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

Moinuddin K. Qureshi David Thompson Yale N. Patt
Department of 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-store 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 achieves an average miss rate reduction of 13% on sixteen benchmarks from the SPEC CPU2000 suite. 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 because memory references in a program exhibit non-uniformity, causing some sets to be accessed heavily while other sets remain underutilized.

To achieve the lowest possible miss rate, a cache should be organized as fully-associative with Belady's OPT replacement policy [2]. However, the power, latency, and hardware costs of a fully-associative organization make it impractical, and OPT replacement is impossible to achieve without knowledge of the future. 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. Doubling the associativity to 16 ways marginally improves the miss rate, whereas making the cache fully-associative results in a much more significant improvement. The fully-associative cache with OPT replacement even reduces miss rate more than a set-associative cache of double its size. This result highlights the significant impact a cache's organization and replacement policy have in reducing cache misses. Furthermore, the upper bound provided by the ideal cache indicates that there is much room for improvement with existing cache designs.

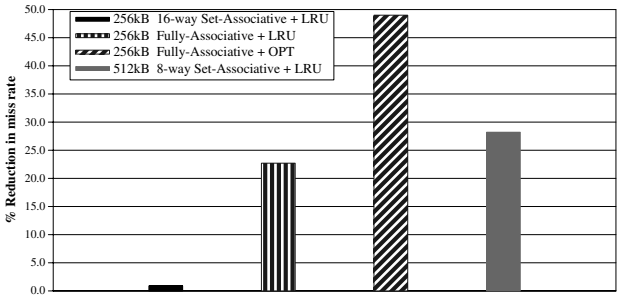


Figure 1. Percent reduction in miss rate compared to a 256kB 8-way set-associative cache.

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. 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. Accessing a fully-associative cache is impractical, however, as it requires a tag comparison with every tag-store entry, prohibitively increasing both access latency and power consumption. Our work addresses the tradeoff between cache performance and cost in a new way.

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 **IEEE**

The major contributions of our work are as follows:

1. We propose a novel mechanism to provide the benefit of global replacement while maintaining the constant hit latency of a set-associative cache. We call this the *Variable-Way Set Associative Cache*, or simply, the *V-Way Cache*.

2. For the V-Way cache, we propose a practical global replacement policy based on access frequency called *Reuse Replacement*. Reuse Replacement performs comparably to a perfect least recently used (LRU) policy at a fraction of the hardware cost and complexity.

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 [13] [10]. 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 [9] 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 [13], and the predictive sequential-associative cache [3] trade variable hit latency for increased associativity. With these schemes, 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, and their effectiveness reduces as associativity increases due to the inherent performance benefit of increased associativity.

Our work focuses on reducing the miss rate of large, set-associative, secondary caches.[1] 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.

[1]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.

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.

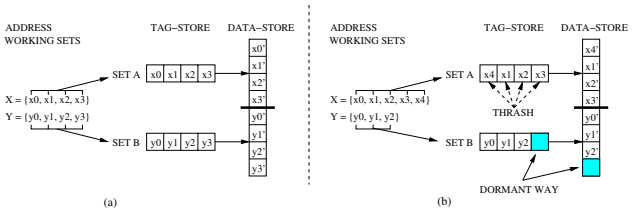


Figure 2. Traditional set-associative cache using local replacement.

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 variable demand on sets in the cache. 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 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 entries 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 additional 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.

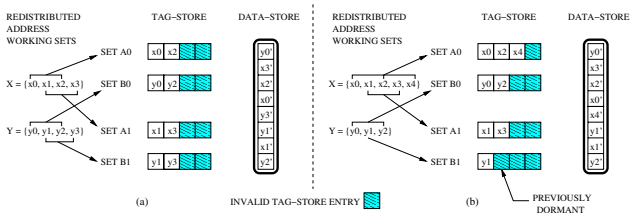


Figure 3. Variable-way set-associative cache using global replacement.

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

- *The memory references are redistributed across the cache sets.*
  After doubling the number of sets, the number of bits used to index the tag-store increases by one. In Figure 3(a), the new most-significant bit of the index redistributes working set X across sets A0 and A1. Working set Y is similarly redistributed across sets B0 and B1.

- *There no longer exists a static one-to-one mapping between tag-store entries and data lines.*
  Every valid tag-store entry now contains a pointer to a unique location in the data-store. This mapping may change dynamically and implies that tag comparison and data lookup must be performed serially.[2]

With twice as many tag-store entries as data lines, each set in the cache contains, 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.

[2]Existing processors serialize tag comparison and data lookup to reduce the power dissipation of large cache arrays[6][18]. Prior research has made use of serialization to increase flexibility and to improve performance in large caches [7][4]. Section 7.3 discusses energy savings due to serialization.

3. V-Way Cache

3.1. Terminology

The defining property of the V-Way cache is the existence of 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 TDR $\geq$ 1. The case TDR $= 1$ is equivalent to a traditional cache. In the example in Section 2, TDR $= 2$ because there are twice as many tag-store entries as data lines. Unless otherwise specified, we assume TDR $= 2$ for the remainder of the paper.

3.2. Structure

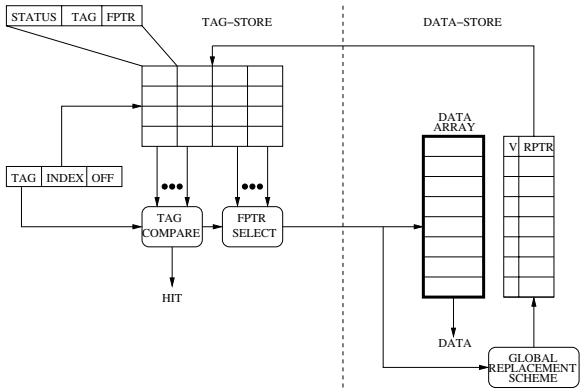


Figure 4. V-Way Cache.

Figure 4 shows the structure of the V-Way cache. 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 (a valid bit, a dirty bit, and replacement information), tag bits, and a forward pointer (FPTR) which identifies the unique entry in the data-store to which the tag-store entry is mapped. If the valid bit in a tag-store entry is cleared, all other information in the entry, including the FPTR, is considered invalid. Each data-store entry contains a data line, a valid bit, 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 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.

3.3. Operation

A V-Way cache access consists of one or two sequential array lookups, depending on whether the first lookup results in a miss or a hit. The tag-store lookup is performed first. If the tag-store lookup results in a hit, the FPTR of the matching entry is used to perform a direct-mapped lookup into the data-store to retrieve the appropriate data line. Replacement information is updated appropriately in both the tag-store and the data-store after each access. If the tag-store lookup fails to find a matching entry, a cache miss is signaled. Because the tag-store and data-store are decoupled, two victims must be identified for the ensuing line fill: 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 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 occurred due to the implementation of a cache coherency protocol. In the V-Way cache, however, each time the data-store's global replacement engine is invoked to find a data-victim, a tag-store entry that is unlikely to be used in the near future is invalidated.

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 set-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 ability of the V-Way cache to reduce miss rate depends primarily on the intelligence of the global replacement policy. A naïve 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 $O(n^2)$[17]. Considering the fact that large caches typically contain thousands of data lines, perfect LRU is an impractical choice. Two-handed clock replacement[5], commonly used for page replacement in an operating system, uses only a single bit per entry. Though inexpensive, it does not perform as well as perfect LRU when applied to caches[7]. Our goal is to design a replacement algorithm that yields performance comparable to a 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 effect typically results in a much lower measure of locality in the L2 than in 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.

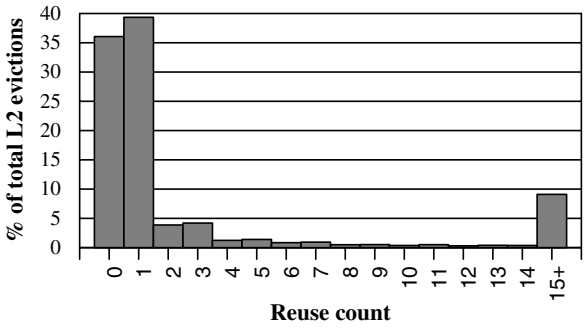


Figure 5. Distribution of L2 cache line reuse.

4.2. Cost Effectiveness of Frequency Based Replacement

Prior research has identified the significance of access frequency in relationship to cache performance [16]. Several authors have proposed the use of frequency information for placement and replacement of cache lines. Johnson [8] uses frequency information to make cache bypass decisions. A non-uniform cache architecture [11] uses access frequency to direct the placement of a cache line to an appropriate distance group. Hallnor et al [7] use access frequency with hysteresis to implement an algorithm called generational replacement, which requires 33 bits of information per tag-store entry. Tracking reuse information, on the other hand, requires far less storage for a comparable measure of frequency. In Figure 5, over 80% of L2 cache lines are reused three or fewer times, and 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.

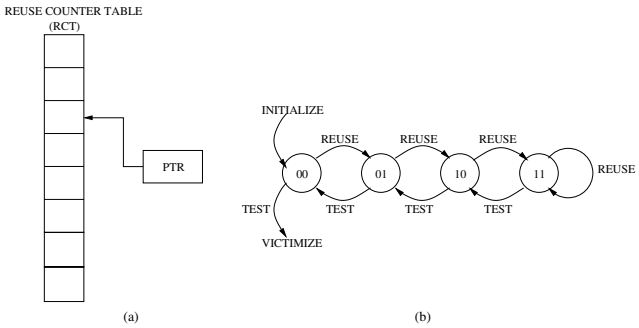


Figure 6. Reuse Replacement : (a) RCT and PTR register. (b) State machine for the reuse counters.

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 the reuse counters for newly installed cache lines to reach a representative value before being tested and decremented.

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_REUSE_COUNTERS for an N-bit reuse counter. Using two-bit reuse counters, the theoretical maximum victim distance is 6144 for a cache with 2048 data lines. 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 12 F04 (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%, and five cycles 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, yields an average replacement latency of 1.2 cycles, 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 similar to the Itanium II processor [18]. The benchmarks used in this study do not stress a very large sized cache. For this reason, 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. Cache Configuration.

L1 I-Cache	16kB; 64B linesize; 2-way with LRU repl.
L1 D-Cache	16kB; 64B linesize; 2-way with LRU repl.
Baseline L2	256kB; 128B linesize; 8-way with LRU repl.

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 using the `ref` input set were fast-forwarded for 15 billion instructions and simulated for 2 billion instructions. For benchmarks bzip2, gcc, mcf, vpr, and ammp, a slice of 2 billion instructions from the `ref` input set was unable to capture the varying phase behavior of the benchmarks. For these benchmarks, the experiments were run with the `test` input set from start to completion with the exception 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 a 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. Table 4 lists the total instruction count, the number of L2 cache accesses, the baseline L2 miss rate, and the total size of the L2 footprint for each benchmark used in our experiments. 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 linesize (128 bytes).

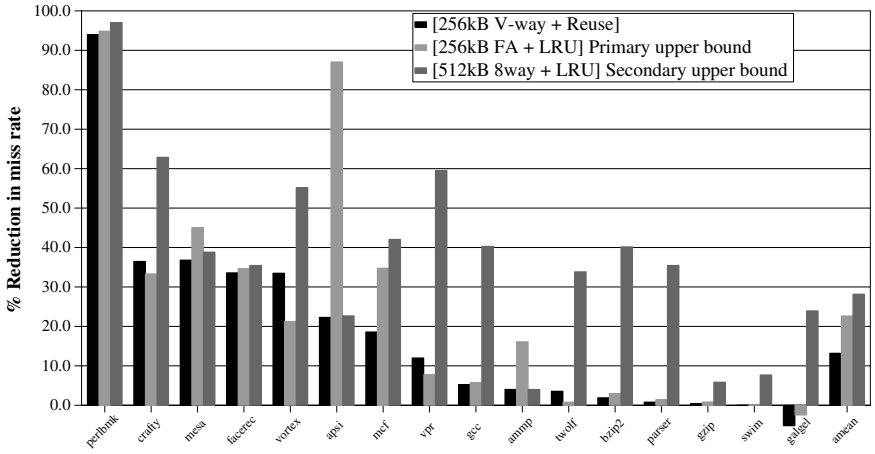


Figure 7. Reduction in miss rate with : V-way cache (TDR=2), fully-associative cache, and double sized baseline.

Table 4. Benchmark characteristics.

Benchmark	Inst. count	Num L2 accesses	Baseline L2 miss rate	Footprint
bzip2	418 M	4.2 M	35.7%	6.8 MB
crafty	2000 M	117 M	1.6%	1.5 MB
gcc	218 M	6.7 M	4.0%	1.7 MB
gzip	2000 M	37 M	2.4%	69 MB
mcf	173 M	15 M	36.7%	193 MB
parser	2000 M	30 M	33.1%	15 MB
perlbmk	2000 M	3.4 M	2.3%	3 MB
twolf	2000 M	59 M	36.8%	1.7 MB
vortex	2000 M	5.8 M	10.7%	19 MB
vpr	567 M	16 M	11.9%	1.7 MB
ammp	1000 M	98 M	52.1%	9.9 MB
apsi	2000 M	59 M	44.8%	129 MB
facerec	2000 M	23 M	76.2%	15 MB
galgel	2000 M	218 M	8.1%	16 MB
mesa	2000 M	17 M	5.5%	8.1 MB
swim	2000 M	90 M	65.3%	177 MB

6. Results

6.1. Performance of V-Way Cache

Figure 7 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 linesize 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. The V-Way cache provides an average miss rate reduction of 13.2% compared to the baseline cache.

The V-Way cache centers around the use of global replacement for line fills. Ultimately, the V-Way cache attempts to reduce the miss rate by the same measure as a fully-associative cache. Thus, a *primary upper bound* on the miss rate reduction for the V-Way cache is provided by a fully-associative cache of the same size. In Figure 7, we see the V-Way cache approach this upper bound for perlbmk, facerec, and gcc. We refer to the primary upper bound as a *loose upper bound* because differences in the LRU and Reuse Replacement policies can result in the V-Way cache outperforming the fully-associative cache, as seen with crafty, vortex, vpr, and twolf. Another upper bound for the V-Way cache is determined by the TDR value, which determines the size of the V-Way tag-store. This *secondary upper bound* for the V-Way cache is provided by a traditional cache of TDR times its data-store size. In Figure 7, where TDR = 2, the secondary upper bound is the reduction in miss rate given by a double sized cache.

In general, the miss rate reduction provided by the V-Way cache is limited by the lower of the two upper bounds. When both upper bounds are high, as for perlbmk, the miss rate reduction with V-Way is very significant. V-Way reduces the miss rate by 94% for perlbmk. For most benchmarks, increasing the size of the cache results in a greater miss rate reduction than increasing the associativity. In these cases, the V-Way cache is limited by the primary upper bound. The aberrant behavior of galgel can be attributed to anti-conflict misses [12].

In some cases, a fully-associative cache outperforms a larger sized cache due to the presence of sets with unusually high demand. Mesa, apsi, and ammp exhibit this behavior. In these cases, the V-Way cache is limited by the secondary upper bound. In the case of apsi and ammp, the disparity between the two upper bounds is significant. Note that in these cases, the secondary upper bound can be controlled by the cache designer by changing the TDR value. Section 8.2 explores the impact of varying the TDR in greater detail.

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%

Table 6. Comparison of miss rate percentage (lower is better) for perfect LRU replacement and Reuse Replacement.

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**

6.2. Comparing Reuse Replacement to 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.

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 interesting results from Figure 7 is that the V-Way cache outperforms the fully-associative cache for crafty, vortex, vpr, and twolf. 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 counters each time the replacement algorithm is invoked, evicting low-reuse data lines more quickly than LRU. Also, Reuse Replacement can potentially retain high reuse lines four times as long as perfect LRU. Table 6 compares the miss rate of a V-Way cache using perfect LRU replacement to that of a V-Way cache using Reuse Replacement.

For vortex and vpr, Reuse Replacement 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, albeit 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. To model cache access latency and energy we used Cacti v3.2[19].

7.1. Storage

The additional hardware for the V-Way cache consists of the following: (1) Extra tags; The exact number is determined by the TDR, (2) Forward pointers (FPTR) for each tag-store entry, and (3) 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 (v+dirty+LRU)	5 bits	5 bits
tag (36-log_2sets-$log_2$128)	21 bits	20 bits
FPTR	–	11 bits
Size of tag-store entry	26 bits	36 bits
Each data-store entry contains:		
status (v+reuse)	–	3 bits
data	128*8 bits	128*8 bits
RPTR ($log_2$4096)	–	12 bits
Size of data-store entry	1024 bits	1039 bits
Number of tag-store entries	2048	4096
Number of data-store entries	2048	2048
Size of the tag-store	6.7 kB	18.4 kB
Size of the data-store	256 kB	259 kB
Total (tag-store+data-store) Size	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 linesize, however. Table 8 shows the cost and performance benefit for various linesizes. As the linesize increases, the benefit provided by V-Way increases and the storage overhead decreases.

7.2. Latency

The V-Way cache incurs a latency penalty due to the additional tag-store entries combined with the addition of the

Table 8. Cost-benefit analysis for various linesizes.

Linesize	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%

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: 65nm and 90nm. 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. We assume this added latency will result in at most one extra cycle for the cache access.

Table 9. Cache access latency.

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

7.3. Energy

The additional tag-store entries and control information in the V-Way cache increase energy consumption compared to the baseline. Table 10 shows the energy required per cache hit for both the baseline and the V-Way cache. Both the baseline and V-Way use serial tag and data lookup.

Table 10. Energy per cache access.

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

The energy consumption for a traditional cache with parallel tag and data lookup is shown for reference. 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 by 0.08nJ and 0.05nJ for 90nm and 65nm technologies, respectively.

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, 64k-entry PAs and 64k-entry meta-predictor misprediction penalty is 12 cycles min.
L1 Instruction Cache	16kB, 64B linesize, 2-way with LRU repl.
L1 Data Cache	16kB, 64B linesize, 2-way, 2 cycle hit
L2 Unified Cache	256kB, 128B linesize, 8 way, 10 cycle hit
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/bus-cycle

Figure 8 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 replacement latency of a V-Way cache miss was limited to a worst case of five cycles. IPC improvements are shown for both 10 and 11 cycle hit latencies The bar labeled *gmean* is the geometric mean of the individual IPC improvements seen by each benchmark.

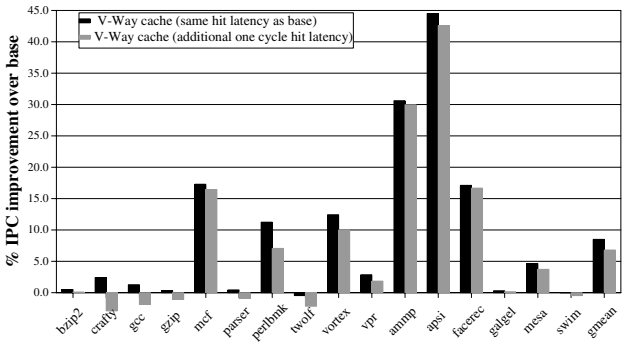


Figure 8. Percentage IPC improvement over the baseline for a system with a V-Way L2 cache.

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 significant performance improvement 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 9 shows the reduction in miss rate relative to the baseline for several different TDR values in a V-Way cache. 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 individual ways until the TDR is satisfied. For example, a TDR of 1.125 is satisfied by adding a ninth way to an eight way cache.

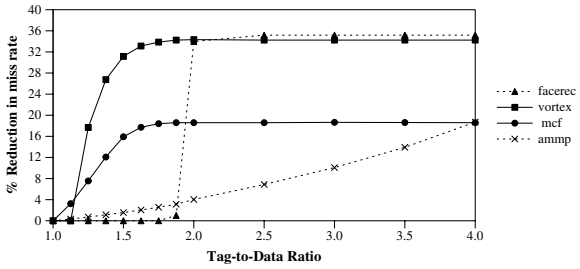


Figure 9. Miss rate reduction (higher is better) relative to baseline vs. TDR.

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 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 shows a steady improvement 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. 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 7, 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 secondary upper bound increases (Section 6.1), providing more potential for miss rate reduction.

8.3. Impact of Varying Cache Size

We analyze the impact of cache size on the performance of the V-Way cache by varying the size from 256kB to 1MB. Figure 10 shows the miss rate averaged across all sixteen benchmarks for the traditional cache and the V-Way cache.

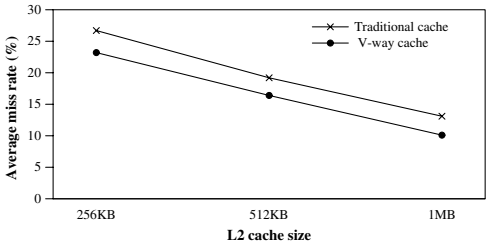


Figure 10. Average miss rate (lower is better) for cache size of 256kB, 512kB, and 1MB.

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

8.4. Variable Set Demand

The demand on an individual set in a V-Way cache can be measured by the number of valid tags present in the set 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 based on the number of valid tags in each set: low demand (0-2), medium demand (3-5), and high demand (6-8).

We sample the second level V-Way cache every 100K accesses and measure the demand on each set. Figure 11 shows the variation in set demand during execution for mcf and facerec. 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 to 100%.

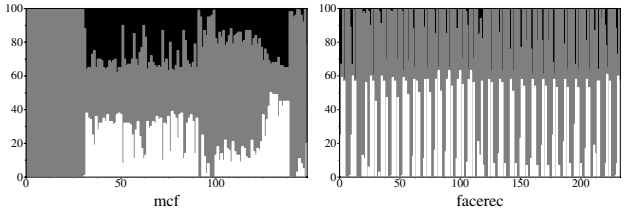


Figure 11. Distribution of low demand(white), medium demand(gray), and high demand(black) sets.

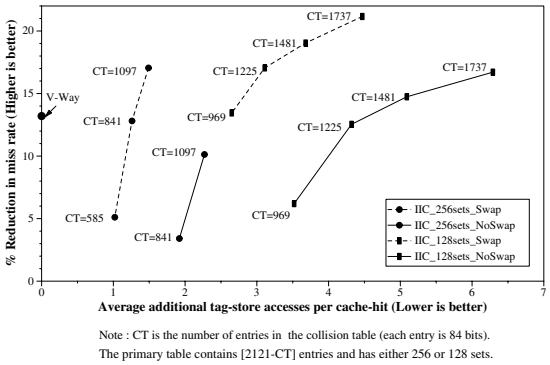
Mcf consists of distinct program phases where the set demand varies from almost completely uniform to evenly distributed. Facerec exhibits two distinct repeating phases. One phase shows uniform medium demand, and the second phase is composed almost entirely of high and low demand sets. The V-Way cache supports this variable demand by increasing associativity of high demand sets, while reducing the associativity of low demand sets.

9. Related Work

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

Hallnor et al [7] 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. If a matching tag is not found in the set-associative tag-store, a pointer associated with the set is used as a direct mapped index into a collision table. Each entry in this second table provides a pointer to the next member of the collision chain. The chain is traversed until either a match is found or the maximum chain length is reached. We distinguish the V-Way cache from the IIC with the following points:

1. IIC requires collision chain traversal, resulting in variable hit latency and port contention. V-Way indexes the tag-store only once and has constant hit latency.
2. IIC requires swapping of tag entries (i.e. promoting entries to the set-associative table on a collision table hit) to reduce average hit latency to a reasonable value. Swapping is undesirable because it requires four accesses to the tag-store, consumes energy, and increases port contention. V-Way does not require swapping.
3. IIC management is very complex due to the variable number of tag-store accesses, insert/remove operations in the singly-linked collision table, and insert/remove operations in the doubly-linked queues used in generational replacement. IIC requires software management of the replacement algorithm. V-Way is managed entirely in hardware.



Note : CT is the number of entries in the collision table (each entry is 84 bits).
The primary table contains [2121-CT] entries and has either 256 or 128 sets.

Figure 12. Comparison of V-Way with IIC. Both swapping-enabled and swapping-disabled configurations are evaluated.

IIC can trade miss rate reduction for access latency by varying the number of entries in the collision table. Figure 12 compares the V-Way cache and Reuse Replacement with various configurations of IIC and generational replacement, devoting equal area to both V-Way and IIC. Though the miss rate reduction provided by IIC and V-Way are comparable, V-Way avoids the increased access latency, swapping, and complexity associated with IIC.

In the NuRAPID cache [4] 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 accommodate 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. NuRAPID targets access latency as opposed to V-Way, which targets miss rate. Moreover, NuRAPID has the same number of entries in the tag and data stores, has a fixed associativity for all sets, and exclusively uses local replacement to find data-victims.

The adaptive group-associative cache (AGAC) [13] attempts to improve the performance of first level, direct-mapped caches. AGAC has variable access latency, needs multiple banks to aid swapping, and has a structure that does not lend itself to global replacement. For replacement, AGAC essentially chooses a victim randomly from the lines that are not tracked by its history gathering structure. The benefit provided by AGAC reduces with increasing associativity. Our studies show that AGAC reduces average miss rate by 3.16%, compared to 13.2% with V-Way.

Prime modulo hashing [10] and skewed associativity[15] 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 [14] 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 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. The V-Way cache has a built-in shadow directory that can provide feedback information to the replacement policy. Future work includes evaluating the impact of these optimizations.

11. Acknowledgments

We thank Brian Prasky, Francis Tseng, Mary Brown, and Steven Reinhardt for their helpful comments. We also thank other members of the HPS research group for the fertile environment they help create. This work was supported by gifts from IBM, Intel, and the Cockrell Foundation. Moinuddin Qureshi is supported by an IBM Ph.D. fellowship.

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

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

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] A. R. Lebeck. Cache conscious programming in undergraduate computer science. In *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 99)*, 1999.

[13] 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.

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

[15] 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.

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

[17] A. J. Smith. Cache memories. *Computing Surveys*, 14(4), 1982.

[18] 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.

[19] 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.