An Analytical Model for Software-Only Main Memory Compression

Irina Chihaia and Thomas Gross
Departement Informatik
ETH Zürich
CH 8092 Zürich, Switzerland

ABSTRACT

Many applications with large data spaces that cannot run on a typical workstation (due to page faults) call for techniques to expand the effective memory size. One such technique is memory compression.

Understanding what applications under what conditions can benefit from main memory compression is complicated due to various tradeoffs and the dynamic characteristics of applications. For instance, a large area to store compressed data increases the effective memory size considerably but also decreases the amount of memory that can hold uncompressed data.

This paper presents an analytical model that states the conditions for a compressed-memory system to yield performance improvements. Parameters of the model are the compression algorithm efficiency, the amount of data being compressed, and the application memory access pattern. Such a model can be used by an operating system to compute the size of the compressed-memory level that can improve an application's performance.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; I.6 [**Simulation and Modeling**]: Model Validation and Analysis; D.4.2 [**Operating Systems**]: Storage Management—*main memory*

General Terms

Performance, Measurement

1. INTRODUCTION

Despite larger and larger main memories in PCs and workstations, there are still many applications with memory demands that exceed the amount of memory that is available. These applications have working sets that are larger than the main memory in the user's host, and these applications resort to thrashing since the amount of physical memory is less than what is required. In the end, these applications run at the speed of the swapping system (i.e., the speed of the disk) and cannot benefit from any increase in processor speed.

One example of an application that is limited by the physical memory available in a system is symbolic model checking. Although significant progress has improved the space performance of model checkers [17], there are many interesting models that cannot be checked. With thrashing, the execution time to check such a model exceeds the time that is acceptable, and therefore the corresponding models are not checked.

Compression has been used in many settings to increase the effective size of a storage device or to increase the effective bandwidth, and other researchers have proposed to build a *compressed-memory* system by integrating compression into the memory hierarchy. The basic idea of a compressed-memory system is to set aside a part of main memory to hold compressed pages. Then, instead of swapping a page P to disk (when its main memory is needed), the evicted page P is compressed (producing P_{comp}). P_{comp} is kept in the compressed-memory level. If P_{comp} is needed again, it is decompressed and moved to the main (uncompressed) memory. As long as compression and decompression of P (resp. P_{comp}) take less time than swapping P in and out, there exists the opportunity to improve overall system performance. Of course, a number of conditions must be met: setting aside part of the main memory to hold compressed pages *reduces* the effective size of the application's main memory area and *increases* the number of page faults. And if the compressed-level is not large enough to hold a sufficient number of compressed pages, some compressed pages must be evicted to disk, increasing the overhead of the memory system.

The size of the data segment(s) is the critical issue for most applications. We therefore concentrate on the use of data compression at the main memory level and do not consider compression of the instruction space. Furthermore, we are interested in a design that works with commodity PCs. Therefore changes to the L1 and L2 cache levels of the memory hierarchy are not an option (in our design space); these levels of the memory hierarchy are usually controlled by processor architects and PC designers. However, by restricting changes to the software system (most prominently the OS kernel), we can use compression for only those applications that benefit from it. Even with these restrictions, there is still considerable freedom available to system architects. For instance, we have completed an implementation of a compressed-memory system using a dual-processor workstation so that one CPU is dedicated to de/compression activities.

In this paper, we use a compressed-memory level to hold evicted pages in compressed form. The system intercepts page faults to check if the requested page is available in compressed form before

This work was founded, in part, by the NCCR "Mobile Information and Communication Systems", a research program of the Swiss National Science Foundation, and by a gift from Intel's Microprocessor Research Laboratory.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WMPI '04, Munich Germany
Copyright 2004 ACM 1-59593-040-X ...$5.00.

a disk access is initiated. When the compressed-memory level becomes filled, parts of the compressed data are swapped to disk. We have completed an implementation of a compressed-memory system into the Linux OS [3].

The potential benefits of main memory compression depend on the relationship between the size of the compressed area, an application's compression ratio, and an application's access pattern. Because accesses to compressed pages take longer than accesses to uncompressed pages, compressing too much data decreases an application's performance. Moreover, if an application's pages do not compress well, compression will not show any benefit. Furthermore, if an application accesses its data set such that compression does not save enough accesses to disk, memory compression will slowdown the application.

To accurately decide how much data to compress during an application's execution, we must keep track of recent program behavior. However, the more accurate the prediction scheme is, the more information it requires. Therefore, a very accurate scheme will also have a high overhead. To investigate whether there is need for simple and fast (yet maybe not so accurate) schemes, we investigate the effect of memory compression on two memory intensive applications. The measurements show that because for these applications the performance improvement is not very high, using simple schemes to decide on the optimal size of the compressed area is crucial. We propose a simple model that relies on a few data points, such as an application's number of accesses to the uncompressed memory, to the compressed memory, and to the disk. We show how the model can be used to compute the size of the compressed-memory level that can improve an application's performance.

2. PERFORMANCE POTENTIAL

To assess the benefits of main memory data compression, we examine the performance of two applications that use large data sets. We use a Pentium 4 PC at 1.9 GHz to generate the memory reference string of the selected applications. To measure an application's performance on a compressed-memory system, we use a compressed-memory prototype built into the Linux kernel. We select two programs that are simulators with different input sets, the SMV model checker [14] and the NS2 network simulator [13]. We use Yang's SMV implementation since it demonstrated superior performance over other implementations [17]. The SMV inputs model the FireWire protocol [12]; although the selected inputs have the same memory footprint of 625MB, they perform different amounts of computation. NS2 simulates the DSR protocol over a wireless network of 2500 and 3000 nodes. The first two simulations have a memory footprint of 600MB, the last two of 750MB.

For each input set we select memory sizes for which an application's execution time is at least twice as slow, but no more than 100 times slower than the execution time without thrashing. When memory available is less than is required to hold the whole working set, the 1.9 GHz CPU spends 1-99% of the total running time paging. We configure the system such that the amount of memory available is 95%, 90%, and 80% of memory required to run the application without thrashing and the size of the compressed-memory level is 5%, 10%, and 20% of the size of memory available. The measurements show that when compression is enabled, the execution of most of the SMV models is slowed by 40-80%. However, one of the SMV models executes 18-35% faster with compression. Also for the selected NS2 inputs, the results were mixed: main memory compression degrades the performance of some inputs by 4-17%, and improves the performance of other inputs by 10-29%.

The measurements show that software compression can both degrade and improve the performance of applications that use large data sets, but the improvement is not very high. Therefore, for the applications we select, the adaptive scheme that resizes the compressed-memory level must be simple and should introduce low overheads.

3. COST/BENEFIT ANALYSIS

3.1 Application Performance

The execution time of a memory intensive application (T) is the sum of the times the application spends at each memory level. The time spent at a memory level (T_{total_i}) is a linear combination of the number of hits at that level (N_i) multiplied by the access time to that memory level (T_i). Formally, an application's execution time is

$$T = \sum_{i=1}^{n} T_{total_i} = \sum_{i=1}^{n} N_i \times T_i \quad (1)$$

where i iterates over the memory levels (e.g., L1, L2, DRAM).

Given the complexity of out-of-order execution processors, there is no simple characterization of the memory system performance with a single parameter (T_i). We use the definition of Chihaia and Gross [4] and for each memory level we compute an application's access time as the weighted average of the time spent executing continuous accesses (T_{cont}), accesses within the same cache line (T_{sameCL}), and non-continuous accesses ($T_{non-cont}$). This definition of an application's access time captures a wide range of processor optimizations, such as how many of the application's accesses to cache, memory and disk are overlapped. Formally, an application's read access time to a memory level is

$$T_{Rd} = \frac{N_{Rd_{cont}}}{N_{Rd}} \cdot T_{Rd_{cont}} + \frac{N_{Rd_{sameCL}}}{N_{Rd}} \cdot T_{Rd_{sameCL}} + \frac{N_{Rd_{non-cont}}}{N_i} \cdot T_{Rd_{non-cont}}$$

where $N_{Rd} = N_{Rd_{cont}} + N_{Rd_{sameCL}} + N_{Rd_{non-cont}}$. An application's write access time (T_{Wr}) is computed using the same formula and is the weighted average of the application's write access times.

An application's access time to a memory level (T_i) is the weighted average of its read and write access times ($T_{i_{Rd}}$ and $T_{i_{Wr}}$), and is

$$T_i = \frac{N_{i_{Rd}}}{N_i} \cdot T_{i_{Rd}} + \frac{N_{i_{Wr}}}{N_i} \cdot T_{i_{Wr}} \quad (2)$$

where $N_i = N_{i_{Rd}} + N_{i_{Wr}}$.

We compute the L_1 and L_2 cache access times (T_{L_1} and T_{L_2}), and the DRAM access time (T_{DRAM}) using Eq. 2. As we do not investigate changes to the L_1 and L_2 caches, we consider them a single cache level (L_{12}). Formally, the time an application spends at the (L_1 and L_2) cache level is

$$T_{L_{12}} = N_{L_1} \cdot T_{L_1} + N_{L_2} \cdot T_{L_1}$$

where N_{L_1} and N_{L_2} are the L_1 and L_2 cache hits, and T_{L_1} and T_{L_2} are the L_1 and L_2 cache access times.

The disk access time (T_{Disk}) includes an application's characteristics (by considering the weighted average), and is computed as

$$T_{Disk} = \frac{N_{Disk_{seq}}}{N_{Disk}} \cdot T_{Disk_{seq}} + \frac{N_{Disk_{non-seq}}}{N_{Disk}} \cdot T_{Disk_{non-seq}} \quad (3)$$

where $T_{Disk_{seq}}$ is the performance of a sequential disk access, and $T_{Disk_{non-seq}}$ is the average time of a random disk access, and $N_{Disk} = N_{Disk_{seq}} + N_{Disk_{non-seq}}$.

The access time to the compressed-memory level (T_{ComprL}) is the de/compression speed (T_{Cspeed}) plus the time spent waiting for old compressed pages to be swapped to disk[1]. The de/compression

[1]This happens when the compressed-memory level becomes filled and the compression rate is bigger than the decompression rate.

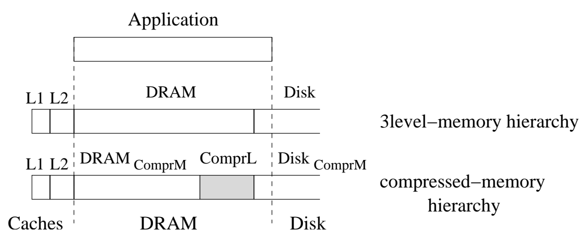


Figure 1: A 3-level-memory hierarchy and a compressed-memory hierarchy

speed includes an application's characteristics, and is

$$T_{Cspeed} = \frac{N_{Decompr}}{N_{ComprL}} \cdot T_{Decompr} + \frac{N_{Compr}}{N_{ComprL}} \cdot TCompr$$

where $T_{Decompr}$ and T_{Compr} are the de/compression times, and $N_{ComprL} = N_{Decompr} + N_{Compr}$.

We consider a three-level memory hierarchy ("3-level-memory hierarchy") with a (L_1 and L_2) cache level, main memory, and disk. On this system, an application's performance when the amount of physical memory is less than is required to run the application without thrashing (see Fig. 1) is computed based on Eq. 1 (n=3), and is

$$T_{with-swap} = T_{L_{12}} + N_{DRAM} \cdot T_{DRAM} + N_{Disk} \cdot T_{Disk} \quad (4)$$

The basic idea of a compressed-memory system is to reserve some memory that would normally be used directly by an application and use this memory region instead to hold compressed pages (see Fig. 1). We call the compressed area "ComprL" and compute the execution time of the same application on the compressed-memory system based on Eq. 1 (n=4).[2] Formally,

$$\begin{aligned} T_{ComprM} = & T_{L_{12}} + N_{DRAM_{ComprM}} \cdot T_{DRAM} \\ & + N_{ComprL} \cdot T_{ComprL} + N_{Disk_{ComprM}} \cdot T_{Disk} \end{aligned} \quad (5)$$

3.2 Size of the Compressed-Memory Level

This section defines the minimum size of the compressed-memory level that can improve an application's performance. We consider the performance of the memory system constant, and we use lower case notation to denote it (t_{DRAM}, t_{ComprL}, and t_{Disk}). An application's characteristics are considered variable, and are denoted by an upper case notation (N_{DRAM}, N_{ComprL} and N_{Disk}).

As the 3-level-memory and compressed-memory systems have the same (L_1 and L_2) caches, an application's number of L_2 cache misses is the same on both system. On the 3-level-memory system the L_2 cache misses become DRAM hits (N_{DRAM}) and disk accesses (N_{Disk}), while on the compressed-memory system they result in DRAM hits ($N_{DRAM_{ComprM}}$), *ComprL* hits (N_{ComprL}) and disk accesses ($N_{Disk_{ComprM}}$). After substituting the number of L_2 misses with data above, we have

$$N_{ComprL} = (N_{DRAM} - N_{DRAM_{ComprM}}) + (N_{Disk} - N_{Disk_{ComprM}}) \quad (6)$$

where $n_{DRAM} > n_{DRAM_{ComprM}}$ and $n_{Disk} > n_{Disk_{ComprM}}$.

Main memory compression improves an application's performance for sizes of the compressed-memory level for which the condition $T_{ComprM} \leq T_{with-swap}$ is fulfilled. After substituting $T_{with-swap}$ and T_{ComprM} with data in Eq. 4 and 5, and N_{ComprL} with data in Eq. 6, we have

$$N_{DRAM} - N_{DRAM_{ComprM}} \leq \frac{t_{Disk} - t_{ComprL}}{t_{ComprL} - t_{DRAM}} \cdot (N_{Disk} - N_{Disk_{ComprM}}) \quad (7)$$

The term $N_{DRAM} - N_{DRAM_{ComprM}}$ of Condition 7 is the overhead due to smaller effective uncompressed DRAM (fewer DRAM hits), and is called "Overhead". $\frac{t_{Disk}-t_{ComprL}}{t_{ComprL}-t_{DRAM}} \cdot (N_{Disk} - N_{Disk_{ComprM}})$ is the gain due to fewer disk accesses, and is called "Gain".

3.3 Influence of an Application's Characteristics

Flynn [7] defines the page miss rate of an application as a function of the amount of memory provided, application memory footprint, and application memory access behavior. He shows that an application's number of accesses to disk is given by the following formula: $h \cdot 10^{-V \cdot z}$, where V is the fraction of memory required that is available, and h and z are application dependent constants. The number of DRAM hits is equal with the number of L_2 misses minus the number of disk accesses, or formally $N_{L2_{misses}} - h \cdot 10^{-V \cdot z}$.

To assess the influence of the *ComprL* size, we select the size of the memory available to be 60% of the working set size of the *nodes_3_4_2* SMV model. The computed values[3] of "Overhead" and "Gain" (Condition 7) for different sizes of the compressed-memory level are depicted in Figure 2. Based on this figure, we can make several observations. First, if the compression speed is slow, the value of $\frac{t_{Disk}-t_{ComprL}}{t_{ComprL}-t_{DRAM}}$ is small and Condition 7 is not fulfilled; hence, "Gain" is smaller than "Overhead", as shown in Figure 2(a). In this case, a compressed-memory system will slowdown an application for all sizes of the compressed-memory level. For faster compression speeds, "Overhead" is smaller than "Gain" for small sizes of the compressed-memory level and bigger for larger sizes, as shown in Figure 2(b). In this case, a compressed-memory system improves an application's performance for those sizes of the compressed-memory level that are smaller than the intersection point of the two slopes. Third, fast compression speeds result in big values for $\frac{t_{Disk}-t_{ComprL}}{t_{ComprL}-t_{DRAM}}$ that fulfill Condition 7. Figure 2(c) shows such a case, where "Gain" is bigger than "Overhead". In this case, a compressed-memory system will show a clear benefit for all sizes of the compressed-memory level.

Our analysis confirm other researchers' measurements that showed that devoting too much memory to hold compressed data hurts as much as devoting not enough [15]. Moreover, our analytical model explains why main memory compression can both improve and degrade an application's performance.

4. VALIDATION

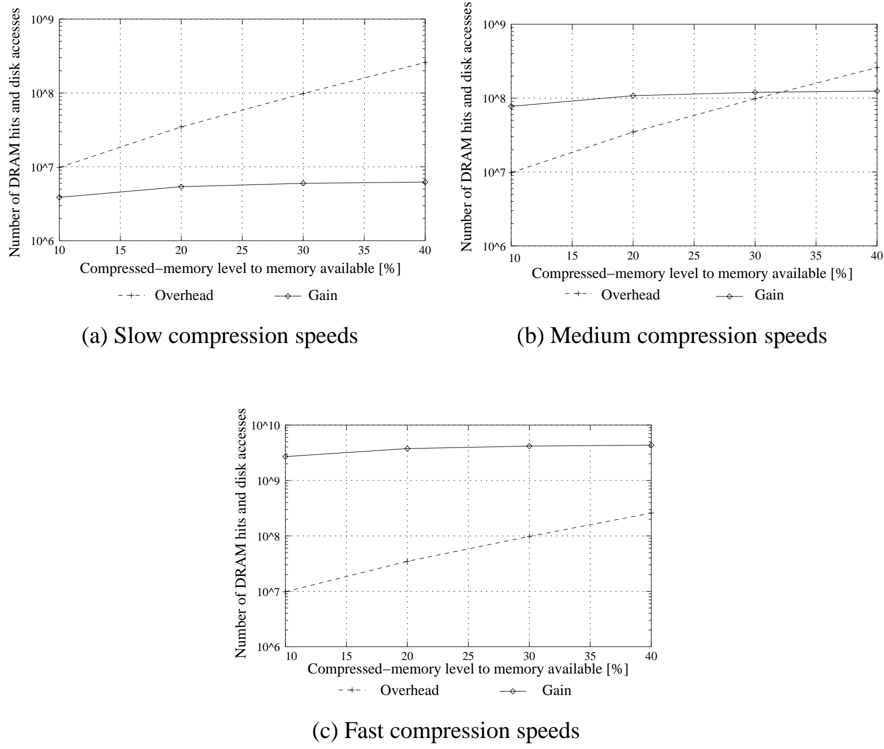
4.1 Experimental Setup

To generate the memory reference string of the selected applications, we use a commodity PC (Pentium 4 at 1.9 GHz, with a 8KB L_1 data cache, 256KB L_2 cache, and 1GB DRAM) that has its swap partition on a ST340016A ATA disk. The PC runs the Linux OS.

An application's DRAM read and write access time is the weighted average of the time spent executing continuous accesses, accesses

[2] As both uncompressed and compressed systems have the same (L_1 and L_2) caches, we assume that an application's reference behavior is on both systems the same. Therefore, also the time spent by an application at the (L_1 and L_2) cache level is the same ($T_{L_{12}}$).

[3] We consider two memory sizes smaller than the application footprint (two values of V) and measure the number of disk accesses for these memory sizes. The values of h and z are the solutions of a system of two equations with two variables. Having h and z, we use Flynn's formula to compute the number of disk accesses for all fractions of memory required that is available ($0 \leq V \leq 1$).



(a) Slow compression speeds

(b) Medium compression speeds

(c) Fast compression speeds

Figure 2: The values of the "Overhead" and "Gain" for different compression speeds for the SMV application

within the same cache line, and non-continuous accesses. SMV and NS2 are pointer-based applications that access their data non-continuously. On the selected PC, the performance of the SMV and NS2 DRAM reads is 50 *ns* and of DRAM writes is 62.5 *ns*. An application's DRAM access time is computed using Eq. 2, and is the weighted average of the application's read and write access times (90% of the SMV DRAM accesses are reads; for NS2 the DRAM reads are 33%). The disk performance is computed using the formula in [8]: for the ST340016A ATA disk $T_{Disk_{seq}}$ and $T_{Disk_{non-seq}}$ are 5.15 *ms* and 13.05 *ms*. An application's disk access time (T_{Disk}) is computed using Eq. 3, and is the weighted average of the application's sequential and non-sequential access times (17% of the SMV disk accesses are sequential; 45% of the NS2 disk accesses are sequential).

The de/compression speed (T_{Compr} and $T_{Decompr}$) of different algorithms was measured using binary files generated by the selected applications; the unit of compression is equal to the page size. We chose the WKdm compression algorithm, as it shows superior performance over other algorithms. For SMV, the WKdm de/compression speed T_{Cspeed} is 18.8 μs (45% of its *ComprL* accesses are compression operations), and for NS2, T_{Cspeed} is 19 μs (48% of the *ComprL* accesses are compression operations). The access time to the compressed-memory level (T_{ComprL}) is given by an application's de/compression speed (T_{Cspeed}) plus the time spent waiting for old compressed pages to be swapped to disk.

The number of DRAM hits (N_{DRAM}) is equal with the number of L_2 misses (gathered by the Pentium performance counters) minus the number of page faults (gathered using the Linux `/proc` file system). The number of hits at the compressed-memory level (N_{ComprL}) is gathered by an in-house compressed-memory prototype. The number of disk accesses (N_{Disk}) is equal with the number of page faults (reported by the Linux kernel) minus the number of compressed-memory hits.

To measure an application's performance on a compressed-memory system and to collect data about the compressed-memory level, we use a compressed-memory prototype built into the Linux kernel (version 2.4.18). The prototype is implemented as a module that allocates memory for the compressed-memory level. When a page is to be written to disk, it is compressed and copied into the compressed-memory level. Because the management unit of the compressed-memory level is 128B, a compressed page is stored as a list of blocks of 128B. The module uses a hash table to keep information about all pages that are stored in the compressed-memory level. When the compressed-memory level becomes filled, old compressed pages (using the LRU technique) are sent to disk by the compressed-memory daemon and the number of free compressed pages is kept below a certain threshold. (As a compressed page is sent to disk, it is decompressed to lower the latency of a future access.) When a page is to be read from the disk, it is first searched for in the compressed-memory level. If the page is there, it is decompressed and copied into DRAM, otherwise it is read from disk. The user selects one of the four compression algorithms implemented by the prototype (WKdm, WK4x4, LZRW1, LZ0 [16]) and specifies the performance data to be collected (e.g., number of compressed-memory hits, time spent waiting for old compressed pages to be swapped to disk).

4.2 Experimental Results

This section presents the performance of the SMV and NS2 applications that execute on a system with main memory compres-

sion, when the size of the compressed-memory level is fixed. We configure the system such that the amount of memory provided is 95% of memory required to run the application without thrashing; for this setup, the selected SMV models execute two times slower than without thrashing. We set the size of the compressed-memory level to be 5%, 10%, and 20% of memory available. We summarize the measurements in Table 1. The results confirm our cost/benefit analysis: main memory compression improves an application's performance for sizes of the compressed-memory level that fulfill Condition 7. The measurements show a small performance improvement for *nodes_3_3_3*. Although the selected SMV models have the same memory footprint of 625MB, *nodes_3_3_3* performs less computation than the other two models. Therefore, the lower amount of computation explains the performance improvement for this model when memory compression is employed.

NS2 allocates a large amount of data but only uses a small subset of this data at any one time. The memory footprint of a NS2 simulation is determined by the number of nodes simulated, while the working set size is given by the number of traffic connections that are simulated. Although both *NS2 sim1* and *NS2 sim2* have the same memory footprint of 600MB, they have different working set sizes. *NS2 sim3* and *NS2 sim4* differ in the same way: although they have the same memory footprint of 750MB, they have different working set sizes.

We configure the system such that the amount of memory provided is 80% of memory required to run the application without thrashing; for this setup, the selected NS2 inputs execute two times slower than normal. This small slowdown is explained by the fact that although the amount of memory available is smaller than the size of the NS2 memory footprint, the size of memory available is bigger than the size of the NS2 working set. We set the size of the compressed-memory level to be 12.5%, 25%, and 50% of memory available, and we summarize the measurements in Table 2. The results indicate that once the size of memory available is smaller than the NS2 working set size, many accesses are to the compressed-memory level and few disk accesses are avoided.

Our analysis examines the performance of two applications that use large data sets to asses the benefits of main memory compression. The measurements show that this technique improves the performance for these two applications when the size of the compressed-memory level fulfills Condition 7. Although in these experiments the sizes of the compressed-memory level are fixed, they do not only verify that our cost-benefit calculation is reasonable, but also help reveal the relationship between the size of memory available and compressed-memory level.

5. RELATED WORK

Several researchers have investigated the use of compression to reduce paging by introducing another level into the memory hierarchy (we call this level the compressed-memory level). The key idea of a compressed-memory system is to hold evicted pages in compressed form in the compressed-memory level, and intercept page faults to check if the requested page is available in compressed form before a disk access is initiated. When the compressed area becomes filled, parts of the compressed data are swapped to disk. The compressed-memory systems can be classified in software-based and hardware-based approaches. Since we want our solution to work with stock hardware, we consider software-based approaches only. For a description of the hardware-based approaches, we refer the interested reader to a recent study by Alameldeen and Wood [1].

Using compression to keep data in compressed form in main memory was first suggested by Wilson [15]. He notes that as CPU cycles become increasingly cheap relative to disk seeks, it becomes increasingly attractive to keep pages in memory in compressed form, adding a new level to the memory hierarchy.

Douglis [6] presents a detailed investigation of a compressed-memory prototype implemented in the Sprite OS. His experimental results do not indicate a consistent benefit from employing compression; some applications run faster (up to 62.3%), but other run slower (up to 36.4%). Because different programs have different memory requirements, Douglis implements an adaptive scheme that varies the size of the compressed-memory level dynamically. However, Kaplan [9] shows latter that Douglis' adaptive scheme might have been maladaptive.

Cervera et al. [2] investigate the performance of a compressed swap space prototype built into the Linux OS. Their results show that memory compression increases system performance by a factor of up to 2 relative to an uncompressed swap system. However, from a physical memory of 64MB only up to 4MB hold compressed data, and this small area may not suffice for programs with large working sets.

Kjelso et al. [11, 10] use simulations to demonstrate the efficacy of main memory compression. The authors develop a performance model for a compressed-memory architecture and investigate the performance impact of a software-based and hardware-based implementation of main memory compression for a number of DEC-WRL workloads. Their results show that software-based compression can improve system performance by up to a factor of 2, while hardware-based compression can improve performance by up to an order of magnitude. Although Kjelso et al. acknowledge that some sizes of the compressed-memory level can damage performance; they did not investigate the amount of data that should be compressed for compressed memory to show performance improvements. Our study addresses this issue and defines the minimum size of the compressed area that can improve an application's performance.

Kaplan et al. [9, 16] focus on CPU performance and reconsider the idea of compressed memory in the context of current technology trends. Their studies, based on simulations, show that the discouraging results of former studies were primarily due to the use of machines that were quite slow computation engines by current standards. Their technique uses the compressed-memory level for all disk I/O, including paging and file I/O. The authors show that the amount of data that should be compressed can be determined adaptively, by keeping track of recent program behavior. They maintain a queue of referenced pages ordered by their recency information; this queue contains records for pages that are in main memory and pages that are not in main memory but were evicted recently. The authors use simulations to validate their model. Although the input required by their decision scheme is easy to obtain in a simulation environment, this information cannot be obtained on current systems. On the other hand, our adaptation scheme relies on information that is easy to acquire on today's systems; we use the performance counters to capture the memory access demands of an application. In addition, Kaplan's model does not capture the influence of the DRAM performance on an application that executes on a compressed-memory system.

Castro et al. [5] evaluate the performance of a compressed-memory prototype implemented into the Linux kernel. The authors use an adaptive mechanism to resize the compressed area dynamically. Castro et al. analyze all accesses to the compressed area and depending on whether the page would be uncompressed or on disk if compression was not used, they shrink or resize the compressed area. Although intercepting every access to the compressed area may work well for small applications with few disk accesses, it may not be feasible for large applications with frequent disk ac-

Table 1: Performance data of the selected SMV models on a compressed-memory system

SMV model	memory available	*ComprL* size	Overhead accesses	Gain accesses	Cond. 7	speedup (>1) slowdown (<1)
nodes_3_3_3	95%	5%	11,720	99,003	Yes	1.21
		10%	24,137	142,926	Yes	1.35
		20%	153,835	158,265	Yes	1.18
nodes_3_4_2	95%	5%	678,519	137,752	No	0.2
		10%	1,622,204	198,867	No	0.4
		20%	6,465,217	2,495,166	No	0.4
nodes_4_2_3	95%	5%	1,276,966	609,537	No	0.2
		10%	3,168,318	879,965	No	0.2
		20%	12,627,181	974,401	No	0.4

Table 2: Performance data of the selected NS2 simulations

NS2 sim	memory available	*ComprL* size	Overhead accesses	Gain accesses	Cond. 7	speedup (>1) slowdown (<1)
NS2 sim1	80%	12.5%	5,290	14,511	Yes	1.1
		25%	8,096	14,423	Yes	1.21
		50%	17,831	14,352	No	0.96
NS2 sim2	80%	12.5%	5,043	14,236	Yes	1.1
		20%	8,029	14,140	Yes	1.25
		50%	17,477	14,092	No	0.93
NS2 sim3	80%	12.5%	1,216	6,226	Yes	1.3
		25%	1,754	6,145	Yes	1.12
		50%	1,403	6,076	Yes	1.25
NS2 sim4	80%	12.5%	264	5,991	Yes	1.1
		25%	441	5,902	Yes	1.14
		50%	353	5,846	Yes	1.29

cesses. The authors report performance improvements up to 171% for some applications that run on a Pentium III PC. Our adaptation scheme is simpler than Castro's scheme: our cost/benefit analysis does not have to be done each time the system accesses the compressed area.

6. CONCLUDING REMARKS

This paper represents a step toward understanding to what extend software memory compression can improve the performance of large applications. We have chosen to study a symbolic model checker and a network simulator because they constitute real applications with high CPU and memory requirements and therefore are good candidates for the compressed-memory technology. The measurements show that, for these applications, a compressed-memory system provides an increase in performance for sizes of the compressed-memory level that are within a certain range: devoting too much memory to compression hurts as much as devoting not enough. This range is determined by an application's characteristics and the system's compression performance and therefore specific adaptation is needed so that compression shows any benefit.

The measurements indicate that for memory intensive applications it is important that the system uses simple schemes to decide on the optimal size of the compressed-memory level. Good results can be obtained from a simple scheme; the scheme we present requires information that is easy to acquire on today's processors. The performance counters and a kernel module capture the memory access properties of an application, i.e. the number of accesses at each memory level. We have successfully applied the model to executions of SMV and NS2 on a commodity PC that uses main memory compression.

Although the amount of main memory in a workstation has increased with declining prices for semiconductor memories, application developers have even more aggressively increased their demands. Therefore, if the access times to memory and disk continue to improve over the next decade at the same rate as they did during the last decade, there are plenty of opportunities for software-only compressed-memory systems.

7. REFERENCES

[1] A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. In *Proc. ISCA*, pages 212–223, Munich, Germany, June 2004. ACM.

[2] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Proc. 1999 USENIX Tech. Conf.: FREENIX Track*, pages 207–218, Monterey, CA, June 1999.

[3] I. Chihaia and T. Gross. Adaptive Main Memory Compression. Technical report, ETH Zurich, 2004.

[4] I. Chihaia and T. Gross. Effectiveness of Simple Memory Models for Performance Prediction. In *Proc. ISPASS*, pages 98–105, Austin, TX, March 2004. IEEE.

[5] R. de Castro, A do Lago, and D. Da Silva. Adaptive Compressed Caching: Design and Implementation. In *Proc. SBAC-PAD*, pages 10–18, Sao Paulo, Brazil, Nov. 2003. IEEE.

[6] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Proc. Winter*

USENIX Conference, pages 519–529, San Diego, CA, Jan. 1993.

[7] M. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, 1995.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2002.

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

[10] M. Kjelso, M. Gooch, and S. Jones. Design and Performance of a Main Memory Hardware Compressor. In *Proc. 22nd Euromicro Conf.*, pages 423–430. IEEE Computer Society Press, Sept. 1996.

[11] M. Kjelso, M. Gooch, and S. Jones. Performance Evaluation of Computer Architectures with Main Memory Data Compression. *Journal of Systems Architecture*, 45:571–590, 1999.

[12] V. Schuppan and A. Biere. A Simple Verification of the Tree Identify Protocol with SMV. In *Proc. IEEE 1394 (FireWire) Workshop*, pages 31–34, Berlin, Germany, March 2001.

[13] The Network Simulator - NS2. http://www.isi.edu/nsnam/ns/.

[14] Symbolic Model Verifier. http://www-2.cs.cmu.edu/ bwolen/software/.

[15] P. Wilson. Operating System Support for Small Objects. In *Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, CA, Oct. 1991. IEEE.

[16] P. Wilson, S. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proc. 1999 USENIX Tech. Conf.*, pages 101–116, Monterey, CA, June 1999.

[17] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi. A Performance Study of BDD-Based Model Checking. In *FMCAD'98*, pages 255–289, Palo Alto, CA, Nov. 1998.