be an average processor two years from now. Compressed caching works even better on a faster processor.
3. **UltraSPARC-2 168 MHz**: A slower SPARC machine which provides an interesting comparison to the Pentium Pro, due to its different architecture (e.g., faster memory subsystem).

We used three different compression algorithms in our experiments:

1. **WKdm**: A recency based compressor that operates on machine words and uses a direct-mapped, 16 word dictionary and a fast encoding implementation.
2. **LZO**: Specifically, **LZO1F**, is a carefully coded Lempel-Ziv implementation designed to be fast, particularly on decompression tasks. It is well suited to compressing small blocks of data, using small codes when the dictionary is small. While all compressors we study are written in C, this one also has a speed-optimized implementation (in Intel x86 assembly) for the Pentium Pro.
3. **LZRW1**: Another fast Lempel-Ziv implementation. This algorithm was used by Douglis in [Dou93]. While it does not perform as well as LZO, we wanted to demonstrate that even this algorithm would allow for an effective compressed cache on today's hardware.

The runtimes of the test suite. Our results are presented in terms of time spent paging, but it is helpful to know the processing time required to execute each program in the test suite. Figure 2 shows the time required to execute each of our six programs on each of the three processors, when no paging occurs. These times can be added with paging time information to obtain total turnaround time for a given architecture, memory size, and virtual memory configuration.

Program name	**P-Pro 180MHz**	**SPARC 168MHz**	**SPARC 300MHz**
gnuplot	46.89	32.99	20.61
rscheme	8.26	11.77	7.59
espresso	10.07	12.35	7.41
gcc	9.89	14.66	9.41
ghostscript	18.95	26.89	16.84
p2c	2.38	2.91	2.08

Figure 2: The processing times for each program in the test suite on each processor used in this study. If enough memory is available such that no paging occurs, these times will be the turnaround times.

A brief note on compressor performance. All of our compression algorithms achieve roughly a factor of two in compression on average for all six programs. All can compress and decompress a page in well under half a millisecond on all processors. The WKdm algorithm is the fastest, compressing a page in about 0.25 milliseconds and decompressing in about 0.15 milliseconds on the Pentium Pro, faster on the SPARC 168 MHz, and over twice as fast on the SPARC 300 MHz. (This is over 20 MB compressed *and* uncompressed per second, about the bandwidth of a quite fast disk.) LZO is about 20% slower, and LZRW1 about 20% slower still.

Tracing. Our simulator takes as input a trace of the pages a program touches, augmented with information about the compressibility and cost of compression of each touched page for a particular compression algorithm. To create such a trace and keep the trace size manageable, we used several steps and several tracing and filtering tools.

We traced each program using the portable tracing tool *VMTrace* [WKB]. We added a module to VMTrace that made it emit a complete copy of each page as it was referenced. We refer to such traces as *page image traces*.

Creating compression traces. To record the actual effectiveness and time cost of compressing each page image, we created a set of *compression traces*. For each combination of compression algorithm and CPU, we created a trace recording how expensive and how effective compression is for each page image in the reduced page image trace. Since we have 6 test programs, 3 compression algorithms, and 3 CPU's, this resulted in 54 compression traces.

The tool that creates compression traces is linked with a compressor and decompressor, and consumes a (reduced) page image trace. For each trace record in the page image trace, it compresses and decompresses the page image and outputs a trace record. This record contains the page number, the times for compressing and decompressing the page's contents at that moment, and the resulting compressed size of the page. Each page image is compressed and decompressed several times, and the median times are reported. Timing is very precise, using the Solaris high-resolution timer (all of our compression timings were done under the Solaris operating system). To avoid favorable (hardware) caching effects, the caches are filled with unrelated data before each compression or uncompression. (This is conservative, in that burstiness of page faults will usually mean that some of the relevant memory is still cached in the second-level cache in a real system.)

Simulation parameters. We used four different target compression sizes with values equal to 10%, 23%, 37%, and 50% of the simulated memory size. Thus, during persistent phases of program behavior (i.e., when the system has enough time to adapt) either none, or 10%, or 23%, or 37%, or 50% of our memory pages are holding compressed data. Limiting the number of target compression sizes to four guarantees that our cost/benefit analysis incurs a low overhead. The decay factor used is such that the M-th most recent event (with M being the size of memory) has a weight equal to 20% of the most recent event. Our results were not particularly sensitive to the exact value of the decay factor.

Estimates used. During simulation we had to estimate the costs for reading a page from disk or writing it to disk. We conservatively assumed that writing "dirty" pages to disk incurs no cost at all, to compensate for file systems that keep low the cost of multiple writes (e.g., log-structured file systems). Additionally, we assumed a disk with a uniform seek time of 5ms. Admittedly, a more complex model of disk access could yield more accurate results, but this should not affect the validity of our simulations (a 5ms seek time disk is fast by modern standards). In Section 4.3 we examine the effect of using a faster disk (up to a seek time of 0.625ms).

4.2 Results of Detailed Simulations

4.2.1 Wide Range Results

For each of our test programs, we chose a wide range of memory sizes to simulate. The plots of this section show the entire simulated range for each program. Subsequent sections, however, concentrate on the *interesting* region of memory sizes. This range usually begins around the size where a program spends 90% of its time paging and 10% of its time executing on the CPU, and ends at a size where the program causes very little paging.

Figure 3 shows log-scale plots of the paging time of each of our programs as a function of the memory size. Each line in the plot represents the results of simulating a compressed cache using a particular algorithm on our SPARC 168 MHz machine. The paging time of a regular LRU memory system (i.e., with no compression) is shown for a comparison. As can be seen, compressed caching yields benefits for a very wide range of memory sizes, indicating that our adaptivity mechanism reliably detects locality patterns of different sizes. Note that all compression algorithms exhibit benefits, even though there are definite differences in their performance.

Figure 3 only aims at conveying the general idea of the outcome of our experiments. The same results are analyzed in detail in subsequent sections (where we isolate interesting memory regions, algorithms, architectures, and trends).

4.2.2 Normalized Benefits and the Effect of Compression Algorithms

Our first goal is to quantify the benefits obtained by using compressed caching and to identify the effect of different compression algorithms on the overall system performance. It is hard to see this effect in Figure 3, which seems to indicate that all compression algorithms obtain similar results.

A more detailed plot reveals significant variations between algorithm performance. Figure 4 plots the normalized paging times for different algorithms in the interesting region. (Recall that this usually begins at the size where a program spends 90% of its time paging and 10% of its time executing on the CPU, and ends at a size where the program causes very little paging). By "normalized paging time" we mean the ratio of paging time for compressed caching over the paging time for a regular LRU

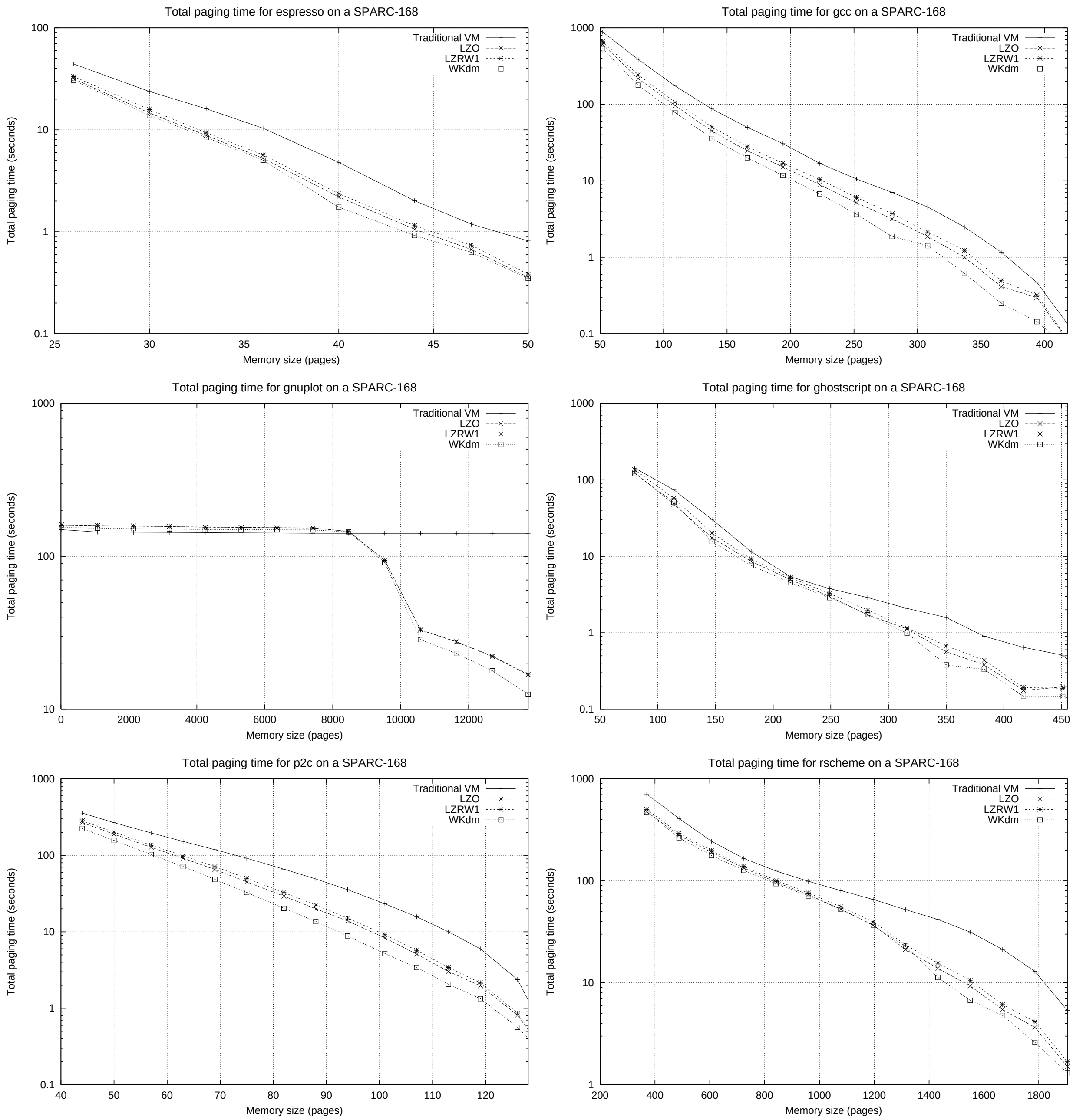

Figure 3: Compressed caching yields consistent benefits across a wide range of memory sizes.

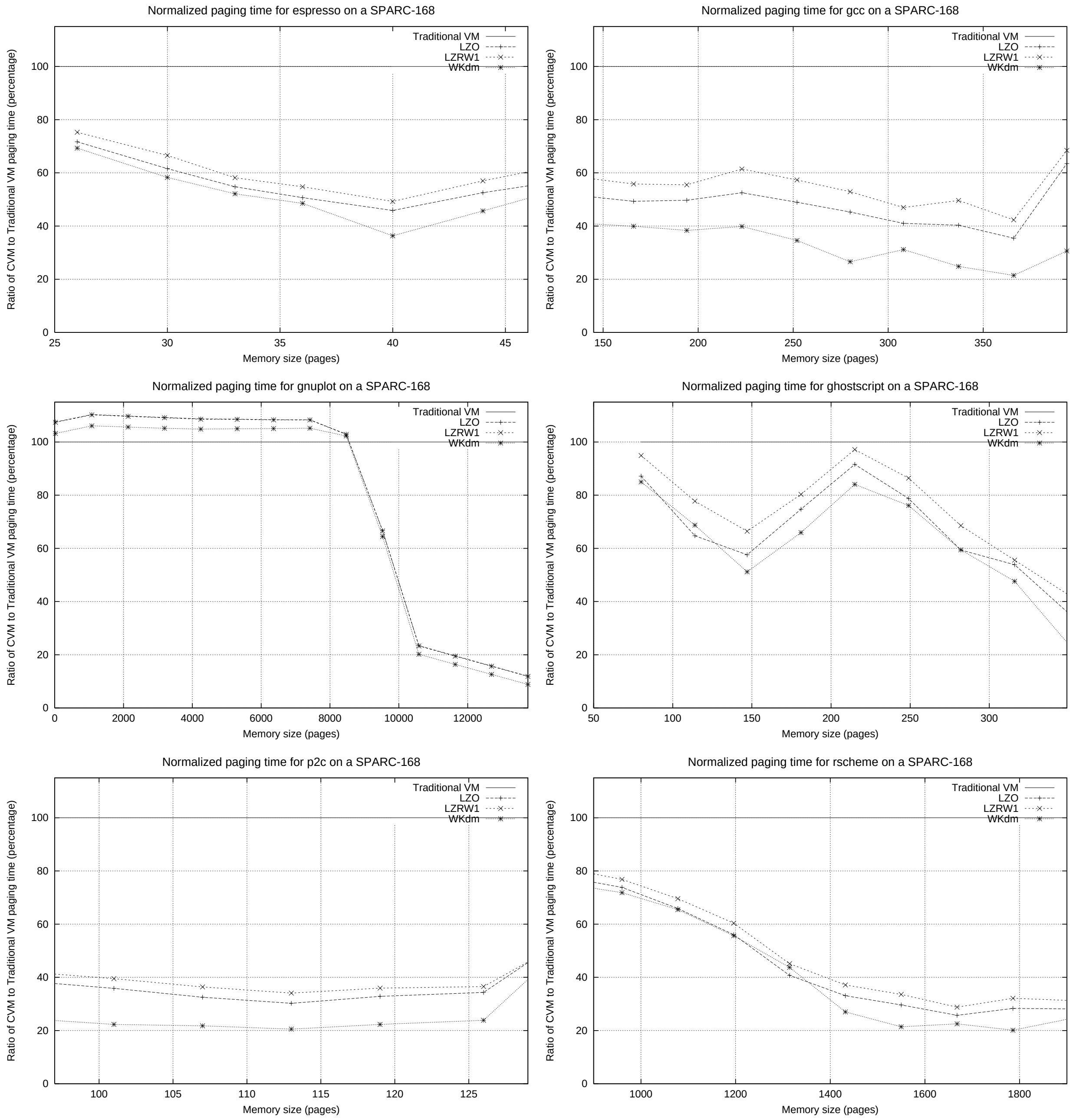

Figure 4: Varying compression algorithms can affect performance significantly. Even though all algorithms yield benefits compared to uncompressed virtual memory, some are significantly better than others.

replacement policy.

As can be seen, all algorithms obtain significant benefit over uncompressed virtual memory for the interesting ranges of memory sizes. Benefits of over 40% are common for large parts of the plots in Figure 4. At the same time, losses are rare (only exhibited for gnuplot) and small. Additionally, losses diminish for faster compression algorithms (and faster processors, which is not shown in this plot). That is, when our adaptivity does not perform optimally, its cost can be reduced by having a fast compression algorithm, since it is a direct function of performing unnecessary compressions and decompressions.

Gnuplot is an interesting program to study more closely. The program stores data that are highly compressible (exhibiting a ratio of over 4:1 on average). This way, the compressed VM policy can look at quite large memory sizes, expecting that it can compress enough pages so that all the required data remains in memory. Nevertheless, gnuplot's running time is dominated by a large loop iterating only twice on a lot of data. Hence, for small memory sizes the behavior that the compressed caching policy tries to exploit ends before any benefits can be seen. For larger sizes, the benefit can be substantial, reaching over 80%.

As shown in Figure 4, the performance difference of compressed caching under different compression algorithms can often be over 15%. Our WKdm algorithm achieves the best performance for the vast majority of data points, due to its speed and comparable compression rates to LZO. The LZRW1 algorithm, used by Douglis yields consistently the worst results. This fact, combined with the slow machine used (for current standards) are at least partially responsible for the rather disappointing results that Douglis observed.

4.2.3 Implementation and Architecture Effects

In the past sections we only showed results for our SPARC 168 MHz machine. As expected, the faster SPARC 300 MHz machine has a lower compression and decompression overhead and, thus, should perform better overall. The Pentium Pro 180 MHz machine is usually slower than both SPARC machines in compressing and uncompressing pages (not unexpectedly as it is an older architecture—see also out later remarks on memory bandwidth).

Figure 5 shows three of our test programs simulated under WKdm and LZO in all three architectures. For WKdm, the performance displayed agrees with our observations on machine speeds. Nevertheless, the performance of LZO is significantly better on the Pentium Pro 180 MHz machine than one would expect based on the machine speed alone. The reason is that, as pointed out earlier, the implementation of LZO we used on the Pentium Pro is hand optimized for speed in Intel x86 assembly language. Perhaps surprisingly, the effect of the optimization is quite significant, as can be seen. For ghostscript, for instance, the Pentium Pro is faster than the SPARC 168 MHz using LZO.

4.3 Technology Trends

4.3.1 Is Memory Bandwidth a Problem?

Compressed caching mostly benefits from the increases of CPU speed relative to disk latency. Nevertheless, a different factor comes into play when disk and memory *bandwidths* are taken into account. A first observation is that moving data from memory takes at most one-third of the execution time of our WKdm compression algorithm. (This ratio is true for both the Pentium Pro 180 MHz machine, which has a slow memory subsystem, and the SPARC 300 MHz, which has a fast processor. It is significantly better for the SPARC 168 MHz machine.) Hence, memory bandwidth does not seem to be the limiting factor for the near future. Even more importantly, faster memory architectures (e.g., RAMBUS) will soon become widespread and compression algorithms can fully benefit as they only need to read contiguous data. The overall trend is also favorable. Memory bandwidths have historically grown at 40%, while disk bandwidths and latencies have only grown at rates around 20%. (An analysis of technology trends can be found in M. Dahlin's "Technology Trends" Web Page at http://www.cs.utexas.edu/users/dahlin/techTrends/ .)

4.3.2 Sensitivity Analysis

The cost and benefits of compressed caching are dependent on the relative costs of compressing (and uncompressing) a page vs. fetching a page from disk. If compression is insufficiently fast relative to disk paging, compressed virtual memory will not be worthwhile.

On the other hand, if CPU speeds continue to increase far faster than disk speeds, as they have for many years, then compressed virtual memory will become increasingly effective and increasingly attractive. Over the last decade, CPU speeds have increased by about 60% a year, while disk latency and bandwidth have increased by only about 20% a year. This works out to an increase in CPU speeds *relative to disk speeds* of one third a year—or a doubling every two and a half years, and a quadrupling every five years.

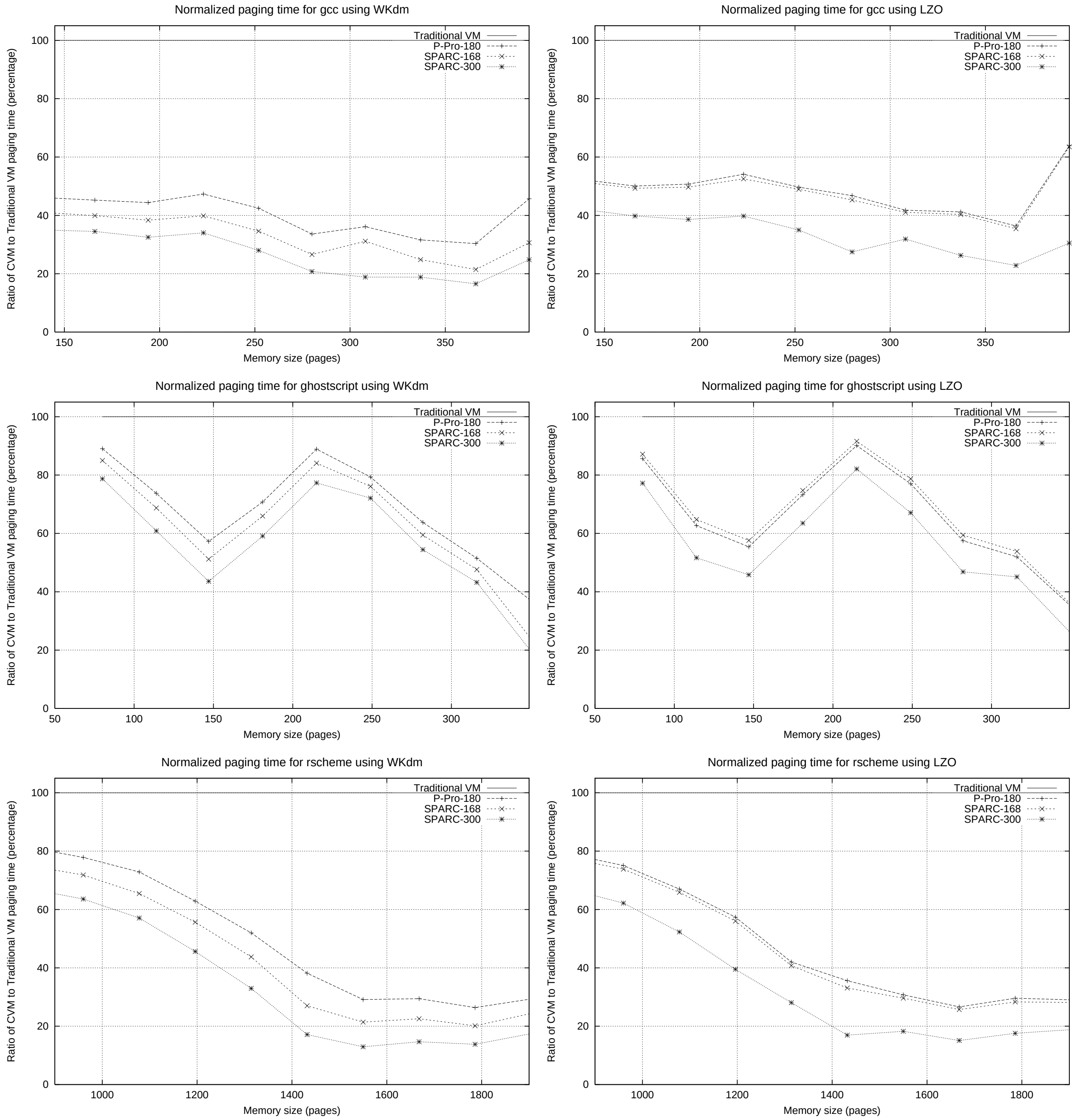

Figure 5: A SPARC 168 MHz usually has better performance than a Pentium Pro 180 MHz, while a SPARC 300 MHz is significantly better than both. Nevertheless, the Pentium Pro 180 MHz is much faster for a hand-optimized version of the LZO algorithm, sometimes surpassing the SPARC 168 MHz.

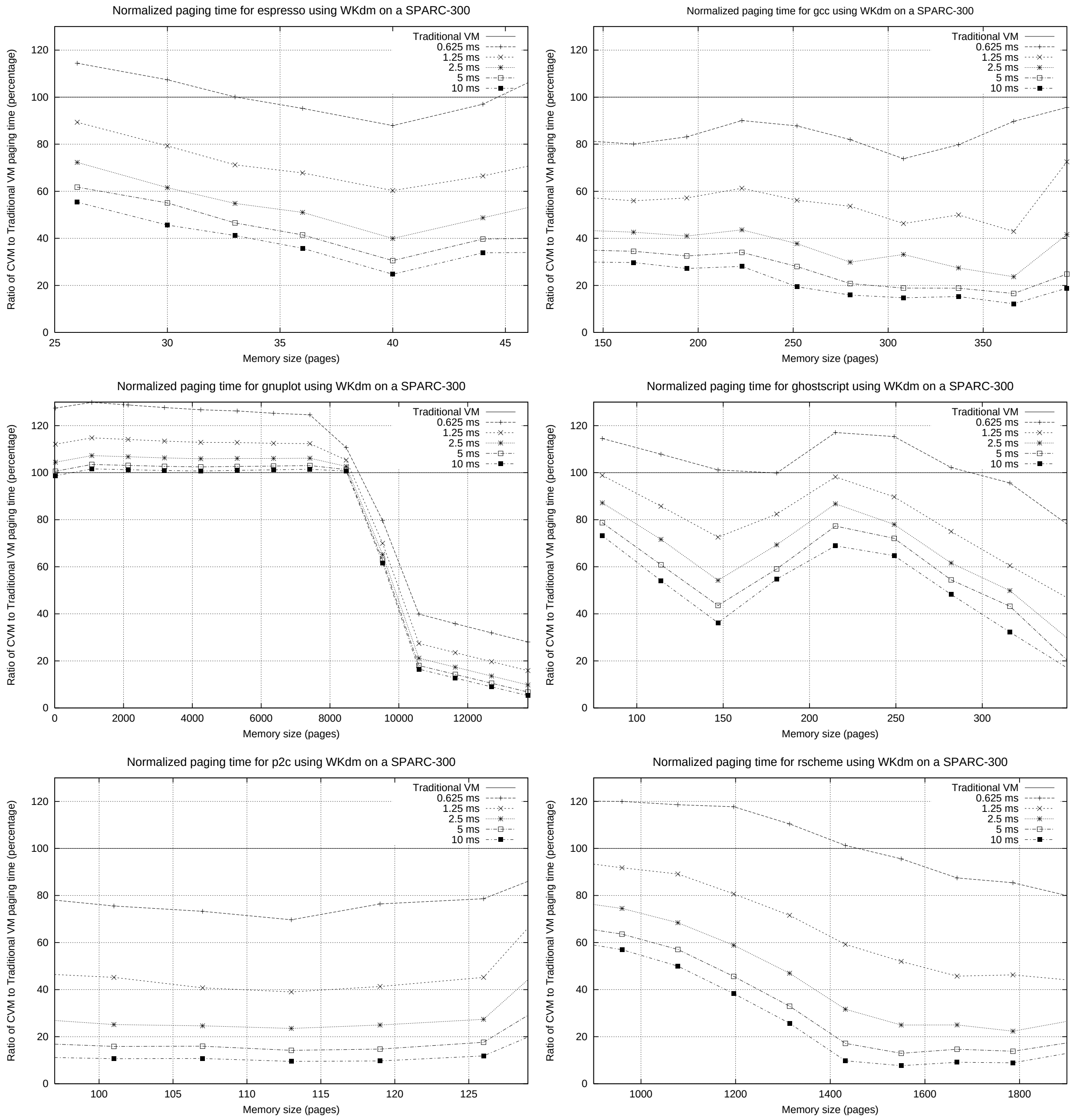

Figure 6: A sensitivity analysis studying disks of various speeds. This conservatively covers the cases of slower CPUs, perfect prefetching, and larger page sizes.

Figure 6 shows plots of simulated performance of our adaptive caching system, using page compression timings measured on a 300 MHz UltraSPARC. Each line represents the paging costs for simulations using a given disk fault cost. Costs are normalized to the performance of a conventional LRU memory with the *same* disk page access time; that is, each curve represents the speedup or slowdown that comes from using compressed caching.

The middle line in each plot can be regarded as the performance of a machine the speed of a 300 MHz UltraSPARC with an average page fetch cost (for 4KB pages) of only 2.5ms, about one third the average disk seek time of a fast disk. Note that, in normalized performance terms, assuming a twice as fast disk is exactly equivalent to assuming a twice as slow CPU. At the same time, studying the case of a fast disk conservatively covers the case of perfect prefetching of multiple pages (a twice as fast disk is equivalent to always prefetching the next two needed pages with one seek). This, in turn, conservatively covers the case of using larger page sizes. Hence, our sensitivity analysis (taking into account fast disks) also subsumes many other scenarios.

Looking at the middle line of each plot, we can see that with a disk page access cost of 2.5ms, most programs show a reduction of paging times by 30 to 70 percent, averaged across the interesting range of memory sizes. Thus, compressed virtual memory is a very clear win even for a disk access cost of 2.5ms per 4KB page. The line above the middle one can be taken to represent a system with the same CPU speed and disk costs a factor of two lower, at 1.25ms per 4KB page. Even though performance for this system is significantly worse, still much speedup is obtained. The top line represents a system where disk page accesses cost only 0.625ms per 4KB page. For some programs, this degrades performance overall to the point that compressed caching is not worthwhile.

Going the other direction, along with the technology trends, we can look at the next lower line to see the performance of a system with twice as fast a processor relative to its disk. For most of our programs, each doubling of CPU speed offers a significant additional speedup, typically decreasing remaining paging costs by ten to forty percent.

5 Related Work

Our compression algorithms are roughly similar to the well-known MTF ("move-to-front") algorithm, which maintains an LRU ordering, but is unusual in its use of partial matching and a fixed 32-bit word as its basic granularity of operation. (The general MTF scheme is fairly obvious and has been invented independently at least four times [BCW90] before we reinvented it yet again.)

The use of partial matching (only the high bits) can be viewed as a simple and fast approximation of *delta coding*, a technique used for purely numeric data (such as sensor input data or digitized audio) [Nel95].[4] Delta coding (a form of differential coding) encodes a numerical value as a numerical difference from the previous numerical value. Unlike a traditional delta coder, our algorithm can encode a value by its difference (low bits) from any of the values in an MTF dictionary, rather than the unique previous value.

In [KGJ96], Kjelso, Gooch, and Jones presented a compression algorithm also designed for in-memory data. Their X-match algorithm (which is designed for hardware implementation) is similar to ours in that both use a small dictionary of recently used words. Rizzo, in [Riz97], also devised a compression algorithm specific to in-memory data. His approach was to compress away the large number of zeros found in such data. Rizzo asserts that more complex modeling would be too costly. We have shown that it is possible to find more regularities without great computational expense.

While we have not addressed the compression of machine code, others have shown that it is possible to compress machine code by a factor of 3 using a specially tuned version of a conventional compressor [Yu96] and by as much as a factor of 5 using a compressor that understands the instruction set [EEF$^+$97]. We believe that similar techniques can be made very fast and achieve a compression ratio of at least 2, similar to the ratios we get for data, so an overall compression ratio of 2 for both code and data should generally be achievable. This is within 20% of the size reduction found by Cogswell and Russinovich using an extremely fast, simple, and untuned "general purpose" compression algorithm [RC96]. (Their paging data also support the assumption that full workloads exhibit the kind of locality needed for compressed paging, making our focus on data paging more reasonable.)

A significant previous study of compressed caching was done by Douglis, who implemented a compressed virtual memory for the Sprite operating system and evaluated it on a DECStation 5000, which is several times to an order of magnitude slower than the machines we used in our experiments.

Douglis's results were mixed, in that compressed virtual memory was beneficial for some programs and detrimental to others. As should be apparent from our dis-

[4]"Delta coding" is something of a misnomer because it's really a modeling technique with an obvious encoding strategy.

cussion of performance modeling, we believe that this was primarily due to the slow hardware (by today's standards) used. This is supported by our sensitivity analysis, which showed that an 8 times slower machine than a 300 MHz UltraSPARC would yield mixed results, even with better compression algorithms than those available to Douglis.

As discussed earlier, Russinovich and Cogswell's study [RC96] showed that a simple compression cache was unlikely to achieve significant benefits for the PC application workload they studied. Nevertheless, their results do not seem to accurately reflect the trade-offs involved. On one hand, they reported compression overheads that seem unrealistically low (0.05ms per compression on an Intel 80486 DX2/66, which is improbable even taking only the memory bandwidth limitations into account). But the single factor responsible for their results is the very high overhead for handling a page fault that they incurred (2ms—this is overhead not containing the actual seek time). This overhead is certainly a result of using a slow processor but it is possibly also an artifact of the OS used (Windows 95) and their implementation.

A study on compressed caching, performed in 1997 but only very recently published, was done by Kjelso, Gooch, and Jones [KGJ99]. They, too, used simulations to demonstrate the efficacy of compressed caching. Additionally, they addressed the problem of memory management for the variable-size compressed pages. Their experiments used the LZRW1 compression algorithm in software and showed for most programs the same kinds of reduction in paging costs that we observed. These benefits become even greater with a hardware implementation of their X-match algorithm.

Kjelso, Gooch, and Jones did not, however, address the issue of adaptively resizing the compressed cache in response to reference behavior. Instead, they assumed that it is always beneficial to compress more pages to avoid disk faults. This is clearly not true as when more pages are compressed, many more memory accesses may suffer a decompression overhead, while only a few disk faults may be avoided. The purpose of our adaptive mechanism is to determine when the trade-off is beneficial and compression should actually be performed. Kjelso, Gooch, and Jones did acknowledge that some compressed cache sizes can damage performance. Indeed, their results strongly suggest the need for adaptivity: two of their four test programs exhibit performance deterioration under software compression for several memory sizes.

6 Conclusions

Compressed virtual memory appears quite attractive on current machines, offering an improvement of tens of percent in virtual memory system performance. This improvement is largely due to increases in CPU speeds relative to disk speeds, but substantial additional gains come from better compression algorithms and successful adaptivity to program behavior.

For all of the programs we examined, on currently available hardware, a virtual memory system that uses compressed caching will incur significantly less paging cost. Given memory sizes for which running a program suffers tolerable amounts of paging, compressed caching often eliminates 20% to 80% of the paging cost, with an average savings of approximately 40%. As the gap between processor speed and disk speed increases, the benefit will continue to improve.

The recency based approach to adaptively resizing the compression cache provides substantial benefit at nearly any memory size, for many kinds of programs. In our tests, the adaptive resizing provided benefit over a very wide range of memory sizes, even when the program was paging little. The adaptivity is not perfect, as small cost may be incurred due to failed attempts to resize the cache, but performs well for the vast majority of programs. Moreover, it is capable of providing benefit for small, medium, and large footprint programs.

The WK compression algorithms successfully take advantage of the regularities of in-memory data, providing reasonable compression at high speeds. After many decades of development of Ziv-Lempel compression techniques, our WKdm compressor fared favorably with the fastest known LZ compressors. Further research into in-memory data regularities promises to provide tighter compression at comparable speeds, improving the performance and applicability of compressed caching for more programs.

It appears that compressed caching is an idea whose time has come. Hardware trends favor further improvement in compressed caching performance. Although past experiments failed to produce positive results, we have improved on the components required for compressed caching and have found that it could be successfully applied today.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and*

Operating Systems (ASPLOS IV), pages 96–107, Santa Clara, California, April 1991.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[EEF+97] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the 1997 SIGPLAN Conference on Programming Language Design and Implementation*, Las Vega, Nevada, June 1997. ACM Press.

[KGJ96] Morten Kjelso, M. Gooch, and S. Jones. Main memory hardware data compression. In *22nd Euromicro Conference*, pages 423–430. IEEE Computer Society Press, September 1996.

[KGJ99] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. In *Journal of Systems Architecture 45*, pages 571–590. Elsevier Science, 1999.

[Nel95] Mark Nelson. *The Data Compression Book (2nd ed.)*. M & T Books, 1995.

[RC96] Mark Russinovich and Bryce Cogswell. RAM compression analysis, February 1996. O'Reilly Online Publishing Report available from http://ftp.uni-mannheim.de/info/OReilly/windows/win95.update/model.html.

[Riz97] Luigi Rizzo. A very fast algorithm for RAM compression. In *Operating Systems Review 311997*, pages 36–45, 1997.

[SW91] Walter R. Smith and Robert V. Welland. A model for address-oriented software and hardware. In *25th Hawaii International Conference on Systems Sciences*, January 1991.

[Wil90] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also appears in *SIGPLAN Notices 23*(3):45–52, March 1991.

[Wil91a] Ross N. Williams. An extremely fast Ziv-Lempel compression algorithm. In *Data Compression Conference*, pages 362–371, April 1991.

[Wil91b] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, California, October 1991. IEEE Press.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[WKB] Paul R. Wilson, Scott F. Kaplan, and V.B. Balayoghan. Virtual memory reference tracing using user-level access protections. In Preparation.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[Yu96] Tong Lai Yu. Data compression for PC software distribution. *Software Practice and Experience*, 26(11):1181–1195, November 1996.