Reactive-Associative Caches

Brannon Batson
Alpha Design Group
Compaq Computer Corporation
bbatson@pa.dec.com

T. N. Vijaykumar
School of Electrical & Computer Engineering
Purdue University
vijay@ecn.purdue.edu

Abstract

While set-associative caches typically incur fewer misses than direct-mapped caches, set-associative caches have slower hit times. We propose the reactive-associative cache (r-a cache), which provides flexible associativity by placing most blocks in direct-mapped positions and reactively displacing only conflicting blocks to set-associative positions. The r-a cache uses way-prediction (like the predictive associative cache, PSA) to access displaced blocks on the initial probe. Unlike PSA, however, the r-a cache employs a novel feedback mechanism to prevent unpredictable blocks from being displaced. Reactive displacement and feedback allow the r-a cache to use a novel PC-based way-prediction and achieve high accuracy; without impractical block swapping as in column associative and group associative, and without relying on timing-constrained XOR way prediction. A one-port, 4-way r-a cache achieves up to 9% speedup over a direct-mapped cache and performs within 2% of an idealized 2-way set-associative, 1-cycle cache. A 4-way r-a cache achieves up to 13% speedup over a PSA cache, with both r-a and PSA using the PC scheme. CACTI estimates that for sizes larger than 8KB, a 4-way r-a cache is within 1% of direct-mapped hit times, and 24% faster than a 2-way set-associative cache.

1. Introduction

The growing gap between processor speeds and memory speeds is resulting in increasingly expensive cache misses, underscoring the need for sophisticated cache hierarchy techniques. Increasing the associativity of the cache is one way to reduce the miss rate of the cache. While set-associative caches typically incur fewer misses than direct-mapped caches, set-associative cache implementations are usually slower than direct-mapped caches [9]. Because even for a direct-mapped cache the common case is a hit, set associativity should be provided without a large increase in hit latency over a direct-mapped cache.

We propose the reactive-associative cache (r-a cache), which provides flexible associativity by placing most blocks in direct-mapped positions and reactively displacing only conflicting blocks to set-associative positions. To achieve direct-mapped hit times, the r-a cache uses an asymmetric organization in which the data array is organized like a direct-mapped cache and the tag array like a set-associative cache, similar to the IBM 3081 L1, and MIPS R8000 L1 [15]. Unlike a set-associative cache, data from the r-a cache's data array proceeds without any way-select multiplexors (which would have to wait for the tag comparators to identify the matching way) in the data output path [9]. Because a set-associative tag array is almost as fast as a direct-mapped tag array for small associativities (although the set-associative data array is significantly slower than direct-mapped data array) [13], this organization achieves near direct-mapped speeds.

To locate a block in one of the many set-associative positions, the above-mentioned machines probe the tag array first and *then* sequentially probe the matching data array, lengthening the hit time. To avoid this serialization, other schemes first probe the direct-mapped positions of the tag and data arrays in parallel, and then probe the set-associative positions [1,2,18, 14, 16]. To increase the probability of finding blocks in the first probe, these schemes swap a block found on the second probe with the block in the direct-mapped position. Unfortunately, cache block swapping degrades both latency and bandwidth because swapping involves two reads and two writes which is slow if done sequentially, and expensive, if done in parallel.

To increase the first probe hit rate without block swapping, the predictive sequential associative cache (PSA cache) proposed using way-prediction for D-caches [6]. The PSA cache predicts the way-number of a block's location, which avoids waiting for the tag array to identify the matching way-number. Way-prediction accuracy crucially affects *both* access latency and L1 bandwidth demand. Not only do mispredicted accesses incur higher latency, they degrade valuable L1 bandwidth due to additional probes. The PSA cache predicts *all* accesses without any selectivity or control, resulting in poor way-prediction accuracy and first probe miss rates which are significantly worse than direct-mapped miss rates. Consequently, the PSA cache worsens the latency and bandwidth of accesses that would hit in a direct-mapped cache.

While the r-a cache also uses way prediction, it is the first proposal to combine asymmetric organization with way-prediction without compromising hit time. To avoid the bandwidth depletion inevitable with PSA's poor way-prediction, the r-a cache places most blocks in direct-mapped positions, and reactively displaces *only* conflicting blocks to set-associative positions. While reactive displacement may reduce overall miss rate, it takes pressure off the way-prediction mechanism, enabling high first probe hit

rates. In addition to reactive displacement, the r-a cache provides a feedback mechanism which prevents repeatedly-mispredicted accesses from being displaced to set-associative positions. Thus, the r-a cache achieves performance robustness by trading-off overall hit rate for first-probe hit rate and L1 bandwidth.

The predicted way number *must* be made available before the actual data address to avoid any delay in the initiation of every cache access. This stipulation rules out using the data address for prediction lookup. PSA recommends an XOR-based way-prediction, which XORs the instruction offset with the source register value to approximate the address [3] and looks up the prediction table. XOR operation on a register value often obtained late from a register-forwarding path *followed* by a table lookup, is likely to be slower than a full add to compute the address, delaying access initiation. In contrast, the r-a cache uses the instruction PC for prediction lookup, allowing at least six pipeline stages for the lookup, making the predicted way-number available *well* before the data address. The novelty of our PC scheme is in integrating way prediction with reactive displacement and feedback.

The main contributions of this paper are:

- We show that a 4-way r-a cache hit latency is within 1% of a direct-mapped cache, and 25% faster than 2-way.
- R-a cache's reactive and feedback mechanisms conserve L1 bandwidth and achieve low first-probe miss rates at 7.3%, compared to PSA at 16.6%.
- Our novel PC-based scheme enables way-prediction early in the pipeline.
- A one-port, 4-way r-a cache, using 1184 bytes of prediction storage, achieves up to 9% speedup over a direct-mapped cache and up to 13% speedup over a PSA cache, with both the r-a and PSA caches using the PC scheme. The r-a cache performs, on average, within 2% of an idealized 2-way set-associative, 1-cycle cache. We also show that reactive displacement and feedback are essential and without these mechanisms the r-a cache's first-probe miss rates suffer.

In Section 2., we describe the r-a cache's organization and in Section 3., the way-prediction mechanism. In Section 4., we qualitatively compare against previous schemes. We present experimental results in Section 5. and conclude in Section 6.

2. Reactive-Associative Cache Organization

The r-a cache is formed by using the tag array of a set-associative cache with the data array of a direct-mapped cache, as shown in Figure 1. For an n-way r-a cache, there is a single data bank, and n tag banks.

The tag array is accessed using the conventional set-associative index, probing all the n-ways of the set in parallel, just as in a normal set-associative cache. The data array index uses the conventional set-associative index concatenated with a way number to locate a block in the set. The way number is $\log_2(n)$ bits wide. For the first probe, it may come from either the conventional set-associative tag field's lower-order bits (for the direct-mapped blocks), or the way-prediction mechanism (for the displaced blocks). If there is a second probe (due to a misprediction), then the matching way number is provided by the tag array.

The r-a cache simultaneously accesses the tag and data arrays for the first probe, at either the direct-mapped location or a set-associative position provided by the way-prediction mechanism. If the first probe, called probe0, hits, then the access is complete and the data is returned to the processor. If probe0 fails to locate the block due to a misprediction (i.e., either the block is in a set-associative position when probe0 assumed direct-mapped access or the block is in a set-associative position different than the one supplied by way-prediction), probe0 obtains the correct way-number from the tag array if the block is in the cache, and a second probe, called probe1, is done using the correct way-number. Probe1 probes only the data array, and not the tag array. If the block is not in the cache, probe0 signals an overall miss, and probe1 is not necessary.

Thus there are three possible paths through the cache for a given address: (1) probe0 is predicted to be a direct-mapped access, (2) probe0 is predicted to be a set-associative access and the prediction mechanism provides the predicted way-number, and (3) probe0 is mispredicted but obtains the correct way-number from the tag array, and the data array is probed using the correct way-number in probe1. On an overall miss, the block is placed in the direct-mapped position if it is non-conflicting, and a set-associative position (LRU, random, etc.) otherwise.

2.1. Probe0 hit latency

The fundamental reason that a conventional set-associative cache is slower than a direct-mapped cache is the multiplexor in the data array path; the select signals for this multiplexor are derived from the tag array output, which is slow [9]. We analyze the r-a cache to show that the r-a cache is almost as fast as a direct-mapped cache, assuming that way-prediction information (the predicted way-number and whether probe0 is direct-mapped or way-predicted) is available well before the data address. In Section 3., we show how way-prediction can be done before the data address is available.

Compared to a direct-mapped cache, the r-a cache introduces the extra multiplexor *probe0 way# mux* in the data array index path (Figure 1). Note that this multiplexor always chooses one out of three inputs, irrespective of the set-associativity of the r-a cache. The multiplexor select signal is available earlier than the data address because the signal comes from the way-prediction mechanism. There-

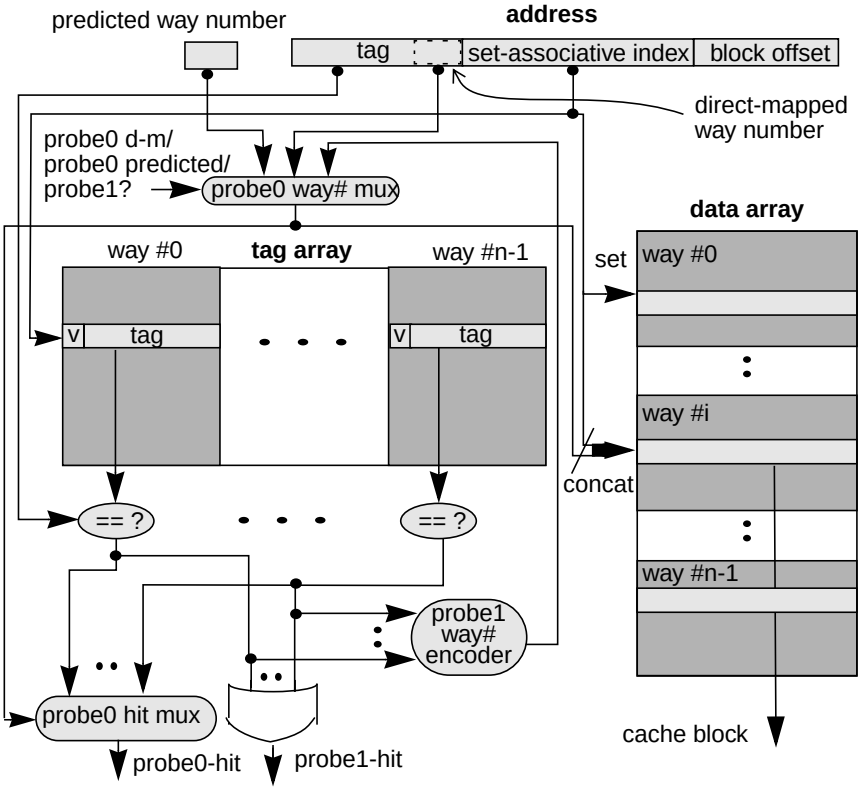

FIGURE 1: The reactive-associative cache.

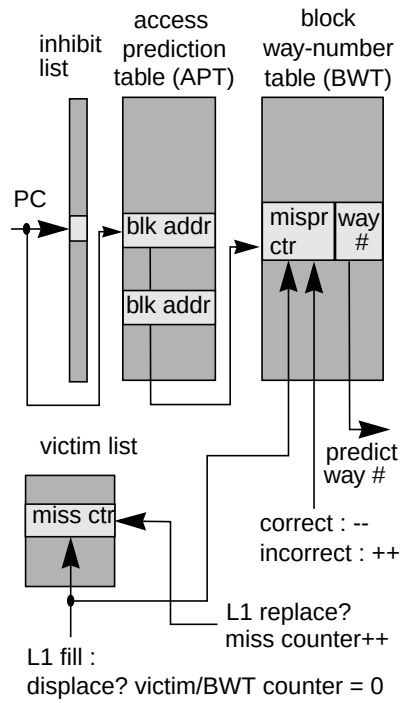

FIGURE 2: PC-based way-prediction structures.

fore, the only delay added to the critical path is propagation through the multiplexor. One possible implementation of *probe0 way# mux* would be a single level of pass gates using a one-hot encoding of the select lines, in which case the extra delay would be negligible. Probe1 may incur a whole extra cycle to account for the *probe1 way# encoder* (Figure 1) and the pass gate.

The tag array does not incur any extra delay because it uses the conventional set-associative index directly from the address and not through *probe0 way# mux (*Figure 1). The multiplexor *probe0 hit mux* generates the probe0 hit signal by selecting the tag match for the probe0 way-number from among the tag matches of all the tag array banks. Note that the probe0 way-number (either direct-mapped way-number or predicted probe0 way-numbers) is the select for *probe0 hit mux,* and is available no later than the address. While the tag array is being accessed with the conventional set-associative index, the probe0 way-number is sent, in parallel, to *probe0 hit mux* select. Using a similar pass-gate mux as we suggested for the *probe0 way# mux*, the probe0-hit signal would incur extra delay of one pass gate, compared to a conventional direct-mapped cache hit signal. The probe1-hit signal incurs extra delay of one OR gate, over the equivalent direct-mapped hit signal.

2.2. Complications due to dual probes

The r-a cache's dual probing may complicate cache pipelining, because of the variable access latency (probe0 hit time vs. probe1 hit time). Scheduling subsequent pipelined accesses without knowing the latency of the previous access is difficult. On a misprediction, a new access has entered the cache pipeline by the time we determine that we need to do a second probe for the prior access. It is mainly for this reason that we charge two additional cycles for probe1. Unlike prior multi-probe caches, however, the r-a cache can signal a probe1 hit or an overall miss by the end of probe0, which may simplify the scheduling.

3. Way prediction

The r-a cache employs hardware way-prediction to obtain the way-number for the blocks that are displaced to set-associative positions before address computation is complete. The strict timing constraint of performing the prediction in parallel with effective address computation requires that the prediction mechanism use information that is available in the pipeline earlier than the address-compute stage. The equivalent of way-prediction for i-caches is often combined with branch prediction [5, 9], but because D-caches do not interact with branch prediction, those techniques cannot be used directly. An alternative to prediction is to obtain the correct way-number of the displaced block using the address, which delays initiating cache access to the displaced block, as is the case for statically probed schemes such as column-associative and

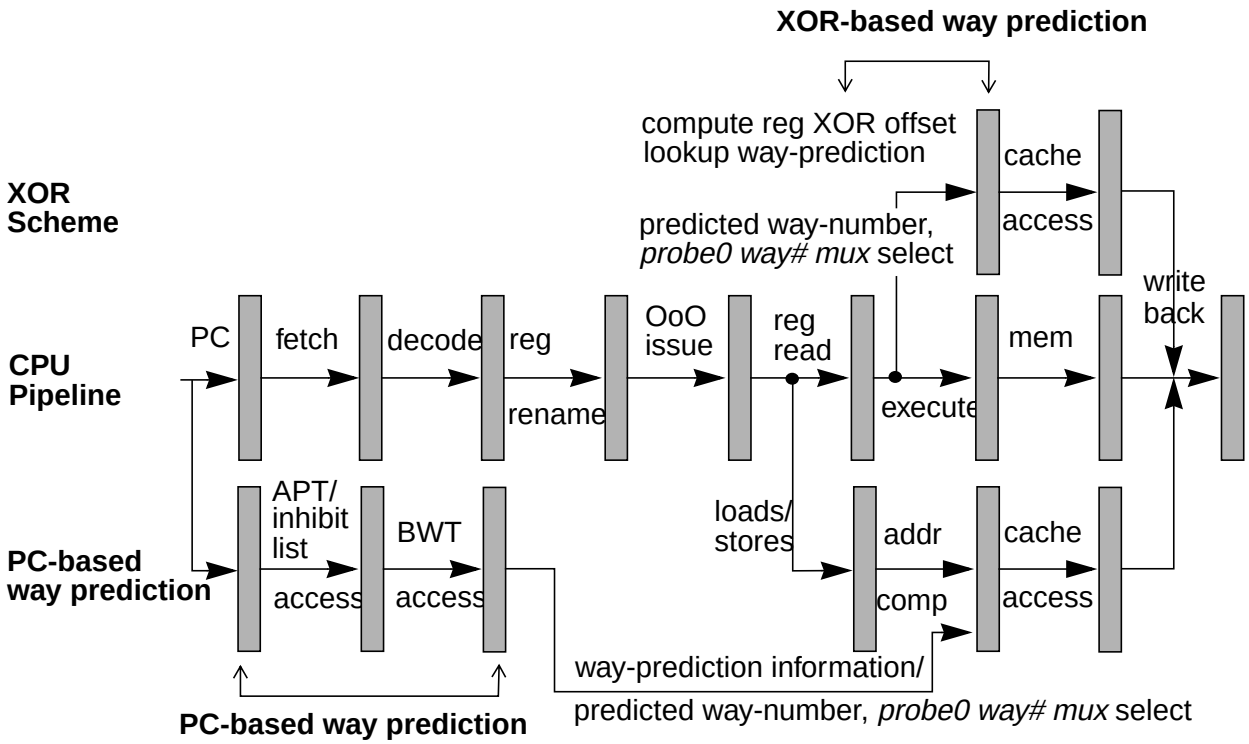

FIGURE 3: Pipeline timing of way-prediction.

group-associative caches.

We examine two handles that can be used to perform way prediction: instruction PC and approximate data address formed by XORing the register value with the instruction offset (proposed in [3], and used in [6]), which may be faster than performing a full add. These two handles represent the two extremes of the trade-off between prediction accuracy and early availability in the pipeline, as shown in Figure 3. PC is available much earlier than the XOR approximation but the XOR approximation is more accurate because it is hard for PC to distinguish among different data addresses touched by the same instruction. Other handles such as instruction fields (e.g., operand register numbers) do not have significantly more information content from a prediction standpoint, and the PSA paper recommends the XOR scheme for its high accuracy.

In an out-of-order processor pipeline (Figure 3), the instruction PC of a memory operation is available much earlier than the source register. Therefore, way-prediction can be done in parallel with the pipeline front end processing of memory instructions so that the predicted way-number and *probe0 way# mux* select input are ready well before the data address is computed. The XOR scheme, on the other hand, needs to squeeze in an XOR operation on a value often obtained late from a register-forwarding path followed by prediction table lookup to produce the predicted way-number and the *probe0 way# mux* select, *all within* the time the pipeline computes the real address using a full add. Note that the prediction table must have more entries or be more associative than the cache itself to avoid conflicts among the XORed approximate data addresses, and therefore will probably have a significant access time, exacerbating the timing problem.

3.1. PC-based way-prediction

PC-based way-prediction tracks memory access behavior to associate the cache blocks with the PCs of the instructions that access the blocks, by exploiting the locality of reference within one cache block on a per-instruction basis. The prediction mechanism tracks the instructions that access displaced blocks and associates the instruction PCs with the way-number of the displaced blocks. If the instruction accesses the displaced block again (i.e., the instruction exhibits locality of reference within the cache block), the prediction mechanism returns the associated way-number, which results in a probe0 hit. This kind of locality can be expected from (1) an instruction that accesses the same data throughout program execution, such as an instruction accessing static globals using a global pointer register, (2) an instruction that accesses the same data for a period of time during program execution, such as an instruction accessing stack frame variables within a function invocation using the stack pointer (the

stack pointer changes infrequently within a function invocation), and (3) an instruction that accesses the different words in a cache block, such as an instruction accessing successive array elements within a cache block. If an instruction accesses a different block on every access (e.g., strided accesses using strides larger than the block size), the PC scheme will not work well. More sophisticated correlational prediction schemes may solve this problem, but such schemes may require large prediction storage [10].

3.1.1. Access-prediction and block way-number tables

When a block is displaced to a set-associative position, it may cause more conflicts in which case it is replaced from the cache. The r-a cache places a block replaced from a set-associative position in the block's direct-mapped position, anticipating that it may not conflict anymore. If the block continues to conflict in its direct-mapped position then the block is displaced to a, presumably different, set-associative position. Thus a conflicting block may transit through a few positions before settling into a non-conflicting position. During this transition, the way-number of the block is constantly changing and unless way-prediction is updated with the correct way-number, many mispredictions will ensue. If multiple instructions access the same block, block transit problems are exacerbated because each of these instructions incurs a misprediction. Because way-prediction maps PCs to way-numbers, and not the other way, updating way-number is hard because the mapping PC is not known when the block transits to a new way-number.

Adding one level of indirection solves the block transit problem. Figure 2 depicts the structures used by the way-prediction scheme. The instructions' PCs are associated with the block address, and not the way-number of the block, in the access-prediction table (APT). A second table, called the block way-number table (BWT), is used to associate the block address with the way-number of the block. Multiple entries in the APT may hold the same block address, but there is only one BWT entry per block address. Using the block address, transiting blocks update the BWT with the correct way-number, enabling all the instructions that access the same block again to get the correct way-number. Through the APT, the r-a cache exploits locality within one cache block on a per-instruction basis.

The APT is accessed using the instruction PC, and then the block address supplied by the APT entry is used to access the BWT. If the block address supplied by the APT is found in the BWT, the way-number from the BWT entry is sent to the cache as the predicted probe0 way-number. If the block address obtained from the APT is not found in the BWT then the access is predicted to be direct-mapped. Similarly, instructions that do not find a matching PC entry in the APT are predicted to access direct-mapped positions. The BWT is also accessed on a L1 cache fill, so that if the block being retrieved is found in the BWT, the way-number is updated; this update keeps the way-prediction accuracy high for blocks in transit.

Because the APT and BWT need to hold information on only the displaced blocks, small size may suffice. Also, because displaced blocks, by definition, conflict in the cache and the BWT uses block addresses to index, conflicts in the BWT may be common. Conflicts in the APT and BWT lead to mispredictions because such accesses are predicted to be direct-mapped. Therefore, the APT and BWT may need to be moderately associative (or indexed through a skewing function). Fortunately, both the APT and the BWT access are well ahead of the cache access in the pipeline (Figure 3), high associativity of the APT or BWT is not likely to delay probe0 initiation.

3.2. XOR-based way-prediction

XOR-based way prediction, used in the PSA paper [6}, relies on the idea that while a pipeline stage computes the data address by adding the source register value to the instruction offset, the register value can be XORed with the instruction offset to compute an approximate of the address [3] and access a way-prediction table. This scheme exploits the fact that most memory instructions have small enough offsets so that the block address from the XOR approximation is usually same as or at least correlates well with the block address from the actual data address.

An APT is not need for the XOR scheme. Instead, the XOR scheme simply uses the BWT indexed by the XOR value. The other key difference between the XOR and PC schemes is that the XOR scheme accesses the BWT during the late address-compute pipeline stage, whereas the PC scheme accesses the BWT in the early instruction-decode pipeline stage. While the XOR scheme is naturally more accurate than the PC scheme, we claim that this late access causes the timing problems pointed out in Section 3., especially because of BWT's large size requirements (the PSA paper suggests 1024 entries for 256 blocks in the L1 cache), as discussed in the previous section.

3.3. Reactive displacement and feedback

While displacing conflicting blocks reduces overall misses, first-probe miss rate typically worsens due to increased pressure on way-prediction. Because probe0 misses due to mispredictions result in a second data array access (if the block is in the cache), overall hit latency and bandwidth to the cache are significantly degraded. The main reason for this is that probe1 hits occupy the data array for extra cycles beyond probe0, causing subsequent cache accesses to queue in the load/store queue. In the worst case, a substantial fraction of all L1 hits may be from probe1, considerably degrading valuable L1 bandwidth.

Because way-prediction has to use inexact information due to pipeline timing constraints, it is hard to make way-prediction perfect. Therefore, the r-a cache attempts to reduce the number of accesses it predicts, such that either data is in a direct-mapped position or is highly-predictable in a set-associative position, keeping the number of mispredictions in check. Regardless of the way-prediction scheme used, the r-a cache (1) reactively displaces only those blocks that frequently conflict, avoiding prediction accuracy degradation due to a large number of displaced blocks, (2) tracks prediction accuracy so that unpredictable blocks are moved back to their direct-mapped positions, avoiding repeated mispredictions and (3) disallows unpredictable blocks from being displaced again, using a simple feedback mechanism.

Thus, the r-a cache achieves performance robustness by trading-off overall hit rate for probe0 hit rate, and lowering bandwidth demand. In Section 5.4., we show that probe0 miss rates worsen drastically without reactive displacement and feedback.

3.3.1. Victim list: reactive displacement

Ideally, the r-a cache would displace only conflicting blocks to set-associative positions; in a real implementation, it is difficult to isolate capacity and conflict misses. Consequently, the r-a cache approximates isolation of conflict misses by tracking the set of recently replaced blocks in a table called the victim list, similar to [8]. Each victim list entry consists of a block address and a saturating counter. The block address of a replaced block is inserted in the victim list and the corresponding counter, which counts the number of times the block has been replaced in the past, is incremented. After the block gets replaced a few times, the victim list counter reaches saturation, signaling a conflicting block; the next time the block is brought back into the cache, the block is displaced to a set-associative position, and the victim counter is reset.

The victim list needs to be a high-associative structure to avoid conflicts in the victim list itself. Because the victim list is not on the cache access critical path but only in the replacement path, the high associativity of the victim list does not impact hit latency.

3.3.2. Feedback to evict unpredictable blocks

Regardless of whether the PC-based or XOR-based scheme is used, the BWT maps a block address to its way-number in the cache. Apart from the way-number, each BWT entry contains a saturating counter, which decrements on a correct prediction and increments on an incorrect prediction to track the accuracy of way-prediction for the corresponding block. The purpose of tracking misprediction counts is to evict blocks whose addresses are not predictable. If the way-number for a block that is displaced to a set-associative position is repeatedly mispredicted then the block is evicted from the cache; if, later, the block is brought back into the cache, it is placed in the direct-mapped position (because the victim list miss counter would have been reset when it was put in the set-associative position). While such evictions increase overall cache miss rate, continuing to hold such unpredictable blocks in set-associative positions may cause probe0 miss rate to become worse than that of a conventional direct-mapped cache. Thus the r-a cache trades-off overall hit rate for probe0 hit rate, which lowers average cache access time.

It should be noted that in the case of the PC-based scheme, the BWT mispredict counter is shared among all instructions that access that displaced block. This does a good job of capturing group prediction behavior.

3.3.3. Feedback to prevent repeated mispredictions

The r-a cache relies on its ability to turn off associative displacement for accesses that have poor prediction accuracy. This is done on a per address basis, as described in the previous section. In general, instructions that are unpredictable for one address are also unpredictable for other addresses. To prevent the performance penalty of relearning unpredictable behavior for each address that an unpredictable instruction touches, the r-a cache also provides the inhibit list as a way to turn off reactive displacement on a per instruction basis.

The inhibit list is a single bit per instruction which is set when that instruction is inhibited from causing an associative displacement. Because the inhibit list entry is a single bit indexed by instruction, it would be natural to place the bit in the i-cache.

When the misprediction counter in the BWT saturates, not only is the block evicted from the cache, but the inhibit list entry is set for the instruction causing the mispredict. As long as the inhibit list entry is set, the instruction always accesses only direct-mapped locations without using way-prediction (PC or XOR), and does not displace any blocks to set-associative positions. Furthermore, an inhibited instruction will cause evictions of set-associative blocks that it touches, and will also saturate the BWT misprediction counter for that block. Therefore, the inhibit bit naturally has a poison property, in that once an instruction is inhibited, all the blocks it touches will be forced to direct-mapped locations, and all the instructions that touch those blocks will be inhibited, and so on.

We learned that predictability of instructions changes over the course of program execution (usually when the data that they touch changes). Therefore, it is desirable to clear the inhibit list when instructions start to work on new data. We use a DTLB miss to indicate that we are entering a new data phase, and it is time to clear the inhibit list.

Addresses may also become more predictable when they are touched by different instructions. Therefore, we wish to clear the BWT mispredict counters when we are

Component	Description
CPU	Out of order, 8-issue, 64-entry reorder buffer, 32-entry ld/st queue
Prediction	bimodal 4096 entries + gshare 10-bit history
L1 I-cache	16 KB, 2-way, 32 byte blocks, 1 cycle
L1 D-cache	8 KB, 32 byte blocks, 1 cycle probe0, 3 cycles probe1, lock-up free
L2 cache	256 KB, 8-way, 64 byte blocks, 12 cycle
memory	Infinite capacity, 60 cycle latency
Way-prediction resources	**PC**:128-entry APT & BWT, 2048-bit inhibit list, 256-entry victim list; inhibit threshold: 3 victim threshold: 5; (total 1184 bytes). **XOR**: 1024-entry BWT, 2048-bit inhibit list, 256-entry victim list; inhibit threshold: 3 victim threshold: 2; (total 2560 bytes).

Table 1: Hardware parameters.

Program	input	instructions
vortex	ref	1 billion
gcc	1recog	347 million
li	train	365 million
perl	jumble	1 billion
go	9stone21	1 billion
troff	paper.me	70 million
m88ksim	train	171 million
swim	train	430 million
fpppp	train	235 million

Table 2: Benchmarks.

entering a new instruction phase of the program. We use an ITLB miss to indicate the arrival of a new instruction phase, and this causes a clearing of the BWT mispredict counters.

We tried several dynamic schemes, none of which performed as robustly as the TLB scheme. Simple periodic clearing performed well, but the clearing interval is application dependent and needs to be pre-determined.

4. Related work: qualitative comparison

The hash-rehash cache [1] introduced the idea of multiple probes to the cache to achieve high hit rates while maintaining direct-mapped speeds. Accesses perform a static probe0 *always* in the direct-mapped position and on a probe0 miss, a second probe is done by hashing the address. The column-associative cache [2] improved on hash-rehash by associating rehash information with each block and improving the replacement algorithm, which decreases the number of second probes required. The parallel multicolumn cache [18] generalizes column-associative to n-way associative through a set-associative tag array with a direct-mapped data array to perform the tag search in parallel, and uses MRU information to optimize the search, much like other implementations considered in [17,12,7]. All the above schemes increase the probability of finding blocks in the static probe0 by swapping a block found on the second probe with the block in the probe0 position. Unfortunately, swapping of entire cache blocks is hard to implement because swapping involves two reads and two writes, which is slow if done sequentially, and prohibitively expensive if done in parallel, and degrades valuable L1 bandwidth.

The Difference-bit cache is a two-way associative cache which achieves almost direct-mapped speeds for the special case of two-way associativity without employing any way-prediction by using the fact that the tags in the set have to differ in at least one bit [11]. The group-associative cache [14] pioneered the idea of using under-utilized cache frames (or 'holes') to displace any block into any frame in the cache—achieving fully-associative miss rates in the limit. It also uses cache block swapping, and has a notion of selective displacement of recently accessed blocks. Group associativity is achieved by maintaining the location and tags of the displaced blocks in a fully-associative OUT directory. In Section 5.3., we show that the fully-associative OUT directory severely lengthens the hit time.

To avoid cache block swapping, the predictive sequential associative cache [6] proposed way prediction to access any way, as opposed to only the direct-mapped way, on probe0. The PSA paper uses direct-mapped tag and data arrays, unlike the r-a cache's set-associative tag array, and suggests sequentially probing the tag array to find the correct way-number. The PSA paper recommends the XOR scheme for its high accuracy, but incurs timing problems (See Section 3.2.). But even with XOR prediction, PSA's way prediction accuracy is low, mainly because all accesses are predicted, resulting in many probe0 misses, even for accesses that would hit in the direct-mapped cache. PSA does achieve low overall miss rates. Because of many second probes, average hit latency and L1 cache port pressure increase. In Section 5.3., we show that this increase usually nullifies the advantage of the low overall miss rate, under realistic bandwidth constraints.

5. Experimental Results

We modified the Simplescalar3.0 simulator [4] to model the L1 D-cache as an r-a cache. Table 1 shows the system configuration parameters used throughout the experiments, unless specified otherwise. The processor core including the out-of-order issue and branch prediction mechanisms remain unchanged. We assume a modest on-chip cache hierarchy of 8 Kbytes L1 D-cache and 256 Kbytes L2 so

that the SPEC95 benchmarks exercise the memory hierarchy to a reasonable extent. Using a larger L1 D-cache results in negligible miss rates for the SPEC95 benchmarks, thwarting any effort to study data cache performance using SPEC95. We assume that the r-a cache probe0 hit is 1 cycle, and the probe0 and probe1 hit signals are available at the end of probe0, as per the discussion in Section 2.1.. Probe1 takes 2 additional cycles (i.e., data from probe1 takes a total of 3 cycles). L2 access is initiated after probe0 if the block is not in the cache (the tags for all the ways are checked in parallel).

For our experiments, we choose some benchmark/input combinations from the SPECint and SPECfp suite that do not require prohibitively long simulation runs. Table 2 presents the benchmarks and inputs used in this study. In addition to the SPEC programs, we also use *troff*. The benchmarks were compiled for a Compaq Alpha AXP-21164 using the Compaq C and Fortran compilers under -O4 -ifo optimization flags. All of the simulations are run to completion except for *vortex* and *go*, which we halt at 1 billion instructions.

5.1. Hit time of the reactive-associative cache

Because the r-a cache employs an unusual organization combining a direct-mapped-like data array with a set-associative-like tag array, we use the cache geometry optimizing tool, CACTI2.0 [15], configured for 0.18 micron technology, to estimate probe0 hit time. Readers should note that they should download CACTI2.0 from the official web site if they want to verify these numbers; other CACTI versions may report vastly different numbers.

In Table 3, we present the tag array hit signal (not including the OR gate in the hit signal path in set-associative caches) and data array data-out (not including the precharge phase) latencies for direct-mapped through 8-way conventional set-associative, 8-KB caches. The r-a cache's probe0 timings can be derived from direct-mapped data array and the 4-way set-associative tag array timings. We added an 10-ps delay, obtained by Hspice simulations, for each of the pass gates in the probe0 hit signal path and the data array index path, as discussed in Section 2.1.. The data out latency includes the output-way multiplexor, which must wait for the result of the tag comparison (for a set-associative cache), and the output driver. The total hit time is the longer of the data out latency, and the tag hit/miss signal latency.

Because each 2-way tag bank is half the size of the direct-mapped tag array, the 2-way tag array is actually faster than the direct-mapped tag array. As expected, the 4-way data array is considerably slower than the direct-mapped data array but the 4-way tag array is almost as fast as the direct-mapped data array, which is the critical path through the direct-mapped cache. Thus, a 4-way r-a probe0 critical path (for both data and tag) is almost as fast as that of a direct-mapped cache. For an 8 KB cache, the 4-way r-a is about 1% slower than direct-mapped, but still 41% faster than 2-way, and 43% faster than a 4-way set-associative cache. For cache sizes greater than 8 KB, however, 4-way associative tag array is no longer any slower than direct-mapped tag array, and a 4-way r-a cache is less than 1% slower than a direct-mapped cache. For sizes of 32 KB and greater, an 8-way r-a cache is less than 1% slower than a direct-mapped cache. We examine an 8-KB cache only because using a larger cache results in negligible miss rates for the SPEC95 benchmarks, but many microprocessors use L1 D-caches of size 16 KB or larger, so replacing a direct-mapped cache with a r-a cache should not affect the clock rate.

	1-way normal	2-way normal	4-way normal	8-way normal	4-way r-a
8K data (ns)	0.792	1.129	1.141	1.240	0.802
8K tag (ns)	0.590	0.554	0.524	0.537	0.534
8K total (ns)	0.792	1.129	1.141	1.240	0.802
% increase over 1-way	0	43	44	57	1
16K data (ns)	0.960	1.202	1.222	1.294	0.970
16K tag (ns)	0.612	0.588	0.585	0.550	0.595
16K total(ns)	0.960	1.202	1.222	1.294	0.970
% increase over 1-way	0	25	27	35	1

Table 3: Cache hit times.

5.2. Performance of the reactive-associative cache

In this section, we present the performance of the r-a cache using the PC and XOR schemes, compared against direct-mapped and 2-way set-associative caches. We show an idealized 2-way set-associative, 1-cycle cache as a reference point. We graph the speedups of the various cache configurations by normalizing against a direct-mapped cache in Figure 4. To underscore the r-a cache's robustness with respect to L1 bandwidth, we vary the number of L1 cache ports from 1 (top graph) to 2 (bottom graph). We model the extra bandwidth demand of probe1 accesses by holding the L1 port for an additional cycle. As discussed earlier, we consider the XOR scheme to be difficult to implement, but we still present its performance to show how it compares to the PC scheme.

In Figure 4, we classify our benchmarks into two groups. The first group of benchmarks (*vortex, gcc, li, perl*) are relatively insensitive to associativity, and achieve only modest improvements (2%-4%) even with the ideal, 2-way cache. The second group of benchmarks (*go, troff, m88ksim, swim, fpppp*) achieve speedups from 6%-10% with the ideal, 2-way cache. *Swim* has a pathological map-

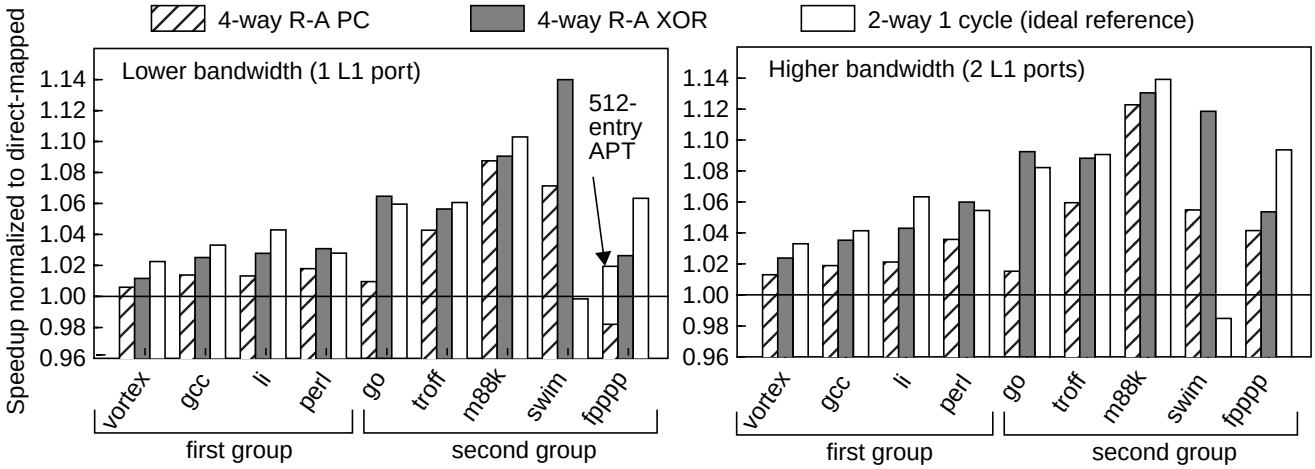

FIGURE 4: Performance of the reactive-associative cache.

	direct mapped	r-a 4-way, pc	r-a 4-way, pc	r-a 4-way, pc	r-a 4-way, xor	r-a 4-way, xor	r-a 4-way, xor	2-way set assoc.
SPEC-95 Bench	overall miss rate	probe0 miss rate	overall miss rate	prediction accuracy	probe0 miss rate	overall miss rate	prediction accuracy	overall miss rate
vortex	5.1	6.7	4.5	97.8	6.5	4.1	97.5	3.9
gcc	8.0	8.8	7.0	98.2	8.4	6.4	97.9	6.4
li	6.0	5.7	5.6	99.9	5.4	5.1	99.7	4.8
perl	5.4	5.6	4.2	98.5	5.9	3.3	97.4	3.9
go	8.9	9.5	8.3	98.7	6.9	5.8	98.9	6.3
troff	5.0	4.3	3.4	99.1	3.7	2.7	98.9	2.8
m88ksim	5.2	3.3	2.4	99.1	4.2	2.2	98.0	2.0
swim	49.7	48.7	46.9	96.6	49.8	44.6	90.6	50.8
fpppp	7.4	24.8	3.5	77.9	6.2	5.2	98.9	2.7
MEAN	7.9	8.7	5.9	96.0	7.3	5.3	97.5	5.1

Table 4: Reactive associative miss rates compared to set associative caches.

ping problem which causes the idealized 2-way cache to perform slightly worse than direct-mapped, but the problem subsides with increasing associativity, which is why 4-way r-a performs better than the ideal, 2-way cache.

Focusing on the one port case using the PC scheme in the top graph of Figure 4, the r-a cache achieves 1%-3% speedups for the first group, and 1%-9% speedups for the second group. The r-a cache performs within 2% of the idealized 2-way cache in all benchmarks except *go* and *fpppp*. Thus, the r-a cache dynamically adjusts to the associativity requirements of the benchmark, providing the first group with low degrees of associativity, and the second group with high degrees of associativity—but at direct-mapped speeds. *go*'s branch prediction accuracy is known to be poor, indicating its instruction stream's unpredictable nature, which makes the PC scheme less effective. The r-a cache incurs a slowdown for *fpppp* due to *fpppp's* large instruction footprint, disrupting the PC-indexed APT, which causes high probe0 miss rates. As indicated, this slowdown turns into a 2% speedup on increasing the APT size to 512 entries.

The difference in performance between the PC and XOR schemes for one port is less than 2% in all benchmarks except *go* and *swim*. Because the XOR scheme is potentially more difficult to implement than the PC scheme, this comparison indicates that the PC scheme is a better alternative.

In the two-port case, the trends are similar to those of the one-port case. But the speedups of the r-a cache and the idealized 2-way cache are higher because the higher bandwidth exposes more of the direct-mapped cache's miss latency, which is reduced by the r-a and 2-way caches. *fpppp*'s bandwidth pressure is absorbed by the extra port, resulting in a 4% speedup without increasing the APT size.

In Table 4, we compare the miss rates for the r-a cache with those of direct-mapped and 2-way caches. Compared to the direct-mapped miss rates, probe0 miss rates under both PC and XOR schemes are less or almost equal for all

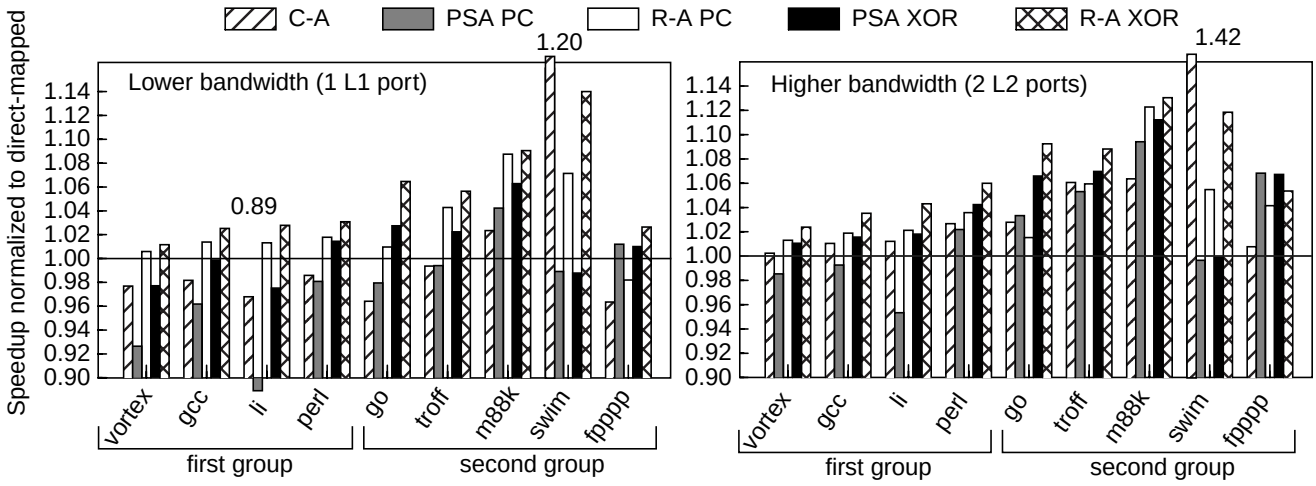

FIGURE 5: Comparison of reactive associativity to previous schemes.

	direct-mapped	c-a	PSA pc	r-a pc	PSA xor	r-a xor	c-a	PSA pc	r-a pc	PSA xor	r-a xor
Benchmark	overall	p0	p0	p0	p0	p0	overall	overall	overall	overall	overall
vortex	5.1	6.0	36.1	6.7	15.7	6.5	4.0	3.4	4.5	3.4	4.1
gcc	8.0	8.9	32.8	8.8	17.2	8.4	6.0	5.8	7.0	5.8	6.4
li	6.0	6.6	34.5	5.7	15.5	5.4	4.7	4.4	5.6	4.4	5.1
perl	5.4	6.2	20.9	5.6	12.5	5.9	3.4	3.5	4.2	3.5	3.3
go	8.9	10.3	25.8	9.5	12.0	6.9	5.9	5.4	8.3	5.4	5.8
troff	5.0	5.6	19.2	4.3	12.3	3.7	2.3	2.4	3.4	2.3	2.7
m88ksim	5.2	5.7	17.5	3.3	10.3	4.2	2.2	1.8	2.4	1.8	2.2
swim	49.7	53.6	66.4	48.7	65.1	49.8	26.2	50.7	46.9	50.3	44.6
fpppp	7.4	9.6	35.3	24.8	19.1	6.2	5.8	2.3	3.5	2.3	5.2
MEAN	7.9	9.0	29.6	8.7	16.6	7.3	4.9	4.5	5.9	4.5	5.3

Table 5: Initial probe (p0) and overall (ov) miss rates for several multi-probe cache schemes.

benchmarks except for *vortex* and *fpppp*, indicating the success of our way-prediction scheme. If the inhibit list were cleared less frequently than our TLB-based approach (Section 3.3.3.), *Vortex's* probe0 miss rate decreases to 5.3%. *Fpppp's* large instruction footprint causes thrashing in the APT, resulting in high probe0 miss rate using the PC scheme, even though the overall miss rate approaches that of 2-way. The prediction accuracy columns for PC and XOR schemes show the performance of the way predictor. When comparing the r-a cache miss rates with those of the 2-way cache, it is clear that feedback achieves high accuracy at the cost of higher overall miss rate. Because XOR is a more accurate prediction handle in general, we relax the victim list threshold from 5 to 2, which encourages more displacement. In most cases, this relaxation results in lower probe0 and overall miss rates for XOR, but worse way-prediction accuracies, as compared to PC.

Aside from *fpppp*, the other benchmarks are essentially not sensitive to varying the APT size from 64 to 512 entries, the BWT size from 64 to 256 entries, or the victim list size from 128 to 256 entries. However, if we scale up the tables to sizes larger than the cache itself, the probe0 miss rate converges to lower than 2-way set-associative miss rates.

5.3. Reactive-associative cache vs. prior schemes

In this section, we compare the r-a cache with the column-associative cache (c-a cache), which is representative of caches that rely on swapping because they are all similarly limited in performance due to the bandwidth demands imposed by swapping. We also compare the r-a cache against the PSA cache, which uses way-prediction instead of swapping. For the PSA-cache, we assume the same latency assignments as the r-a cache, as they have similar timing constraints for both probe0 and probe1; the PSA-cache has a 1-cycle probe0 hit with an additional 2 cycles for a probe1 hit. As with r-a, the cache port is only held for 1 additional cycle for a probe1 hit. The PSA cache uses a 1024-entry way-prediction table (as in [6]), with 1 hash-

	direct-mapped	r-a pc, with no reactive displacement	r-a pc, with no reactive displacement	r-a pc, with no feedback	r-a pc, with no feedback	r-a pc, original config	r-a pc, original config
Benchmark	overall	probe0	overall	probe0	overall	probe0	overall
vortex	5.1	31.9	4.4	11.7	4.4	6.7	4.6
gcc	8.2	23.3	7.4	16.3	6.7	9.0	7.2
li	6.3	7.9	6.2	9.2	5.4	6.0	5.9
perl	5.6	17.6	4.5	10.3	3.5	5.6	4.2
go	9.2	19.7	8.1	17.2	6.6	9.8	8.5
troff	5.2	13.4	3.1	9.0	2.8	4.3	3.3
m88ksim	5.3	12.0	3.1	11.0	2.1	3.4	2.4
swim	54.3	50.7	45.5	50.3	48.4	50.3	48.5
fpppp	7.4	69.0	2.3	22.4	2.8	25.9	3.5
MEAN	8.1	21.9	5.8	14.8	5.2	9.0	5.9

Table 6: Effect of reactive displacement and feedback mechanisms.

rehash bit per cache block—for a total of 160 bytes of extra state. We show PSA using the XOR scheme as well as the PC scheme so that we can compare PSA and r-a. The PSA paper recommends the XOR scheme, but we include the PC scheme also due to the XOR scheme's timing problems mentioned in Section 3.2.. In Figure 5, we compare speedups over direct-mapped cache of the 4-way r-a cache against those of the column-associative and PSA caches.

We do not compare against the group-associative cache (g-a cache) because it incurs the same bandwidth problems of swapping as the c-a cache. In addition, the g-a cache has fundamental circuit problems, which prohibit it from achieving direct-mapped hit latency. Group associativity is achieved by maintaining the location and tags of the displaced blocks in a fully-associative OUT directory. Because the OUT directory *must* be checked on every access, the previous lookup must complete before the next cache access. However, our analysis using CACTI2.0 indicates that this fully-associative directory is 48% slower (21% slower if the OUT directory is only 2-way set associative) than a direct-mapped data cache (the OUT directory has 1/4 as many entries as cache blocks, and 5 bytes per entry—as recommended in [14]), which would severely lengthen hit time.

For the 1-port (lower bandwidth) case in Figure 6 (the upper graph), PSA using PC and XOR schemes and column-associative (c-a) incur slow-downs on many benchmarks due to their higher bandwidth demand than direct-mapped, caused by poor probe0 miss rates and swapping, respectively. The slow-downs are more pronounced in the first group of benchmarks because the higher bandwidth demand is not compensated by lower overall miss rate.

Overall, the r-a cache using PC or XOR performs better than the corresponding PSA and c-a caches, because of r-a's lower bandwidth demand. Using the PC scheme, the one-port r-a cache is 3%-13% better than the PSA cache, except for *fpppp*. *fpppp*'s large instruction footprint wreaks havoc with the PC scheme for an r-a cache, as mentioned in Section 5.2.. The c-a cache achieves large speedups for *swim* because it uses a skewing hash function, which alleviates the pathological set-associative mapping problem. Similarly, if the APT and BWT sizes are increased to about 16K entries, the 4-way r-a cache does achieve comparable speedups over direct mapped for *swim*.

For the two-port (higher bandwidth) case in Figure 6 (the lower graph), column associative and the predictive sequential caches achieve speedups because the extra port absorbs the increased bandwidth pressure, although there are still a few cases of slow-down. The r-a cache using PC or XOR mostly outperforms the corresponding PSA as well as c-a caches, although by smaller margins than the 1-port case because the bandwidth advantage of the r-a cache is less important for the two-port case.

Table 5 shows the first-probe (indicated by p0) and overall (indicated by overall) miss rates for 4-way r-a (using PC and XOR), column associative, and PSA (using PC and XOR) caches. The r-a caches using PC or XOR usually have the lowest probe0 miss rates of all the caches. Although column-associative has low probe0 miss rates, overall performance is poor due to high latency and bandwidth demand of cache block swapping. Probe0 miss rates of PSA using XOR are much worse than direct-mapped miss rates, resulting in high bandwidth demand. Although the overall miss rates for the r-a caches are usually higher than those for PSA and c-a because of reactive displacement and feedback, the r-a cache performs better than PSA or c-a, affirming the notion of trading-off overall hit rate for probe0 hit rate.

5.4. Effect of reactive displacement and feedback

To isolate the effects of reactive displacement and feed-

back on the r-a cache, in Table 6 we present probe0 and overall miss rates using the PC scheme but without reactive displacement (i.e., no victim list), and without feedback (i.e., no mispredict counters in the BWT, and no inhibit list). The last row of Table 6 shows the geometric means of the miss rates for that column. In the case of no reactive displacement, probe0 miss rate increases dramatically because the cache attempts to displace all accesses, even those that do not cause any contention in a direct-mapped cache. In the case of no feedback, the probe0 miss rate increases, albeit not as dramatically as before, but the overall miss rate approaches those achieved by PSA or c-a because the cache now displaces all cache blocks regardless of predictability. Such increased probe0 miss rates would, in general, cause an increased average cache hit time and higher cache port contention. Therefore, reactive displacement and feedback are essential for the r-a cache.

6. Conclusions

We proposed the reactive-associative cache (r-a cache) based on the key observation that associativity is needed only for conflicting blocks—and should not be provided at the expense of higher hit latencies for all accesses. The r-a cache provides flexible associativity by placing most blocks in direct-mapped positions and reactively displacing only conflicting blocks to set-associative positions. To achieve direct-mapped hit times, the r-a cache uses the well-known asymmetric organization in which the data array is organized like a direct-mapped cache, and the tag array like a set-associative cache. The r-a cache uses way prediction coupled with feedback for high prediction accuracy, regardless of the prediction handle. Our reactive mechanisms (selective displacement and feedback) allow the use of PC as a viable way prediction handle. Since the instruction PC is available early in the pipeline, allowing at least six pipeline stages for the lookup, we claim that the PC scheme easier to implement than the XOR prediction proposed in the PSA paper.

A one-port, 4-way r-a cache achieves up to 9% speedup over a direct-mapped cache and performs within 2% of an idealized 2-way set-associative, 1-cycle cache. A 4-way r-a cache achieves up to 13% speedup over a PSA cache, with both r-a and PSA using the PC scheme. Finally, CACTI2.0 estimates indicate that for sizes larger than 8KB, a 4-way r-a cache is within 1% of direct-mapped hit times, while still 25% faster than a 2-way set-associative cache.

Acknowledgements

This work was in part supported by the NSF CAREER award no. 9875960-CCR, and was done when Brannon Batson was at Purdue university.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, Nov. 1988.

[2] A. Agarwal and S. Pudar. Column associative caches: A technique for reducing miss rate of direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, May 1993.

[3] T. M. Austin, D. N. Pnevmatikatos, and G. Sohi. Streamlining data cache access with fast address calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 369–380, June 1995.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report CS TR-1308, University of Wisconsin, Madison, July 1996.

[5] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 287–296, June 1995.

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

[7] J. H. Change, H. Chao, and K. So. Cache design of a submicron CMOS System/370. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 208–213, June 1987.

[8] J. Collins and D. Tullsen. Hardware identification of cache conflict misses. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 126–135, Nov. 1999.

[9] M. Hill. A case for direct-mapped cache. *IEEE Computer*, 21(12):25–40, Dec. 1988.

[10] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[11] T. Juan, T. Lang, and J. J. Navarro. The difference-bit cache. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 114–120, May 1996.

[12] R. E. Kessler, R. Jooss, A. Lebeck, and M. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 131–139, May 1989.

[13] J.-K. Peir, W. Hsu, H. Young, and S. Ong. Improving cache performance with balanced tag and data paths. In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 268–278, Oct. 1996.

[14] J.-K. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the Eighth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 240–250, Oct. 1998.

[15] G. Reinman and N. P. Jouppi. *An Integrated Cache Timing and Power Model*. COMPAQ Western Research Lab http://research.compaq.com/wrl/people/jouppi/CACTI.html, 1999.

[16] A. Seznec. DASC cache. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 134–143, Jan. 1995.

[17] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709, June 1988.

[18] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE Micro*, pages 40–49, Sept. 1997.