11-1-1993

The Feasibility of Using Compression to Increase Memory System Performance

Jenlong Wang
Purdue University School of Electrical Engineering

Russell W. Quong
Purdue University School of Electrical Engineering

THE FEASIBILITY OF USING COMPRESSION TO INCREASE MEMORY SYSTEM PERFORMANCE

JENLONG WANG
RUSSELL QUONG

TR-EE 93-37
NOVEMBER 1993

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

The Feasibility of Using Compression to Increase Memory System Performance

Jenlong Wang and Russell W. Quong
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
{wangj,quong)@ecn.purdue.edu

November 9, 1993

The Feasibility of Using Compression to Increase Memory System Performance

Abstract

We investigate the feasibility of using instruction compression at some level in a multi-level memory hierarchy to increase memory system performance. For example, compressing at main memory means that main memory and the file system would contain compressed instructions, but upstream caches would see normal uncompressed instructions. Compression effectively increases the memory size and the block size reducing the miss rate at the expense of increased access latency due to decompression delays. We present a simple compression scheme using the most frequently used symbols and evaluate it with several other compression schemes. On a SPARC processor, our scheme obtained compression ratio of 150% for most programs.

We analytically evaluate the impact of compression on the average memory access time for various memory systems and compression approaches. Our results show that feasibility of using compression is sensitive to the miss rates and miss penalties at the point of compression and to a lesser extent the amount of compression possible. For high performance workstations of today, compression already shows promise; as miss penalties increase in future, compression will only become more feasible.

Keywords: Memory system performance, multi-level memory system, cache, data compression.

1 Introduction

As the ratio of processor speeds to memory speeds continues to rise, design of faster memory systems has become a crucial of computer systems design. Multi-level memory hierarchies [HI88][PRHH88][PRHH89][SHL88] are the standard way to reduce average memory access time in a cost-effective manner. A memory hierarchy uses one or more levels of cache between the processor and the main memory to reduce the average memory access time. Fast, small upstream caches match the processor's speed, while larger, downstream caches reduce traffic to slower main memory.

The average access time of a cache is function of its hit time, miss rate, and miss penalty. We can reduce the miss rat': of a cache either by making cache bigger or by making the program smaller. The latter can be done in two ways.

1. Use an instruction set [FLMM87, WAF87] with a higher code density. Unfortunately, designing an instruction set is a complicated issue as it affect many areas of the system including the processor decoding complexity, and memory traffic. Also, from a commercial standpoint, a new instruction is undesirable because it will not be compatible with previous designs.

2. Compress the instruction stream. This approach are that it can be used with any processor, so that backward instruction-set compatibility can be maintained if desired. A small amount additional hardware is needed.

We consider the second approach in this paper. Namely, we investigate improving system performance by compressing instructions in a multi-level memory hierarchy. Our approach is transparent to the processor in that it sees normal instructions. We require extra hardware for runtime decompression and address translation. Use of compression will also reduce executable sizes on disk, however we are not concerned with this side effect. Finally, we do not consider compressing data, only instructions.

This paper is organized as follows. In Section 2, we illustrate our memory model using compression. In Section 3, we derive formulas showing when compression is advantageous given various parameters and show that the use of compression is feasible now for today's fastest processors. We discuss the additional hardware needed for our method in Section 4. Finally, in Section 5 we evaluate several different compression schemes.

2 Memory Hierarchy Model

In this section, we describe the memory hierarchy used in our study. Figure 1 shows the memory hierarchy models with and without compression. The memory contents before or upstream of level i, (i.e. closer to the

CPU) are the same for both approaches, so that the processor sees the same set of input symbols in both approaches. The memory contents of all levels after level i are also compressed in the compression approach. There is no architectural difference between these two approaches other than the decompression hardware. ***Compression at level i*** denotes that the decompression is done between levels $i-1$ and i as in Figure 1. In the compression approach, the compiler (or compression software) creates an executable with compressed instructions; at the runtime, decompression hardware in the memory system restores the original instructions. To be feasible, we must be able to build a fast hardware realization of the decompression algorithm.

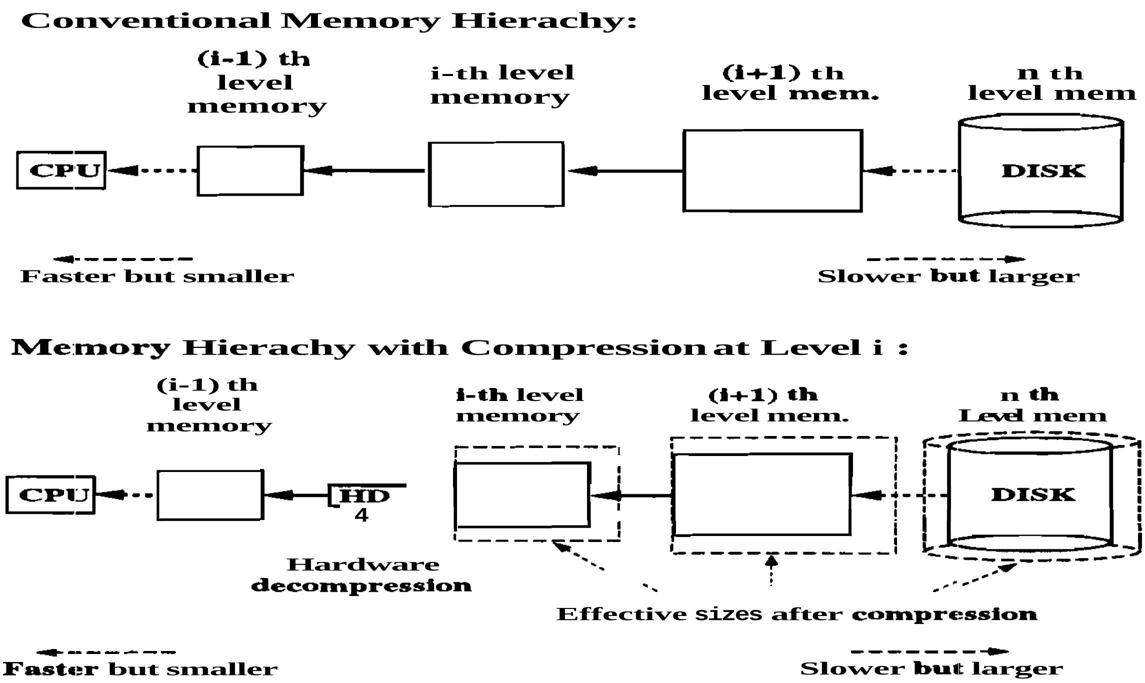

Figure 1: Memory hierarchy models for the compression and non-compression approaches.

We define the ***compression ratio*** as the increase in ***effective memory size*** increase due to compression. Thus, if the compressed instructions are 1/4 the size of the original, the compression ratio = four, because the same memory can hold four times as much information.

$$\text{Compression ratio} = \frac{\text{Original size}}{\text{Compressed size}}$$

The effectiveness of compression at a particular level depends on the following factors.

1. The capacity and miss rate of that level.
2. The miss penalty of that level.
3. The increase in miss penalty due to hardware decompression.
4. The compression ratio and the effectiveness of compression in reducing the miss rate.

For a given design space, these factors are interdependent, because as memory size increases, access time increases, but miss rate decreases. The compression ratio dictates the effective capacity increase, and the compression/decompression methods impose various decoding delays.

We use the following definitions in the rest of this paper. ***A compressed level*** is memory level that contains compressed code; a ***normal*** or ***uncompressed level*** contains uncompressed code. In comparing a memory system using compression versus a normal memory system, we make the following assumptions.

- Both systems use the same processor and the same memory organization except the compression approach has extra decompression hardware. The same memory size, associativity, block/line size, and replacement policy is used in both systems.

- The effect of memory misalignment caused by compression is neglected. The misalignment penalty can be minimized by adding hardware, and it can be considered as part of the decompression delay.

- The ***inclusion principle*** holds: The contents of level i are always in level j for all $i < j \leq n$, where n is the number of levels in the memory hierarchy.

- Uniformity condition: Compression changes the code density of all program parts equally, independent of how often they are executed. The uniformity condition is not true for individual instructions but is shown to be approximately true for extended basic blocks[ST89].

An important question to ask is "At which level should we decompress the program?" We answer this question in the next section where we will model the memory hierarchy and quantitatively predict the benefits of using compression.

3 Tradeoff Between Compression Ratio and Average Memory Access Time

Although compression increases the effective memory size, it also introduces a decoding penalty. Based on empirical data, we use a simple equation to parameterize relationship between the compression ratio and the reduction in miss rate. We use this relationship to determine the change in average access time when using compression at the i-th level memory and the effect on adjacent memory levels.

3.1 A Model for Miss Ratio versus Effective Memory Size

We assume the global miss rate at memory level i changes as the (effective) memory size raised to the power logp as described by Equation 1. Alternatively, Equation 1 shows the miss rate is reduced by the compression ratio raised to the power logp.

$$m'_i \quad = \quad p^{lg\,C} \times m_i = C^{lg\,p} \times m_i \tag{1}$$

where	m'_i	=	miss rate at level i of the new memory size and new block size.
	m_i	=	miss rate at level i of the original memory size.
	C	=	original size / compressed size = size increase ratio
	$lg\,x$	=	$log_2\,x$ function
	p	=	reduction ratio $= m'_i/m_i$ where both new size and new block size are ***twice*** of the original values.

In Equation 1, the parameter p indicates how much the miss rate decreases when both the memory size and the line is size is doubled. For example, $p = 0.3$ means that doubling the memory size and block size will reduce the miss rate to 3/10 of its previous value. Note that $0 \leq p \leq 1$ and $C > 1$. For a fixed C, as p decreases the new miss rate decreases as well. For a fixed p, as C increases the new :miss rate decreases. Therefore, smaller p values (and larger C) values are "good" in that the miss rate decreases quickly.

We used trace-driven simulation to empirically estimate the value of p for various programs. Table 1 describes the four ATUM cache traces [AGSH86] and two other traces, `spice` and `cc1` we used. Using a cache simulator, we gathered the instruction miss information for different cache and block sizes. The value of p for a 4k cache with an 8-byte block size is denoted by $p_{4k/8}$ and is given by $p_{4k/8} = \frac{m_{8k/16}}{m_{4k/8}}$, where $m_{8k/16}$ is the miss rate of an **8k** cache with block size of 16 bytes. We define the other p values similarly. Note that an important effect of compressing instructions is that the effective line size increases by C also. E.g. if $C = 4$, a 92 byte line in a compressed cache has an effective line size of 128 bytes.

Name	Total References	# I-Fetch	# Distinct I-Fetch	Include OS Instr.
cc1	1000002	757341	31195	no
spice	1000001	782764	8964	no
dec0	361982	183023	7276	yes
fora	387934	199799	8716	yes
forf	368212	190915	14123	yes
lisp	291390	169786	929	yes

Table 1: Trace used to evaluate reduction ratio.

For the these programs, we found most p values ranged from 0.4 to 0.7 as shown in Figures 2–4, which was somewhat lower than we had expected. The reason p is so low is that the compression increases both the effective memory size and the effective line size. Figure 2 shows p values for different cache sizes and a line size of 8 bytes; Figures 3 and 4 shows p values for line sizes of 32 and 2048 bytes respectively. In Figure 4, we model main memory with the 2048 byte block size corresponding to a memory page. We observe that p values are constant for each application, as the only misses occur at startup, as the memory sizes are much larger than the number of distinct addresses in the traces. Thus, the increased line size from compression accounts entirely for the reduction in miss rate.

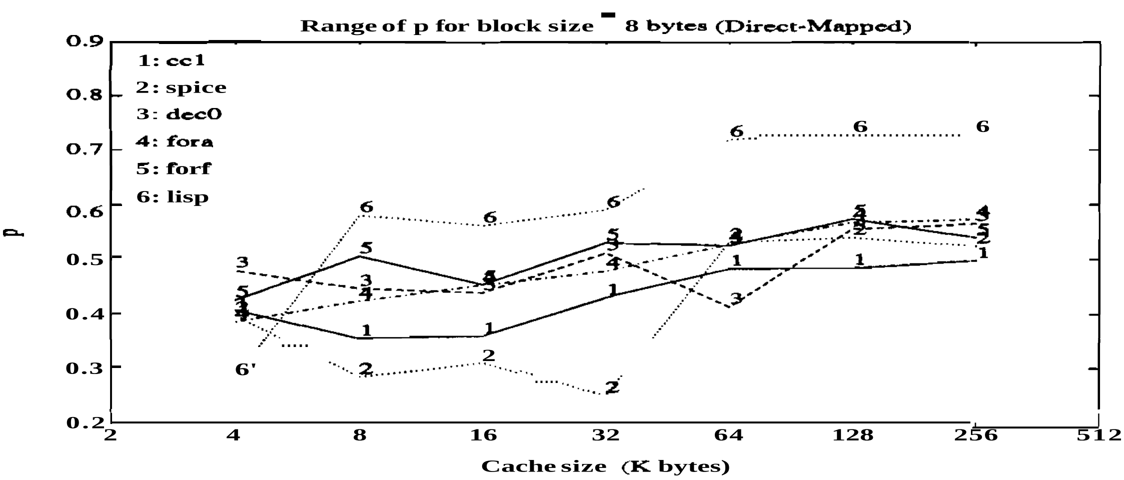

Figure 2: Range of p values in different applications.

3.2 Evaluation of Systems With or Without Compression

In this section, we analyze the average memory access time both with and without considlering block transfer time. First, we ignore block transfer time, assuming early restart and out-of-order fetch [HP90, page 458]. We then consider block transfer time. In both cases, we gives formulas for the average memory access time as a function of C, p and the decompression time d.

We use effective memory access time as our performance metric to evaluate different memory systems. Since we examine the instruction stream only, compression has no effect on the access time for data reads and writes. Hereafter, the analysis concentrates on the time for instruction fetches. In the following analysis, a subscript denotes the memory level and a superscript denotes a compression or a non-compression approach, e.g. m^c versus m^{nc}.

3.2.1 Average Memory Access Time without Block Transfer Time

The effective access time at the i-th level memory t_i is defined as

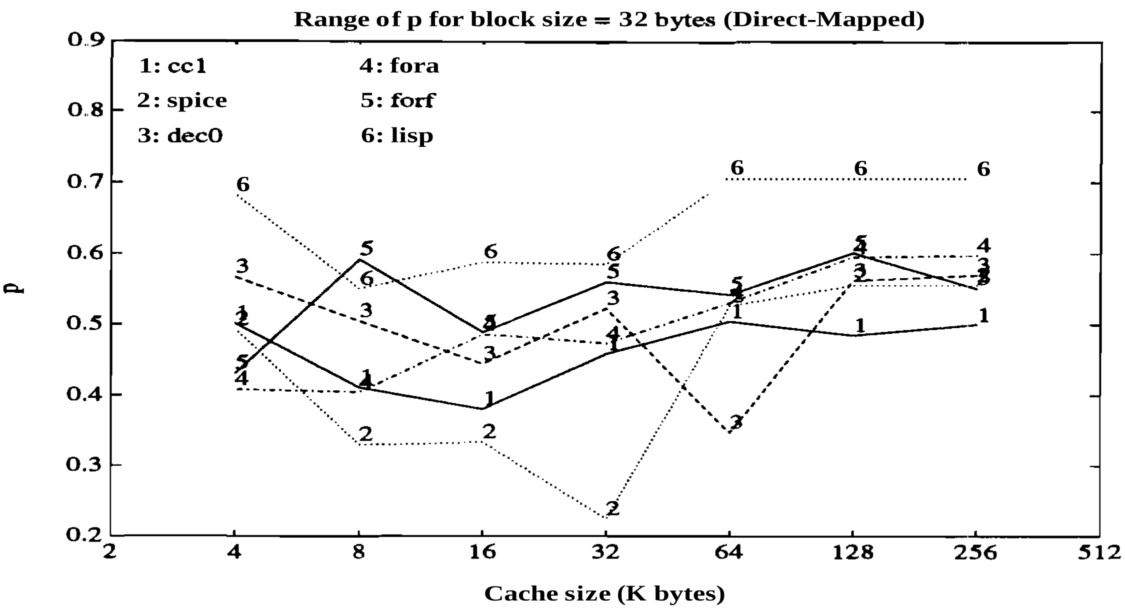

Figure 3: Range of p values in different applications.

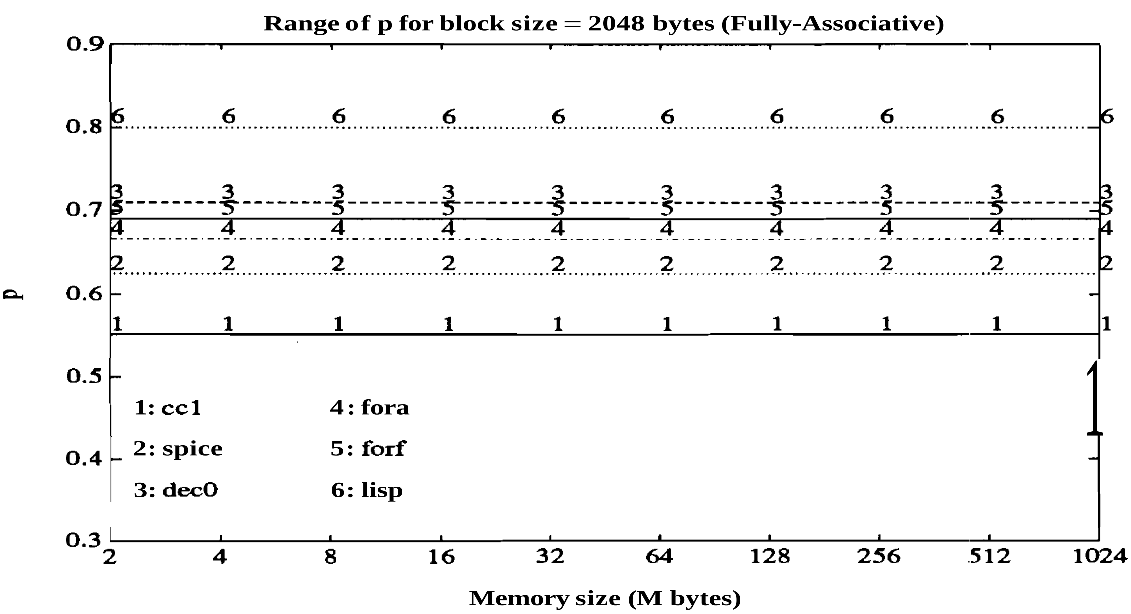

Figure 4: Range of p values for different applications.

$$t_i = h_i + m_i \times P_i \tag{2}$$

where h_i = The access time to the i-th level memory when it is a hit, on a miss from the $(i\text{-}1)$th level.

P_i = t_{i+1} = The penalty incurred when the access to the i-th level is a miss.

= The effective access time at level $i+1$.

m_i = Probability of miss at the i-th level memory. = *Local* miss *ratio* at level i.

$$= \frac{\text{Misses in the i-th level}}{\text{Memory accesses to the i-th level}}$$

The *global* miss *rate* at memory level i, M_i, is defined as:

$$\begin{aligned} M_i &= \frac{\text{Misses at the i-th level}}{\text{Memory accesses generated by the CPU}} \\ &= m_1 m_2 \cdots m_i = \prod_{j=1}^{i} m_j \end{aligned}$$

The effective memory access time of a system is t_1. In an n-level memory system, t_1 can be determined by

$$\begin{aligned} t_1 &= h_1 + m_1 \times t_2 \\ &= h_1 + m_1 \times (h_2 + m_2 \times t_3) \\ &= h_1 + m_1 \times h_2 + \cdots + (\prod_{j=1}^{i-1} m_j) t_i \qquad (3) \\ &= h_1 + \sum_{i=2}^{n-1} M_{i-1} h_i + M_{n-1} t_n \qquad (4) \end{aligned}$$

As there is no miss at the last memory level, $m_n = 0$. Hence, $t_n = h_n$. If we define $M_0 = m_0 = 1 =$ the miss rate at the CPU, then

$$t_1 = h_1 + \sum_{i=2}^{n} M_{i-1} h_i = \sum_{i=1}^{n} M_{i-1} h_i \tag{5}$$

When compression is performed at the i-th memory level, levels closer to the CPU are unaffected, i.e., miss rate and hit time of all levels before i are not affected. Hence,

$$h_j^c = h_j^{nc}, \qquad m_j^c = m_j^{nc}, \qquad M_j^c = M_j^{nc}, \qquad 1 \le j < i$$

where c denotes the compression approach and nc denotes the non-compression approach. As the only hardware difference between these two approaches is the decompression hardware between the levels $i-1$ and i, there is no access delay for levels $> i$. Hence, for levels after i, hit time is not changed

$$h_j^c = h_j^{nc}, \ \forall \ i < j \le n$$

Let $\Delta t = t_1^{nc} - t_1^c =$ the time savings using compression. Thus, the compression approach is advantageous only when

$$\Delta t = t_1^{nc} - t_1^c > 0$$

Using Equation 3 and expanding until level i, we can derive the condition

$$\begin{aligned} \Delta t &= (\prod_{j=1}^{i-1} m_j)(t_i^{nc} - t_i^c) \qquad (6) \\ &= M_{i-1}(t_i^{nc} - t_i^c) > 0 \qquad (7) \end{aligned}$$

for when using compression is favorable. Because $M_{i-1} > 0$ and M_{i-1} is independent of compression, the only difference between the approaches is t_i^{nc} and t_i^c. Equation 2 indicates that t_i is a fuiiction of h_i, mi, and t_{i+1}. giving a recursive dependence down to level n. We use the following lemma to simplify the recurrence relation.

The following lemmas prove that compression does not change the local miss rates and the effective access time of compressed memory levels after level i. We then derive a tradeoff condition to judge when the compression approach gives better performance than the non-compression counterpart.

Applying Equation 1 to the global miss rate, we obtain

$$M_j^c = p^{lg\,C} M_j^{nc}, \qquad i \leq j \leq n.$$

Lemma 1 : $m_j^c =$ my^c, *for* $i + 1 \leq j \leq n$

Proof: By induction on j from $i + 1$ to n.

Basis:

Since $M_j^c = p^{lg\,C} M_j^{nc}$, $\forall$ j such that $i \leq j \leq$ n and $M_j = \prod_{l=1}^{j} m_l$

$$\begin{aligned}
M_{i+1}^c &= m_1^c \cdots m_i^c m_{i+1}^c \\
&= m_1^{nc} \cdots m_{i-1}^{nc} \times p^{lg\,C} m_i^{nc} \times m_{i+1}^c \\
M_{i+1}^{nc} &= m_1^{nc} \cdots m_{i-1}^{nc} \times m_i^{nc} \times m_{i+1}^{nc} \\
\frac{M_{i+1}^c}{M_{i+1}^{nc}} &= \frac{m_i^{nc} p^{lg\,C} \times m_{i+1}^c}{m_i^{nc} \times m_{i+1}^{nc}} = p^{lg\,C}
\end{aligned}$$

Hence, $m_{i+1}^c = m_{i+1}^{nc}$.

Hypothesis: Assume that $m_j^c = m_j^{nc}$, $\forall i + 1 \leq j \leq 6$.

Induction:

$$\frac{M_{k+1}^c}{M_{k+1}^{nc}} = \frac{m_i^{nc} p^{lg\,C} \times m_{i+1}^c \cdots m_k^c \times m_{k+1}^c}{m_i^{nc} \times m_{i+1}^{nc} \cdots m_k^{nc} \times m_{k+1}^{nc}} = p^{lg\,C}$$

Hence, $m_{k+1}^c = m_{k+1}^{nc}$. Therefore, $m_j^c = m_j^{nc}$, $\forall$ j such that $i + 1 \leq j \leq n$. ∎

Lemma 2 : $t_j^c = t_j^{nc}$, *for* $i + 1 \leq j \leq n$

Proof: (by induction)

Basis:

$$t_n^c = t_n^{nc} = \text{h,, because } m_n^c = m_n^{nc} = 0$$

Hypothesis: Assume that $t_j^c = t_j^{nc}$, $\forall$ 6 $\leq$ **j** $\leq$ **n**

Induction: Recall that

$$\begin{aligned}
t_{k-1}^c &= h_{k-1}^c + m_{k-1}^c t_k^c, \text{ and} \\
t_{k-1}^{nc} &= h_{k-1}^{nc} + m_{k-1}^{nc} t_k^{nc}
\end{aligned}$$

Since the compression is performed at level i, we can obtain that

$$h_j^c = h_j^{nc} = h_j, \ \forall\, i < j \leq n, \text{and}$$

and from the previous lemma, $m_j^c = m_j^{nc}$, $\forall\, i < j \leq$ n. Hence,

$$t_{k-1}^c = t_{k-1}^{nc}$$

Therefore, $t_j^c = t_j^{nc}$, $\forall\, i < j \leq n$. ∎

From Lemmas 1 and 2, the difference between t_i^{nc} and t_i^c thus relies only on the following.

The compression ratio at the i-th level.

The miss ratio at the i-th level for compression and non-compression approaches

The access delay at the i-th level introduced by the decompression hardware.

Note the memory access time and miss rate of level j, for all $j > i$, have no effect on At. When compression at the i-th level memory is advantageous, the following conditions can be derived using Lemmas 1 and 2:

$$\begin{aligned} t_i^c &< t_i^{nc}, \\ h_i^c + m_i^c t_{i+1}^c &< h_i^{nc} + m_i^{nc} t_{i+1}^{nc} \\ h_i^c - h_i^{nc} &< m_i^{nc} t_{i+1}^{nc} - (p^{lg\,C} m_i^{nc}) t_{i+1}^c \end{aligned} \tag{8}$$

Letting $d = h_i^c - h_i^{nc} =$ the delay due to decompression, and using $t_{i+1}^c = t_{i+1}^{nc} = t_{i+1} = P_i$, the savings from using compression is

$$t_1^c - t_1^{nc} = M_{i-1}\left[m_i^{nc} P_i (1 - p^{lg\,C}) - d\right] \tag{9}$$

Thus, compression at level i is advantageous when

$$d < m_i^{nc} t_{i+1} (1 - p^{lg\,C}) = m_i^{nc} P_i (1 - p^{lg\,C}) \tag{10}$$

Equation 10 indicates that d is directly proportional to mi, miss rate at level i, and t_{i+1}, the access time of level $i + 1$. For example, if m_i^{nc} or t_{i+1} doubles, d is doubled. We now assess two extreme cases, $i = n$ and $i = 1$ as examples as a intuitive check of our analysis.

3.2.2 Cariestudiesofmemory systems

As examples, we evaluate using compression at the extreme ends of the memory hierarchy, namely at the L1 cache and at secondary storage. We then study the general case, showing our approach is theoretically feasible when used at main memory for next-generation processors.

Case **1:** $i = 1$. Compression is done at the first level cache so that Equation 10 becomes

$$d < m_1^{nc} t_2 (1 - p^{lg\,C})$$

In order to assess the feasibility of compression at this level, we use the parameters from Table 2 which are typical in the early 1990's according to [HP90]. Using Equation 2, $t_2 = h_2 + m_2 P_2 = 8.5 - 34$ cycles. Let $p = 0.5 - 0.8$, and $C = 1.2 - 2.5$, then the extreme values are

$$m_1^{nc} t_2 (1 - p^{lg\,C}) = 0.002 - 3.34 \text{ cycles}$$

Even for an optimistic case where $C = 2.0$ and $p = 0.5$,

$$m_1^{nc} t_2 (1 - p^{lg\,C}) = 0.04 - 3.4 \text{ cycles}$$

As a ballpark figure, the allowed decompression delay for a 100 MHz processor would be .02–33.4 nS. Because of the very :short latency allowed for hardware decompression, compression at the first llevel cache is simply not feasible

m_1^{nc}	h_2^{nc}	m_2^{nc}	$P_2^{nc} = t_3^{nc}$
1% – 20%	4 – 10 cycles	15% – 30%	30 – 80 cycles

Table 2: Parameters for Case 2

Case **2:** $i = n$, i.e., compression is done at the n-th memory level. E.g. the filesystem contains compresses executables, but memory holds normal executables. As $m_n = 0$, Equation 10 gives $d < 0$, which means that any decompression delay slows down memory performance.

Thus, compression at the n-th level degrades average memory access time, which is expected. Although memory response time is not improved by doing compression at the last level, delay is much less than the hit time on the n-th level. Typically t_n^{nc}, the conventional disk access time, is in the range from 8 ms to 20 ms. For a 100 MHz processor, the disk access time is in the range from 8×10^5 – 2×10^6 cycles. The value of d is due to extra hardware decompression delay. In other words, $t_n^c \approx t_n^{nc} \gg$ d. Hence,

$$t_1^c - t_1^{nc} = (\prod_{j=1}^{n-1} m_j)(t_n^c - t_n^{nc}) \approx 0$$

Consequently, the only advantage of compression at this level is to save space.

Case 3: A general analysis. Figure 5 shows the maximum allowed decompression delay if using compression. is to be effective. For the particular set of parameters (p = 0.7, m x P = 300 CPU cycles), points on the '--' curve show where compression neither helps nor hurts the average memory access time. Points below the curve favor compression. As Section 5 will show, we can obtain $C \approx 1.5$, so that the maximum allowed decompression delay is about 60 CPU cycles. As d is proportional to $m \times P$, we can calculate where compression. is advantageous by simply scaling the graph. For example, if p = 0.7, m x P = 600 CPU cycles, then d = 120 cycles. From the graph, if the original design has a large miss rate and the miss penalty is large, a connpression approach gains significant improvement. Clearly, as C grows, compression becomes more feasible.

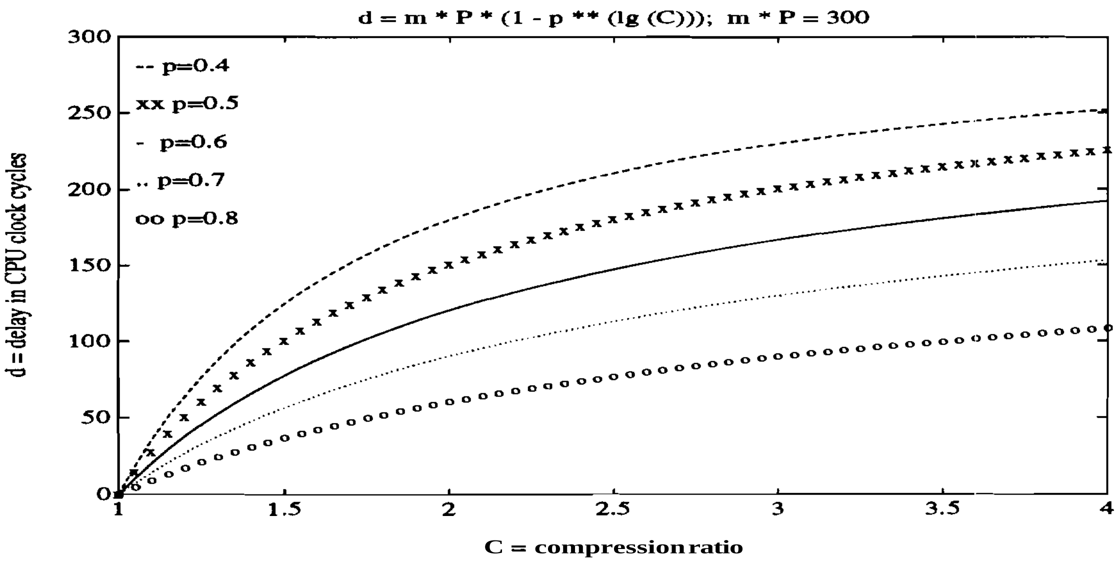

Figure 5: The tradeoff conditions with various miss ratios and miss penalties. P = miss penalty to access level i + 1. m = miss rate at level i. p= reduction ratio.

We empirically estimated m x P by measuring local miss ratios using trace-driven simulations. The miss ratios and memory organization are shown in Table 3. To calculate the average access time and miss penalty of each memory level, we assumed a 200 MHz processor with a disk access time of 5ms – 15ms and a memory system with parameters similar to that in [HP90] as shown in Table 4. In this design, we observe that $m \times P$ value for the second level cache range from impractically small (1–3 cycles) to moderate (20–56 cycles). However, m x P for main memory is large (300 CPU cycles) even with the most pessimistic miss rate measured and the least disk access time (5ms). With a moderate miss rate 0.05% and 9ms disk access time, the m x P :: 900 CPU cycles which makes compression a distinct possibility. For example, with p = 0.7 and C = 1.5, the maximum allowed d is 180 CPU cycles. For a near-future CPU running at 400 MHZ, using compression becomes even more attractive.

Memory level Size (bytes)	First level cache 8K – 256K	Second level cache 256K – 1M	Main memory 1M – 1G
Block size (bytes)	4 – 128	4 – 256	256 – 8192
Associativity	Direct	Direct – Fully associative	Fully associative
	local miss rate (%)		
cc1	0.3100 – 22.3000	0.2076 – 0.4548	0.0305 – 0.0638
dec0	0.4200 – 16.3600	0.2249 – 0.4548	0.0519 – 0.157
fora	0.3100 – 8.6700	0.4245 – 0.6000	0.0505 – 0.1290
forf	0.7300 – 14.7500	0.4466 – 0.4637	0.0477 – 0.1411
lisp	0.3200 – 1.7500	0.0611 – 0.0906	0.2710 – 0.3103
spice	0.0600 – 9.0900	0.1276 – 0.5667	0.0293 – 0.0882

Table 3: Local miss rates in % of various applications.

CPU speed	200 MHz		400 MHZ	
Main memory hit time	50 ns		25ns	
Disk access time	5 ms – 15 ms			
Memory level	2nd level cache	main memorv	2nd level cache	main memory
Avg. access time (cycles)	5 – 66	310 – 9310	6 – 122	610 – 18610
Block transfer time (cycles)	2 – 22	10^4 – 10"	2 – 22	$2 \times 10^4 - 2 \times 10^5$
m = Local miss rate (%)	0.06 – 0.6	0.03 – 0.31	0.06 – 0.6	0.03 – 0.31
P = Miss penalty (cycles)	310 – 9310	$10^6 - 3 \times 10^6$	610 – 18610	$2 \times 10^6 - 6 \times 10^6$
$m \times P$ (cycles)	1 – 56	300 – 9300	1 – 112	600 – 18600

Table 4: Design parameter sets.

3.2.3 Average Memory Access Time with Block Transfer Time

The effective access time at the i-th level memory t_i is still defined as $t_i = h_i + m_i \times P_i$. Every term in this equation remains the same except P_i, which is now defined as

$$\begin{aligned}
P_i &= t_{i+1} + x_{i+1} \\
&= \text{The penalty incurred when the access to the i-th level is a miss.} \\
t_{i+1} &= \text{The latency time to obtain the first data from level i + 1.} \\
x_{i+1} &= \text{Time to transfer a block from level } i + 1 \text{ to level } i. \\
&= \frac{B_i}{X_{i+1}} = \frac{\text{Block size at level } i.}{\text{Transfer rate from level i + 1 to level i.}}
\end{aligned}$$

Equation 10 remains applicable, giving the following bound for d_B for increased memory performance.

$$\begin{aligned}
d_B &< m_i^{nc} t_{i+1}(1 - p^{lg\,C}) + m_i^{nc} x_{i+1}(1 - p^{lg\,C}), \qquad \text{or} \\
d_B &< m_i^{nc}(1 - p^{lg\,C})(t_{i+1} + x_{i+1}) = m_i^{nc} P_i(1 - p^{lg\,C}) \qquad (11)
\end{aligned}$$

As shown in Equation 11, the delay time allowed for decompression is increased when block transfer time can not be hidden by mechanisms such as ***early restart*** and ***out-of-order fetch.***

4 Design of Memory Systems Using Compression

Our compression method requires additional hardware for two reasons, runtime instruction decompression and translation of uncompressed addresses to compressed addresses. For any compression algorithm, we refer to the mapping from normal symbols to compressed symbols as the ***codebook.*** We maintain an address table and a codebook for each process as shown in Figure 6.

The address translation problem occurs because the compressed instruction stream does not preserve a linear addressing space. Thus, if we branch to (uncompressed) address A, where do we find A among the compressed instructions? The ***address mapping table*** contains an index into the compressed instructions for each cache index. For example, if the L2 cache has a line size of 256 bytes and addresses are 32 bits (4 bytes) wide, then the address table would contain a 32-bit index into the compressed program for every 256 bytes of code. Thus, the address table would be be $4/256 = 1.6\%$ of the original program size. As we shall see, this additional overhead reduces the effective compression ratio.

Figure 6 shows that the hardware leaves the data stream intact. Decompression hardware also tracks whether a program is in compressed form. A selective bypassing capability allows the system to run uncompressed programs. In our example, we have assumed all caches are virtually addressed. The decompression hardware stores a copy of the current codebook.

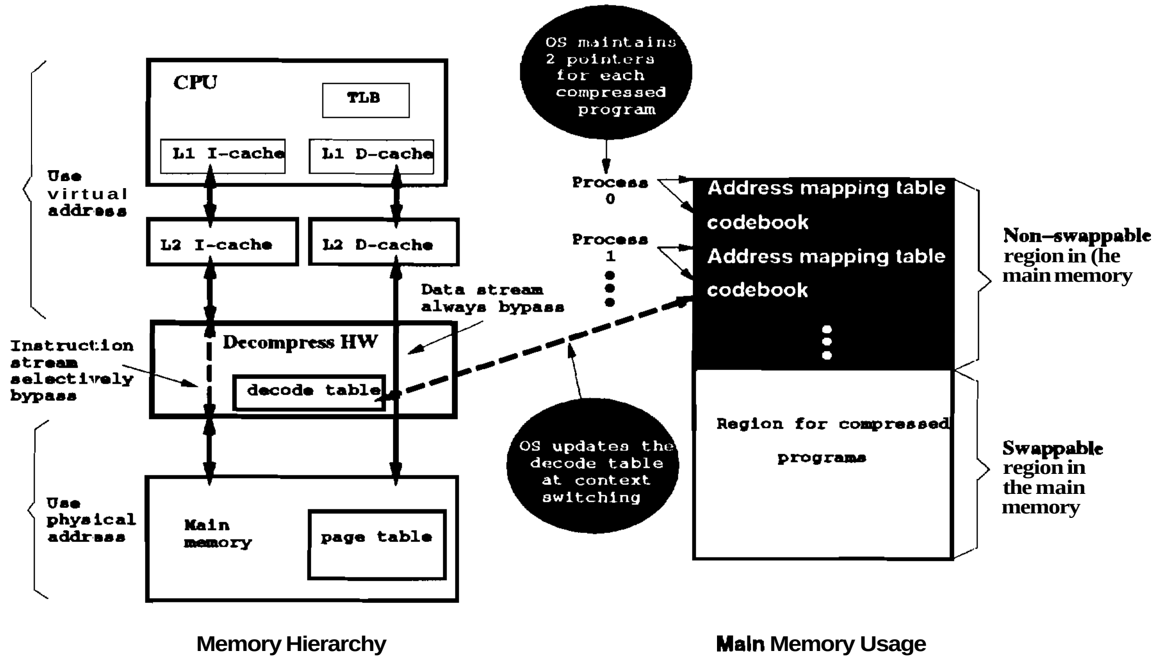

Figure 6: Design of memory system with compression occuring at main memory.

The operating system stores the codebook and the address mapping table in a non-swappable region in the main memory. On a context switching, the operating system must reload the decode table with the appropriate codebook.

As an example, we illustrate the sequence of actions for a L2 cache miss for virtual address A.

- A hit at main memory: Look in the address mapping table for the L2 line holding address A. Read the index X into the compressed instructions in main memory. Since this table is never swapped out, it cost an extra main memory access latency. The decompression hardware then starts decompression at index X in main memory.

- A page fault: After translating the virtnal address to a physical address, the operating system detects a page fault. The page of compressed instructions is loaded from disk into main memory. We then proceed as above when there is a hit on main memory.

Both the address mapping table and codebook require space in main memory and must be saved with the compressed program in the filesystem. Therefore, the actual compression ratio must be adjusted. Figure 7 shows the adjusted compression ratio as a function of compression ratio and the mapping overhead. Most current workstations have L1 caches of size greater than 32 bytes or 8 instructions for most RISC processors [HP90]. Thus, the overhead of address mapping will be less than 1/8. To be effective, the L2 I-cache typically will have much larger size and line size than the L1 I-cache. As the cache size in most workstation is increased, the block size will change correspondingly. We expect the overhead to be less than 1/32. For $C = 1.5$, the adjusted compression ratio is 1.4.

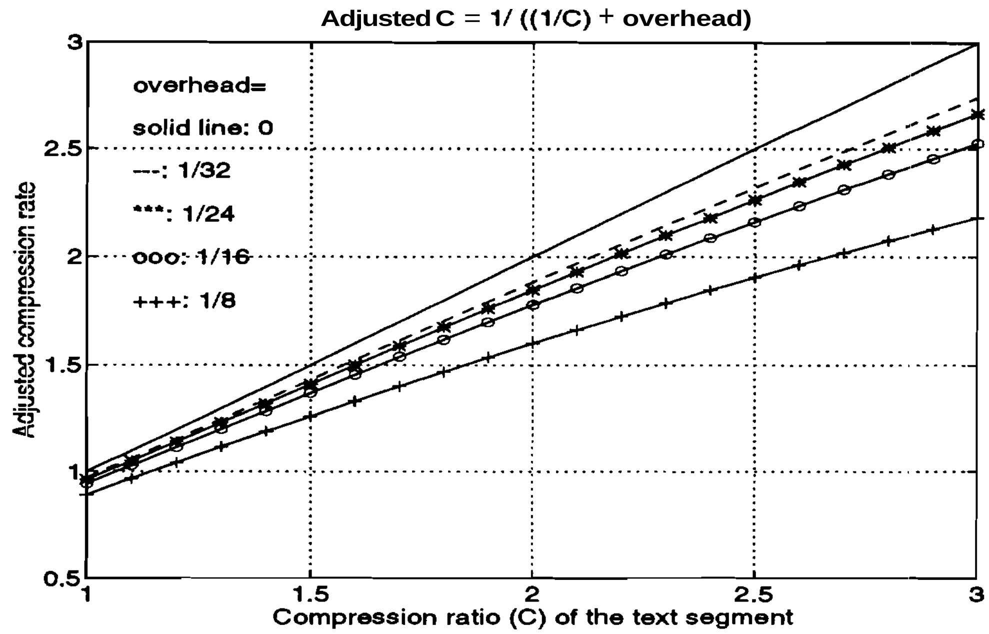

Figure 7: Adjusted compression ratio.

5 Compression Methods and Basic Compression Unit

In this section, we address compression requirements and the choice of the smallest unit of code to be compressed. We also present measurements for a simple compression method suitable for use in a memory system.

5.1 Decompression requirements

Data compression [Hu52][LeH87][St88][We84] has been used extensively to reduce data storage and transmission costs. Recently, data is compressed on secondary storage, with the slight time penalty needed for decompression more than offset by the increase in disk space. As an example, the operating system MS-DOS 6.0 contains a file-compression utility. These utilities compress the entire executable including instructions, data, and the symbol table. Before execution, the entire program is decompressed and copied to main memory. As we have seen compression at the file system level must degrade memory system performance.

Because we will be decompressing fragments (i.e. a cache line) of a program at runtime, we require a compression scheme that requires minimal synchronization between compression and decompression. On a cache miss to instruction **I**, the system must locate **I** among the compressed instructions and decompress **I**, filling the appropriate upstream cache slots. As **I** might be the target of a branch, **I** can have an arbitrary address and the system might not have decompressed neighboring instructions. Thus, the Ziv-Lempel-Welch (LZW) algorithm [We84] is unsuitable because it uses a dynamic codebook for compression/decompression that is built during a sequential pass over the data.

We only compress read-only items, such as instructions or read-only data. Thus, we do not consider compressing the writable data stream because data changes as the program executes so that it would have to be recompressed during a write. We do not know of any fast, effective technique that can compress small amounts (a cache line) of dynamically changing data. By considering only read-only items, the compression can be done at compile/link time. At runtime we only need to do decompression.

Our last consideration was the size of the basic compression unit (BCU). A small BCU offers little opportunity for compression, as there is little repeated information. Hence, a basic block in a program, normally 3–9 instructions, is not an effective BCU. A small sized procedure has the same potential drawback, and there is no guarantee a program will not have small procedures. In addition, procedure calls and returns complicate the use of a procedure as the BCU. Thus, we use an entire program as the BCU.

5.2 Experimental Compression Ratio

In this section, we compare the compression ratios of several compression methods on various Unix executables. We used the entire text segment from an executable as the BCU and fed it to the compression algorithms. The compression ratios are measured on executable files of a SUN SPARC workstation running SUN-OS 4.1.1.

After some experimentation, we found that independent compression of the different fields of a machine instruction performed well. We broke down each instruction by its fields (opcode, operand, jump displacement, immediate value, etc.) [Ro90] and compress each field. For example, an opcode "LD", a register "r31", and an immediate value "#4095" all belong to different fields. Each instruction uses only some of the fields; e.g. a ADD instruction would not have a jump displacement field. We used this approach of compressing fields on all the following strategies except LZW.

- Most frequently used (MFU): For each field, we used a f-cache (field-cache) of fixed-size preloaded with the most frequent values for that field. E.g. an opcode f-cache of size four might be preloaded with $\boxed{0 : \text{empty}}$, $\boxed{1 : \text{LOAD}}$, $\boxed{2 : \text{STORE}}$, $\boxed{3 : \text{BRANCH}}$. In the compressed instruction stream, each field is an index into the appropriate f-cache. In the event the field value is not in the f-cache, we use a special index (say 0) followed by the actual (uncompressed) value. Thus, the most frequently used instructions are represented by f-cache indices, and all others result in f-cache misses. Indices into the f-caches are shorter than the actual fields giving compression.

  For each field, we tried different f-cache sizes (always powers of two) and we selected the size providing the best compression. The sizes of the f-caches differed depending on the field. Larger f-cache sizes reduce the "miss rate" increasing compression, but require larger indices decreasing compression.

  The MFU method is ideal for use in a memory system, as the decompression hardware is always "in sync" because the MFU f-caches are fixed. MFU lends itself to a straight forward implementation of the decompression hardware.

- Static Huffman coding: We estimated the effect of independently compressing each instruction field via Huffman coding. We underestimated the compressed size by calculating the entropy of each field and then adding the space required for the Huffman trees.

- Compression bound: We calculated the entropy for each field, giving a theoretical upper bound for compression schemes that independently compress each field. For field k (say the jump displacement field) with possible values $f_1, f_2, \ldots, f_n$, the entropy is $\mathcal{H}_k = \sum_{i=1}^{n} - \Pr(f_i)\log_2 \Pr(f_i)$, where $\Pr(f_i)$ is the probability of f_i occurring, given that field k exists. The entropy for the entire instruction is the sum of the entropies for each field. Note that by adding the space for a Huffman encoding tree, we get the Huffman bound.

  While better compression might be possible by viewing instructions differently, our measurements indicate our bound is fairly good (making it difficult to beat in practice).

- Lempel-Ziv-Welch (LZW) [We84]. We also measured the popular LZW algorithm used by the UNIX utility. `compress`. The LZW result is used only as a comparison point as LZW is unsuitable for our purposes, as previously mentioned in Section 5.1.

5.3 Results of Compression Methods

Filename	Orig size	MFU + f-cache size	MFU only size	Comp ratio	Huffman size	Comp ratio	Comp bound	Comp ratio	LZW size	Comp ratio
chgrp	4680	3316	2961	141.1	3087	151.6	2076	225.4	3218	145.4
cmp	3968	2816	2547	140.9	2614	151.8	1720	230.7	2704	146.7
cp	5616	3956	3601	142.0	3674	152.9	2412	232.8	3826	146.8
env	2224	1688	1563	131.8	1552	143.3	978	227.4	1602	138.8
hostid	1096	843	808	130.0	795	137.9	469	233.7	784	139.8
kill	2856	2135	2021	133.8	1983	144.0	1289	221.6	1975	144.6
ldd	2864	2131	1771	134.4	1965	145.8	1232	232.5	2026	141.4
xfig	1040384	806122	801482	129.1	717880	144.9	556048	187.1	660188	157.6
bash	278528	196290	193826	141.9	173741	160.3	128120	217.4	164398	169.4
bibtex	122880	81969	81095	149.9	71765	171.2	53339	230.4	75547	162.7
archie	32768	17651	17225	185.6	16183	202.5	11108	295.0	16959	193.2
detex	24576	14032	13589	175.1	12546	195.9	8800	279.3	12550	195.8
dvips	98304	68330	67516	143.9	61698	159.3	45961	213.9	63933	153.8
f2c	270336	181267	180098	149.1	158632	170.4	111538	242.4	137398	196.8
flex	81920	54209	53419	151.1	48681	168.3	35976	227.7	50490	162.2
gcc	49152	29946	29504	164.1	26080	188.5	17770	276.6	24127	203.7
gdb	466944	326756	324626	142.9	287185	162.6	200742	232.6	240985	193.8
yacc	49152	29723	29396	165.4	26765	183.6	18992	258.8	27594	178.1
SCC	212992	156458	155195	136.1	140014	152.1	104506	203.8	138629	153.6
as	180224	128720	127558	140.0	113925	158.2	84536	213.2	109619	164.4
cc	81920	53524	52978	153.1	47037	174.2	36586	223.9	46646	175.6
cpp	163840	117271	116577	139.7	106970	153.2	78687	208.2	100479	163.1
ex	196608	144315	141788	136.2	128144	153.4	96002	204.8	131211	149.8
ld	122880	83341	82078	147.4	73754	166.6	54828	224.1	73017	168.3
nawk	106496	77646	76853	137.2	70561	150.9	54359	195.9	73603	144.7
emacs	1327104	880325	875902	150.8	783179	169.5	587737	225.8	706037	188.0
idraw	786432	549372	545286	143.2	469971	167.3	366864	214.4	351191	223.9
ghostview	835584	645307	640875	129.5	577669	144.6	438736	190.5	532519	156.9
virtex	237568	171996	170421	138.1	153821	154.4	115574	205.6	159686	148.8
f77	57344	36406	35860	157.5	32115	178.6	24938	229.9	31923	179.6
pc	196608	147094	145831	133.7	131868	149.1	98107	200.4	126343	155.6
m2c	57344	35966	35420	159.4	31789	180.4	24431	234.7	31670	181.1
csh	139264	97356	95874	143.0	87119	159.9	65849	211.5	87775	158.7
sh	90112	61050	59784	147.6	54384	165.7	36740	245.3	51393	175.3
lex	49152	30844	30293	159.4	27782	176.9	19974	246.1	28762	170.9

Table 5: Experimental compression ratios. All sizes in bytes; all compression ratios in percent.

Table 5 shows compression ratios of various files on a SUN4. The original size of the text (code) segment, is listed. The size for MFU compression includes the space for the preloaded f-cache values. For smaller programs, the overhead due to the preloaded f-caches significantly decreased the compression ratio. For larger programs, MFU had an compression ratio of roughly 150%, including the space for the f-caches codebook.

The size for static Huffman coding includes the size of the Huffman tree. The compression bound gives the projected best possible compression. The majority of the difference between Huffman encoding and the compression bound is due to the Huffman tree, which amounts to roughly 1/3 of the compression bound size. The compression ratio of Huffman encoding is always greater than the simple MFU encoding.

We also listed the compression ratio of LZW compression method. For small to medium size programs, the Huffman encoding performs slightly better than LZW encoding. For large programs, LZW usually gives better compression ratios.

6 Conclusion

We have analyzed the effect of using compression in a memory system on the average system access time. We have found that if a compression ratio of around 1.5 can be achieved, compression is feasible at main memory for computers of today. We also found that the benefit from compression is quite sensitive to the miss ratio and miss penalty at the level of compression.

We proposed a memory system design to deal with instruction decompression and address translation and suggested OS support for this particular design. This design is capable of running compressed and uncompressed programs. This capability provides a way to utilize compression when it improves memory performance.

We have also measured the compression ratios of several different compression techniques. A simple compression method using a f-cache of MFU values achieved compression ratios of 15096. A static Huffman encoding gives even better compression ratios. With miss penalties increasing in future systems, we believe using compression in the memory system will only become more viable as time progresses.

7 Acknowledgements

We thank Glary Lauterbach for his comments.

References

[AGSH86] Agarwal, A., Sites, R., and Horowitz, M. ***ATUM: A New Technique for Captaring Address Traces Using Microcode.*** Proceedings of the 13th Annual Symposium on Computer Architecture, June 1986, pp. 119-127.

[FLMM87] Flynn, M. J., Mitchell, C., Mulder, H., ***And Now a Case for More Complex Instruction Sets.*** IEEE Computer, Sep. 1987, pp. 71-83

[HP90] Hennessy, J. L., Patterson, D. A., ***Computer Architecture: A Quantitative Approach.*** Morgan Kaufmann Publishers, 1990.

[HI88] Hill, M. D., ***A Case for Direct-Mapped Caches.*** IEEE Computer, Dec. 1988, pp. 25-40.

[HU52] Huffman, D. A., ***A Method for the Construction of Minimum-Redundancy Codes.*** Proc. IRE, 40(9), 1952, pp. 1098-1101.

[LEH87] Lelewer, D. A., Hirschberg, D. S., ***Data Compression.*** ACM Computing Surveys, Vol. 19, No. 3, Sep. 1987, pp. 261-296.

[PRHH88] Przybylski, S., Horowitz, M., Hennessy, J., ***Performance Tradeoffs in Cache Design.*** Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp. 290-298.

[PRHH89] Przybylski, S., Horowitz, M., Hennessy, J., ***Characteristics of Performance-Optimal Multi-Level Cache Hierarchies.*** Proceedings of the 16th Annual International Symposium on Computer Architecture, 1989, pp. 114-121.

[PR90] Przybylski, S., ***Cache and Memory Hierarchy Design: a Performance-Directed Approach.*** Morgan Kaufmann Publishers, 1990

[RO90] ROSS Technology, Inc., ***SPARC RISC User's Guide.*** Cypress Semiconductor Corporation, 2nd Ed., Feb., 1'990

[SHL88] Short, R. T., Levy, H., A *Simulation Study of Two-Level Caches.* Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp. 81-88.

[SM82] Smith, A. J., *Cache Memories.* ACM Computing Survey, Vol.14, No. 3, Sep. 1982.

[ST89] Steenkiste, P., *The Impact of Code Density on Instruction Cache Performance.* Proceedings of the 16th Annual International Symposium on Computer Architecture, 1989, pp. 252-259.

[ST88] Storer, J. A., *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[WAF87] Wakefield, S. P., Flynn, M. J., *Reducing Execution Parameters Through Correspondence in Computer Architecture.* IBM J. Res. Develop., Vol. 31, No. 4, July 1987, pp. 420-434

[WE84] Welch, T. A., A *Technique for High-Performance Data Compression.* IEEE Computer, Jun. 1984, pp. 8-19.