Partitioned Compressed L2 Cache

by

David I. Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 2001
[February 2002]

© Massachusetts Institute of Technology 2001. All rights reserved.
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
October 12, 2001

Certified by .....
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 3 1 2002
LIBRARIES

BARKER

Partitioned Compressed L2 Cache

by

David I. Chen

Submitted to the Department of Electrical Engineering and Computer Science
on October 12, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The effective size of an L2 cache can be increased by using a dictionary-based compression scheme. Since the data values in a cache greatly vary in their "compressibility," the cache is partitioned into sections of different compressibilities. For example, the cache may be partitioned into two roughly equal parts: two-way uncompressed cache having 32 bytes allocated for each line and an eight-way compressed cache have 8 bytes allocated for each line. While compression is often researched in the context of a large stream, in this work it is applied repeatedly on smaller cache-line sized blocks so as to preserve the random access requirement of a cache. When a cache-line is brought into the L2 cache or the cache-line is to be modified, the line is compressed using a dynamic, LZW dictionary. Depending on the size of the compressed string, it is placed into the relevant partition.

Some SPEC-2000 benchmarks using a compressed L2 cache show an 80% reduction in L2 miss-rate when compared to using an uncompressed L2 cache of the same area, taking into account all area overhead associated with the compression circuitry. For other SPEC-2000 benchmarks, the compressed cache performs as well as a traditional cache that is 4.3 times as large as the compressed cache, taking into account the performance penalties associated with the compression.

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Acknowledgments

I would like to thank my thesis advisor Professor Larry Rudolph for his guidance and encouragement. Whenever I hit a wall, he was quick to distill the problem and find solutions. I am also grateful for Professor Srinivas Devadas's advice.

Thanks to fellow students Prabhat Jain, Josh Jacobs, Vinson Lee, Daisy Paul, Enoch Peserico, and Ed Suh for their friendship and for being around to bounce ideas off of. Thanks to Derek Chiou for his help with writing and modifying cache simulators. Thanks to Todd Amicon for settling administrative details painlessly and Daisy for being a fun officemate and supplying generous amounts of Twizzlers.

I would like to thank my parents and my sister Clara, my friends Anselm Wong, Hau Hwang, Jeffrey Sheldon, and many others for their support.

This research was performed as a part of the Malleable Caches group at the MIT Laboratory for Computer Science, and was funded in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

Contents

1 Introduction **9**

1.1 Related Work 11

1.2 PCC in the Context of Related Work 13

2 Motivation **15**

2.1 Compaction 15

3 The Partitioned Cache Compression Algorithm **18**

3.1 The LZW Algorithm 19

3.2 PCC 19

3.3 PCC Compression and Decompression of cache lines 22

3.4 PCC Dictionary Cleanup 23

3.5 Managing the storage 24

3.6 Alternate Compression Methods 26

3.6.1 LZ78 28

3.6.2 LZW 28

3.6.3 LZC 28

3.6.4 LZT 29

3.6.5 LZMW 29

3.6.6 LZJ 29

3.6.7 LZFG 30

3.6.8 X-Match and X-RL 30

3.6.9 WK4x4 and WKdm 31

3.6.10 Frequent Value 31
3.6.11 Parallel with Cooperative Dictionaries 32

4 The Partitioned Compressed Cache Implementation 33

4.1 Dictionary Latency vs. Size Tradeoff 33
4.2 Using Hashes to Speed Compression 34
4.3 Decompression Implementation Details 36
4.4 Compression Implementation Details 37
4.5 Parallelizing Decompression and Compression 40
4.6 Other Details 42

5 Results 44

5.1 Characteristics of Data 44
5.2 Performance metrics 47
5.3 Simulation Environment 48
5.4 PCC performance 50
5.5 Increased Latency Effects 57
5.6 Usefulness of compressed data, effect of partitioning 59

6 Conclusion 61

A Latency Effects Specifics 64

List of Figures

3-1 PCC dictionaries and sample encoding 20

3-2 Pseudocode of LZW compression 21

3-3 Pseudocode to extract a string from a table entry 21

3-4 Pseudocode for dictionary cleanup 24

3-5 Sample partitioning configuration and sizes 25

3-6 PCC access flowchart 27

4-1 Decompression logic 35

4-2 Compression logic 37

4-3 Sample hash function 38

5-1 Data characteristics histograms 45

5-2 `mcf` and `equake` IMREC ratio over time 49

5-3 IMREC ratios and MRR for `mcf` over standard partition associativity and compressibility 50

5-4 IMREC ratios and MRR for `swim` over standard partition associativity and compressibility 51

5-5 IMREC vs. MRR gains 52

5-6 IMREC and MRR for the `art` benchmark 54

5-7 IMREC and MRR for the `dm` benchmark 54

5-8 IMREC and MRR for the `equake` benchmark 55

5-9 IMREC and MRR for the `mcf` benchmark 55

5-10 IMREC and MRR for the `mpeg2` benchmark 56

5-11 IMREC and MRR for the swim benchmark with a 2-way standard partition 56
5-12 IMREC and MRR for the swim benchmark with a 3-way standard partition 58
5-13 art and equake ITEEC ratio for varying dictionary entry size 58

A-1 ITEEC and time reduction for the art benchmark 65
A-2 ITEEC and time reduction for the dm benchmark 65
A-3 ITEEC and time reduction for the equake benchmark 66
A-4 ITEEC and time reduction for the mcf benchmark 66
A-5 ITEEC and time reduction for the mpeg2 benchmark 67
A-6 ITEEC and time reduction for the swim benchmark with a 2-way standard partition 67
A-7 ITEEC and time reduction for the swim benchmark with a 3-way standard partition 68

List of Tables

2.1 Measure of cache data entropy 16

2.2 Miss Rate Reduction for Compacted 4K direct mapped L1 data cache 17

2.3 Miss Rate Reduction for Compacted 16K 4-way set associative L1 data cache 17

5.1 Latency model parameters 57

Chapter 1

Introduction

The obvious technique to increase the effective on-chip cache size is to use a dictionary-based compression scheme, however, a naive compression implementation does not yield acceptable results, since many values in the cache cannot be compressed. The main innovation of the work presented here is to partition the cache and apply compression to only part of the cache. Partitioning allows traditional replacement strategies with random access to cache blocks while preventing excessive fragmentation.

A good deal of research has gone into compression of text, audio, music, video, code, and more [1]. Compression of data values within microprocessors has only begun to be studied recently, for example bus transactions[7] and DRAM [2]. Recently a scheme to compress frequently occurring values in L1 cache has been proposed and evaluated for direct-mapped caches [23].

Compression is a good match for caches since there is no assumption that a particular memory location will be found in the cache. Besides performance, nothing is lost if an address is not found in the cache. Our Partitioned Compressed Cache (PCC) algorithm is applied to the data values in the L2 cache, using a dictionary-based compression, as opposed to sliding-window compression, thereby avoiding coherency problems. As main memory moves further away from the processor, it makes sense to spend a few extra cycles to avoid off-chip traversal costs.

We only apply this compressed cache scheme to an L2 cache. The most important reason for this is that the decompression process takes a not insignificant amount of

time, and it is on the critical path. While buffers storing recently requested decompressed data could help hide some of the latency should the scheme be used in L1, the performance impact from the increased latency would likely be too severe. A secondary reason is that compression tends to do a better job given more data, and so the larger L2 cache in general compresses better than the smaller L1. The scheme should also work well with L3 and higher level cache.

We have found significant improvements in L2 hit rates when compression is applied to only part of the cache. A PCC cache is always compared with a traditional cache of the same size, i.e., the same number of bits. We have found improvements of up to 65% of the miss rate. This is due to simply having an effectively larger cache. Another metric is the increase of the effective size of the cache. That is, given a PCC cache of size S with a hit ratio of R, how much larger must we make the traditional L2 cache to get a hit ratio of R. On some benchmarks, we have found that the traditional cache must be more than 7.5 S (seven and a half times as large).

A final metric is the performance, i.e., the reduction of the running time of the application, or increase in IPC when a PCC cache is used rather than a normal cache. For fairness, this comparison should take into account all area overheads and clock cycle penalties associated with compression. We have found improvements of up to 39% in IPC by using a PCC.

1.1 Related Work

While compression has been used in a variety of applications, it has yet to be researched extensively in the area of processor cache. Previous research includes compressing bus traffic to use narrower buses, compressing code for embedded systems to reduce memory requirements and power consumption, compressing file systems to save disk storage, and compressing virtual and main memory to reduce page faults. Yang et al. [23, 24] explored compressing frequently occurring data values in processor cache, focusing on direct-mapped L1 configurations.

Citron et al. [7] found an effective way to compact data and addresses to fit 32-bit values over a 16-bit bus. This method prompted our early work on compacting cached data.

Work has been done on compression of code, which is a simpler problem than that of compressing code and data. Since code is read-only, compression can be done off-line with little concern for computation costs. The only requirement is that the decompression be quick. The low code density of RISC code made RISC less attractive for embedded systems, since low code density means greater memory requirements which increases cost, and an increase in the number of memory accesses which increases power consumption. This motivated modifications to the instruction set such as Thumb [17], a 16-bit re-encoding of 32-bit ARM, and MIPS16 [12], a 16-bit re-encoding of 32-bit MIPS-III. Other attempts include using compressed binaries with decompression in an uncompressed instruction cache [22], compressed binaries with decompression between cache and processor [15], and pure software solutions[16].

Burrows et al. add compression [5] to Sprite LFS. The log-based Sprite LFS eliminates the problem of avoiding fragmentation while keeping block sizes large enough to compress effectively. Commercial products like Stacker for MS-DOS and the Desktop File System [8] for UNIX use compression to increase disk storage.

Douglis [9] proposed using compression in the virtual memory system to reduce the number of page faults and reduce I/O. He proposes having a compressed partition in the main memory which acts as an additional layer in the memory hierarchy between

standard uncompressed main memory and disk. Data is provided to the processor from the uncompressed partition, and if data is not available in the uncompressed partition, the page needed is faulted in from the compressed partition. When the page is neither in the uncompressed or compressed partitions of memory, it is brought in from disk. Since the performance of this scheme is highly dependent on the size of main memory and the size of the working set, the size of the compressed partition is made to be variable. For example, if the working set is the size of main memory or smaller, no space is allocated to the compressed partition - otherwise unnecessary paging between the compressed and uncompressed partition could cause performance degradation. Douglis' experiments using a software implementation of LZRW1 [20] show several-fold speed improvement in some cases and substantial performance loss in others.

Kjelso et. al [14] evaluate the performance of a compressed main memory system which uses the additional compressed level of memory hierarchy proposed by Douglis. They compare a hardware implementation using their X-Match compression and a software implementation using LZRW1 to the standard uncompressed paging virtual memory system. Using the DEC-WRL workloads, they found up to an order of magnitude speedup using hardware compression, and up to a factor of two speedup using software compression, over standard paging.

Similar to the work of Kjelso et al., Wilson et al. [21] used the same framework as that proposed by Douglis, but with a different underlying compression algorithm. Their WK compression algorithms use a small 16 entry dictionary to store recently encountered 4 byte words. The input is read a word at a time, and full matches, matches in the high 22 bits, and 0 values are compressed. They found that using their compression algorithms and more recent hardware configurations, compression of main memory has become profitable.

Benveniste et al. [3] also worked on compression on main memory, but their system feeds the processor with data from both uncompressed and compressed parts of main memory, unlike the Douglis design. Since in their system compressed data can be used without incurring a page fault, it is necessary to reserve enough space in main

memory so that all of the dirty lines in the cache can be stored if flushed, even if the compression deteriorates due to the modified values (guaranteed forward progress). To find requested data in main memory whether it is compressed or uncompressed, a directory is used, incurring an indirection penalty. To limit fragmentation, the main memory storage is split into blocks 1/4 the size of the compression granularity (the smallest contiguous amount of memory compressed at a time), and partially filled blocks are combined to limit the space wasted by the blocking. The underlying compression they use is similar to LZ77, but with the block to be compressed divided into sub-blocks, the parallel compression of which shares a dictionary in order to maintain a good compression ratio [10].

IBM has recently built machines using its MXT technology [18] which uses the scheme developed by Benveniste et al. with 256 byte sub-blocks, a 1KB compression granularity, combining of partially filled blocks, along with the LZ77-like parallel compression with shared dictionaries compression method. As of the time of this writing, they are selling machines with early versions of this technology.

Compressing data in processor cache has gotten less attention. Yang et al. [23, 24] found that a large portion of cache data is made of only a few values, which they name Frequent Values. By storing data as small pointers to Frequent Values plus the remaining data, compression can be achieved. They propose a scheme where a cache line is compressed if half or more of its values are frequent values, so that the line can be stored in half the space (not including the pointers, which are kept separate). They present results for direct-mapped L1 which with compression can become a 2-way associative cache with twice the capacity.

1.2 PCC in the Context of Related Work

The PCC is similar to Douglis's Compression Cache in its use of partitions to separate compressed and uncompressed data. A major difference is that the Compression Cache serves data to the higher level in the hierarchy only from the uncompressed partition, and so if the data requested is in the compressed partition, it is first moved

to the uncompressed partition. The PCC on the other hand returns data from either type of partition, and does not move the data when it is read. While the Compression Cache aims to have the working set fit in the uncompressed partition, the PCC hopes to keep as much of the working set as possible across all of its partitions.

The scheme developed by Benveniste et al. and the Frequent Value cache developed by Yang et al. serve data from both compressed and uncompressed representations as the PCC does, but both lack partitioning.

Chapter 2

Motivation

In this chapter we review some of our first attempts to apply compression to data cache. First we investigate the data of the cache to get an idea about the degree of redundancy and therefore its overall compressibility. Then we implement a simple compaction algorithm to attempt to take advantage of the redundancy observed. The negative results led to devising new schemes presented in Chapter 3.

2.1 Compaction

Compaction attempts to take advantage of the property that certain bit positions in a word have less entropy[1] than others. For example, counters and pointers may have high order bits which do not vary much, while the low order bits vary much more.

In order to estimate the entropy of the data in cache, we first want the probability that some piece of data in the cache has a certain value, for all possible values. We estimate this probability by taking a snapshot of the cache when it is running an application, and then dividing the number of times that a particular value appears in the cache by the total number of values in the cache. For example, examining entropy

[1] We abuse the term entropy, which requires a random variable. Once a snapshot of the cache data is taken, there is no randomness to the data and so the data has zero entropy. What we mean more precisely is that, if we construct sources which output 0 or 1 with a probability distribution corresponding to that of the distribution of 0's and 1's in a given bit position for the words in the cache, the output of some sources have less entropy than others.

	art	dm	equake	mcf	mpeg2	swim
least significant	0.6470	0.5389	0.6340	0.4293	0.7907	0.9386
byte 1	0.6499	0.6017	0.5716	0.2624	0.7258	0.9379
byte 2	0.5721	0.5862	0.5557	0.1700	0.7400	0.8897
most significant	0.4390	0.3119	0.4711	0.1386	0.7214	0.7030

Table 2.1: Entropy for 16K 4-way set associative L1 data cache, 1 byte granularity. An entropy value of 1 indicates all possible data values are equally likely, while an entropy value of 0 indicates that the byte always has the same value.

at a single bit level, we can approximate the probability of the most significant bit being a 0 with the number of 0s that are in the most significant bit position divided by the number of words. Table 2.1 shows estimated entropy for a byte of information at each of the four positions of a 32-bit value. It was calculated for various benchmarks, using a 16K 4-way set associative L1 data cache.

These results show that values in the more significant byte positions have less entropy than those in the less significant positions. We can take advantage of this by caching the high order bit values and replacing them with a shorter sequence of bits which index into the table of cached values.

We tried a scheme where 32-bit values are compacted to 16-bit values by replacing the upper 24 bits with an 8 bit index. 32 byte cache lines are then only compressed to 16 bytes if all of the values in the line can be compacted, and kept uncompressed otherwise. Two of these compacted cache lines fit in the space of one uncompressed cache line, thus doubling the capacity of the cache. We compact the address tags in the same way, thus gaining a doubling of associativity without increasing the space for comparators. The reduction in miss rate for a 4K direct mapped L1 data cache using this compaction is shown in Table 2.2. Note that although the cache is direct mapped when its entries are not compressed, sets which contain two compressed lines are two-way set associative.

The results were underwhelming. The benchmark with the most gains, `mpeg2`, had a modest improvement of nearly 8% in the miss rate, while the other benchmarks had less than 3% miss rate reduction. Furthermore, increasing the associativity of the cache brings up issues in how to conduct replacement. Using a replacement

Benchmark	Standard MR (%)	Compacted MR (%)	MR Reduction (%)
art	27.0044	26.7306	1.01
dm	6.5741	6.3779	2.98
equake	12.7280	12.4690	2.03
mcf	41.7296	41.4111	0.76
mpeg2	6.7158	6.1804	7.97
swim	17.0729	17.0729	0.00

Table 2.2: Miss Rate Reduction for Compacted 4K direct mapped L1 data cache

Benchmark	Standard MR (%)	Compacted MR (%)	MR Reduction (%)
art	25.7009	25.7372	-0.14
dm	3.2895	3.2807	0.27
equake	6.0638	6.0495	0.24
mcf	37.1333	37.0505	0.22
mpeg2	0.4127	0.4019	2.62
swim	78.6509	78.6509	0.00

Table 2.3: Miss Rate Reduction for Compacted 16K 4-way set associative L1 data cache

policy where LRU information is kept for each compressed or uncompressed line, and incoming lines replace the LRU line regardless of the compressed or uncompressed state of either line, the miss rate reduction for a 16K 4-way set associative L1 data cache is as shown in Table 2.3.

The best performance here is only a few percent improvement in the case of the mpeg2 benchmark. For the art benchmark, performance has actually decreased, despite the increase in the cache's capacity. This negative performance gain is due to the replacement, as newly uncompressible lines kick out compressed lines, and newly compressible lines kick out uncompressible lines.

The unsatisfactory performance of the compacted cache due to a poor replacement behavior, despite promising entropy figures, prompted the development of the partitioned cache presented in the following section.

Chapter 3

The Partitioned Cache Compression Algorithm

This chapter describes our Partitioned Compressed Cache (PCC) algorithm while implementation and optimization details are presented in the subsequent section. While the encoding used by the PCC is based on the common Lempel-Ziv-Welch (LZW) compression technique [19], there are interesting differences from standard LZW in how the PCC maintains its dictionary and reduces the number of lookups needed to uncompress cache data. Moreover, the PCC algorithm partitions the cache into compressed and uncompressed sections so as to provide direct access to cache contents and avoid fragmentation.

A small dictionary is maintained by PCC. When an entry is first placed in the cache or when an entry is modified, the dictionary is used to compress the cache line. If the compressed line is smaller than some threshold, it is placed in the compressed partition otherwise it is placed in the uncompressed partition. The dictionary values are purged of useless entries by using a "clock-like" scheme over the compressed cache to mark all useful dictionary entries. The details are elaborated in what follows after a brief review of the basic LZW compression algorithm.

3.1 The LZW Algorithm

For simplicity, PCC uses a compression scheme based on Lempel-Ziv-Welch (LZW)[19], a variant of Lempel-Ziv compression. It is certainly possible to use newer, more sophisticated compression schemes as they are orthogonal to the partitioning scheme.

With LZW compression, the raw data, consisting of an input sequence of uncompressed symbols, is compressed into another, shorter output stream of compressed symbols. Usually, the size of each uncompressed symbol, say of d bits, is smaller than the size of each compressed symbol, say of c bits. The dictionary initially consists of one entry for each uncompressed symbol.

Input stream data is compressed as follows. Find the longest prefix of the input stream that is in the dictionary and output the compressed symbol that corresponds to this dictionary entry. Extend the prefix string by the next input symbol and add it to the dictionary. The dictionary may either stop changing or it may be cleared of all entries when it becomes full. The prefix is removed from the input stream and the process continues.

3.2 PCC

Unlike LZW, Partitioned Compressed Cache (PCC) compresses only a cache line's worth of data at a time rather than compressing an entire input stream. Although compressing larger amounts of data provides better compression, it adds extra latency in decompressing unrequested data and complicates replacement.

Cache lines are compressed using the dictionary. Consider first a simple dictionary representation as a table with 2^c entries; each entry being the size needed by a maximum length string. While this length is unbounded for LZW, in the PCC strings are never longer than an L2 cache line (usually only 32 or 64 bytes). The compressed symbol is just an index into the dictionary.

A space-efficient table representation maintains a table of 2^c entries, each of which contains two values. The first value is a compressed symbol, of c bits, that points to

Space-efficient Dictionary

2^d entries not stored

2^c-2^d entries

Index	Pointer	Append	
1		a	"a"
2		b	"b"
3		c	"c"
...	...	...	
2^d	1	b	"ab"
2^d+1	2	a	"ba"
2^d+2	2^d	c	"abc"
2^d+3	3	d	"cd"
2^d+4	4	b	"db"
2^d+5	2^d+1	c	"bac"
2^d+6	2^d+3	e	"cde"

Reduced-latency Dictionary

2^d entries not stored

2^c-2^d entries

Index	Pointer	Append	
1		a	"a"
2		b	"b"
3		c	"c"
...	...	...	
2^d	1	b	"ab"
2^d+1	2	a	"ba"
2^d+2	1	bc	"abc"
2^d+3	3	d	"cd"
2^d+4	4	b	"db"
2^d+5	2	ac	"bac"
2^d+6	3	de	"cde"

Input: "ababcdbacde"

Step	Input read	String added
1	ab	ab
2	aba	ba
3	ababc	abc
4	ababcd	cd
5	ababcdb	db
6	ababcdbac	bac
7	ababcdbacde	cde

Figure 3-1: LZW algorithm and corresponding dictionaries on sample input: The space-efficient dictionary stores only one uncompressed symbol per entry, while the reduced-latency dictionary stores the entire string (up to 31 uncompressed symbols). The space-efficient implementation of the dictionary uses $(2^c - 2^d)(c+d)$ bits of space and requires $(s_l/d) - (s_a/c)$ lookups to decompress a compressed cache line, where c is the size of the compressed symbol, d is the size of the uncompressed symbol, s_l is the size of the uncompressed cache line, and s_a is the size of the compressed cache line. The reduced-latency implementation of the dictionary as described in Section 4.1 uses $(2^c - 2^d)(c + (d \times (s_l/d - 1)))$ bits of space and requires 1 lookup in the best case and (s_a/c) lookups in the worst case. With an uncompressed symbol size of 8 bits, a compressed symbol size of 12 bits, an uncompressed cache line size of 256 bits, and a compressed cache line size of 72 bits, the space-efficient dictionary is 9600 bytes with a latency of 26 cycles to decompress a cache line, while the reduced-latency dictionary is 124,800 bytes and requires from 1 to 6 lookups. The table at the lower half of the figure shows the order in which entries are added to the initially empty dictionaries.

```
while input[i]
   (length, code) <-- dict_lookup(&input[i])
   output(code)
   if dictionary not full
      dict_add(input, i, length + 1)
   i <-- i + length
```

Figure 3-2: Pseudocode of LZW compression

```
length <-- 0, string <-- ""
do
   cat(string, table_uncompressed[input])
   input <-- table_compressed[input]
   length <-- length + 1
   while input does not start with endcode
while length not 0
   output(string[length - 1])
```

Figure 3-3: Pseudocode to extract a string from a table entry

some other dictionary entry, and the second is an uncompressed symbol, of d bits. A dictionary entry consists of $c + d$ bits. All the uncompressed symbols need not be explicitly stored in the dictionary by making the first 2^d values of the compressed symbols be the same as the values of an uncompressed symbol. So, the entire dictionary requires $(2^c - 2^d)(c + d)$ bits.

Given a table entry, the corresponding string is the concatenation of the second value to the end of the string pointed to by the first value. The string pointed to by the first value may need to be evaluated in the same way, recursively. The recursion ends when the pointer starts with an end code of $(d - c)$ bits of zeros, at which point the remaining d bits of the pointer are treated as an uncompressed symbol and added as the last symbol to the end of the string before terminating. The use of an end code is equivalent to setting the first 2^d entries of the table to contain an uncompressed symbol equivalent in value to the table index, and ending recursion whenever one of these first 2^d entries is evaluated.

While LZW constantly adds new strings to its dictionary, PCC is more careful

about additions. Data which cannot be compressed sufficiently only makes contributions to the dictionary if the dictionary is sufficiently empty. Hard-to-compress data strings do not replace easy-to-compress data strings, thus preventing dictionary pollution.

3.3 PCC Compression and Decompression of cache lines

To compress a cache line, go through its uncompressed symbols looking through the dictionary to find the longest matching string, then outputting the dictionary symbol. Repeat until the entire line has been compressed. With $2^c - 2^d$ table entries to look through, and at most s_l/d repetitions (where s_l is the number of bits in an uncompressed cache line), the compression uses $\frac{(2^c-2^d)s_l}{d}$ table lookups in the worst case. While this is a large number of lookups, it is not on the critical path since it is done for L2—as data is brought in from main memory, the requested data can be sent to L1 before trying to compress it. Buffers can alleviate situations where many L2 misses occur in a row, and if worse comes to worst, we can give up on some of the data and store it in the uncompressed cache partition. Optimizing the compression using hash functions, Content Addressable Memory (CAM), and parallelization are discussed in Sections 4.2 and 4.5.

Decompression is much faster than compression. Each compressed symbol in a compressed cache line indexes into the dictionary to provide an uncompressed string. For a compressed cache line containing n_c compressed symbols, $s_l/d - n_c$ table lookups are needed for decompression. In other words, the fewer the number of compressed symbols (the better the compression), the greater the number of table lookups needed, with a worst case of $s_l/d - 1$ table lookups. Since the amount of space allocated to store a compressed cache line is known and constant in the PCC, compression is performed until the result fits exactly in the amount of space allocated and no less. This results in the best case number of $(s_l/d) - (s_a/c)$ table lookups for decompression, where s_a is

the number of bits allocated for a compressed cache line. The decompression latency can be improved by increasing the dictionary size and parallelization, as described in Sections 4.1 and 4.5.

Naturally, increasing the compressed symbol size c while keeping the uncompressed symbol size d constant will increase the size of the associated table and enable more strings to be stored. With more strings being stored, it is more likely that longer strings can be compressed into a smaller number of symbols, but at the expense of dedicating space to store the larger table and the increased space needed to store each output symbol. Increasing the uncompressed symbol size d will reduce the number of table lookups needed, but also probably increase the number of different strings needed for good compression. It is beyond the scope of this work to study these tradeoffs in further detail.

3.4 PCC Dictionary Cleanup

Since our compression scheme adds but never removes entries to the dictionary, at some point the dictionary becomes full and no more entries can be added. Moreover, if the compression characteristics change throughout the trace, dictionary entries must be purged. PCC continuously cleanses the dictionary of entries that are no longer required by any symbols in the compressed cache.

One way to purge dictionary entries is by maintaining reference counts for each entry. When a compressed cache line is evicted or replaced, the reference count is decreased and the entry purged when the count becomes zero.

PCC uses a more efficient method to purge entries. It sweeps through the contents of the cache slowly, using a clock scheme with two sets of flags. Each of the two sets has one flag per dictionary entry, and the status of the flag corresponds to whether or not the dictionary entry is used in the cache. If there is a flag in either set, it is assumed that the entry is being referenced, otherwise the dictionary entry can be purged.

Two sets of flags are used, one active and one inactive. A sweep through the

```
for each symbol s in block
   active_set[s] <-- TRUE
block <-- next block in cache
if block is first block of cache
   inactive_set <-- active_set
   for all i
      active_set[i] <-- FALSE

if (active_set[i] == FALSE and
    inactive_set[i] == FALSE and
    for all j table_compressed[j] != i)
   table_compressed[i] <-- INVALID
```

Figure 3-4: Pseudocode for dictionary cleanup

compressed cache partition entries sets flags in the active set for the corresponding dictionary entries. When a complete pass of the contents of the cache has been made, the inactive set is emptied and the sets are swapped. Compression or decompression also cause the appropriate dictionary entries referenced also cause the flag to be set. A second process sweeps through the dictionary purging entries.

While checking data more quickly will result in a cleaner dictionary, it also requires more accesses to the cache data and the dictionary table. One can determine, via simulation, the best rate to sweep through the cache; one "tick" per cache reference appears to work fine.

3.5 Managing the storage

In a standard cache with fixed sized blocks, managing the data in the cache is trivial. However, when variable sized blocks are introduced, managing the storage of data becomes an issue. A first attempt at storage management might be to have each cache line data entry be a pointer into a shared pool of compressed strings representing entire cache lines, where the strings can be of variable length. However, since the compressed lines are unlikely to all be the same size, data moved in and out of the cache will likely cause fragmentation.

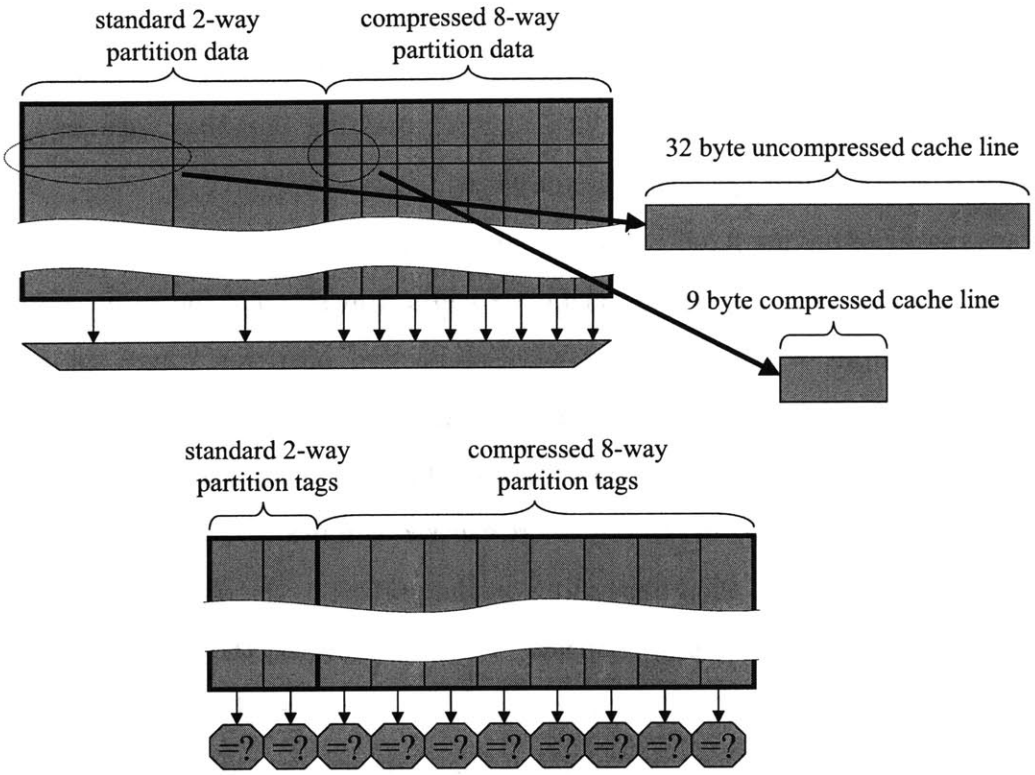

Figure 3-5: Sample partitioning configuration: if there are 8192 sets per way, each of the 2 uncompressed ways of data are $8192 \times 32B = 256KB$, and each of the 8 compressed ways of data are $8192 \times 9B = 72KB$. This adds up to a 1088KB cache, with 512K uncompressed and 576KB compressed. The tags show in the lower half of the figure are the same size across both partitions.

PCC avoids fragmentation by noticing that each application tends to have a collection of easily compressible data and a set of incompressible data. For example, a data structure may have several fields which are counters or flags, data which tends to be easily compressible, while other fields may be pointers to a large database, thereby having values are fairly random and hard to compress. PCC maintains multiple partitions each of which stores compressed cache lines that are up to a different maximum size, as illustrated in Figure 3-5. While the space taken by the data of a way for each partition is different, tags remain uncompressed across partitions and are looked up in the normal manner. Although compressing tags may save area, we believe that it is not worth the ensuing complications.

One advantage of having fixed compressed cache line sizes for entire ways of cache is that a greater associativity can be achieved while drawing out the same amount of data. For example, a PCC with a 4 way compressed partition where the cache lines stored are at most 8 bytes each has 4 times the associativity of a direct mapped cache, but in both cases 32 bytes of data are pulled out to the muxes.

Upon an L2 cache lookup, all partitions are checked simultaneously for the address in question. If there is a hit, then depending on the partition, the data might need to be first uncompressed before being returned. On a cache miss, the data is retrieved from main memory, returned to the L1 cache, then compressed and stored in the partition targeted for the highest compression that can accommodate the size of the newly compressed line in L2 cache. At most one partition will have a valid copy of the requested data. A writeback from L1 back into L2 cache is treated like an eviction of the entry and an insertion of a new entry. The handling of accesses to a PCC is illustrated in Figure 3-6.

3.6 Alternate Compression Methods

Although the compression method used to investigate the performance of a Partitioned Compressed Cache is based on LZW, other methods are available.

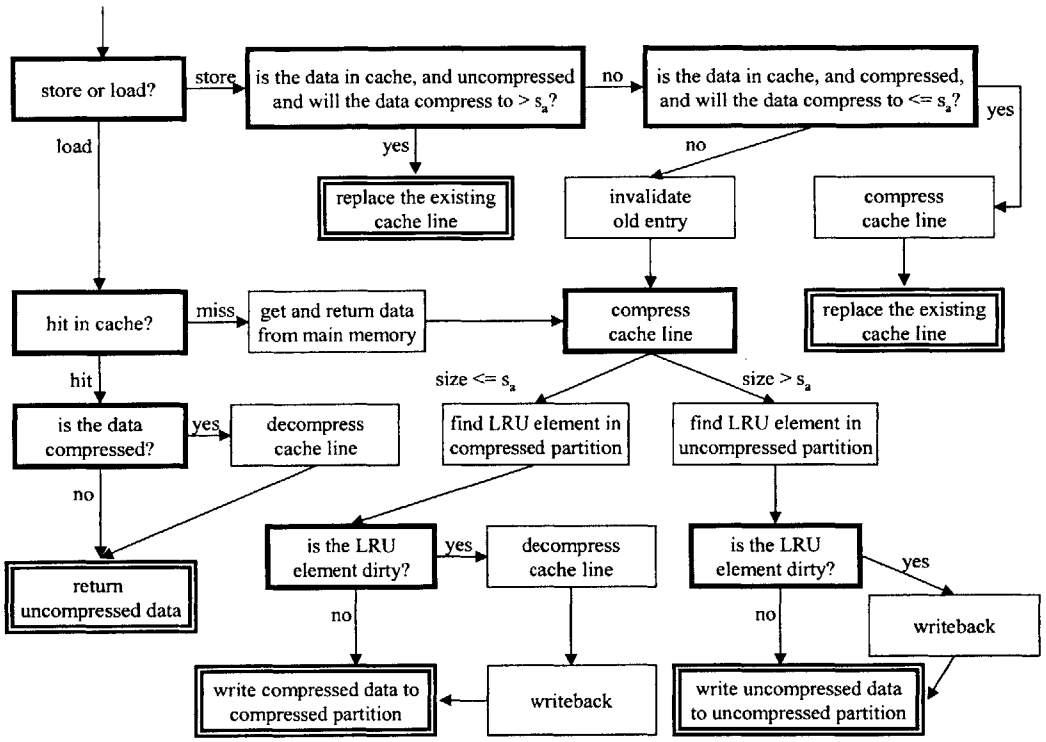

Figure 3-6: flowchart illustrating how accesses to a PCC are handled: control starts at the upper left corner and flows until a double-bordered box is reached.

3.6.1 LZ78

LZ78 is based on chopping up the text into phrases, where each phrase is a new phrase (i.e., has not been seen before) and consists of a previously-seen phrase followed by an additional character. The previously-seen phrase is then replaced by an index to the array of previously-seen phrases. As text is compressed and the number of previously-seen phrases increases, the size of the pointer increases as well. When we run out of memory, we clear out memory and restart the process from the current position in the text.

Encoding uses a trie, which is a tree where each branch is labelled with a character and the path to a node represents a phrase consisting of the characters labeling the branches in the path. In recognizing a new phrase, the trie is traversed until reaching a leaf, at which point we have traversed the previously-seen phrase and the addition of the next character results in the new phrase.

3.6.2 LZW

While LZ78 uses an output consisting of (pointer, character) pairs, LZW outputs pointers only. LZW initializes the list of previously-seen phrases with all the one-character phrases. The character of the (pointer, character) pair is now eliminated by counting the character not only as the last character of the current phrase but also as the first character of the next phrase. To speed up the transmission and processing of the pointers, they are set at a fixed size (typically 12 bits, resulting in a maximum of 4096 phrases).

3.6.3 LZC

LZC is used by the "compress" program available on UNIX systems. It is an LZW scheme where the size of the pointers is varying, as in LZ78, but has a maximum size (typically 16 bits), as in LZW. LZC also monitors the compression ratio; instead of clearing the dictionary and rebuilding from scratch when the dictionary fills, LZC does so when the compression ratio starts to deteriorate.

3.6.4 LZT

LZT is based on LZC, but instead of clearing the dictionary when it becomes full, space is made in the dictionary by discarding the LRU phrase. In order to keep track of how recently a phrase has been used, all phrases are kept in a list, indexed by a hash table. In effect, the LRU replacement of phrases is imposing a limitation similar to that which is imposed in the sliding-window based LZ77 and its variants, of allowing the use of only a subset of previously-seen phrases. This limitation encourages a better utilization of memory, at the cost of some extra computation. In addition, LZT uses a phase-in binary encoding which is more space efficient than the encoding of phrase indices used by LZC at the cost of added computation.

3.6.5 LZMW

Instead of generating new phrases by adding a new character to a previously-seen phrase, LZMW generates new phrases by adding another previously-seen phrase. With this method, long phrases are built up quickly, but not all prefixes of a phrase will be found in the dictionary. The result is better compression, but a more complex data structure is needed. Like LZT, LZMW discards phrases to bound the dictionary size.

3.6.6 LZJ

LZJ rapidly adds dictionary entries by including not only the new phrase but also all unique new sub-phrases as new dictionary entries. To keep this manageable, LZJ also bounds the length of previously-seen phrases to a maximum length, typically around 6. Each previously-seen phrase is then assigned an ID of a fixed length, typically around 13 bits. All single characters are also included in the dictionary to ensure that the new phrase can be formed. When the dictionary is filled, previously-seen phrases that have occurred only once in he text are dropped from the dictionary. LZJ allows fast encoding and has the advantage of a fixed-size output code due to the use of the phrase ID. On the other hand, the method for removing dictionary

entries imposes a performance penalty, and much memory is required to achieve a given compression ratio. An important aspect of LZJ is that encoding is easier than decoding. LZJ' is the same as LZJ but with a phase-in binary encoding for its phrase indices.

3.6.7 LZFG

LZFG occupies a middle position between LZ78 and LZJ. LZJ is slow partly because every position in the text is the potential start of a phrase. LZFG allows a phrase to start only at the start of a previously-seen phrase. However, where LZ78 requires that a new phrase consist of exactly a previously-seen phrase plus one character, LZFG allows the previously-seen phrase to be extended beyond its original end point in the text, by including a length field in he encoded output. Like LZ78, therefore, exactly one new phrase is inserted into the dictionary for every new phrase encoded. However, the new phrase in the dictionary actually represents a number of new phrases. Each of these new phrases consists of the original new phrase plus a string (of arbitrary length) consisting of the characters that follow that new phrase in the text. LZFG requires a more complex data structure and more processing than LZ78, but overall it still achieves good compression with efficient storage utilization and fast encoding and decoding.

3.6.8 X-Match and X-RL

The X-Match [13] compression method uses a dictionary of 4 byte strings. The input is read 4 bytes at a time, and compared to the dictionary entries. If two or more of the 4 bytes are the same as those of a dictionary entry, a compressed version of these 4 bytes are sent to the output along with a bit indicating that the information is compressed. The ability to send a compressed encoding when there is only a partial match (not all 4 bytes match) is where algorithm's name comes from. The compressed encoding consists of the index of the dictionary entry (encoded using a phased-in binary code), the positions of the matching bytes (encoded using a static

Huffman code), and the remaining unmatched bytes if any (sent unencoded). If no such partial or full match exists, the 4 bytes are sent without modification, along with a bit indicating that the output is not compressed. The dictionary uses a move-to-front strategy where the first entry is the most recently used entry, and subsequent entries monotonically decrease in recency of use. Each 4 byte chunk of input is added as an entry to the front of the dictionary unless it already exists in the dictionary (a full match occurs), in which case the dictionary size stays constant while the matched entry moves to the front and the displaced entries shuffle to the back.

X-RL adds to X-Match a run length encoder which encodes only runs of zeros.

3.6.9 WK4x4 and WKdm

The WK4x4 and WKdm algorithms developed by Wilson and Kaplan [21] works on 4 byte words at a time and looks for matches in the high 22 bits of each word. Each word of the input is looked up in a small dictionary which stores 16 recently encountered words. The WK4x4 variant uses a 4 way associative dictionary, while WKdm uses a direct-mapped dictionary. A two-bit output code describes whether the input matched exactly with a dictionary entry, matched in the high 22 bits, did not match at all, or contained 0 in all 32 bits. In the case of a full match, the index of the match in the dictionary is then added to the output. For a partial match, the index and the low 10 bits are output. Finally when there is no match and the input is not 0, the uncompressed input word is sent to the output. The output is packed so that like information (two-bit codes, indexes, indexes plus low 10 bits, and full words) is stored together.

3.6.10 Frequent Value

Frequent Value compression has been proposed by Yang et. al, and consists of keeping a small table of the most frequently occurring data values. Each compressed block contains a bit vector which specifies whether the data is a compressed frequent value or if it is uncompressed, and when compressed it specifies an index into the frequent

value table.

3.6.11 Parallel with Cooperative Dictionaries

Franaszek et al. researched the use of multiple shared dictionaries to preserve a high compression ratio while parallelizing the compression process [10]. Their algorithm uses LZSS (a variant of LZ77) as a base, then takes the compression block size and divides it into sub-blocks which are compressed in parallel. While a dictionary is maintained for each compressing process, each process searches across all dictionaries as part of the LZSS compression.

Chapter 4

The Partitioned Compressed Cache Implementation

This chapter describes the details of implementing the PCC in hardware, including optimizations and tradeoffs. These include making the dictionary representation less space efficient in the interests of reducing latency, searching only a strict subset of the dictionary entries during compression to reduce latency, hashing the inputs of searches to the dictionary in order to improve the compression ratio, and parallelizing compression and decompression in the interests of latency.

4.1 Dictionary Latency vs. Size Tradeoff

The number of lookups needed to decompress a cache line can be reduced by increasing the size of the dictionary. While storing only one compressed symbol and one uncompressed symbol per dictionary entry is fairly space efficient, decompressing a compressed symbol potentially requires many lookups. Specifically, the number of lookups needed to decompress a compressed symbol is the number of symbols in the encoded string minus one. To reduce the number of lookups needed, each entry can store more than one uncompressed symbol along with a compressed symbol. In this case, all of the multiple uncompressed symbols are added to the output string after decoding the string pointed to by the compressed symbol. For example, if two un-

compressed symbols are stored at each entry, then a string that is 5 uncompressed symbols long requires only 2 lookups instead of 4. In order for decompression to work, an additional entry length field is needed to indicate the number of valid uncompressed symbols in each entry, which adds $\log n$ bits per entry, where n is the maximum entry length. Taken to an extreme, each dictionary entry could store the entire string, so that only one lookup is needed per symbol. Since only one lookup is needed, the compressed symbol pointer and the entry length fields are no longer needed, and each entry uses $s_l - d$ bits (not counting valid and cleanup bits). An added benefit in this extreme case is that dictionary cleanup becomes much easier, as it is no longer necessary to check that an entry is not used by any other entry before invalidating it.

4.2 Using Hashes to Speed Compression

The number of lookups can be reduced dramatically by searching through only a strict subset of the entire dictionary for each uncompressed symbol of the input. This may harm the compression ratio, so to increase the likelihood of encountering a match in this reduced number of entries, we can hash the input of the lookup to determine which entries to examine. As an example of this scheme, we could use 16 different hash functions so that for each dictionary access, we hash the input these 16 ways and test the resulting 16 dictionary entries for a match. This example would limit the number of accesses to $\frac{16s_l}{d}$. If the dictionary is stored in multiple banks of memory, choosing hash functions such that entries are picked to be in separate banks allows these lookups to be done in parallel. Alternatively, content addressable memory (CAM) can be used to search all entries at the same time, reducing the number of dictionary accesses to the number of repetitions needed, or s_l/d accesses.

The cost of using hashes is the increased space and complexity required for their implementation, and the additional latency in performing the hash to find the desired dictionary entry before each dictionary lookup.

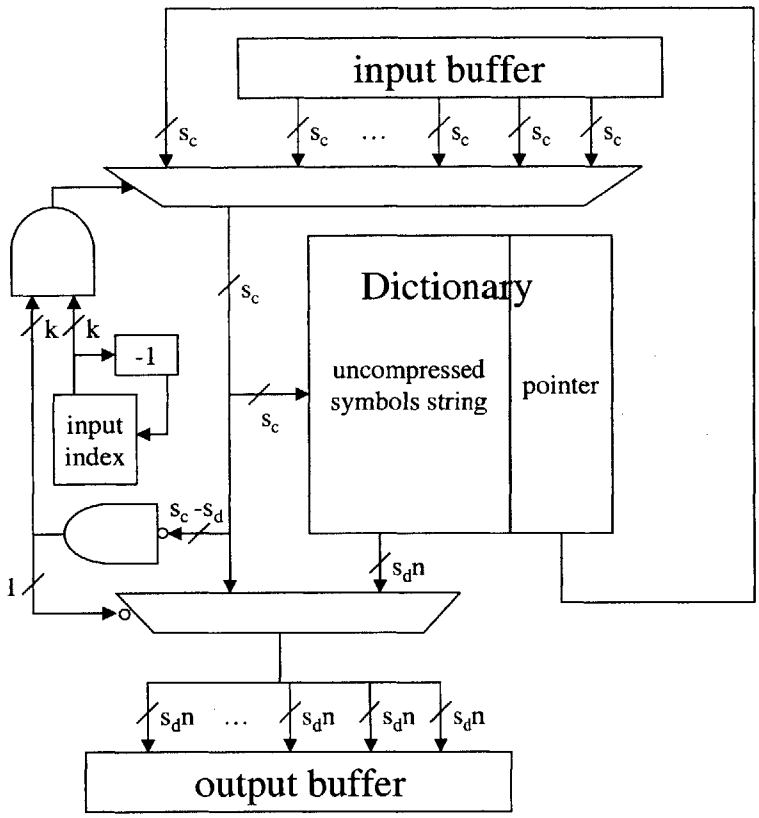

Figure 4-1: Decompression Logic: note that the design shown here does not include parallel decompression, and therefore exhibits longer latencies for larger compression partition line sizes. n represents the number of uncompressed symbols stored in each dictionary entry.

4.3 Decompression Implementation Details

Decompression begins by storing the compressed line in the input buffer, setting the input mux to provide the last compressed symbol of the input buffer, and setting the output mux to drive the end of the output buffer. If the compressed symbol of the input does not need to be looked up in the dictionary (it encodes only one uncompressed symbol and therefore has zero in its upper $c - d$ bits), the output mux is set to store the value indicated by the compressed symbol (its lower d bits) into the output buffer, and then the input mux and output demux are updated accordingly.

If the compressed symbol of the input does need to be looked up in the dictionary, then the symbol provides an index into the dictionary. The result of the dictionary lookup is selected by the output mux and stored to the output buffer through the output demux, and then the output demux is updated according to a field in the dictionary entry which contains the length of the portion of string decoded by the entry. If the decompression of the compressed symbol is not complete because the dictionary entry's pointer field needs to be looked up in the dictionary, the input mux selects this pointer as the compressed symbol and it is used to index into the dictionary for another lookup. Dictionary lookups are repeated until the pointer field of the dictionary entry indicates the end of the encoded string by encoding only one decompressed symbol (it contains zero for its upper $c - d$ bits). This entire process repeats until the entire input buffer has been consumed.

This decompression implementation is shown in Figure 4-1. A mux at the input buffer selects either the next compressed symbol to decode or a dictionary pointer to look up. A mux at the output buffer selects between storing the input (in the case the compressed symbol encodes only one uncompressed symbol) and the results of a dictionary lookup. A demux at the output buffer selects where in the block to store. Finally, there is some logic to determine whether decoding of a compressed symbol has finished or if another dictionary lookup is required. Since the demux at the output buffer provides the results of a dictionary lookup, its output width is equal to the size of the n uncompressed symbols in the dictionary, plus the width of one

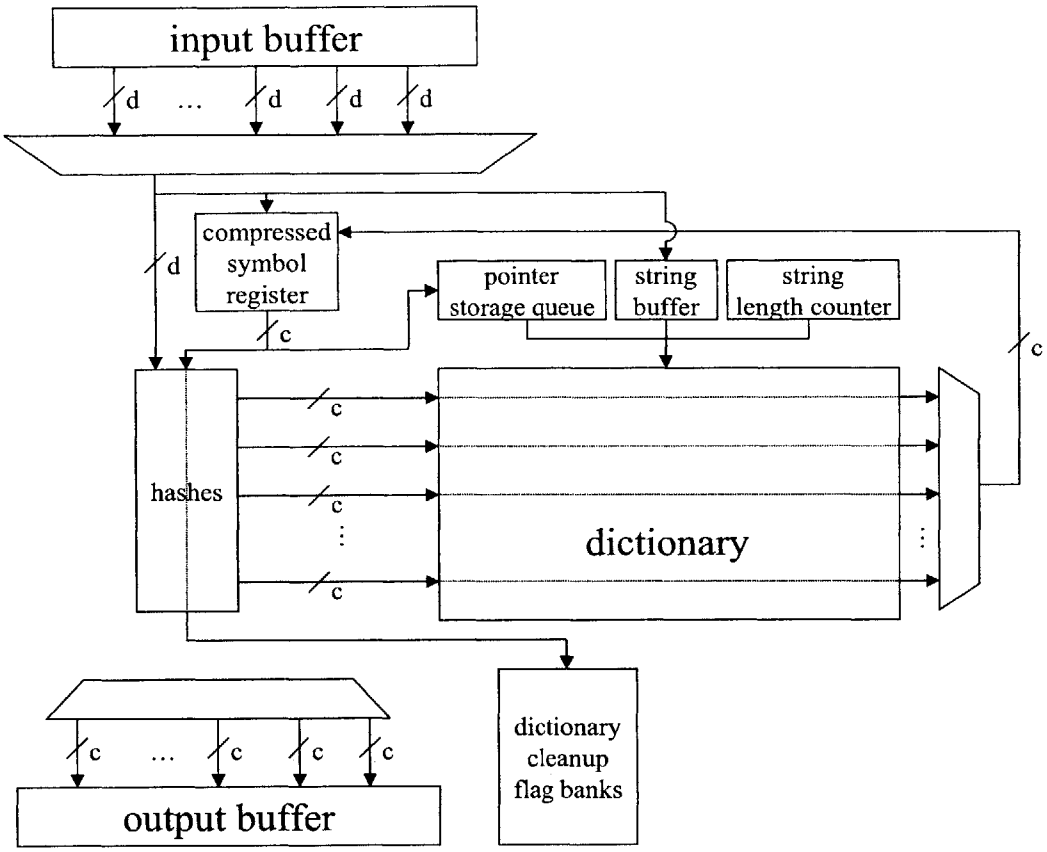

Figure 4-2: Compression logic

more uncompressed symbol for the pointer field of a dictionary entry.

The implementation shown in the figure is serialized. In this non-parallel case, decompression starts at the end of the input buffer and works its way to the front of the input. This avoids the use of a stack or additional dictionary information otherwise needed to recursively look up a string in the dictionary. Section 4.5 describes a faster parallel version.

4.4 Compression Implementation Details

Compression begins by storing the uncompressed line in the input buffer, setting the input mux to read the first uncompressed symbol of the input buffer, setting the

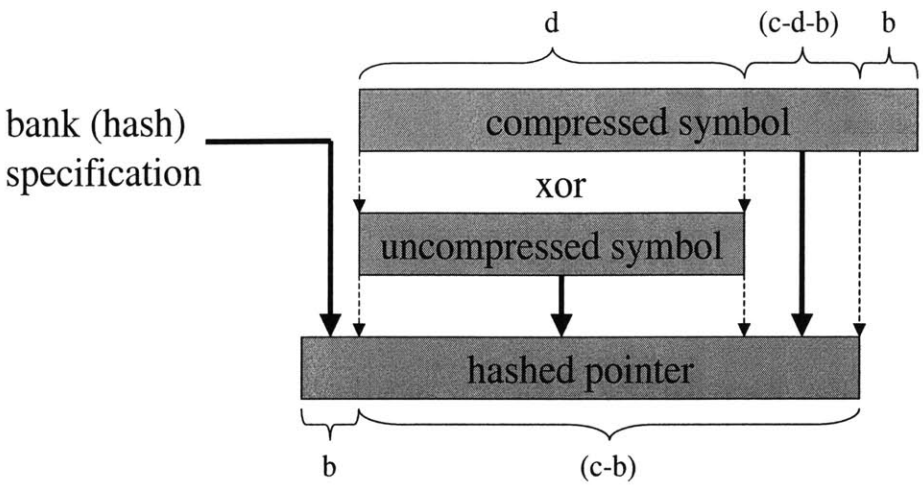

Figure 4-3: Sample hash function

output demux to store to the first compressed symbol of the output buffer, setting the string length counter to 1, and setting the pointer storage queue to contain the input buffer's first uncompressed symbol's corresponding compressed symbol (the same value with zeros in the upper $c - d$ bits).

Next, an uncompressed symbol from the input buffer is turned into its corresponding compressed symbol (the same value with zeros in the upper $c - d$ bits), and is hashed along with the next uncompressed symbol in the input buffer. A possible hash function is shown in Figure 4-3: the uncompressed symbol shifted left by $c - d - \log_2 b$ xor with the compressed symbol shifted right by $\log_2 b$, with b being the number of banks comprising the dictionary. The results of the hash functions are then used to look up entries in the dictionary.

For the results of each dictionary lookup, the valid bit is checked. For each valid entry, the entry's compressed symbol pointer and first element of the uncompressed string are compared against the input of the hash functions. On a match, the matched entry's dictionary cleanup usage flag is turned on, the string length counter is incremented, and the entry index is added to the pointer storage queue. If the pointer storage queue is longer than n, the maximum number of uncompressed symbols stored in an entry, then an entry is removed from the queue in FIFO order. This completes

the compression of one of the uncompressed symbols, so the matched entry's index is hashed with a new uncompressed symbol from the input buffer, and compression continues from the hashing stage.

If none of the dictionary lookups yields a valid matching entry, but one of the banks contains an invalid entry, then the invalid entry is marked valid and the newly encountered string is stored in it. To fill the entry, a pointer is removed from the pointer storage queue in FIFO order and stored in the entry's compressed symbol pointer field. The entry string buffer contents are stored in the entry's n uncompressed symbol fields and the string length counter is copied to its respective field in the entry. Before processing the next input symbol, the dictionary cleanup usage bit is toggled on for the entry, the entry's index is sent to the output buffer as a compressed symbol, and the various state is reset. The state reset consists of setting the string length counter to 1 and setting the pointer storage queue to contain only the latest uncompressed symbol's corresponding compressed pointer. Finally the processing of the current uncompressed symbol is finished, so the uncompressed symbol's corresponding compressed symbol along with a new uncompressed symbol from the input buffer are sent to the hash functions, and compression continues from there.

In the case that none of the dictionary lookups yields a valid matching entry and none of the entries are invalid, the current string needs to be output by simply sending the current compressed symbol to the output buffer, and the various state needs to be reset. Then compression continues by sending the current uncompressed symbol's corresponding compressed symbol and a new uncompressed symbol from the input buffer to the hash functions. When all of the uncompressed symbols of the input buffer have been exhausted, the last compressed symbol is stored in the output buffer and compression for the cache line is finished.

The implementation of the compression algorithm is shown in Figure 4-2 and features

- a mux at the input buffer which selects uncompressed symbols
- a demux at the output buffer which selects where in the output buffer to store

- a string buffer which stores the string data to be copied into a newly created dictionary entry
- a pointer storage queue which keeps track of potential pointer values
- access to the dictionary cleanup flag banks
- a counter which keeps track of the current string length
- hash functions to increase the likelihood of checking relevant dictionary entries, with each hash generating an entry in a different dictionary bank

4.5 Parallelizing Decompression and Compression

Decompression and compression can each be done in parallel to reduce their latency. To do so effectively, a method of performing multiple dictionary lookups in parallel is needed. One solution is to increase the number of ports to the dictionary. Another possibility is to keep several dictionaries, each with the same information. This provides a reduction in latency at the expense of the increased area needed for each additional dictionary. Storing the dictionary in multiple banks can also provide multiple simultaneous lookups. As long as lookups are to entries that are in different banks, they can proceed in parallel.

Once multiple parallel dictionary lookups are possible, significant gains can be obtained by parallelizing decompression and compression. In decompressing a compressed cache line, there are multiple compressed symbols which need to be decompressed. Since these symbols are independent of one another, they can be decompressed in parallel. While most compressed strings have length greater than 1 and will require dictionary lookups, strings which contain only one symbol do not.

A problem with decompressing lines in parallel is that without having the previous compressed symbols already decompressed, it is unclear where in the output subsequent compressed symbols should be decompressed to. This problem can be avoided by decompressing each symbol into a separate buffer, and then combining

each buffer to create the uncompressed line. However, this requires a network with many connections. Alternatively, each entry in the dictionary can store additionally the length of the string encoded by that entry. Thus each subsequent compressed symbol can be decompressed in parallel after the previous symbol's first lookup has completed.

Storing the length of the string encoded by an entry also allows decoding from the beginning of the input buffer without the use of a stack. Since the representation of the dictionary requires recursion to decode strings, storing the results of each dictionary lookup as they occur can only be done if the partial output's position is known. To avoid this, intermediate storage can be used to store the results of each dictionary lookup and then the results of all lookups for a given string can be merged. To avoid this expense, the length stored in the dictionary entry can be used to find the position needed in the output buffer, and decoding can proceed from the beginning of the input buffer. Now that the additional length field has provided the ability to decode from the beginning, decompression can proceed from both the beginning and the end of the input buffer in parallel.

Another way to provide string length information is to choose entry indices during compression such that the length of a compressed string can be determined by the index. For example, the first 2^d entries are known to encode strings of length 1. The entries from 2^d to 2^{d+1} might encode strings of length 2, and so on. While this saves the space of explicitly storing the string length at each dictionary entry, it may adversely affect the compression ratio such that this modification will not provide additional benefit.

In practice, parallelizing the decompression process may not actually reduce latency significantly. The experiments in this work show that performance is best when dictionary sizes are such that only one or two lookups are needed per compressed symbol. This is largely due to the low cost of increasing dictionary size in comparison to the benefits of decreasing the number of lookups.

To parallelize compression, searches for strings can start at different points in the uncompressed cache line simultaneously. For example, compression could start at

the beginning of the input buffer while simultaneously the second half of the input buffer can be compressed. While this has the same problem as parallel decompression in that the position in which to store output is not known, the overall compression process is sufficiently long that it offsets the added latency of merging partial output. This method of parallelizing compression is similar to reducing the compression block size, but differs in that when the overall compression is poor, the results of the process compressing a later part of the input can be discarded in the hopes of improving the compression ratio.

While the previous method decreases the latency of compression, the compression ratio may be improved without significantly increasing latency by taking advantage of parallelism as well. Instead of compressing multiple blocks in parallel, the same block at different offsets can be compressed in parallel in the hopes of finding longer strings. For example, compression can start at the first input symbol, and simultaneously at the second input symbol. Then the shorter of the two compressed results is used. There are many possible variants to this method.

A note to make with compressing lines in parallel is that additions to the dictionary must be made atomically, so that two compression units adding entries to the dictionary do not use the same entry. This is only a problem when using multiple dictionaries, since the addition of ports would be read ports and not write ports, and since the use of banks insures that the entries are different anyways.

4.6 Other Details

The implementation of the dictionary cleanup is quite straightforward. Two banks of usage flags are maintained by continuously reading lines from the cache and turning the appropriate flags on. Specifically, for each compressed symbol in each valid line read, the active bank's usage bit for that compressed symbol is turned on. When a full sweep of the cache has been completed, all of the usage flags in the inactive bank are turned off and the inactive bank is swapped with the active bank.

The implementation of the partitioning is also quite straightforward. Lines being

written to the PCC undergo compression, and the resulting compressed line is written to the partition in which it will fit.

The compression and decompression logic area and the latency incurred by compression and decompression are independent of the size of the L2 cache. The main effect of increasing the physical size of the L2 is a likely decrease in the compression ratio.

Compression/decompress logic area and latency are not dependent on the number of ways or the number of sets in L2. Thus the size of L2 can be changed by altering either of these parameters without changing the PCC implementation costs in either area or latency performance.

Of course, changing the size of L2 may affect the average compressibility of data, which will affect performance. Increasing the size of L2 may require increasing the dictionary size to maintain compression performance. Alternatively, multiple dictionaries can be used, one for each part of the compressed partition. This would help to maintain compression performance at the cost of an increase in logic but not in latency.

Increasing the line size has several effects. A benefit is that it is likely to improve overall compression. Unfortunately, it will increase latency. Since access to random bytes within a cache line is not possible, if the L2 line size is greater than the L1 line size, the entire L2 line must be decompressed before sending the requested part of the line needed by L1. Increased cache line size will also minimally increase the decompression logic area, as the space taken by the registers storing a resulting decompressed line will increase. Increased cache line size should not increase the compression logic area needed.

Chapter 5

Results

This chapter evaluates the benefit of PCC via simulation. Since the meaning of compressibility of data is not very clear, we detail the various dimensions of compressibility and our choice of a compressibility measure in a separate section. Using this choice we examine the data in the cache generated by each benchmark. Describing the performance of a compressed cache is not straightforward either, so another section describes the performance metrics chosen to evaluate the effectiveness of the PCC. Then we present the actual performance figures according to the described metrics, for varied settings of partition sizes and compressed partition compressed line sizes. The simulation results show that PCC performance is quite sensitive to the cache configuration; some benchmarks do as well as 65% in miss rate reduction with some configurations of PCC, but do as poorly as -109% in miss rate reduction with others. We finish with observations of the effects of partitioning and possible improvements motivated by these results.

5.1 Characteristics of Data

To understand where the PCC provides improvement, we look at the compressibility of the data. To do this, the data that is being compressed must first be defined. One possibility is to compress the set of all unique data values used by an application over its entire execution. In the context of caches, this is not very meaningful as it ignores

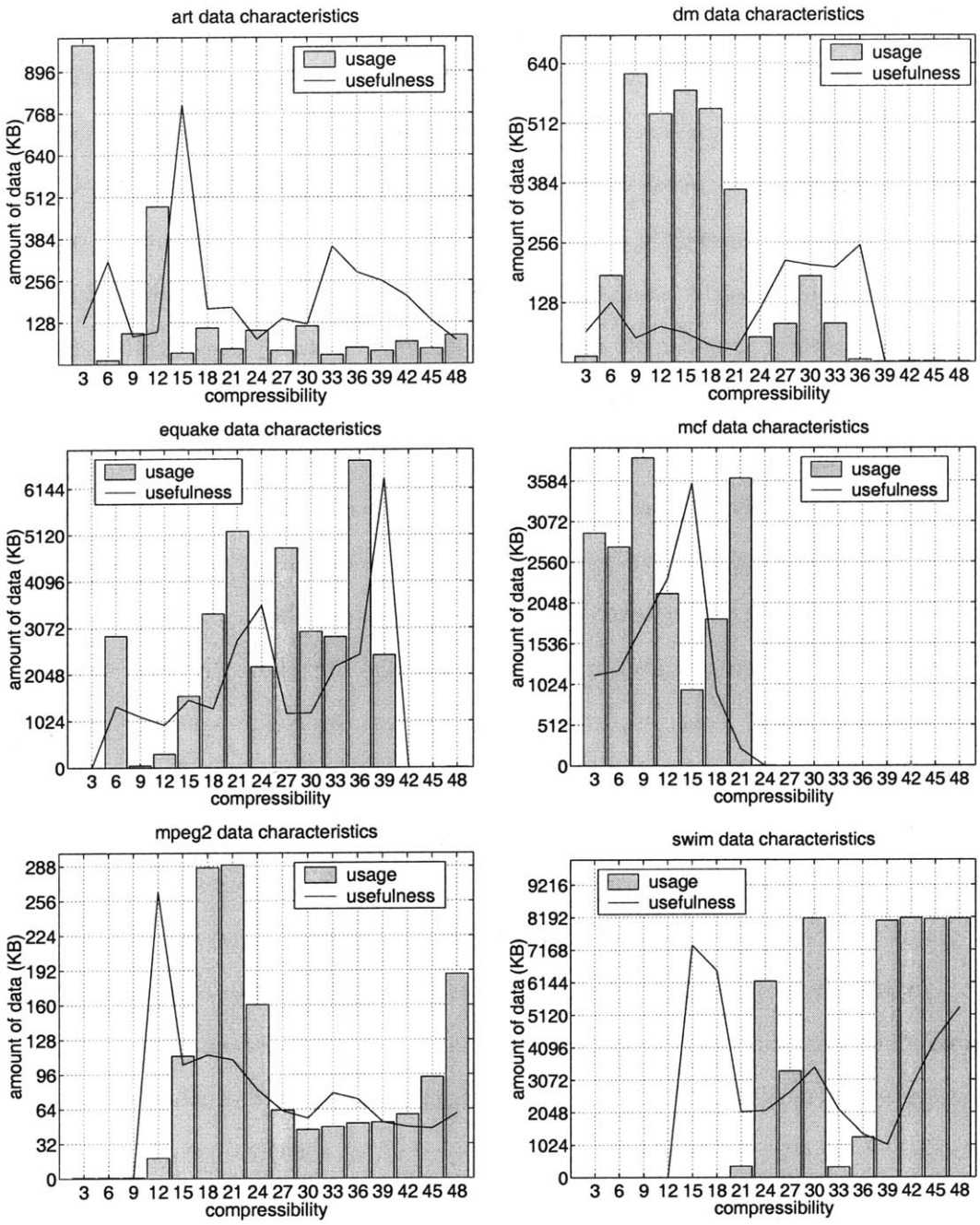

Figure 5-1: The histograms indicate the amount of data available at different levels of compressibility. The x-axis gives the size of the compressed line in bytes. The y-axis gives the amount of data in kilobytes, covering all unique memory addresses accessed during the simulation (an infinite sized cache). The top two histograms show that most data values are highly compressible, while the bottom-most right histogram shows that many data values would require more than 39 bytes to store a 32 byte cache line if compressed. The overlaying curves show the usefulness of data at different levels of compressibility. The y-axis gives the probability that a hit is on a particular cache line in the corresponding partition. This y value is equivalent to taking the total number of hits to the corresponding partition and dividing by the number of cache lines in that partition as given by the bars.

time; compressing the set of unique data values used within a certain amount of time may be more appropriate. The time aspect could also be used by defining the data to include how the data changes over time, for example the delta between a cache line before and after a write occurs.

Not only is the compressibility of the data interesting, but the amount of reuse also impacts the performance of a PCC. Applications which use a large number of data values over a short period of time but reuse only a few of these values are still good targets for a PCC. Finally, the space dimension is important, as an application which uses a small number of data values in most of its address space and a large number of data values in only a small portion is very different from an application whose data values are spread evenly.

Of these possibilities, we chose to define the data of interest to be the data values across the entire accessed address space, weighted by reuse, for different ranges of compressibility. The resulting graphs for each benchmark are shown in Figure 5-1.

When there is no data of a certain compressibility, there is no point in allocating space to store data of that compressibility as it will just be wasted. As shown in the graphs, this is true of `mpeg2` and `swim`, where there are not a significant number of cache lines which can be compressed under 12 and 21 bytes respectively.

The reuse overlays show the usefulness of the data; even though there may be a lot of data of a certain compressibility, it may not be accessed frequently and therefore not be worth storing. Likewise there may be a small amount of data of a certain compressibility which is accessed very frequently, and so a great deal of effort should be spent to make sure that this data is stored.

The PCC's use of partitioning helps ensure that if there is some small amount of data which is accessed very frequently, it will be stored in the cache. If this data is not very compressible, it will be held in the standard partition, while if it is compressible, it will be held in the compressed partition.

Although not immediately obvious, the `art` and `equake` benchmarks are likely to perform well because they have large numbers of cache lines compressible to 3 and 6 bytes respectively.

It is important to note the scale of the y-axis in each of the graphs. While at first glance `mpeg2` may appear to have a fair amount of data compressible to the 18-24 byte range, the accumulation of all of the data in this range amounts to under one megabyte. On the other hand, `equake` has a modest amount of data compressible to 6 bytes in comparison to the less compressible data, but in absolute terms there is almost 3 megabytes of this highly compressible data.

These usage and usefulness graphs provide a good indication of which benchmarks will benefit greatly from PCC and which will not. The `art`, `equake`, and `mcf` benchmarks have large amounts of very compressible data and are likely to do well. Data in the `swim` and `mpeg2` benchmarks is not very compressible, and so it will be harder for PCC to provide improvement. Since the `mpeg2` benchmark has a fairly small memory footprint in addition to its data being hard to compress, it is likely to only do well if the compressed partition is small.

5.2 Performance metrics

Two performance metrics are considered: the effective cache size and the reduction in cache misses. The common metric for the performance of a compression algorithm is to compare the sizes of the compressed and uncompressed data, i.e., the compression ratio [2].

A more interesting metric for a cache is the commonly used miss rate reduction metric. However, the two configurations are not easily comparable as the partitioned cache uses more tags and comparators per area while at the same time using much less space to store data than the traditional cache. We therefore propose a modified metric of interpolated miss rate equivalent caches (IMRECs). This metric measures the effective cache size of a PCC by taking the ratio of the size of a standard cache and the size of a PCC when the two caches have equivalent miss rates. For a given PCC configuration and miss rate, there is usually no naturally corresponding cache size with the same miss rate. Consequently, we interpolate linearly to calculate a fractional cache size. Our sample points are chosen by picking the size of a cache way,

and then increasing the number of ways, thereby increasing the size and associativity of the cache at the same time. Thus, the performance metric to maximize is the IMREC ratio, or the ratio of the interpolated sizes of MRE caches, where one is compressed and the other is not. The size of a standard cache is calculated as being the number of cache lines in the cache multiplied by the cache line size. The size of a PCC cache is calculated as the number of cache lines in its compressed partition multiplied by the compressed cache line size, plus the number of cache lines in the standard partition multiplied by the uncompressed cache line size, plus the space taken by the dictionary.

Thus, when $MR(C_j)$ is the miss rate of a j-way standard cache and $S(C)$ is the size of cache C, the IMREC ratio is the ratio of the size used by the standard cache and the size used by the PCC when the standard cache has the same miss rate as the PCC:

$$\text{IMREC ratio} = S(C_i) + \frac{(S(C_{i+1})-S(C_i))(MR(C_i)-MR(PCC))}{MR(C_i)-MR(C_{i+1})}$$

$$\textit{when } MR(C_i) >= MR(PCC) \textit{ and } MR(C_{i+1}) < MR(PCC)$$

Along with the IMREC ratio, we also provide the miss rate reduction (MRR), or the percent reduction in miss rate. When no simulated standard cache configuration is the same size as the PCC in question, we interpolate linearly between standard cache configurations. Thus, using the same definitions of functions MR and S:

$$\text{Percent Miss Rate Reduction} = \left(MR(C_i) - \frac{(MR(C_i)-MR(C_{i+1}))(S(PCC)-S(C_i))}{S(C_{i+1})-S(C_i)}\right) \times 100\%$$

$$\textit{when } S(C_i) <= S(PCC) \textit{ and } S(C_{i+1}) > S(PCC)$$

5.3 Simulation Environment

We use simulation to evaluate the effectiveness of the PCC. Simulation is done using a hand-written cache simulator whose input consists of a trace of memory accesses.

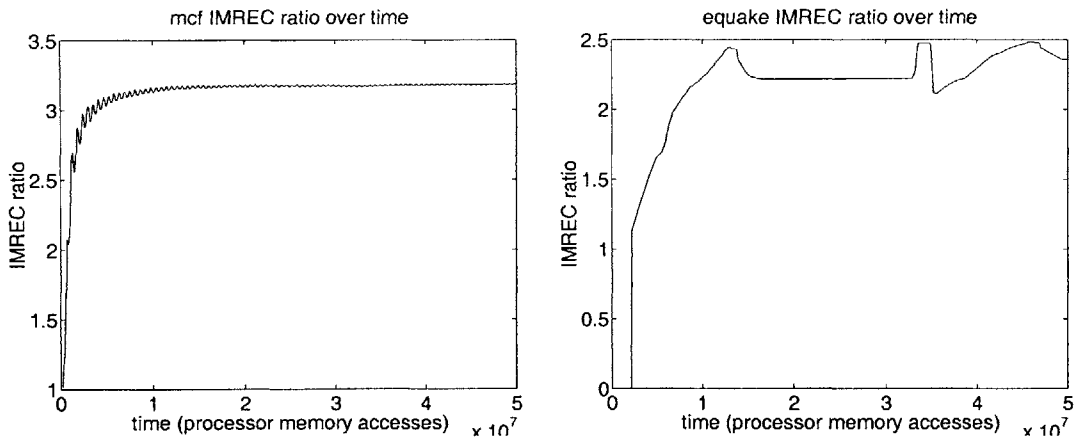

Figure 5-2: `mcf` and `equake` IMREC ratio over time. Both PCCs are configured with 6 byte compressed cache lines, an 8-way compressed partition, and a 2-way standard partition.

A trace of memory accesses is generated by the Simplescalar simulator[4], which has been modified to dump a trace of memory accesses in a PDATS[11] formatted file. Applications are compiled with gcc or F90 with full optimization for the Alpha instruction set and then simulated with Simplescalar. The benchmark applications are from the SPEC2000 benchmark suite and simulated for 30 to 50 million memory references.

The L1 cache is 16KB, 4 way set associative, with a 32 byte line size, and uses write-back. The L2 cache is simulated with varying size and associativity, with a 32 byte line size, and write-allocate (also known as fetch on write).

We assume an uncompressed input symbol size d of 8 bits, and a compressed output symbol size c of 12 bits. The dictionary stores 16 uncompressed symbols per entry, making the size of the dictionary $(2^c - 2^d)(d*16+c)$, which evaluates to 537,600 bits, or 67,200 bytes. A 32 byte cache line size, 1.5 byte compressed symbols, and 1 byte uncompressed symbols, means that a completely uncompressible line takes $32 * 1.5 = 48$ bytes.

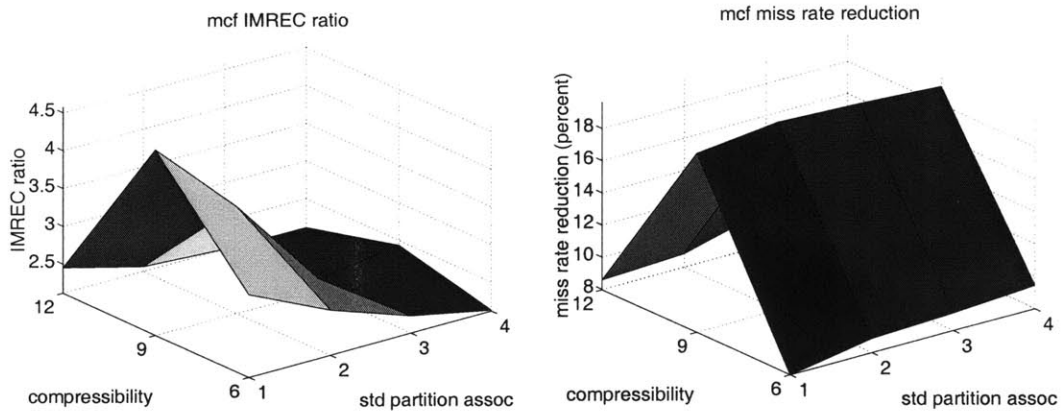

Figure 5-3: We give IMREC ratios and MRR for `mcf` using a PCC with an 8 way compressed partition as a function of both standard partition associativity and compressibility. Higher IMREC ratios are good as they indicate that the PCC configurations have the same performance as larger caches. Higher MRRs are also good as they indicate greater improvement in miss rate over equally sized standard caches. Note that for the `mcf` benchmark as the standard partition increases in size, the IMREC ratio decreases but the MRR increases slightly, showing that neither metric is sufficient in describing the benchmark's behavior. Although the IMREC ratio decreases, this does not mean that performance of the PCC is worse than a standard cache, but rather that the amount of improvement is less. Performance almost always improves when moving from a smaller PCC cache to a larger PCC cache; this improvement can be confirmed by taking the size of the PCC caches, multiplying them by their respective IMREC ratios, and comparing the results. Assuming that larger standard caches have lower miss rates than smaller standard caches, the PCC cache with the larger product has better performance. This does not take into account the additional latency incurred by accessing a compressed partition, which is analyzed in Section 5.5. Thus, an increase in both metrics does not necessarily mean that the performance of the PCC is increasing, but rather that its advantage over a standard cache is more pronounced.

5.4 PCC performance

First we check that our simulation has run for long enough that the values presented are representative of the benchmark by plotting IMREC ratios over time. While some benchmarks like `mcf` clearly reach steady state quickly, others, like `equake`, have more varied behavior and take longer, as shown in Figure 5-2. The performance results obtained from simulating the runs of the various benchmarks are displayed in Figures 5-3 through 5-12.

The change in IMREC ratios and MRR as standard partition associativity and

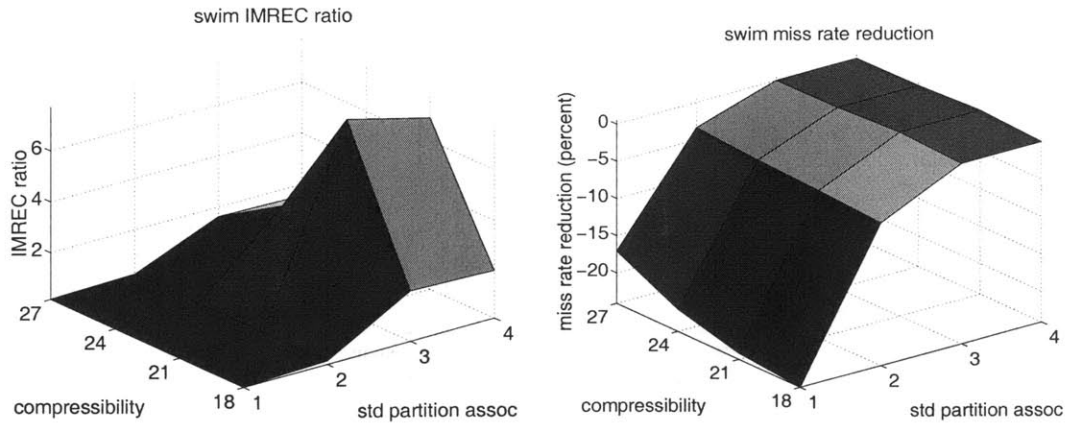

Figure 5-4: We give IMREC ratios and MRR for `swim` using a PCC with an 8 way compressed partition as a function of both standard partition associativity and compressibility. Notice that the PCC actually performs worse than a standard cache of equal size when its standard partition is less than 2 ways large. When the standard partition is 3 ways or larger, the PCC shows large improvements in IMREC ratio, but only small improvements in MRR. The large changes in IMREC ratio coupled with small changes in MRR show that once the cache is able to store 3 or more ways worth of uncompressed data for the `swim` benchmark, it takes a large amount of cache to make small miss rate gains. In this situation, the PCC nets the same performance as a larger cache, but using much less hardware.

targeted compressibility are varied can be seen in Figures 5-3 and 5-4. Since the number of sets and uncompressed cache line size is kept constant, as the standard partition associativity is increased, the size of the partition increases as well. The associativity of the compressed partition is kept constant at 8 ways in these graphs. As can be seen from these graphs, IMREC ratios range from 0.16 to 7.72, while MRR ranges from -109% to 65%.

While the `mcf` benchmark does well with only 1 way of standard cache, `swim` needs at least 3 ways of standard cache. Since `swim` does not have a significant number of cache lines compressible under 21 bytes, performance variations for compressibility of less than 21 bytes is uniform in MRR and steadily decreasing in IMREC ratio as compressed cache line sizes increase, and so these configurations are not very interesting.

Comparing the two metrics shows that neither metric alone is sufficient in describing the behavior of the benchmarks. While `swim` shows significant performance gains by the IMREC ratio, it does not show much miss rate reduction. While `mcf` at

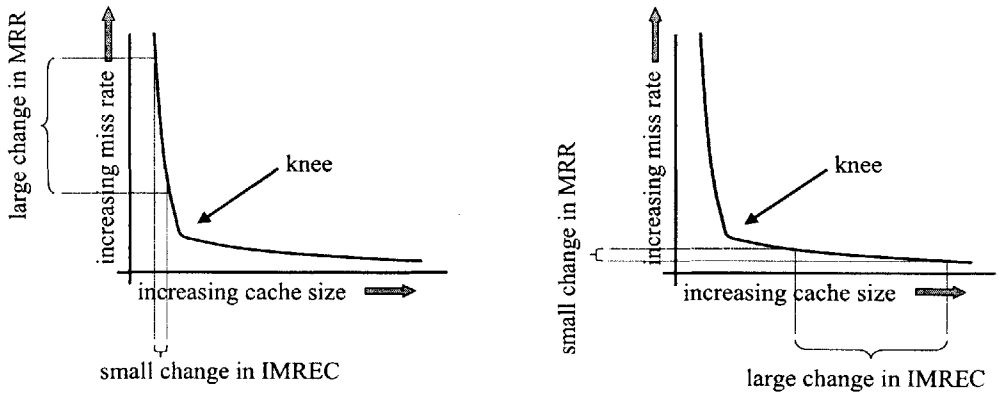

Figure 5-5: To the left of the knee, small increases in IMREC ratio correspond to large increases in MRR. To the right of the knee, small increases in MRR correspond to large increases in IMREC ratio.

9 byte compressibility and a 1 way standard partition has a lower IMREC ratio when another way of standard cache is added, the miss rate reduction increases slightly.

This discrepancy can be understood by looking at what causes large swings in IMREC ratio and MRR. Figure 5-5 shows typical curves of miss rate versus cache size. Miss rate curves typically have a prominent knee where miss rate decreases rapidly until the knee and then very slowly afterwards. The graph to the right shows that to the right of the knee, a small increase in MRR corresponds to a large increase in IMREC ratio. The graph to the left shows that to the left of the knee, a small increase in IMREC ratio corresponds to a large increase in MRR. While it may seem that the small miss rate improvements gained when to the right of the knee are unimportant, applications operating to the left of the knee are likely to be performing so badly that the issue of whether to use a PCC is not a primary concern. Thus most situations of interest occur to the right of the knee, where large IMREC ratios indicate that a PCC provides the same performance gains as a large cache but with much less hardware.

The PCC provides a substantial performance gain for the `art` benchmark. The IMREC ratios show that the PCC configurations perform as well as standard caches ranging from 1.07 to 1.72 times as large, while the number of misses has been reduced by more than 50% in all but one of the configurations. While the IMREC ratio

improvement is not very large, the miss rate reduction is very impressive. This indicates that `art` benefits greatly in time performance (due to miss rate reduction) from a slightly larger cache, an observation which is supported by the drop in improvement as the maximum potential cache size increases from an 8-way associative compressed partition to 16 ways.

The poor performance for the `dm` benchmark remains a mystery. Its working set is considerably smaller than `equake`, `mcf`, and `swim`, but should be large enough to see some performance improvement. Its data is fairly compressible, unlike `mpeg2`, and although the hard to compress data is more frequently reused than the easy to compress data, the difference is not very large. Further study is required to explain the performance results of this benchmark.

The IMREC ratios for PCC running the `equake` benchmark range from 1.19 to 2.36, while the miss rate reduction only hovers around a few percent. This set of data points shows where PCC is an effective use of hardware as opposed to increasing the size of cache. The cache size versus miss rate tradeoff is to the right of the knee of the miss rate curve, such that spending large amounts of hardware gains only a small amount of performance. Using a PCC nets the same performance gains with much less hardware.

The PCC does well in both IMREC ratio and miss rate reduction metrics on the `mcf` benchmark, with IMREC ratios ranging from 2.09 to 3.53 and miss rate reductions of 8% to 36%. It is clear from the difference in the behavior of the two metrics that both are needed to show the effect of different configurations on PCC performance.

The PCC does very poorly for the `mpeg2` benchmark. The poor performance is probably partially due to the benchmark having a relatively small memory footprint and its data being fairly hard to compress. For example, with a compressed cache line size of 21 bytes, an 8-way compressed partition, and a 2-way standard partition, less than one third of the compressed partition is used (has valid data).

The `swim` benchmark really needs to be able to store 3 ways of uncompressible data before it does well at all. Without the ability to store this amount of data,

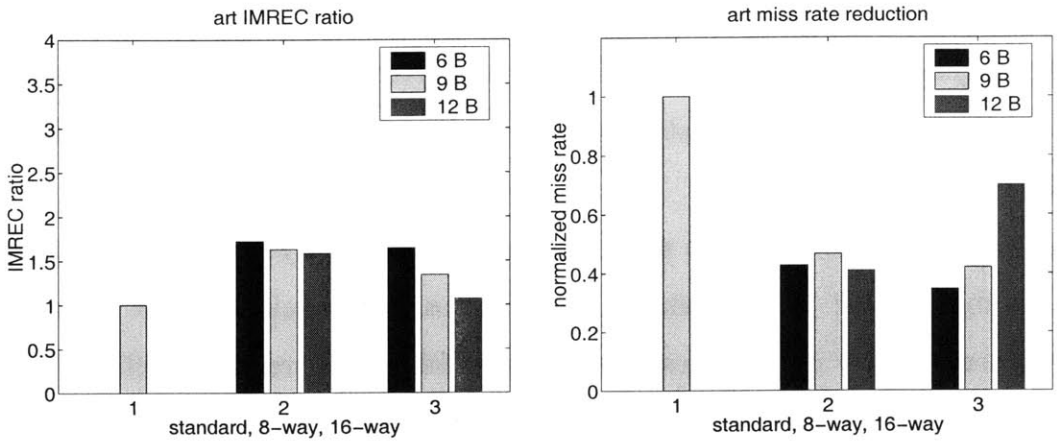

Figure 5-6: IMREC and MRR for the `art` benchmark: substantial performance gain in both IMREC and MRR. Higher IMREC ratios are good as they indicate that the PCC configurations have the same performance as larger caches. Lower normalized miss rates are good as they indicate greater improvement in miss rate over equally sized standard caches. The larger improvement in MRR in comparison to IMREC indicates that at this range of configurations, increasing the cache size slightly provides a large decrease in miss rate. This is supported by the decrease in benefit as the PCC increases in size.

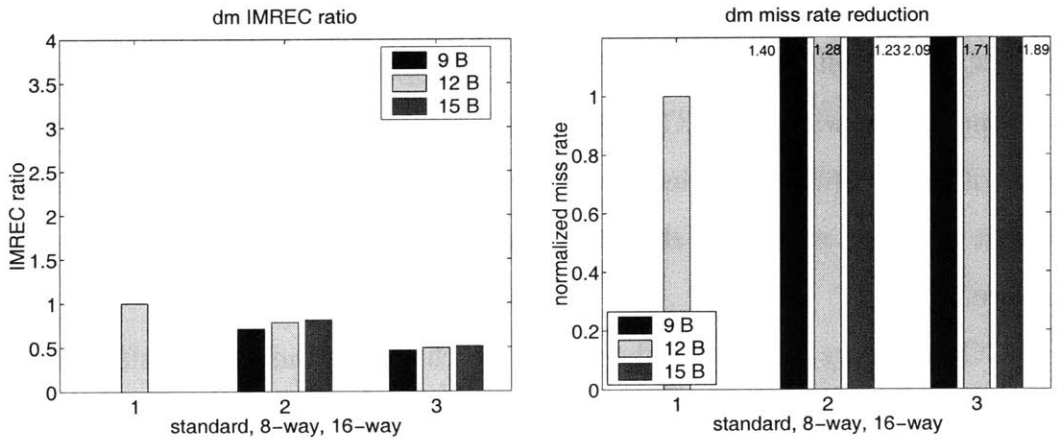

Figure 5-7: IMREC and MRR for the `dm` benchmark: poor performance in both IMREC and MRR is a mystery. Its working set is smaller than `equake`, `mcf`, and `swim`, but should be large enough to see some performance improvement. Its data is fairly compressible, and although the hard to compress data is more frequently reused than the easy to compress data, the difference is not very large.

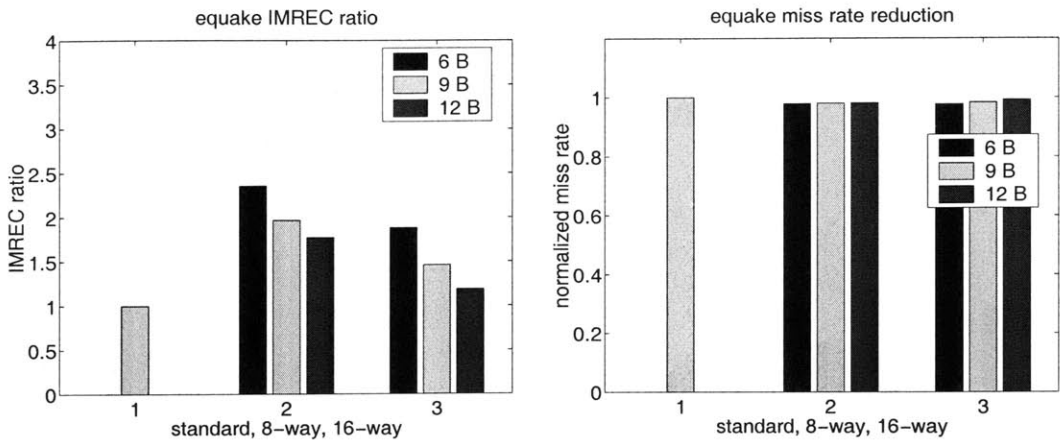

Figure 5-8: IMREC and MRR for the **equake** benchmark: substantial performance gain in IMREC, marginal gain in MRR. This shows that the benchmark is at a point where increasing the cache size results in only small speed improvements. At this point, a PCC obtains the same improvements using much less hardware.

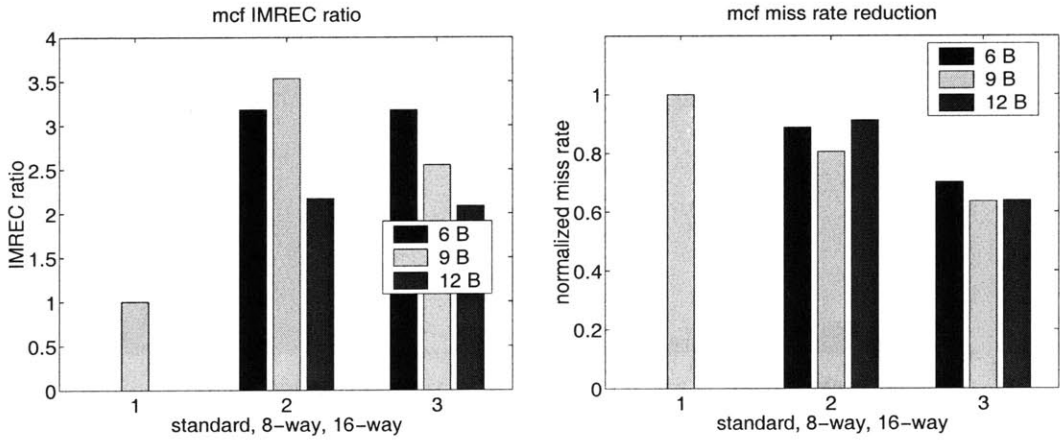

Figure 5-9: IMREC and MRR for the `mcf` benchmark: substantial performance gain in both IMREC and MRR. The difference in behavior of the two metrics in response to changes in PCC configuration show that either metric by itself is insufficient.

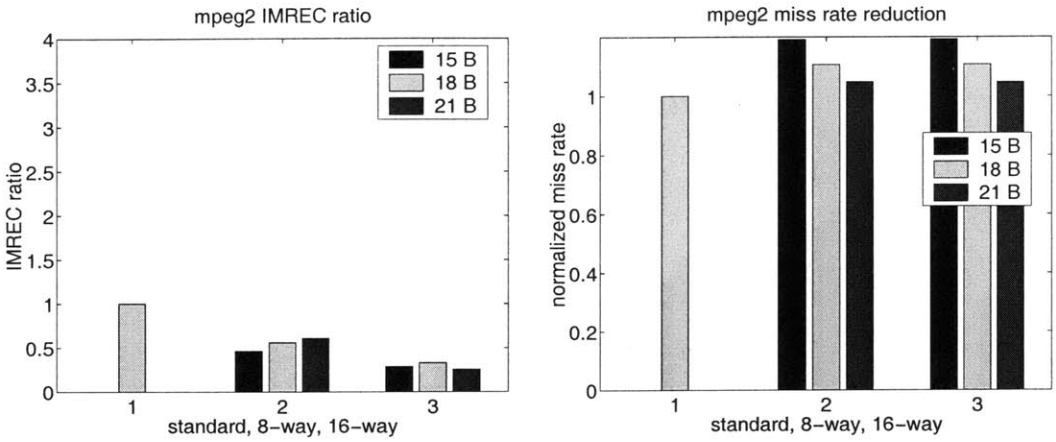

Figure 5-10: IMREC and MRR for the **mpeg2** benchmark: poor performance in both IMREC and MRR. This benchmark has a relatively small footprint and its data is hard to compress.

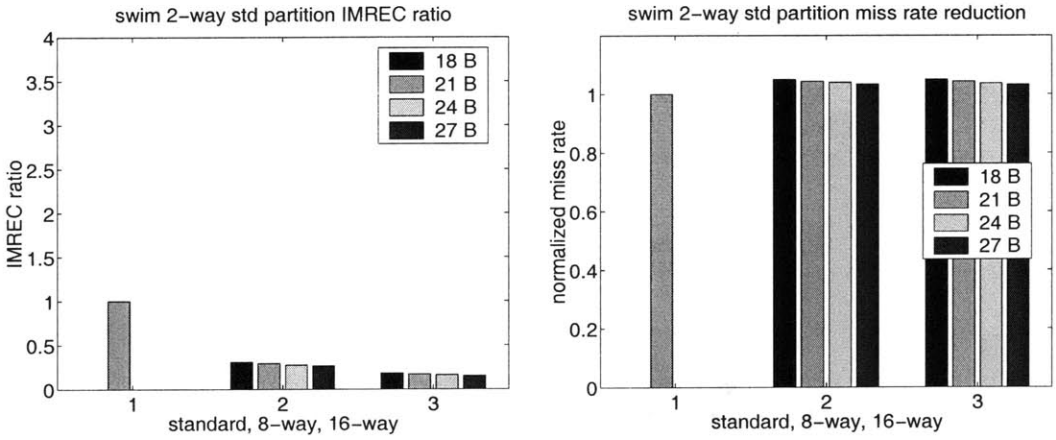

Figure 5-11: IMREC and MRR for the **swim** benchmark: poor performance in both IMREC and MRR. This benchmark really needs to be able to cache 3 ways (768KB) of uncompressible data. When it can only cache 2 ways, performance is poor compared to standard caches which can store the 3 ways of uncompressible data.

Function	Latency (processor cycles)
one dictionary lookup	6
L1 access	1
L2 read, standard	10
L2 write, standard	10
L2 read, compressed	10 + lookup latency
L2 write, compressed	10
Main memory access	100

Table 5.1: Latency model parameters

the PCC does very poorly. With IMREC ratios of under 1 or equivalently, miss rate reductions that are negative, the PCC does worse than a standard cache using the same amount of hardware. The data in the `swim` benchmark is also hard to compress, with almost no 32 cache lines being compressible to less than 21 bytes.

For caches larger than 3 ways, the gain from successively larger caches comes very slowly. Consequently, a small miss rate reduction corresponds to a very large IMREC ratio. For the `swim` benchmark, the small gains which come only after drastically increasing the cache size are obtained by using a PCC. It is very interesting to note that the optimal compression ratio of the PCC is much less than the IMREC ratio achieved while running the `swim` benchmark. This indicates that the partitioned replacement is providing huge gains.

5.5 Increased Latency Effects

A major shortcoming of the IMREC ratio and MRR metrics is that they do not take into account the higher latency of accessing a cache line that is compressed. Once the latency has been modeled, a metric similar to the IMREC ratio, the Interpolated Time Elapsed Equivalent Cache (ITEEC) ratio can be used. For standard caches of varying size, we estimate the amount of time taken to execute some sample of the benchmark. Then we find the ratio in size of a PCC and a standard cache of the same time performance, interpolating as needed. The following results assume the configuration given in Table 5.1.

Since the size of the dictionary can be changed to adjust the number of lookups

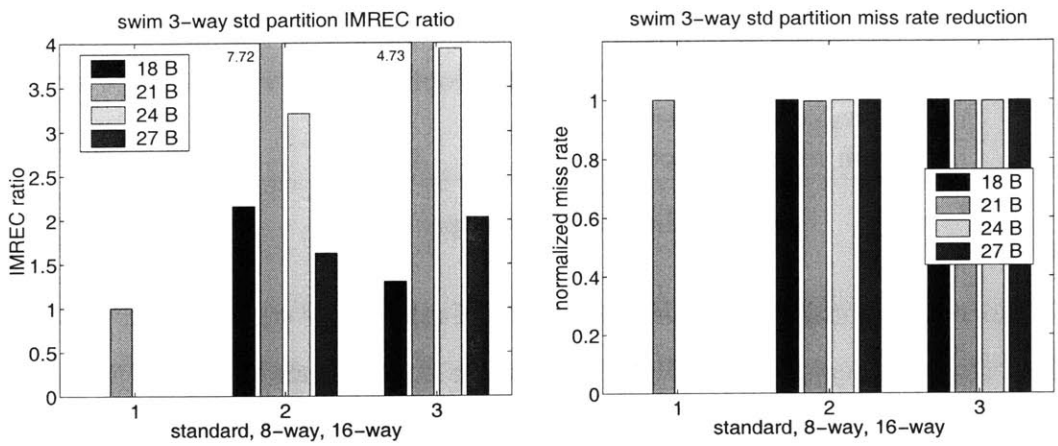

Figure 5-12: IMREC and MRR for the `swim` benchmark, using a 3 way standard partition: very large performance gain in IMREC, marginal gain in MRR. The IMREC ratio here is so large that it exceeds the optimal compression ratio of the cache. This indicates that the partitioned replacement is providing huge gains.

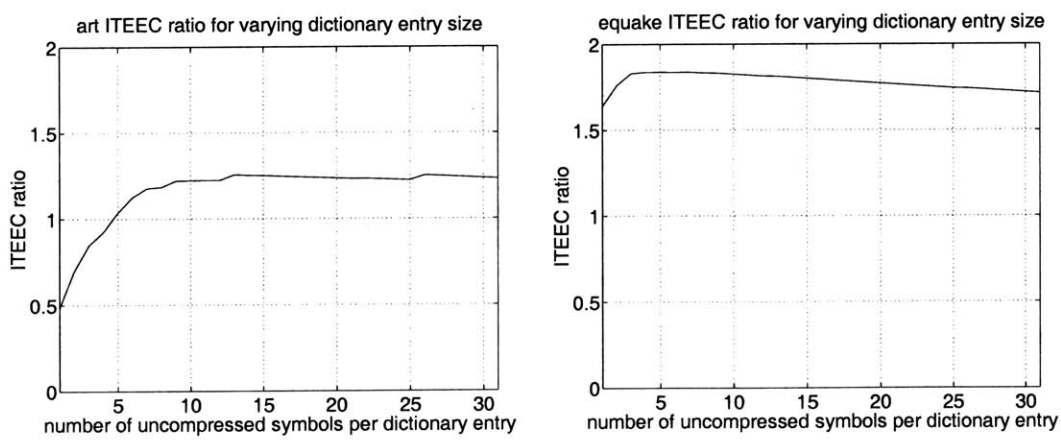

Figure 5-13: `art` and `equake` ITEEC ratio for varying dictionary entry size. Both PCCs are configured with 9 byte compressed cache lines, an 8-way compressed partition, and a 2-way standard partition.

needed to decompress a cache line, ITEEC ratios are shown as a function of dictionary size. The graphs in Figure 5-13 show that the most space efficient dictionary with only one uncompressed symbol per entry do poorly, but by 5 to 10 uncompressed symbols per entry, the ITEEC ratio shows an improvement over standard cache.

Results for each benchmark with latency effects are shown in Appendix A. Accounting for the additional latency in accessing the compressed partition, ITEEC ratios range from 0.08 to 4.32, and time reduction varies from -38% to 28%.

5.6 Usefulness of compressed data, effect of partitioning

The performance gains of a PCC over a standard cache of equivalent size can be attributed to three factors. First, a PCC potentially stores more data than a standard cache, which can reduce capacity misses. Second, a PCC has more associativity than a standard cache of equivalent size, which can reduce conflict misses. Third, using an LRU replacement policy in a PCC is slightly different from LRU in a standard cache, as there is the additional constraint that compressible data can only replace lines in the compressed partition, and uncompressible data can only replace lines in the standard partition.

To get an idea of the effect of the change in replacement policy, we can compare the performance of a PCC to that of a standard cache which is as associative and stores the same amount of data as that of the PCC when the PCC is completely filled. For example, a PCC with a 1 way, 8192 set standard partition, and a 8 way, 8192 set compressed partition, is compared with a 9 way, 8192 set standard cache. If the PCC does better than this standard cache, the differences in replacement must be responsible for at least this difference and possibly more.

For some benchmarks (`mcf`, `swim`), the performance in some configurations is much better than the performance of a standard cache of the same size as that of an expanded compressed cache, which means that the change in replacement has

provided a significant large performance benefit. The replacement policy has changed in that only compressible data can be stored in compressible ways of the cache, and only uncompressible data can be stored in the standard ways. This can be advantageous in comparison to treating all of the data the same, when one type of data pollutes the other. For example, if uncompressible data tends to be fairly streamy (exhibits little or no temporal locality) while compressible data is not, and the references are mixed together, the uncompressible data will pollute the compressible data in cache, causing compressible data to miss more often than is optimal. These partitioning effects were studied in great detail by Chiou [6].

To get an idea of the effect of the increase in associativity, we can hash the set of each cache line so that physical memory addresses map to random set indices. If the performance gains of a PCC with random hashing and a standard cache with random hashing are lesser than without the hashing, this difference can be attributed to the increased associativity of the PCC.

Chapter 6

Conclusion

Compression can be added to caches to improve capacity, but creates problems of replacement strategy and fragmentation; these problems can be solved using partitioning. A dictionary-based compression scheme allows for reasonable compression and decompression latencies and compression ratios. Keeping the data in the dictionary from becoming stale can be avoided with a clock scheme.

Various techniques can be used to reduce the latency involved in the compression and decompression process. Searching only part of the dictionary during compression, using multiple banks or CAMs to examine multiple dictionary entries simultaneously, and compressing a cache line starting at different points in parallel can reduce compression latency. Decompression latency can be reduced by storing more symbols per dictionary entry and decompressing multiple symbols in parallel.

We simulated Partitioned Compressed Caches which implement these ideas, and found a wide variance in its performance across the benchmarks and cache configurations. The worst case cache configuration simulated with the worst performing benchmark saw a decrease in effective cache size by a factor of 6 and a doubling of the miss rate. For the best case, the effective cache size increased almost 8 times and cache misses were reduced by more than half.

It is clear that one partitioning configuration does not work for all applications. Not only the amount of data which can be compressed but also the maximum compressibility at which there is a significant amount of data varies from application to

application. For example, the `swim` benchmark needs at least 3 ways of uncompressed cache, at 256KB a way. The `mpeg2` benchmark does not have much compressible data and thus only benefits from a PCC if its compressed partition is very small. Other benchmarks like `mcf` and `equake` gain the most from PCC when they have only a few ways of uncompressed cache. This motivates the development of a dynamic partitioning scheme to change the sizes of the compressed and uncompressed partitions.

Multiple compressed ways can be combined to be used as fewer ways of uncompressed cache. For example, a compressed partition which has a 16 byte compressed line size can convert two of its ways into a 32 byte uncompressed way. When the cache is accessed, one of the two tag entries is used, and the other is simply ignored.

To determine when the cache would benefit from a larger uncompressed partition at the expense of a smaller compressed partition, we can compare the miss rate improvement of the least recently used compressed ways and of an additional uncompressed way. To perform this comparison, extra tags are managed for an additional uncompressed way. Then counters keep track of hits to the least recently used way of the larger uncompressed tags, and of hits to the least recently used ways of the compressed tags. If the number of hits to an additional uncompressed way is substantially greater than the sum of the hits to the least recently used ways of the compressed partition, overall performance will improve by converting several ways of compressed partition to an additional way in the uncompressed partition.

To determine when the uncompressed partition is too large and should be shrunk to make more compressed ways, periodically convert an uncompressed way into several compressed ways. If the change in partitioning happened to be bad for the miss rate, the partitioning will revert as the process described in the previous paragraph takes effect.

This dynamic partitioning bounds the worst case performance of a PCC to that of a standard cache close to the same size. The performance is not the same as a standard cache exactly the same size due to the space used by the PCC's dictionary and extra tag space. There may also be space wasted if the compressed line size is such that all of the compressed lines can be converted into uncompressed lines.

In addition to dynamic partitioning, there are small modifications to the PCC scheme which can be investigated. For example, for some benchmarks like `art`, the hashing function for the cache is such that only a portion of the compressed cache is actually used. In these cases, a layer of indirection can be used for bigger gains, with each lookup for data returning a pointer into a pool of cache lines. In particular, in the `art` benchmark, data is compressed 2:1 but only half of the compressed partition is filled with valid data. Thus a close to 4:1 compression gain can be obtained by using an extra layer of indirection. Other small improvements include using a double-buffering scheme to keep dictionary contents useful or changing the underlying compression scheme completely.

The benefits of having a partitioned compressed cache have not yet been fully explored. For example, CRCs of the cache data can be done for only a small incremental cost, an idea which is proposed also in [18]. The partitioning based on compressibility may also naturally improve the performance of a processor running multiple jobs, some of which are streaming applications. The streaming data is likely to be hard to compress, and will therefore automatically be placed in the compressed partition and separate from the compressible non-streaming data. In conjunction with dynamic partitioning, only as much uncompressible stream data as needed could be kept in the cache. Although the performance improvements of PCC have been evaluated and found to be large in the best case, there remain interesting topics to investigate in dynamic partitioning so that the worst case behavior is improved to be close to that of a same sized standard cache.

Appendix A

Latency Effects Specifics

The results in Figures A-1 to A-7 are derived using the parameters shown in Table 5.1. The figures show performance as measured by ITEEC ratio and time reduction. The ITEEC ratio is based on time elapsed equivalent caches, or caches which provide the same running time for a certain benchmark.

The ITEEC ratio is essentially the IMREC ratio with the addition of time. When $TIME(C_j)$ is the time needed to run a benchmark using a j-way standard cache and $S(C)$ is the size of cache C, the ITEEC ratio is the ratio of the size used by the standard cache and the size used by the PCC when the standard cache takes the same amount of time as the PCC:

$$\text{ITEEC ratio} = S(C_i) + \frac{(S(C_{i+1})-S(C_i))(TIME(C_i)-TIME(PCC))}{TIME(C_i)-TIME(C_{i+1})}$$

$$when\ TIME(C_i) >= TIME(PCC)\ and\ TIME(C_{i+1}) < TIME(PCC)$$

The time reduction measure shown in the figures is the percent reduction in the amount of time elapsed in running a benchmark using PCC and an equivalently sized standard cache. Similar to the miss rate reduction,

$$\text{Percent Time Reduction} =$$

$$\left(TIME(C_i) - \frac{(TIME(C_i)-TIME(C_{i+1}))(S(PCC)-S(C_i))}{S(C_{i+1})-S(C_i)}\right) \times 100\%$$

$$when\ S(C_i) <= S(PCC)\ and\ S(C_{i+1}) > S(PCC)$$

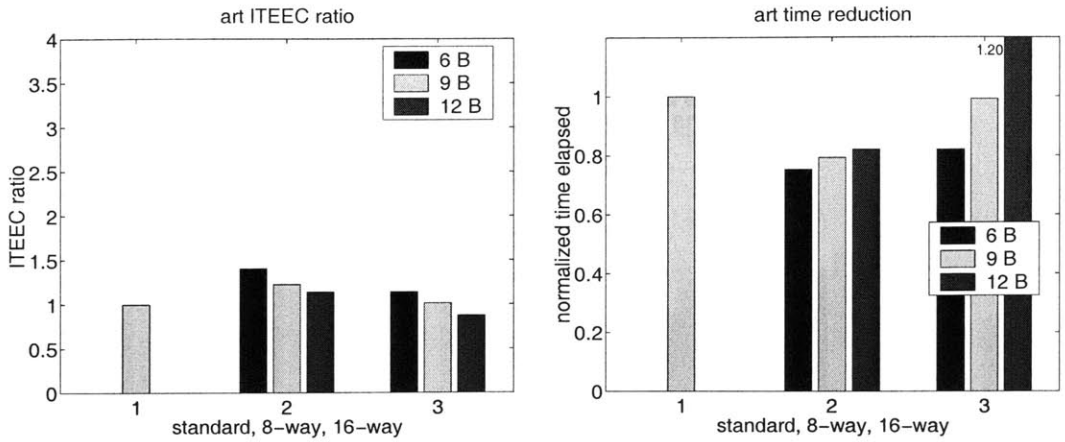

Figure A-1: ITEEC and time reduction for the **art** benchmark: substantial performance gain in both ITEEC and time reduction. Higher ITEEC ratios are good as they indicate that the PCC configurations have the same performance as larger caches. Lower normalized time elapsed is good as it indicates greater improvement in run time over equally sized standard caches.

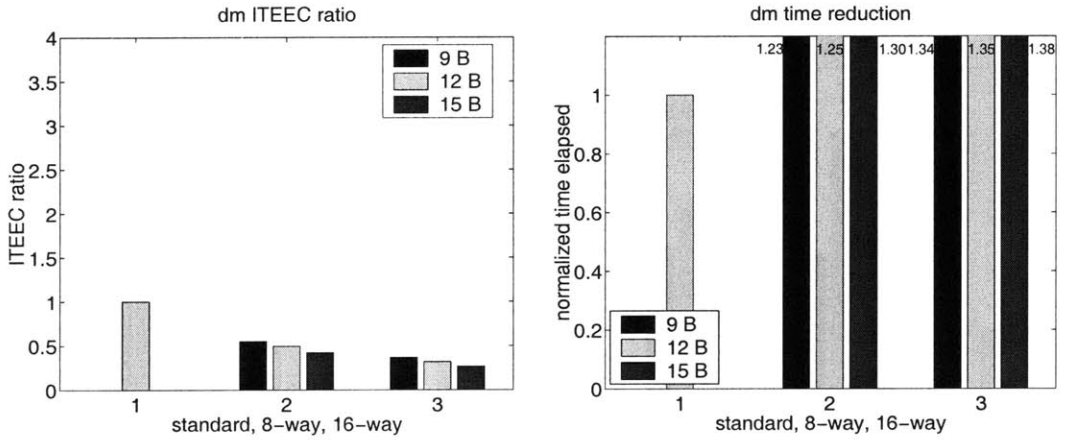

Figure A-2: ITEEC and time reduction for the **dm** benchmark: poor performance in both ITEEC and time reduction is a mystery. Its working set is smaller than **equake**, **mcf**, and **swim**, but should be large enough to see some performance improvement. Its data is fairly compressible, and although the hard to compress data is more frequently reused than the easy to compress data, the difference is not very large.

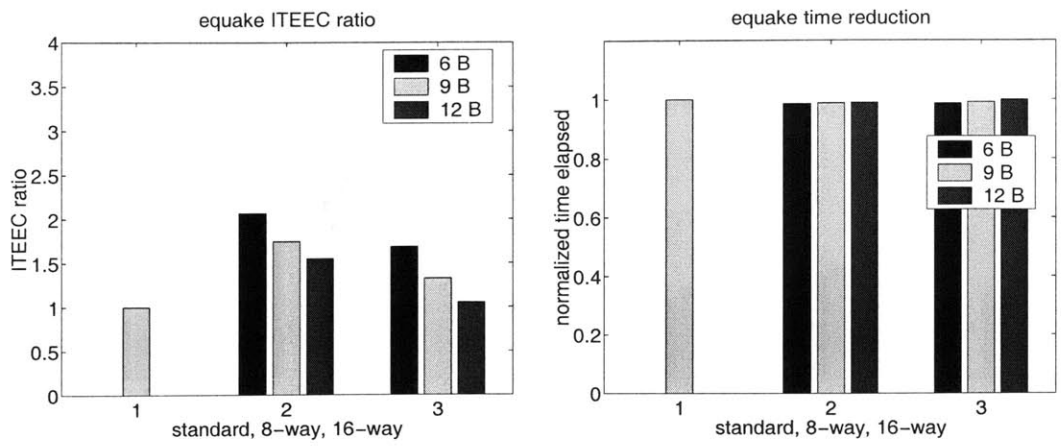

Figure A-3: ITEEC and time reduction for the **equake** benchmark: substantial performance gain in ITEEC, marginal gain in time reduction. This shows that the benchmark is at a point where increasing the cache size results in only small speed improvements. At this point, a PCC obtains the same improvements using much less hardware.

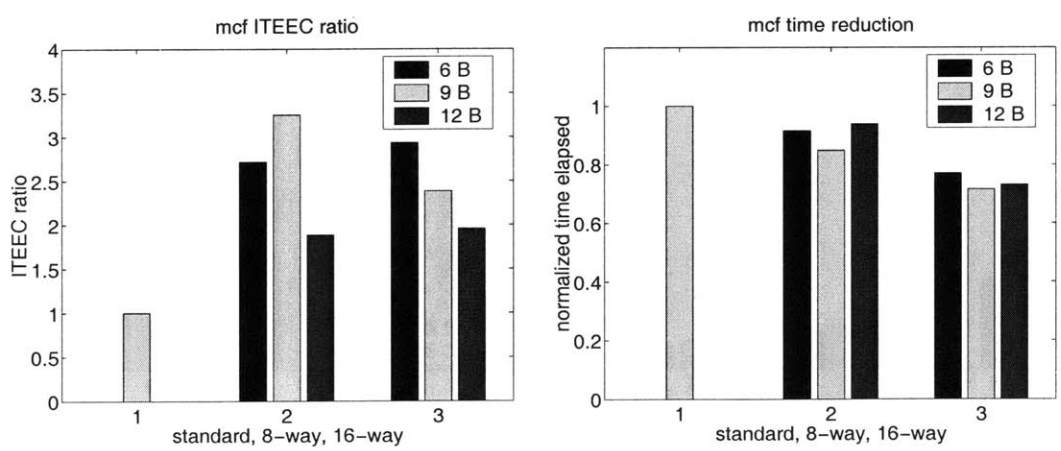

Figure A-4: ITEEC and time reduction for the `mcf` benchmark: substantial performance gain in ITEEC, fair improvement in time reduction. The difference in behavior of the two metrics in response to changes in PCC configuration show that either metric by itself is insufficient.

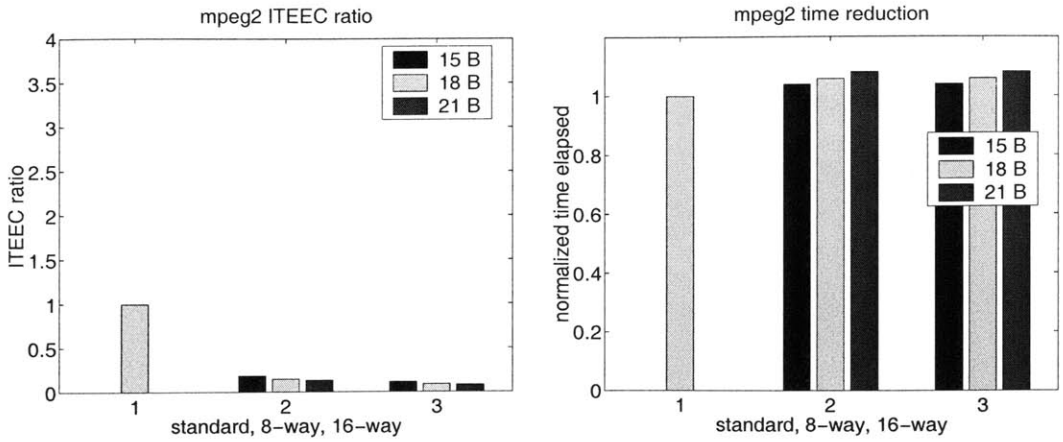

Figure A-5: ITEEC and time reduction for the `mpeg2` benchmark: poor performance in both ITEEC and time reduction. This benchmark has a relatively small footprint and its data is hard to compress.

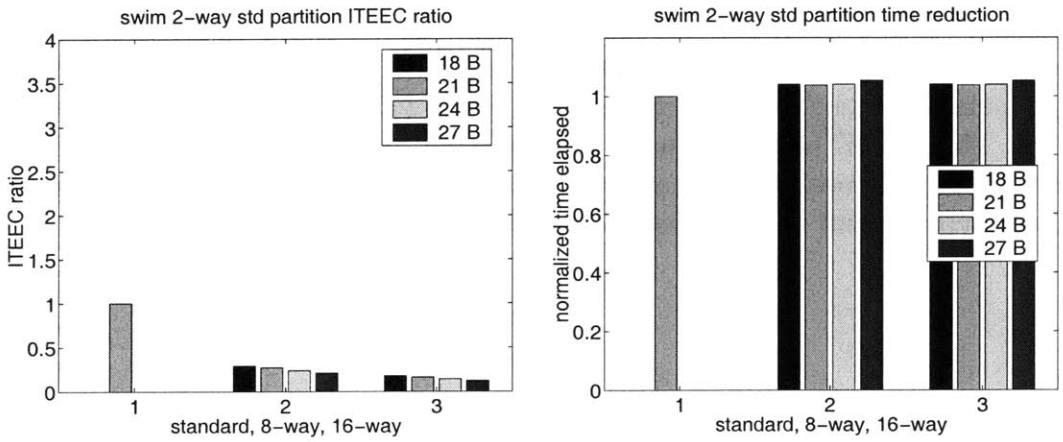

Figure A-6: ITEEC and time reduction for the `swim` benchmark: poor performance in both ITEEC and time reduction. This benchmark really needs to be able to cache 3 ways (768KB) of uncompressible data. When it can only cache 2 ways, performance is poor compared to standard caches which can store the 3 ways of uncompressible data.

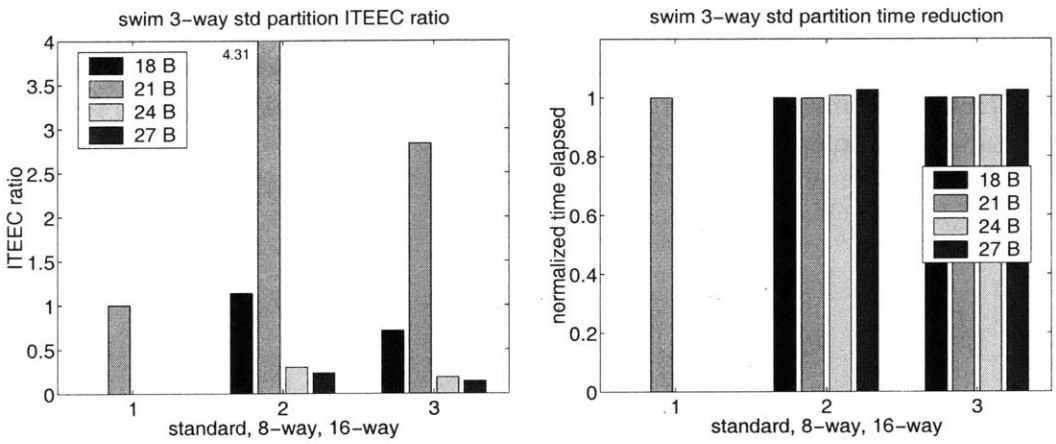

Figure A-7: ITEEC and time reduction for the **swim** benchmark, using a 3 way standard partition: very large performance gain in ITEEC, marginal gain in time reduction. The ITEEC ratios for the 21 B compressed line configurations are so large that they exceed the optimal compression ratio of the cache. This indicates that the partitioned replacement is providing huge gains.

Bibliography

[1] Data Compression Conference.

[2] B. Abali, H. Franke, X. Shen, D. Poff, and T. B. Smith. Performance of hardware compressed main memory, 2001.

[3] C. Benveniste, P. Franaszek, and J. Robinson. Cache-memory interfaces in compressed memory systems. In *IEEE Transactions on Computers, Volume #50 number 11*, November 2001.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. In *Technical report, University of Wisconsin-Madison Computer Science Department*, 1997.

[5] Miachael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 2–9, October 1992.

[6] D. T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching.* PhD thesis, Massachusetts Institute of Technology, 1999.

[7] Daniel Citron and Larry Rudolph. Creating a wider bus using caching techniques. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 90–99, Raleigh, North Carolina, 1995.

[8] M. Clark and S. Rago. The Desktop File System. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 113–124, Boston, Massachusetts, 6–10 1994.

[9] 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, 1993.

[10] Peter A. Franaszek, John T. Robinson, and Joy Thomas. Parallel compression with cooperative dictionary construction. In *Data Compression Conference*, pages 200–209, 1996.

[11] E. E. Johnson and J. Ha. PDATS: Lossless addresss trace compression for reducing file size and access time. In *IEEE International Phoenix Conference on Computers and Communications*, 1994.

[12] Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proceedings of Real Time Systems '97 (RTS97)*, 1997.

[13] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd Euromicro Conference*, pages 423–430, September 1996.

[14] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression, 1999.

[15] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 194–203, Research Triangle Park, North Carolina, December 1997.

[16] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, June 1996.

[17] Simon Segars, Keith Clarke, and Liam Goudge. Embedded control problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, 1995.

[18] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT).

In *IBM Journal of Research and Development vol. 45, No. 2*, pages 271–285, March 2001.

[19] T. Welch. High speed data compression and decompression apparatus and method, US Patent 4,558,302, December 1985.

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

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

[22] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th International Symposium on Microarchitecture*, Portland, Oregon, December 1992.

[23] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *33rd International Symposium on Microarchitecture*, Monterey, CA, December 2000.

[24] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *The 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.