The V-Way Cache : Demand Based Associativity via Global Replacement

Moinuddin K. Qureshi *David Thompson* *Yale N. Patt*

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2004-003
October, 2004

This page is intentionally left blank

The V-Way Cache : Demand Based Associativity via Global Replacement

Moinuddin K. Qureshi   David Thompson   Yale N. Patt

Electrical and Computer Engineering
The University of Texas at Austin
{moin, dave, patt}@hps.utexas.edu

Abstract

As processor speeds increase and memory latency becomes more critical, intelligent design and management of secondary caches becomes increasingly important. The efficiency of current set-associative caches is reduced because programs exhibit a non-uniform distribution of memory accesses across different cache sets. We propose a technique to vary the associativity of a cache on a per-set basis in response to the demands of the program. By increasing the number of tag entries relative to the number of data lines, we achieve the performance benefit of global replacement while maintaining the constant hit latency of a set-associative cache. The proposed variable-way, or *V-Way* set-associative cache, when combined with *Reuse Replacement* reduces the second-level cache miss rate by an average of 13%. This translates into an average IPC improvement of 8%.

1 Introduction

Cache hierarchies in modern microprocessors play a crucial role in bridging the gap between processor speed and main-memory latency. As processor speeds increase and memory latency becomes more critical, intelligent design and management of secondary caches becomes increasingly important. The performance of a cache system directly depends on its success at storing data that will be needed by the program in the near future while discarding data that is either no longer needed or unlikely to be used soon. A cache manages this through its replacement policy.

In a set-associative cache, the number of entries visible to the replacement policy is limited to the number of ways in each set. On a miss, a victim is identified from one of the ways within the set. The replacement policy could potentially select a better victim by considering the global access history of the cache rather than the localized set access history. This is particularly true since memory references in a program exhibit non-uniformity, causing some sets to be accessed heavily while other sets remain underutilized.

Ideally, to achieve the lowest possible miss-rates, a cache should be organized as fully-associative with Belady's OPT replacement [3]. However, the power, latency, and hardware costs of a fully-associative organization make it impractical, and OPT replacement is impossible to achieve without oracular knowledge of the future. In Figure 1, the upper bound on performance provided by an ideal cache indicates that there is much to be gained by improving cache organization and management.

Figure 1 shows the average reduction in miss rate for four different configurations of a second-level cache relative to a 256kB cache with 8-ways.[1] Doubling the associativity from 8-way to 16-way marginally improves performance,

[1] Section 5 discusses experimental methodology.

whereas making the cache fully-associative results in a significant performance improvement. The fully-associative cache with OPT replacement even outperforms a set-associative cache of double its size.

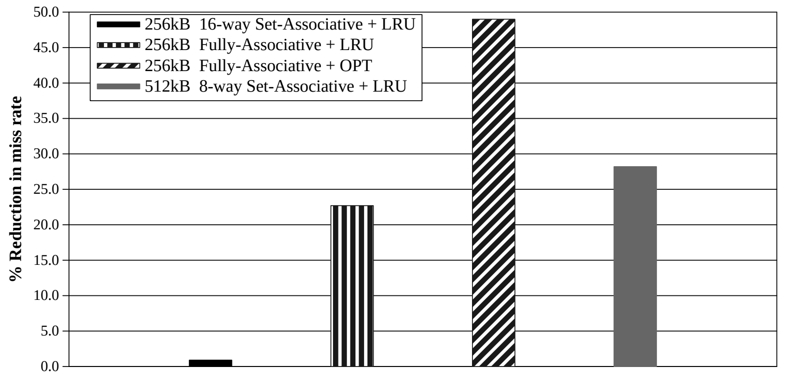

Figure 1: Percent reduction in miss rate compared to 256kB 8-way set-associative.

A fully-associative cache has two distinct advantages over a set-associative cache: *conflict-miss minimization* and *global replacement*. There is an inverse relationship between the number of conflict-misses in a cache and the associativity of the cache, and a fully-associative cache minimizes conflict-misses by maximizing associativity. Furthermore, as the associativity of a cache increases, the scope of the information used to perform replacement also increases. A four-way set-associative cache, for example, considers the four cache lines in the target set when selecting a victim for replacement. A fully-associative cache, on the other hand, benefits from considering the entire contents of the cache when selecting a victim. Global replacement allows a fully-associative cache to choose the best possible victim every time, limited only by the intelligence of the replacement algorithm. This performance comes at a great cost, however. Accessing a fully-associative cache requires a tag comparison with every tag-store entry, making both access latency and power prohibitively large.

In this paper, we propose a novel mechanism to provide the benefit of global replacement to a set-associative cache without incurring the costs of full-associativity. By increasing the number of tag-store entries relative to the number of data lines, the associativity of a cache is allowed to vary on a per-set basis in response to program demand. We call this the *Variable-Way Set Associative Cache*, or simply, the *V-Way Cache*.

To augment the V-Way cache, we propose a practical global replacement policy based on access frequency called *Reuse Replacement*. Reuse Replacement performs comparable to perfect LRU at a fraction of the cost. The proposed implementation is entirely hardware based and is fully compatible with existing Instruction Set Architectures (ISA).

The V-Way cache using Reuse Replacement achieves an average miss rate reduction of 13% on sixteen benchmarks from the SPEC CPU2000 suite. This translates into an IPC improvement of up to 44%, and an average IPC improvement of 8%.

Section 2 further motivates the proposed technique. Section 3 describes the structure of the V-Way cache, and Section 4 explains Reuse Replacement. Experimental methodology is presented in Section 5, followed by results in Section 6. Cost and performance analysis are presented in Sections 7 and 8, respectively. Related work is discussed in Section 9, and concluding remarks are given in Section 10.

2 Motivation

2.1 Problem

Memory accesses in general purpose applications are non-uniformly distributed across the sets in a cache [14][12]. This non-uniformity creates a heavy demand on some sets, which can lead to conflict misses, while other sets remain underutilized. Substantial research effort has been put forth to address this problem for direct-mapped caches. Victim caches [11] are small, fully-associative buffers that provide limited additional associativity for heavily utilized entries in a direct-mapped cache. The hash-rehash cache [1], the adaptive group-associative cache [14], and the predictive sequential-associative cache[4] trade variable hit latency for increased associativity. If the first attempt to access the cache results in a miss, the hash function that maps addresses to sets is changed, and a new cache access is initiated. This process may be repeated multiple times until either the data is found or a miss is detected. These techniques were proposed for first level direct-mapped caches, but their effectiveness reduces as associativity increases due to the inherent performance benefit of increased associativity.

Our work focuses on improving the effectiveness of large, set-associative, secondary caches.[2] Caches at this level are typically four to eight way set-associative, diminishing the impact of the techniques described above. In the *V-Way Cache*, associativity varies on a per-set basis in response to the demands of the application. By limiting the maximum degree of associativity, we maintain the constant hit latency of a set-associative cache.

2.2 Example

We illustrate the V-Way cache with an example. Consider the traditional four-way set-associative cache shown in Figure 2(a). For simplicity, the cache contains only two sets: set A and set B. The data-store is shown as a linear array for illustrative purposes. The memory references in working set X all map to set A, while working set Y maps to set B. The data lines for addresses x0, x1, etc. are represented in the figure as x0', x1', etc.

In a traditional set-associative cache there exists a static one-to-one mapping between each tag-store entry and its corresponding data-store entry. In the figure, set A is mapped to the top half of the data-store, and set B is mapped to the bottom half of the data-store.

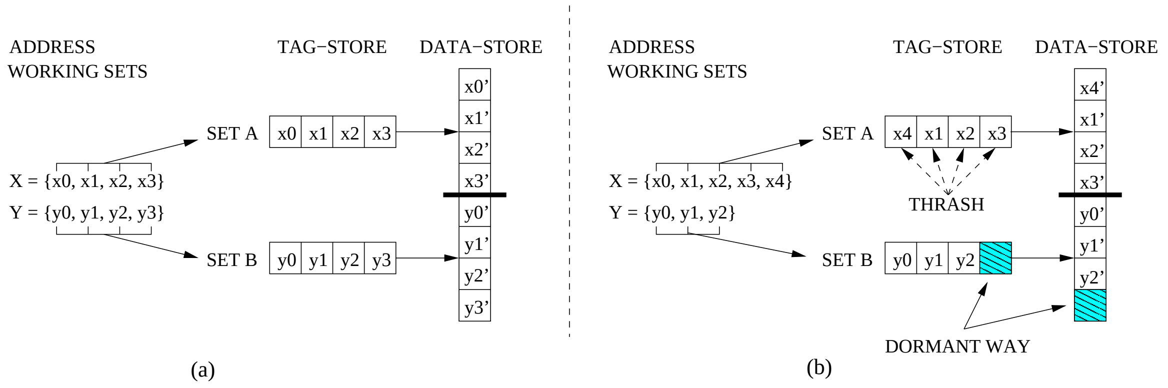

Figure 2: Traditional set-associative cache using local replacement.

[2]Though our model consists of only two levels of caches, the techniques described in this paper may be applied to all non-primary (i.e. L2, L3, etc.) caches in the memory hierarchy.

If cache accesses are totally uniform, as in Figure 2(a), the demand on sets A and B is equal, and both halves of the data-store are equally utilized. This is not the case in actual applications, however, due to hot spots in working sets.[3] In Figure 2(b), presumably at a different phase in the program, working set X increases by one element, and working set Y decreases by one element. Set A is unable to accommodate all the elements of working set X, resulting in conflict misses and potential thrashing. Set B, on the other hand, has a dormant way. If set B could share its dormant way with set A, the conflict misses would be avoided.

A traditional set-associative cache cannot adapt its associativity because lines in the data-store are *statically mapped* to sets in the tag-store. This static partitioning necessitates local replacement. When a cache miss occurs, a victim is identified within the target set, and the corresponding entries in the tag-store and data-store are replaced. We refer to this as *local replacement*. This combination of static mapping and local replacement prevents traditional caches from exploiting underutilized sets and results in reduced cache performance.

2.3 Solution

Increasing the number of tag-store entries relative to the number of data lines provides the flexibility to accommodate cache demand on a per-set basis. Figure 3(a) shows the same example from Figure 2, except the number of tag-store entries in the cache has doubled. Doubling the number of tag-store entries is relatively inexpensive, typically adding 2-3% to the overall storage requirements of a secondary cache. The extra tag-store entries have been added as additional sets rather than ways in order to keep the number of tag comparisons required on each access unchanged at four. The number of data lines remains constant. Note that the total number of *valid* tag-store entries also remains constant. Invalid entries are shaded.

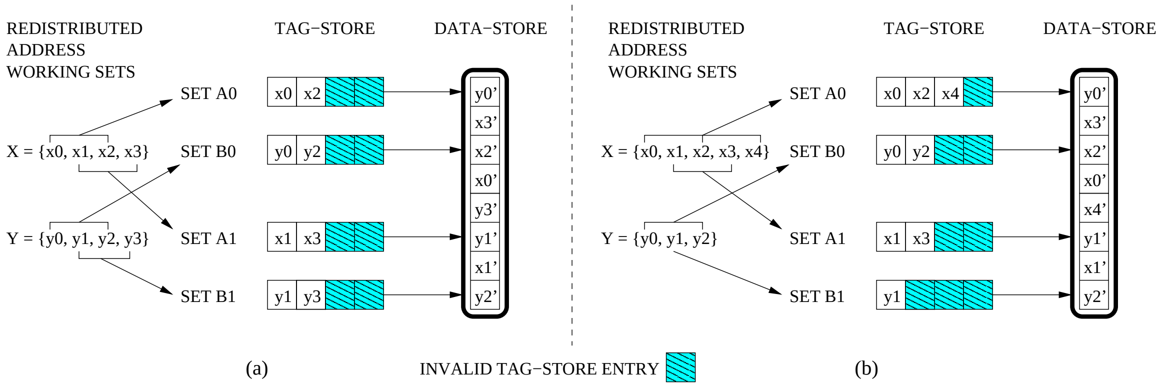

Figure 3: Variable-way set-associative cache using global replacement.

Increasing the size of the tag-store creates the following effects:

- *The memory references are re-distributed across the cache sets.*
  After doubling the number of tags, the number of index bits increases by 1. The new most-significant bit of the index re-distributes working set X across sets A0 and A1. Working set Y is similarly re-distributed across sets B0 and B1.

[3]Section 8.4 describes the problem of variable set demand in more detail.

- *There no longer exists a static one-to-one mapping between tag-store entries and data lines.*
  Every valid tag entry now contains a pointer to a location in the data-store. This mapping may change dynamically and implies that tag comparison and data lookup must be performed serially.[4]

With twice as many tag-store entries as data lines, each set in the cache will contain, on average, two out of four valid entries. As the demand on individual sets fluctuates, the cache responds by varying the associativity of the individual sets, as shown in Figure 3(b). When working set X increases by one element, the demand on set A0 increases, and a new tag-store entry and data line must be allocated. As in Figure 2(b), there exists a dormant way, now in set B1. The data line associated with the dormant tag-store entry is detected by a global replacement policy and allocated to the new tag-store entry in set A0. The tag-store entry of the dormant way (previously belonging to y3) is then invalidated. The presence of additional tags, combined with the use of a global replacement policy, allows the associativity of sets A0 and B1 to vary in response to changing demand.

3 V-Way Cache

3.1 Terminology

Figure 4 shows the structure of the V-Way cache. The defining property of the V-Way cache is that there are more tag-store entries than data lines. We define the tag-to-data ratio (TDR) as the ratio of the number of tag-store entries to the number of data lines, where $\text{TDR} \geq 1$. The case $\text{TDR} = 1$ is equivalent to a traditional cache. $\text{TDR} < 1$ is invalid because there must be at least one tag-store entry for every data line. In the example in section 2, $\text{TDR} = 2$ because there are twice as many tag entries as data lines. Unless otherwise specified, we assume $\text{TDR} = 2$ for the remainder of the paper.

3.2 Structure

The V-Way cache consists of two decoupled structures: the tag-store and the data-store. Each entry in the tag-store contains status information (valid bit, dirty bit, replacement information), tag bits, and a forward pointer (FPTR) which identifies the entry in the data-store to which the tag entry is mapped. If the valid bit in a tag-store entry is cleared, all other information in the entry is considered invalid, including the FPTR. Each data-store entry contains a data line, replacement information, and a reverse pointer (RPTR). The RPTR identifies a unique entry in the tag-store. For every valid tag-store entry, there exists a (FPTR, RPTR) pair that point to each other.

The tag-store and data-store form two structurally independent entities linked only by the FPTR and RPTR, and both structures implement independent replacement algorithms. The tag-store uses a traditional replacement scheme such as least recently used (LRU) on a local, or per-set, basis. The data-store uses frequency information to implement global replacement. We describe the global replacement scheme in detail in section 4.

[4]Existing processors use this serialization technique to reduce the power dissipation of large cache arrays[7][19]. Prior research has made use of serialization to increase flexibility and to improve performance in large caches [9][5].

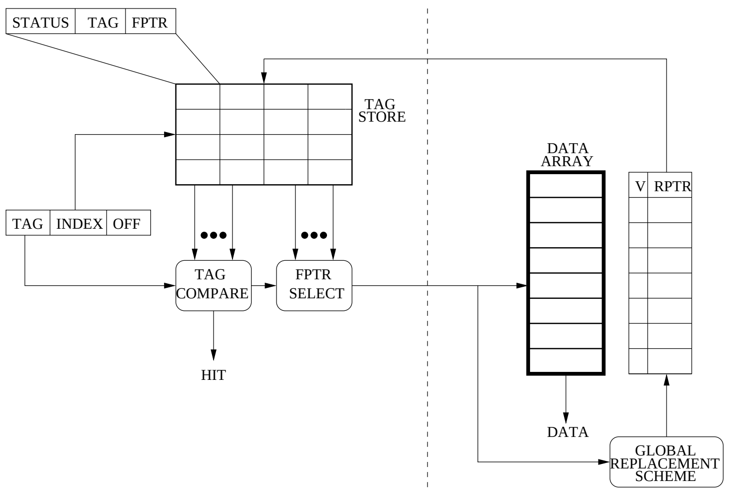

Figure 4: V-Way Cache.

3.3 Operation

Tag-store accesses are exactly the same as in a traditional set-associative cache. If the access hits in the cache, the FPTR of the matching entry is used to directly index the corresponding line in the data-store. Replacement information is appropriately updated in both the tag-store and the data-store after each access. When a cache miss occurs, two victims must be identified: a *tag-victim* and a *data-victim*. The tag-victim is always one of the entries in the target set of the tag-store and is chosen before the data-victim. The selection of a data-victim is based on one of two scenarios that can arise when selecting the tag-victim:

- *There exists at least one invalid tag-store entry in the target set.*

  Since there are twice as many tags as data lines, the probability of finding an invalid tag-store entry in the target set is high. In twelve out of sixteen benchmarks studied, more than 90% of the tag-victims were provided by invalid entries. When this occurs, the data-victim is supplied by the data-store's global replacement policy, and the tag-store entry identified by the RPTR of the data-victim is invalidated. The data-victim is then evicted from the cache, and a write-back is scheduled if necessary, followed by a line fill with the new data. The tag-victim is updated with the new tag bits, the FPTR is updated to point to the data-victim, and the valid bit is set. The RPTR of the data-victim is then updated to point to the newly validated tag-store entry. Finally, replacement information is updated in both the tag-store and the data-store.

- *All the tag-store entries in the target set are valid.*

  In this uncommon case, the tag-victim is chosen using the local replacement scheme of the tag-store. We use LRU as the local replacement scheme in our experiments. The tag-victim in this case contains a valid FPTR, and the data line to which it points is used as the data-victim, bypassing the data-store's global replacement policy. The existing data line is evicted from the cache, and a write-back is scheduled if necessary, followed by a line fill with the new data. The RPTR remains unchanged, as it already points to the correct entry in the tag-store. Replacement information is then updated appropriately in both the tag-store and data-store.

In a traditional set-associative cache, after an initial warm-up period all the tag-store entries in the cache are valid, barring any invalidations that occured due to the implementation of a cache consistency protocol. In the V-Way cache, however, each time the data-store's global replacement engine is invoked to find a data-victim,a way that is unlikely to be used in the near future is replaced. The V-Way cache in Figure 4 has a maximum associativity of four-ways, but the V-Way cache technique can be applied, in general, to any non fully-associative cache. A V-Way cache can potentially achieve miss-rates comparable to a traditional cache of twice its size or a fully-associative cache of the same size. The success of the V-Way cache depends on how well the global replacement engine chooses data-victims. We describe the global replacement algorithm and implementation in the next section.

4 Designing a Practical Global Replacement Algorithm

The success or failure of the V-Way cache depends on the intelligence of the global replacement policy. A naive policy such as random or FIFO replacement increases the miss-rate when compared to the baseline configuration(TDR=1). Perfect LRU is far more effective than random, but has a space complexity of $O(n^2)$[18]. Considering the fact that large caches typically contain thousands of data lines, perfect LRU is an impractical choice. Two-handed clock replacement[6], commonly used for page replacement in an operating system, uses only a single bit per entry. Though inexpensive, it does not provide performance comparable to perfect LRU when applied to caches[9].

Our goal is to design a replacement algorithm that yields the performance of perfect global LRU scheme at a substantially lower cost in both hardware and latency. We start by examining the characteristics of the memory reference stream presented to the second level cache.

4.1 Reuse Frequency

The stream of references that access the second level cache (L2) is a filtered version of the memory reference stream seen by the first level cache (L1). In other words, only those addresses that miss in the L1 are propagated to the L2. This filtering results in a loss of temporal locality information for L2 cache lines while they are in L1.

Temporal locality as seen by the second level cache is only recognized when cache lines are evicted from the L1 and subsequently re-accessed before being evicted from the L2. This view of temporal locality in the L2 is totally independent of the number of accesses to the cache line before its eviction from the L1 and after its subsequent reinstatement, typically resulting in a much lower measure of locality than recorded by the L1. We define *reuse count* as the number of L2 accesses to a cache line *after* its initial line fill. When an L2 cache line is installed, its reuse count is initialized to zero and then incremented by one for each subsequent L2 access to the cache line. When a line is evicted from the L2 cache its reuse count is read, and the appropriate bucket of the global reuse distribution is incremented by one. Figure 5 shows the distribution of reuse counts for all evicted L2 cache lines from all sixteen benchmarks using a baseline cache configuration.

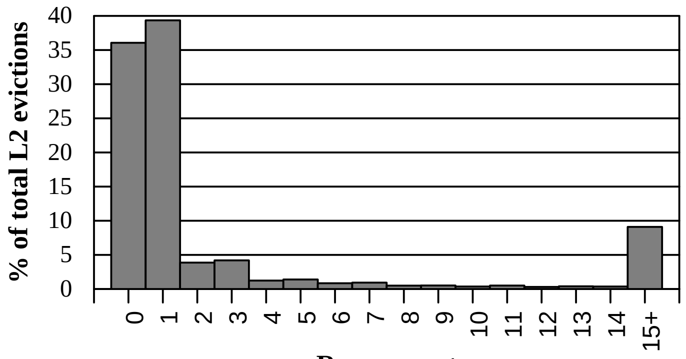

Figure 5: Distribution of L2 cache line reuse.

4.2 Cost Effectiveness of Frequency Based Replacement

Prior research has identified the significance of access frequency in relationship to cache performance [17]. Several authors have proposed the use of frequency information for placement and replacement of cache lines. Johnson[10] uses frequency information to make cache bypass decisions. A non-uniform cache architecture [13] uses access frequency to direct the placement of a cache line to an appropriate distance group. Reinhardt et al [9] use access frequency with hysteresis to implement generational replacement. Generational replacement is expensive in terms of both hardware and management complexity. The proposed implementation requires 33 bits of information per entry, and the data structures involved must be either managed by software or controlled by a dedicated micro-engine. Reuse, on the other hand, requires far less storage to maintain comparable information. Looking at Figure 5, over 80% of L2 cache lines are reused three or fewer times. Fewer than 10% of the lines are reused more than 14 times. Two bits can be used to track four unique reuse count states: 0, 1, 2, and 3+.

4.3 Reuse Replacement

We propose *Reuse Replacement*, a frequency based global replacement scheme that is both fast and inexpensive. Figure 6(a) shows the structures required to implement Reuse Replacement. Every data line in the cache has an associated reuse counter. The reuse counters are two-bit saturating counters and are kept in a structure called the *Reuse Counter Table (RCT)*. The RCT may be physically separate from the cache to avoid accessing the data-store when reading or updating the reuse counters. A *PTR* register points to the entry in the RCT where the global replacement engine will begin searching for the next data-victim.

When a cache line is installed in the cache, the reuse counter associated with the data-store entry is initialized to zero. For each subsequent access to the cache line, the reuse counter is incremented using saturating arithmetic. When a cache miss occurs, the global replacement engine searches the RCT for a reuse counter equal to zero. Starting with the counter indexed by PTR, the replacement engine tests and decrements each non-zero reuse counter. Testing continues until a data-victim is found, wrapping around when the bottom of the RCT is reached. Once a data-victim is found, PTR is incremented to point to the next reuse counter. Incrementing PTR causes every other counter in the RCT to be tested (and decremented) exactly once before the current counter gets tested for the first time. This allows

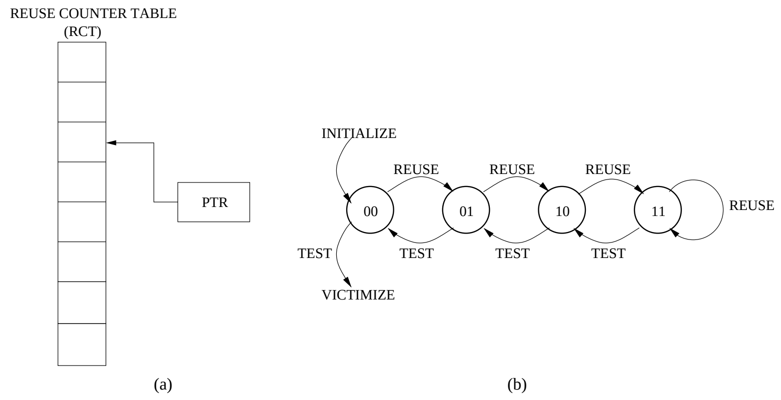

Figure 6: Reuse Replacement : (a) RCT and PTR register. (b) State machine for the reuse counters.

the reuse counters for newly installed cache lines to reach a representative value before being tested and decremented. The algorithm is shown in Figure 7 using C style syntax.

```
NUM_COUNTERS = Total number of reuse counters
RCT[]        = Reuse Counter Table
PTR          = pointer into RCT

find_victim(){
  found = FALSE ;
    while(found != TRUE){       /* REPEAT TILL VICTIM IS FOUND */
      if (RCT[PTR] == 0){        /* TEST  */
           found  = TRUE ;       /* VICTIM FOUND */
           victim = PTR  ;       /* VICTIM IS THE ENTRY POINTED BY PTR */
       }
       else{
           RCT[PTR]--;           /* DECREMENT COUNTER VALUE */
       }
       PTR=(PTR+1)%NUM_COUNTERS; /* PTR POINTS TO NEXT ENTRY */
    }
  return victim;
}
```

Figure 7: Reuse Replacement algorithm.

4.4 Variable Replacement Latency

Although Reuse Replacement is guaranteed to find a victim, the time required to do so can vary depending on the overall level of reuse in the program. We refer to the *victim distance* as the number of times the PTR register is incremented before a victim is found. In the theoretical worst case, where every reuse counter in the RCT is saturated, the victim distance will have the value $(2^N - 1) \times$NUM_COUNTERS for an N-bit reuse counter. For the V-Way cache simulated in this paper, using two-bit reuse counters, the theoretical maximum victim distance is 6144. We expect the typical victim distance to be substantially lower than the theoretical maximum for two reasons.

First of all, the majority of cache lines exhibit little reuse, as shown in Figure 5. Second, decoupling the tag-store entries from the data-store entries has the effect of randomizing the cache lines in the data storage, reducing the likelihood of stride-based memory access patterns generating long victim distances. Table 1 shows the average and worst-case victim distances for each of the benchmarks studied.

Table 1: Average and observed worst case victim distance for Reuse Replacement.

Bmk	bzip2	crafty	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr	ammp	apsi	facerec	galgel	mesa	swim
Avg	2.2	4.7	4.4	2.8	2.8	2.2	18	2.1	4.5	4.6	2	2.7	1.5	3.4	3.6	1.6
Worst	462	258	140	1888	1741	706	1504	298	743	434	28	1593	1504	1338	225	212

The average victim distance in Table 1 is less than five for all benchmarks except perlbmk. The worst case can be several orders of magnitude greater than the average, as it is with gzip and mcf. This variability in the victim distance is unlikely to cause the processor to stall, however. The deadline for selecting a data-victim is the arrival of the incoming data line from the next level of the memory hierarchy, which could be tens or hundreds of cycles.

Furthermore, the logic associated with identifying a data-victim is very simple. Testing a two-bit counter for a zero value can be done with a single NOR gate. A simple logic circuit containing eight parallel NOR gates followed by an 8:3 priority encoder can test and decrement up to 8 reuse counters each cycle based on a gate-delay timing budget of 12F04 (twelve fanout-of-four gate stages). Assuming that eight counters can be tested in one cycle, Table 2 shows the probability of finding a victim as search time increases, based on experimentation.

Table 2: Probability of finding a data-victim as replacement latency increases.

Latency	1 cycle	2 cycles	3 cycles	4 cycles	5 cycles
Probability	91.3%	96.9%	98.3%	98.9%	99.2%

The probability of finding a victim within five cycles is 99.2%. This is well below the miss latency of modern secondary caches. To avoid the latency of worst case victim-distances, however, the global replacement engine may simply terminate the search after five cycles and use the entry pointed by the PTR as the data-victim. Early termination limits the worst case replacement latency to five cycles, yeilds an average replacement latency of 1.2 cyles, and has a negligible impact on miss-rate ($<0.1\%$).

5 Experimental Methodology

The primary performance metric we use to evaluate the V-Way cache is miss rate. We used a trace driven cache simulator to generate all results except IPC. In section 8.1 we analyze the impact of a V-Way cache on overall processor performance (IPC) using an out-of-order, execution-driven simulator. We defer the description of the simulator configuration to that section.

5.1 Cache Hierarchy

Table 3 shows the parameters of the first level instruction (I) and data (D) caches that we used to generate the traces for our second level cache. The L1 cache parameters were kept constant for all experiments. Our baseline includes a 256kB unified second level cache with 8 ways. The size of our first and second level cache is modeled after the Itanium II processor [19]. The benchmarks used in this study do not stress a very large sized cache; therefore, we chose a moderately sized L2. Section 8.3 analyzes the effectiveness of V-Way cache when the cache size is increased. We do not enforce inclusion in our memory model.

Table 3: Configuration for the I-cache, D-cache and baseline L2 cache.

L1 I-Cache	16kB 64B line-size 2-way set-associative with LRU replacement
L1 D-Cache	16kB 64B line-size 2-way set-associative with LRU replacement
Baseline L2	256kB 128B line-size 8-way set-associative with LRU replacement

5.2 Benchmarks

The benchmarks used for all experiments were selected from the SPEC CPU2000 suite and compiled for the Alpha ISA with `-fast` optimizations and profiling feedback enabled. To skip the initialization phase, all benchmarks with ref input set were fast forwarded for 15 Billion instructions and simulated for 2 Billion instructions. For benchmarks bzip2, gcc, vpr, and ammp a slice of 2 Billion instruction with ref input was unable to capture the varying phase behavior. For these benchmarks, the experiments were run with test input from start to completion except in case of ammp which was halted at 1 Billion instructions. Benchmarks eon and fma3d showed extremely low miss-rates ($<$0.1%) for the baseline configuration and were thus excluded from the study. We also excluded benchmarks that showed less than 4% difference in miss rate when the L2 cache size was doubled from 256kB to 512kB. Based on this criteria, gap, art, applu, equake, lucas, mgrid, sixtrack, and wupwise were eliminated from consideration.

All experiments were run without warming up the caches prior to execution. Table 4 lists the input set, the simulation interval, the total instruction count, the number of L2 cache accesses, and the total size of the L2 footprint for each benchmark. The L2 footprint consists of both data and instruction accesses and is measured by multiplying the number of unique L2 cache lines by the L2 line-size (128 bytes).

Table 4: Benchmark characteristics.

Benchmark	Input set	Simulation interval	Inst cnt.	L2 accesses	Footprint
bzip2	test	complete benchmark	418 M	4.2 M	6.8 MB
crafty	ref	fast forward 15B run 2B	2000 M	117 M	1.5 MB
gcc	test	complete benchmark	218 M	6.7 M	1.7 MB
gzip	ref	fast forward 15B run 2B	2000 M	37 M	69 MB
mcf	test	complete benchmark	173 M	15 M	193 MB
parser	ref	fast forward 15B run 2B	2000 M	30 M	15 MB
perlbmk	ref	fast forward 15B run 2B	2000 M	3.4 M	3 MB
twolf	ref	fast forward 15B run 2B	2000 M	59 M	1.7 MB
vortex	ref	fast forward 15B run 2B	2000 M	5.8 M	19 MB
vpr	test	complete benchmark	567 M	16 M	1.7 MB
ammp	test	capped at 1B	1000 M	98 M	9.9 MB
apsi	ref	fast forward 15B run 2B	2000 M	59 M	129 MB
facerec	ref	fast forward 15B run 2B	2000 M	23 M	15 MB
galgel	ref	fast forward 15B run 2B	2000 M	218 M	16 MB
mesa	ref	fast forward 15B run 2B	2000 M	17 M	8.1 MB
swim	ref	fast forward 15B run 2B	2000 M	90 M	177 MB

6 Results

6.1 Performance of V-Way Cache

Figure 8 shows the relative miss-rate reduction for three different cache configurations compared to the baseline cache described in Section 5.1. The V-Way cache has a maximum associativity of 8 ways, and both the V-Way and the fully-associative cache have a 256kB data-store. The third cache configuration is a traditional set-associative cache with the same line-size and associativity as the baseline, but the data-store is doubled to 512kB. The bar marked *amean* was computed by first taking the arithmetic mean of the miss-rates for a given configuration, then comparing this value to the arithmetic mean of the miss-rates for the baseline cache. Thus, the V-Way cache reduces the miss rate on average by 13.2%.

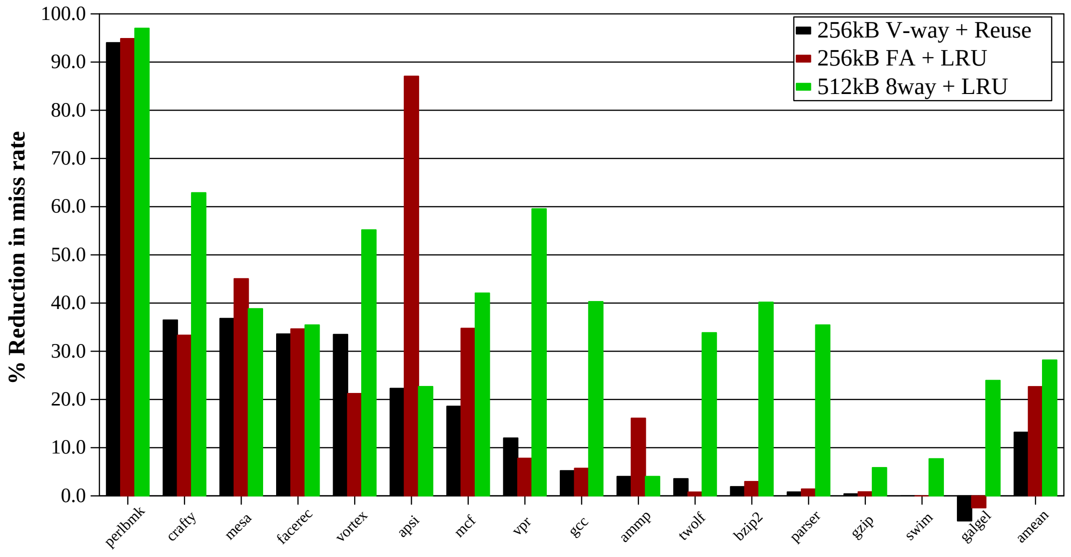

Figure 8: Reduction in miss-rates with : V-way cache (max 8 way), fully-associative cache, and double sized baseline.

Perlbmk, crafty, mesa, facerec and vortex show a miss-rate reduction of 25% or greater using the V-Way cache. The significant improvement that the V-Way cache provides to perlbmk (94%) is due to its small data working set. Doubling the size of the baseline to 512kB eliminates nearly all of the cache misses for perlbmk, reducing the total miss-rate by 97%. In case of galgel, both the fully-associative and the V-Way cache increases the miss rate as compared to the baseline.

Perlbmk, crafty, facrec, apsi, and mcf show a significant reduction in miss rate when a fully-associative cache is used due to the benefit of global data replacement. These benchmarks show a similar improvement using the V-Way cache. In some cases, such as crafty, vortex, vpr, and twolf, the V-Way cache even outperforms the fully-associative cache. This is due to differences in the replacement policy and will be explored further in Section 6.2.

Because the primary benefit of the V-Way cache is global replacement, we do not expect the V-Way cache to significantly improve performance for benchmarks that are insensitive to full-associativity, such as bzip2, parser, gzip and swim.

6.2 Comparing Reuse Replacement to Perfect LRU

When a cache miss occurs, the V-Way cache may or may not use global replacement, depending on whether or not an invalid entry is found in the target set of the tag-store. Table 5 shows the percentage of cache misses that invoked global replacement to find a data-victim.

Table 5: Percentage of misses that invoke global replacement.

bzip2	crafty	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	ammp	apsi	facerec	galgel	mesa	swim
98%	90%	99%	99.9%	70%	98%	91%	98%	95%	98.4%	0.3%	14%	96.5%	99.9%	91%	87%

For twelve of the sixteen benchmarks, global replacement is invoked on more than 90% of the misses. Benchmarks mcf, ammp, and applu are often forced to use local replacement when the demand on sets in the tag-store exceeds the maximum available associativity. Ammp uses local replacement almost exclusively, invoking global replacement for only 0.3% of the L2 misses. The highly skewed set-demand in these benchmarks prevents the use of global replacement and reduces the overall impact of the V-Way technique.

One of the most surprising results from Figure 8 is that the V-Way cache outperforms the fully-associative cache for crafty, twolf, vortex and vpr. This result arises from differences in behavior of the replacement policies used by the two caches. The fully-associative cache uses perfect LRU replacement, whereas the V-Way cache uses Reuse Replacement. Perfect LRU requires that every line remain resident in the cache until all other cache lines have been either accessed or evicted. Reuse Replacement, on the other hand, may test and decrement several lines each time the replacement is invoked, evicting low-reuse data lines more quickly than LRU. Also, Reuse Replacement can potentially retain high reuse lines four times longer than perfect LRU. Table 6 compares a V-Way cache using perfect LRU replacement to a V-Way cache using Reuse Replacement.

Table 6: Comparison of miss rate (lower is better) for perfect LRU replacement and Reuse Replacement. The miss rate of benchmarks for which Reuse Replacement is better than LRU is typed in **bold**

Bmk	bzip2	crafty	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	ammp	apsi	facerec	galgel	mesa	swim	amean
LRU	34.6	1.1	3.8	2.4	29.5	32.7	0.1	36.5	8.5	11.0	50.0	34.8	50.7	8.3	3.4	65.3	23.3
Reuse	35.0	**1.0**	3.8	2.4	29.9	32.9	0.1	**35.4**	**7.1**	**10.5**	50.0	34.8	**50.6**	8.5	3.5	65.3	**23.2**

For vortex and vpr, Reuse Replacement significantly outperforms perfect LRU, whereas perfect LRU outperforms Reuse Replacement in bzip2, mcf and galgel. Overall, Reuse Replacement is better than LRU for five benchmarks, there is a tie for six benchmarks, and LRU is better for the remaining five benchmarks. On average, Reuse Replacement performs marginally better than perfect LRU (at a substantially lower cost and complexity).

7 Cost

In this section we evaluate the storage, latency, and energy costs associated with the V-Way cache. Storage is measured in terms of register bit equivalents (RBE). To model cache access latency and energy we used Cacti v.3.2[20].

7.1 Storage

The additional hardware for the V-Way cache consists of the following:

- Extra tags. The exact number is determined by the TDR
- Forward pointers (FPTR) for each tag-store entry
- Reverse pointers (RPTR) + Reuse Counters for each data-store entry

The total storage requirements for both the baseline and the V-Way cache are calculated in Table 7. A physical address space of 36 bits is assumed.

Table 7: Storage cost analysis.

	Baseline	V-Way Cache
Each tag-store entry contains status	5 bits (v+dirty+LRU)	5 bits (v+dirty+LRU)
tag	21 bits (36-$log_2$256-$log_2$128)	20 bits (36-$log_2$512-$log_2$128)
FPTR	–	11 bits ($log_2$2048)
Size of each tag-store entry	26 bits	36 bits
Each data-store entry contains status	–	3 bits (v+reuse)
data	128*8 bits	128*8 bits
RPTR	–	12 bits ($log_2$4096)
Size of each data-store entry	1024 bits	1039 bits
Number of tag-store entries	2048	4096
Number of data-store entries	2048	2048
Size of tag-store	6.7 kB	18.4 kB
Size of data-store	256 kB	259 kB
Total Size (tag-store + data-store)	262.7 kB	277.4 kB

For the experiments in this paper, the overhead of the V-Way cache increases the total area of the baseline cache by 5.8%. This overhead depends on line-size, however. Table 8 shows the cost and performance benefit for various line-sizes. As the line-size increases, the benefit provided by V-Way increases and the storage overhead decreases.

Table 8: Cost-benefit analysis of V-Way cache for various line-sizes.

Line-size	Baseline miss-rate	V-Way cache miss-rate	Miss-rate reduction	Increase in area
64 B	34.2%	30.6%	10.5%	11.6%
128 B	26.7%	23.2%	13.2%	5.8%
256 B	22.9%	19.5%	14.9%	2.9%

7.2 Latency

The V-Way cache incurs a latency penalty due to the additional tag-store entries combined with the addition of the FPTR to each individual entry. The access latency is also extended by a mux delay to select the correct FPTR. Table 9 shows the access time for two process technologies: 65 nm and 90nm.

Table 9: Cache access latency.

Technology	Configuration	Tag access time	Total Access time (tag+data)
90nm	Baseline 256kB	0.48ns	2.45ns
	V-Way 256kB	0.67ns	2.64ns
65nm	Baseline 256kB	0.35ns	1.76ns
	V-Way 256kB	0.48ns	1.89ns

The latency overhead of the V-Way cache is 0.19ns in 90nm technology and 0.13ns in 65nm technology. This delay can further be reduced with circuit level optimizations. This added latency may result in at-most one extra cycle for the cache access.

7.3 Energy

The additional tag-store entries and control information in the V-Way cache consumes energy. Table 10 shows the energy required per access for the baseline and V-Way cache. Both the baseline and V-Way use serial tag and data lookup. For reference, we have also tabulated the energy required per cache access for the case when tag and data lookup is done in parallel.

Table 10: Energy per cache access.

Technology	Parallel-lookup 256kB	Baseline 256kB	V-Way 256kB
90nm	1.52nJ	0.65nJ	0.73nJ
65nm	1.02nJ	0.35nJ	0.40nJ

Both the baseline and the V-Way cache reduce energy considerably compared to the parallel-lookup cache. The additional tag-store entries in the V-Way cache increase the energy per access only marginally, adding 0.07nJ in 90nm technology and 0.05nJ in 65nm technology.

8 Analysis

In this section, we present the impact of the V-Way cache on overall processor performance. We also evaluate the performance of the V-Way cache for different TDR values and cache sizes. Finally, we provide some intuition behind what makes the V-Way cache work, and we discuss the limitations of the technique.

8.1 Impact on System Performance

To evaluate the effect of the V-Way cache on overall processor performance, we use an in-house execution-driven simulator based on the Alpha ISA. The processor we model is an eight-wide machine with out-of-order execution. Tag comparison and data lookup are serial operations in the baseline L2 cache, resulting in a hit-latency of 10 cycles. The relevant parameters of the model are given in Table 11. As the baseline L2 is 256kB, we assume that the next level in the memory hierarchy is just 80 cycles away.

Table 11: Baseline processor configuration.

Fetch/Issue/Retire Width	8 instructions/cycle, 8 functional units
Instruction Window Size	128 instructions
Branch Predictor	hybrid with 64K entry gshare and 64K entry PAs
Branch Misprediction Penalty	12 cycles minimum
L1 Instruction Cache	16kB, 2-way, LRU replacement, 64B linesize
L1 Data Cache	16kB, 2-way, 2-cycle, LRU repl. 64B linesize
L2 Unified Cache	256kB, 8-way, 10-cycle, LRU repl. 128B linesize
L3/Main Memory	Infinite size, 80 cycle access
L3/Main Memory to L2 bus	Processor to bus frequency ratio 4:1, Latency one bus cycle, Bandwidth 16B per bus cycle

Figure 9 shows the performance improvement measured in instructions per cycle (IPC) between the baseline processor and the same processor with a V-Way L2 cache. The bar labeled *gmean* is the geometric mean of the individual IPC improvements seen by each benchmark. IPC improvements are shown for both 10 and 11 cycle access latencies.

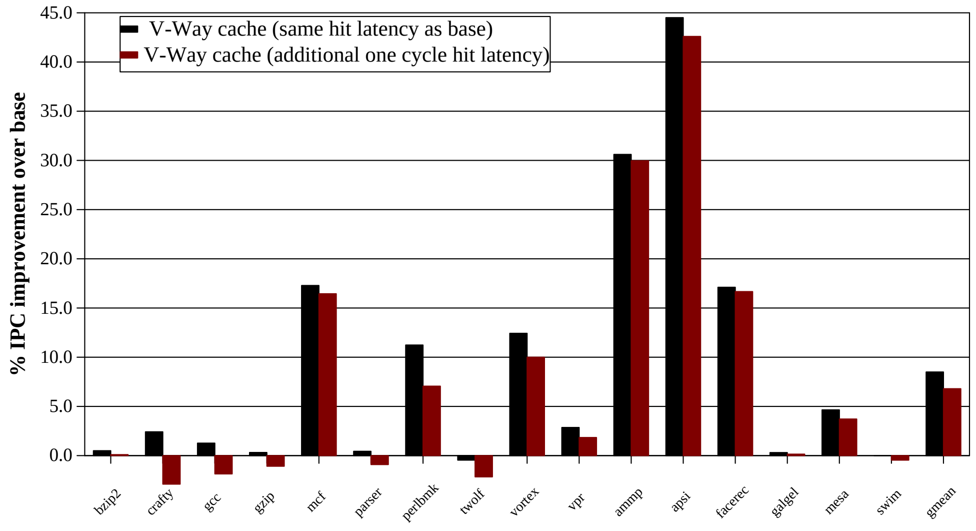

Figure 9: Percentage IPC improvement over the baseline for a system with a V-Way L2 cache.

The processor with the V-Way cache outperforms the baseline by an average of 8.5% for a 10 cycle latency. If the latency of the V-Way cache increases by one cycle, the IPC improves by an average of 6.8% compared to the baseline. Mcf, perlbmk, vortex, ammp, apsi, and facerec show the greatest individual performance improvements using the V-Way cache. The greatest performance improvement is seen in apsi, where IPC increases by 44% for a 10 cycle V-Way cache.

All benchmarks, except twolf, show an IPC improvement at an access latency of 10 cycles. For crafty, gcc, perlbmk, and vortex adding an additional cycle of latency results in a considerable decrease in IPC. This can be attributed to the relatively large instruction working set of these benchmarks. While out-of-order execution can hide the latency of a first level data cache miss by executing additional instructions, misses in the first level instruction cache result in pipeline stalls. In such cases, the additional cycle in the L2 access latency reduces the IPC improvement of global replacement.

8.2 Impact of Varying Tag-to-Data Ratio

Our previous results have assumed TDR = 2. Here, we analyze the impact on miss-rate when the TDR is varied. Figure 10 shows the reduction in miss-rate relative to the baseline for several different TDR values in a V-Way cache. The four benchmarks are chosen to illustrate different program behavior.

Power-of-two TDR values, such as 2 or 4, cause the number of sets in the tag-store to be doubled, quadrupled, etc. while associativity is held constant. For non-power-of-two TDR values, the number of sets in the tag-store is first increased to the largest possible power-of-two. Additional tag-store entries are then added as complete ways until the TDR is satisfied.

Because a V-Way cache with TDR = 1 is equivalent to the baseline, all four curves originate at 0%. The general trend for all four benchmarks is that the miss-rate decreases as the TDR increases. For mcf, and vortex, the reduction in miss-rate grows linearly until a “knee” is encountered in the curve, beyond which the miss-rate remains fairly

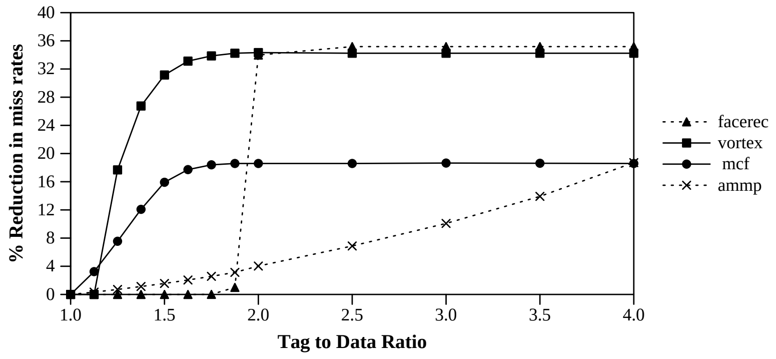

Figure 10: Miss-rate reduction (higher is better) relative to baseline vs. TDR.

constant. The curve for facerec resembles a step-function, showing a sharp improvement in miss-rate for a TDR = 2, but otherwise insensitive to TDR variation. Ammp never saturates, showing a steady reduction in miss rate as the TDR increases from one to four.

The V-Way cache exploits the benefit of global data replacement through additional tag-store entries and variable associativity. One measure of the success of the V-Way cache is the percentage of data-victims chosen using Reuse Replacement as opposed to local replacement (see Table 5). As the TDR increases, the probability of finding an invalid tag-store entry on a cache miss also increases, resulting in selection of the data-victim via Reuse Replacement. Saturation occurs when the tag-store is sufficiently large that adding more entries will not increase the likelihood of finding an invalid tag-store entry upon a cache miss.

Ammp is strictly limited by the size of the tag-store. Referring to Figure 8, ammp is the only benchmark for which the fully-associative cache outperforms the double sized cache. Table 5 shows that ammp uses local replacement to find a data-victim for more than 99% of its cache misses. Simply doubling the size of the cache fails to improve cache performance because the demand on the cache sets remains too high for the cache to support. Furthermore, doubling the number of tag-store entries fails to improve performance considerably. In Figure 8, the V-Way cache with TDR = 2 improves the miss-rate by exactly the same amount as the double sized cache (4%). As the number of tag-store entries increases beyond TDR = 2, however, the cache is better able to support this demand.

8.3 Impact of Varying Cache Size

In order to analyze the impact of cache size on the performance of V-Way cache, we vary the size from 256kB to 1MB. Figure 11 shows the miss rate averaged across all the sixteen benchmarks for the traditional 8-way cache and the V-Way cache.

V-Way reduces miss rate as compared to the traditional 8-way cache for both 512kB and 1MB cache size. However, it should be noted that when the cache size is increased, some benchmarks start to *fit in*, leaving no room for miss rate improvement. For remaining benchmarks, the global replacement of V-Way still helps in reducing miss rate.

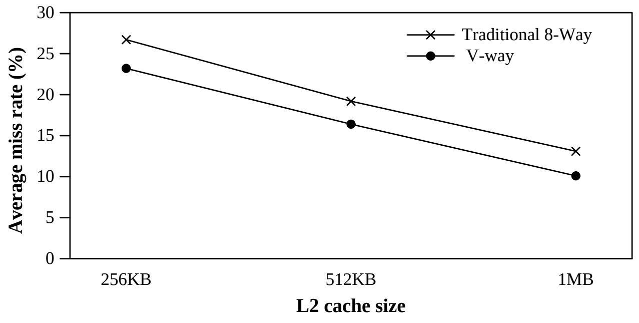

Figure 11: Average miss-rate (lower is better) for cache size of 256kB, 512kB, and 1MB.

8.4 Why It Works

The V-Way cache provides variable associativity on a per-set basis in response to the non-uniform distribution of accesses across the sets in a cache. The demand on a set in a V-Way cache can be measured by the number of valid tags present over time. If accesses are uniformly distributed across all the sets, we would expect all the sets to have exactly $1/TDR$ of their tags valid (i.e. one-half for TDR $= 2$) at any given time in the program. To measure this non-uniformity, we define the following three levels of demand for a V-Way cache with a maximum 8-way associativity:

- Low demand sets : 0 - 2 valid tags
- Medium demand sets: 3 - 5 valid tags
- High demand sets : 6 - 8 valid tags

We sample the second level V-Way cache every 100K accesses and measure the demand on each set. Figure 12 shows the variation in set demand during benchmark execution for mcf, facerec, vortex, and ammp. The horizontal axis is shown in intervals of 100K accesses, and the vertical axis shows the percentage of all sets in the cache from 0-100%.

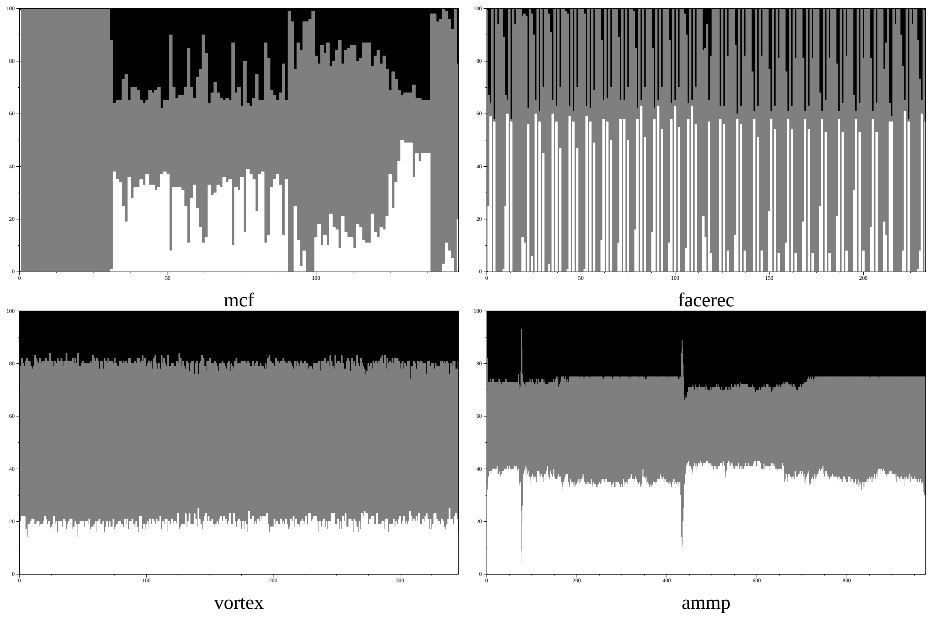

Figure 12: Distribution of low demand(white), medium demand(gray), and high demand(black) sets.

Mcf undergoes clear program phases where the set demand varies from almost completely uniform to evenly distributed. Facerec exhibits two distinct repeating phases. One phase shows uniform medium demand, and the second phase is composed almost entirely of high and low demand sets. The behavior of ammp remains fairly consistent with sets evenly distributed between the low, medium, and high demand categories. Vortex is also very consistent, but contains more medium demand sets than low or high. These benchmarks clearly demonstrate the non-uniform distribution of memory accesses across cache sets. The V-Way cache supports this variable demand by increasing associativity of high demand sets, while reducing the associativity of low demand sets.

8.5 Limitations

The V-Way cache attempts to provide global data replacement without increasing the number of tag comparisons required for each access. The impact of the V-Way cache on performance is limited by the benefit a program receives from global data replacement in general. Some programs respond only to increasing the size of the cache, regardless of the level of associativity. Such programs are unlikely to benefit from the use of a V-Way cache. Given constant access latency and a constant global replacement policy, the performance of a V-Way cache is bounded by that of a fully-associative cache of the same size. The examples of this are bzip2, parser, gzip and swim in Figure 8.

9 Related Work

High performance cache design has received much attention in both industry and academia. We summarize the work in the literature that most closely resembles the techniques proposed in this paper, distinguishing our work where appropriate.

Hallnor et al [9] proposed the Indirect Index Cache (IIC) as a mechanism to achieve full-associativity through software management. The IIC serializes tag comparison and data lookup by storing a forward pointer in the tag-store to identify the corresponding data line. Cache access in the IIC is performed using a structure similar to a hash table with chaining. A hashing function is used to generate a primary index into a two level structure. If a matching tag is not found in the first level, the collision chain is traversed in the second level. The access latency of the IIC is variable, depending on the length of the collision chain. The authors propose a global, software-managed replacement policy called generational replacement.

In the NuRAPID cache [5] the access latency of different cache lines varies depending on the physical placement of data within the data-store. NuRAPID serializes tag comparison and data lookup to accomodate distance replacement – the promotion and demotion of data lines to different distance groups – without affecting the arrangement of the tag-store entries. This serialization is accomplished through the use of forward pointers in the tag-store and reverse pointers in the data-store. While the structure of the V-Way cache is similar in many respects to the NuRAPID cache, the two designs target different fundamentally different aspects of cache performance. The major differences between the two techniques are given below:

1. NuRAPID targets access latency, while V-Way targets miss-rate.
2. Unlike V-Way, NuRAPID has the same number of entries in both the tag-store and data-store.

3. NuRAPID has a fixed associativity for all sets, while V-Way allows the associativity to vary in each set.
4. When a miss occurs, NuRAPID uses local data replacement to find a victim, while V-Way uses global replacement.

NuRAPID and V-Way are orthogonal ideas and can potentially be used in conjunction with one another for even greater performance improvement.

Alameldeen et al [2] proposed variable associativity through data line compression. Each set in the data-store of a set-associative cache can store a fixed number of uncompressed data lines. Individual data lines are compressed, creating room for additional compressed lines. The variation in associativity depends on the actual data values stored in the cache. The V-Way cache, on the other hand, varies associativity in response to program demand, independent of the contents of the data being stored.

Prime modulo hashing [12] and skewed associativity[16] attempt to distribute memory accesses uniformly across cache sets by targeting the indexing function. These approaches suffer from the negative effects of local data replacement due to the static mapping of tag-store entries to data lines in each set.

Puzak [15] proposed the inclusion of extra tags in a *shadow directory* to provide feedback to a local-replacement engine in a set-associative cache. These extra tags are used strictly for maintaining replacement information for evicted data lines, however, and do not provide information about data lines resident in the cache.

10 Conclusion

Traditional cache design implicitly assumes that memory accesses are uniformly distributed across the sets in the cache. In different phases of program execution, however, memory accesses deviate from this uniform behavior, creating an imbalance in the demand on individual sets in the cache. We propose the *V-Way Cache*, a design that allows the associativity to vary on a per-set basis by increasing the number of tag-store entries relative to the number of data lines. We also propose *Reuse Replacement*, a global replacement policy based on frequency information. The Reuse Replacement policy is both fast and implementable, selecting a victim within five cycles for 99.3% of the evictions. A 256kB, 8-way second level V-Way cache using Reuse Replacement outperforms a traditional cache of the same size and associativity by 13%. This results in an IPC improvement of up to 44%, and an average IPC improvement of 8%.

The V-Way cache provides a platform for other optimizations such as cache compression and power management. Invalid tag-store entries can be used to maintain inclusion information without the need for duplicating cache lines in the data-store[8]. The V-Way cache has an inbuilt shadow directory[15] that can provide feedback information to the replacement policy to prevent unnecessary eviction of certain data lines. Future work includes evaluating the impact of these optimizations.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. cache performance of operating systems and multiprogramming. In *ACM Transactions on Computer Systems, 6*, pages 393–431, November 1988.

[2] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, 2004.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. In *IBM Systems journal*, pages 78–101, 1966.

[4] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the Second IEEE International Symposium on High Performance Computer Architecture*, pages 244–253, 1996.

[5] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 55–66, 2003.

[6] F. J. Corbato. A paging experiment with the multics system. *MIT project MAC Report MAC-M-384*, May 1968.

[7] Digital Equipment Corporation, Hudson, MA. *Digital Semiconductor 21164 Alpha Microprocessor Product Brief*, Mar. 1997. Technical Document EC-QP97D-TE.

[8] K. Gharachorloo, A. Nowatzyk, R. McNamara, R. Stets, S. Smith, S. Qadeer, B. Sano, B. Verghese, and L. A. Barroso. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, 2000.

[9] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, 2000.

[10] T. L. Jhonson. *Run-time adaptive cache management*. PhD thesis, University of Illinois, Urbana, IL, May 1998.

[11] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[12] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, pages 288–299, 2004.

[13] C. Kim, B. D., and S. Keckler. An adaptive, nonuniform cache structure for wiredelay dominated onchip caches. pages 211–222, 2002.

[14] J.-K. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 134–143, 1998.

[15] T. R. Puzak. *Analysis of cache replacement algorithms*. PhD thesis, Univ. of Mass., ECE Dept., Amherst, MA., 1985.

[16] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, 1993.

[17] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transaction on Database Systems*, 3(3):223–247, September 1978.

[18] A. J. Smith. Cache memories. *Computing Surveys*, 14(4):473–530, 1982.

[19] D. Weiss, J. J. Wuu, and V. Chin. The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor. In *IEEE journal of solid state circuits*, pages 1523–1529, Nov. 2002.

[20] S. J. E. Wilton and N. P. Jouppi. Cacti: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31:677–688, May 1996.