REACTIVE ASSOCIATIVE CACHES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Brannon Batson

In Partial Fulfillment of the

Requirements for the Degree

of

Masters of Science in Electrical Engineering

May 2000

To my family

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Prof. T. N. Vijaykumar, for spending long hours in our common pursuit to make this project successful. I would also like to thank Prof. Babak Falsafi and Prof. Cheng-Kok Koh for their invaluable service as members of my advisory committee. I am grateful to have had all three of these professors as teachers, advisors, mentors, and friends.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

ABSTRACT

Batson, Brannon. M.S.E.E., Purdue University, May, 2000. Reactive Associative Caches.
Major Professor: Babak Falsafi.

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 its miss rate. 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 dynamically provides associativity in response to application demand. The r-a cache employs a novel multi-probe organization which uses a direct-mapped data array and a set-associative tag array. It is accessed like a regular direct-mapped cache for most accesses, but it also has the ability to be accessed like a set-associative cache (using a way prediction), when it is necessary to alleviate direct-mapped line contention. Circuit analysis indicates that a r-a cache has a hit latency comparable to that of a direct-mapped cache for all direct-mapped accesses and all correctly predicted set-associative accesses. Incorrectly predicted set-associative accesses will incur an additional probe into the data array. Prior multi-probe cache organizations have suffered from poor initial-probe miss rates, and therefore require many secondary probes, which increases average hit time and demand for cache bandwidth over that of a direct-mapped cache. Furthermore, a r-a cache does not require cache block swapping, as in statically-probed multi-probe cache schemes such as column associative and group associative. A r-a cache uses two mechanisms, selective displacement and feedback, to reduce pressure on the prediction resources and achieve low initial-probe miss rates. Selective displacement refers to the notion of only displacing conflicting cache blocks to set-associative positions. The feedback mechanism allows the r-a cache to measure the ‘predictability’ of certain instructions and to prohibit

associative displacement for unpredictable accesses. Simulations show that a 4-way r-a cache, using modest prediction resources, will outperform column associative and predictive sequential associative, as well as achieve 1%-11% speedups over a direct-mapped cache on a subset of the SPEC95 benchmark suite.

1. INTRODUCTION

One of the fundamental limitations of the performance of modern computer systems is the rate at which memory requests can be serviced. While processors have steadily improved in computational performance by several orders of magnitude, memory speeds have not kept pace. This rift, as well as the concept of locality of memory reference, has motivated the development of memory hierarchies, in which small and fast SRAM cache memories are used to satisfy most requests. Successive levels of increasingly larger (and slower) caches are used to gradually insulate the nimble processor from slower main memory DRAMs. The direct relationship between memory system performance and overall system performance has made cache design a relevant issue in computer architecture.

The performance of the memory system is particularly important in the first level of cache (L1) that is accessed by the cpu. In modern processors, the L1 cache is on-chip, and split between an instruction cache (L1 icache) and a data cache (L1 dcache). The high locality and regular access patterns of instruction streams have made the L1 instruction cache perform relatively well, but the L1 data cache of most systems still is a significant performance bottleneck. There are three major reasons that this is the case: (1) loads (and therefore data cache accesses) are in the critical path of program execution, so there are few techniques which can hide the latency of a load operation; (2) for a variety of circuit reasons, it is difficult to increase the number of cache ports to satisfy all the requests from the processor; and (3) as the wire delays start to dominate intra-processor communication, the time to retrieve a cache block from the next level in the hierarchy is growing, which makes misses in the data cache increasing more expensive. In this thesis, we examine optimizations to improve the performance of the L1 data cache.

The traditional measures of a cache are its miss rate (the percent of accesses that are not satisfied by the cache), and the hit latency (the time it takes to service cache hits).

The miss rate indicates how often the next level in the hierarchy is accessed, so it becomes more important as the L2 access latency increases. The hit latency is often in the critical (circuit) path of processor execution, so it can figure prominently in the clock cycle of the cpu. The simplest form of a L1 dcache is a direct-mapped organization, in which each cache block address is mapped to a single possible location in the cache array. This structure allows for fast hit times, and also achieves respectable miss rates, usually lower than 15% for typical integer applications. Set associative caches allow a given cache block to reside in more than one place (way) within the array, which increases occupancy of reused data, and therefore decreases the miss rate. The decrease in miss rate can be drastic. For a subset of the SPEC95 benchmark suite [1] (including vortex, gcc, li, perl, go, troff, m88ksim, swim, and fpppp) with an 8k L1 dcache, a 2-way set associative cache has (on average) 32% fewer misses than a direct-mapped cache. A 4-way set-associative cache has 41% fewer misses than a direct-mapped cache. This decrease in miss rate is unfortunately accompanied by an equally drastic increase in hit latency, due to the additional logic necessary to implement the associativity. Using the cache analysis tool CACTI [2], we estimate that an 8k 2-way set associative cache is 53% slower than an 8k direct-mapped cache. This creates a major trade-off between hit latency and miss rate in the L1 data cache. In Section 1.1, we examine more closely the circuit issues that control the hit time of direct-mapped and associative caches. The large increase in hit latency between direct-mapped and a set-associative caches has motivated interest in multi-probe cache schemes, which have a hit-latency close to that of a direct-mapped cache, but with miss rates closer to an associative cache. These schemes have failed to garner industry attention because of fundamental problems with respect to performance and bandwidth utilization. We briefly examine these issues in section 1.3. In section 1.4, we introduce the reactive associative cache, which addresses the major problems with multi-probe caches.

1.1 Critical Path Analysis of Direct-Mapped and Set-Associative Caches

In this section we will examine the circuit differences between a direct-mapped and set-associative cache as it pertains to the access time of the cache. Knowledge of these differences is critical to understanding the components of cache access time. Figure 1.1

and Figure 1.2 are generic representations of a direct-mapped and a set-associative cache, respectively. For the purposes of this discussion, we ignore the word multiplexing delay and the output drivers, as they are common to all organizations. Also, though our diagrams do not indicate as much, both the set-associative and direct-mapped tag & data arrays are subject to squaring optimizations, which optimize the width and height of the arrays to minimize the access delay. Furthermore, it is uncommon for today's processors to use a single array larger than 8kb, so 16kb and larger data banks are, in general, sub-banked (split into two or more smaller arrays). However, these sub-banks are not independently accessible, and therefore act like a single array (caches with independently accessible banks are referred to as *multi-banked*, or simply *banked*, caches). In this thesis, when we speak of a single array or a single bank, we do so with the understanding that the array may be sub-banked, but acts as though it were a single array.

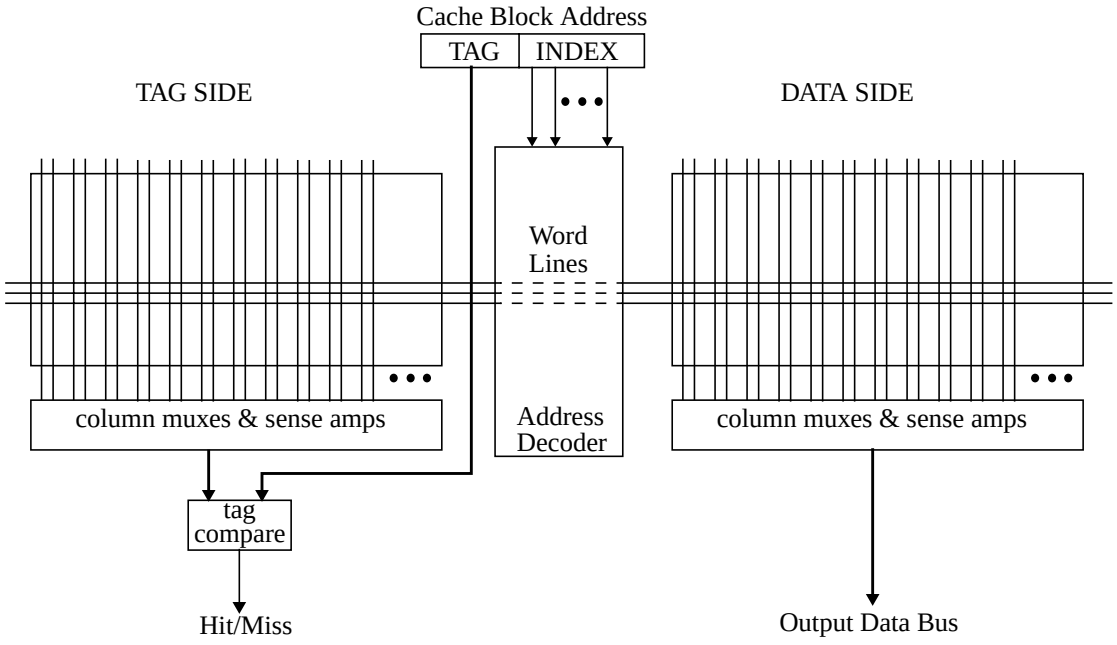

Fig. 1.1. A direct-mapped cache

In the direct-mapped cache (Figure 1.1), each set holds a single cache block. As soon as the address becomes available, the appropriate word line is activated, which causes the bitlines take on the value stored within that cache block. The sense amplifiers detect transitions in the bitlines and immediately make this cache block available on the

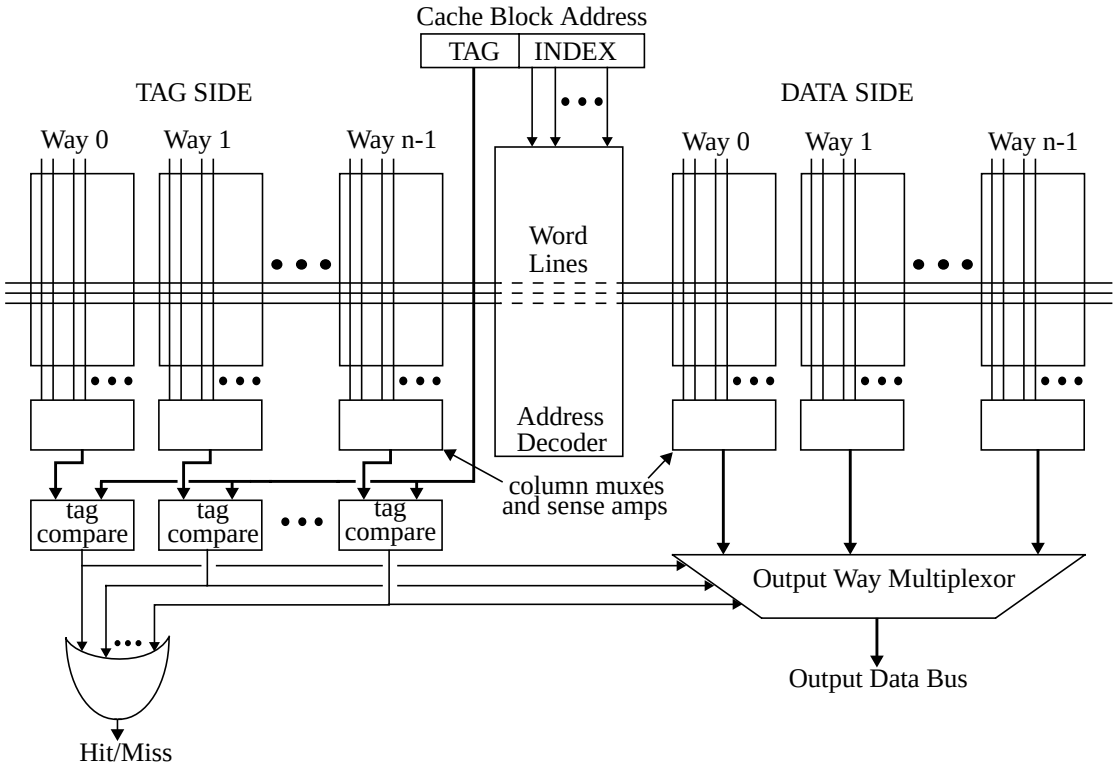

Fig. 1.2. A n-way set-associative cache

output data bus. In parallel with the data cache probe, the tag array is similarly accessed to provide the tag of the stored cache block. This stored tag is compared to the tag bits of the original address, and if they match, the Hit/Miss signal is asserted to indicate a cache hit. By the time the hit is signalled, the data bus should already contain the value of the cache block (as the critical path is, in general, through the tag path). As mentioned earlier, this organization allows for extremely fast access times, which in turn enables fast clock cycles. However, because each address maps to exactly one line in the cache, no two addresses with the same set bits can reside in the cache at the same time. This leads to *hot spots* (frames with high line contention), which cause thrashing, and *holes* (under utilized cache lines), which waste cache space.

The set-associative cache (Figure 1.2) works in a somewhat similar manner, except that multiple cache blocks can reside in a single set, which alleviates most line contention. When the address becomes available, the appropriate wordline is activated. This wordline is shared by each associative way-bank. In each way-bank, the bitlines will take on the val-

ues of the cache block stored in that way. The sense amplifiers will detect transitions on the bitlines and produce logic level values at the inputs to the output way multiplexor. The tag array is also divided into separate way banks, holding the tag information for the cache blocks in corresponding ways of the data array. Each way bank in the tag array is probed in parallel to produce inputs to the n tag comparators (which will compare these stored tags to the tag bits of the address). The results of these tag comparisons are used to generate the select lines for the output way multiplexor on the data side. Once these select lines are available, the output way multiplexor will output the correct cache block onto the output data bus. The set associative cache introduces significant critical path delay in addition to the delays associated with a direct-mapped cache: (1) The time it takes to propagate the way select lines from the tag comparators to the output multiplexor, and (2) The time to switch the multiplexor and provide the results on the data bus. Note that the data bus typically carries a large RC load, and while the evaluation on this bus is overlapped with tag compare in a direct-mapped cache, it is in the critical path of cache access for a set-associative cache. Thus, a set-associative cache can be expected to have a significant increase in hit latency over a direct-mapped cache. This has lead designers to explore alternative associative organizations, such as multi-probe caches. Further discussion of the intricacies of tag and data-side cache timing can be found in [3].

1.2 Multi-Probe Caches

The large increase in hit latency of a set-associative cache over a direct-mapped cache has motivated research in alternative caches that have direct-mapped hit latencies, but with increased associativity to allow for lower miss rates. Most research in this area has been in what we refer to collectively as *multi-probe caches*. The initial concept behind these organizations is to maintain a direct-mapped type cache, but provide the ability to sequentially probe different locations within the cache, and therefore allow any given cache block to be placed in multiple locations. While multi-probe schemes allow for direct-mapped hit latencies on the initial probe into the cache, any subsequent probes require one or more additional cycles of hit latency, as well as stalling access to the cache by other memory operations. In this thesis, we will refer to the initial probe as probe0, the

subsequent probe as probe1, and so on. Most multi-probe organizations will check in a static location on the initial probe into the cache (usually the direct-mapped location). These organizations are referred to as *statically-probed* caches. We refer to caches that employ way-predictors (i.e. probe0 is not necessarily to the direct-mapped location) as *dynamically-probed* caches.

Statically-probed caches include hash-rehash [4], column-associative [5], sequential & parallel multicolumn [6], and group-associative [7]. In Section 3, we will analyze in detail the differences between these organizations. At this point, however, it suffices to point out the fundamental problems shared by all statically-probed caches. One such limitation is that since the initial probe is always to the direct-mapped location, the initial-probe miss rate is always at least as large as the miss-rate of a direct-mapped cache. The probe0 miss rate increases as more blocks are displaced from their direct-mapped locations, as there are fewer blocks that can be reached on the first probe into the cache. To decrease the probe0 miss rate, all of the statically-probed caches will swap a block found on a subsequent probe with the direct-mapped location. Unfortunately, dedicated circuitry to support cache block swapping is impractical for caches, which use regular and highly optimized circuitry. A modest implementation of cache block swapping would require two reads and two writes (using an extra intermediate cache block), which increases average hit time and degrades valuable L1 bandwidth. Furthermore, we will show that even with cache block swapping, the initial-probe miss rate is still nominally larger than the direct-mapped miss rate.

To increase the probability of a first probe hit without block swapping, the predictive sequential associative cache [8] (PSA cache) proposed using way-prediction. Way-prediction has been previously used for low-power purposes in icaches [9], but PSA is the first attempt to apply it to dcaches to create a low-latency multi-probe cache. The initial probe can hit in any way in the set as long as the way-prediction is correct. Thus the PSA cache's probe0 miss rate is dependent only on the prediction accuracy of the way-predictor. The PSA cache predicts all accesses, even those that would have not produced conflict misses in a direct-mapped cache, and without regard to the performance of the predictor. Therefore, the PSA cache has poor initial-probe miss rates, which are usually significantly

worse than direct-mapped miss rates. Each probe0 miss requires the cache to be stalled until probe1 completes. This can actually cause a system with a PSA cache to perform worse than one with a direct-mapped cache, if there is limited L1 bandwidth. Predictive sequential associative and reactive-associative, which we introduce in this thesis, are both considered dynamically-probed caches.

1.3 Reactive Associative Caches

Statically-probed caches have been around for nearly fifteen years, but have not gained industry acceptance, due to the requirement for cache block swapping. Predictive sequential associative is similarly fated, due to its heavy increase in bandwidth utilization. We suggest a new dynamically-probed scheme called reactive-associative (r-a), which does not require cache block swapping, and has a bandwidth demand similar to that of a direct-mapped cache.

The r-a cache employs a novel organization in which the data array is direct-mapped, but the tag array is set-associative (as shown in Figure 1.3). The data side contains a single data array, like in a direct-mapped cache, which can support only one probe at a time. The data side does not require tag match information from the tag array on the initial probe, and therefore the critical path though the data side is equivalent to that of a direct-mapped cache. The tag side contains equivalent logic to the tag side of a set-associative cache (Figure 1.2). Specifically, the tag array is divided into way banks that are probed in parallel for a given set (from the cache block address). The critical-path delay of the tag side of a set-associative cache is comparable to that of a direct-mapped tag side. This is because the set-associative tag array contains two or more way banks that are each smaller and faster than a direct-mapped tag array; since the way banks are probed in parallel, the tag array of an associative cache is actually faster than that of a direct-mapped cache, which offsets the increase in other logic.

Most accesses proceed just as though it were a direct-mapped cache, but when line contention is detected, the appropriate cache bocks are placed into set-associative positions. The initial probe location is controlled by a way selection multiplexor, which allows either the direct-mapped way or a way prediction to be selected depending on whether it is

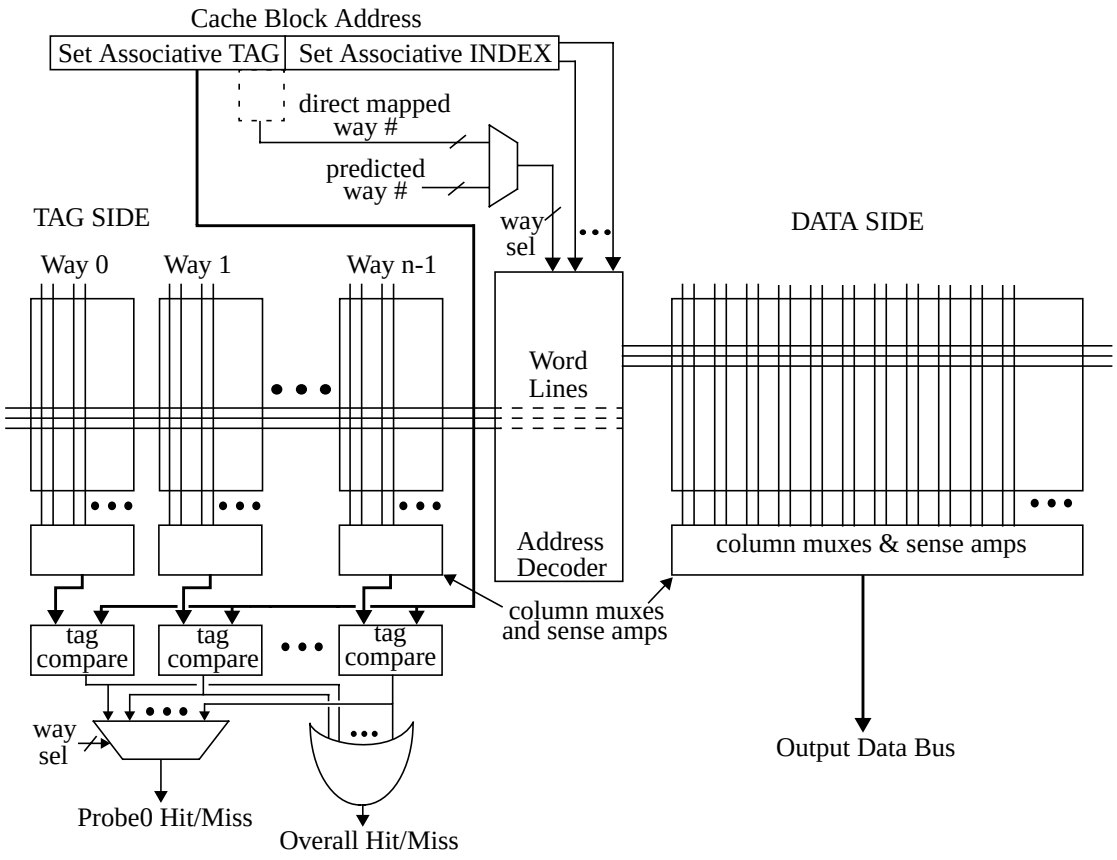

Fig. 1.3. A reactive associative cache

a direct-mapped or set-associative access, respectively. The advantage of using a set-associative tag array is that the location of the cache block is determined by the end of the initial cache probe, which simplifies pipeline timing, requires at most one additional probe on a probe0 miss (even for highly associative r-a caches), and eliminates superfluous additional probes if the block is not in the cache. The idea of using a set-associative tag array with a direct-mapped data array was proposed initially in [6]. However, we show that this is actually a feasible alternative by performing critical path circuit analysis with CACTI. As can be seen in Figure 1.3, the reactive-associative cache is asymmetric in configuration. However, cache designers routinely apply separate squaring and sub-banking optimizations to tag and data arrays, so architectural symmetry rarely maps to circuit symmetry even for traditional direct-mapped or set-associative caches. It is for this reason that the asymmetrical organization of the reactive-associative cache is unlikely to create additional floor planning problems.

Since the data output path is decoupled from the tag comparison path, the r-a cache can support initial probe hit latencies comparable to those of a direct-mapped cache. In Section 4.1, we show that for 16k and larger cache sizes, the difference between the hit latency of a r-a cache and a direct-mapped cache is less than 0.8%. Using a set-associative tag array does not impact hit time, in general, because a set-associative tag array has performance comparable to that of a direct-mapped tag array (as mentioned earlier, in most cases, a set-associative tag array is faster than a direct-mapped tag array, since the set-associative array is divided into smaller and faster way banks that are probed in parallel).

While the organization of the r-a cache is interesting, the true novelty comes from the displacement/prediction subsystems. The r-a cache will only displace conflicting blocks to set associative positions, therefore relieving what would be line contention (and thrashing) in a direct-mapped cache. Conflicting cache blocks are detected using a so-called victim list, which tracks recent L1 dcache misses. By only displacing conflicting blocks, we relieve pressure on the way-predictor, which now must only track way prediction information for displaced (contentious) cache blocks. The notion of only displacing certain blocks is referred to as *selective displacement*. Furthermore, we determined that in each application, there are a group of instructions that have poor predictability. The r-a cache employs a *feedback* mechanism which will measure the dynamic prediction accuracy per individual (or groups of) instructions. Instructions with poor predictability are prohibited from accessing displaced cache blocks (i.e. the cache blocks are forced to reside in their direct-mapped positions). By limiting the candidates for associative displacement, the r-a cache will have a higher overall miss rate, in general, than the earlier multi-probe cache schemes. However, this performance disadvantage is more than offset by the increased performance due to have a much lower probe0 miss rate and by eliminating the need for costly block swapping.

The way predictor uses a prediction handle, which is some function of system state that correlates to data access patterns, and is available prior to the effective address. This prediction handle is used to index into a prediction table, which allows for a way prediction. Predictive sequential associative recommended the use of XOR way prediction, in which the source register contents are logically XORed with the offset value, to produce

an approximation of the data address (this is similar in flavor to zero-cycle loads, which were proposed in [10]). Unfortunately, it is unlikely that this method could be used because of the strict timing constraints of cache accesses. For XOR prediction, the logical operation would have to be performed, and a way prediction table lookup completed, all in the time of a normal address computation. For r-a, we suggest using PC prediction, which is a weaker prediction handle that correlates the address of the memory operation (i.e., the PC value) to the way prediction. Since the PC is available many cycles earlier than the memory request is dispatched, there is plenty of time for table lookups, and there is no risk of compromising the critical path of cache accesses.

We evaluated the performance of several multi-probe caches on a subset of the SPEC95 benchmark suite including applications of varying associativity requirements. For these benchmarks, the geometric mean of the direct-mapped miss rates is 7.9%. The initial probe miss-rate mean for the PSA cache (using XOR prediction) is 16.6%, while the initial probe miss-rate mean for the r-a cache (using XOR prediction) is 7.3%. When using the more implementable, but somewhat weaker, PC prediction scheme, the r-a cache achieves a respectable initial-probe miss rate mean of 8.7%. Execution time simulations with these benchmarks show that a 4-way associative r-a cache, using 1184 bytes of prediction storage, achieves 1%-11% speedups over a system with a direct-mapped cache, and 1%-7% speedups over a system with a predictive sequential associative cache, when the bandwidth to the L1 dcache is limited.

1.4 Thesis Roadmap

In Section 2, we develop the reactive associative cache in detail. In Section 3, we describe the other multi-probe cache schemes, and do a qualitative comparison against reactive associative. In Section 4 we provide a thorough performance analysis using SPEC95 benchmarks on an out of order simulator. We present our conclusions in Section 5.

2. REACTIVE ASSOCIATIVE CACHE

The r-a cache employs a physical organization similar to that of a direct-mapped cache, but allows cache blocks to reside in set-associative positions. Only blocks that cause persistent conflict misses are displaced. Low hit latency is achieved by using a direct-mapped data path. Since a direct-mapped data array can probe only one location at a time, associative cache blocks are found using a way prediction. Incorrectly predicted associative accesses will require an additional probe into the data array. Cache blocks residing in their direct-mapped positions can be found via conventional direct-mapped indexing, without the need for a way prediction.

2.1 Basic Organization

The r-a cache uses a set-associative tag array and a direct-mapped data array, as can be seen in Figure 1.3. A more conceptual view of the cache can be seen in Figure 2.1. 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 hits, the access is complete and the data is returned to the processor. A probe0 miss occurs for one of three reasons: (1) the access is predicted to be direct-mapped, but the block is actually in a set-associative position, (2) the access is predicted to be set-associative, but the way-prediction provided is incorrect, or (3) the block is not resident in the cache at all. Since the tags for all the ways are checked in parallel during probe0, the actual location of the cache block is correctly determined by the end of probe0, even if the initial prediction is incorrect. Therefore, on a probe0 miss, the cache block can be retrieved from the correct way with only one additional probe (called probe1) into the data array. The tag array would not need to be accessed again, as the tag information was verified during the initial probe (therefore, an overall hit signal is generated at the

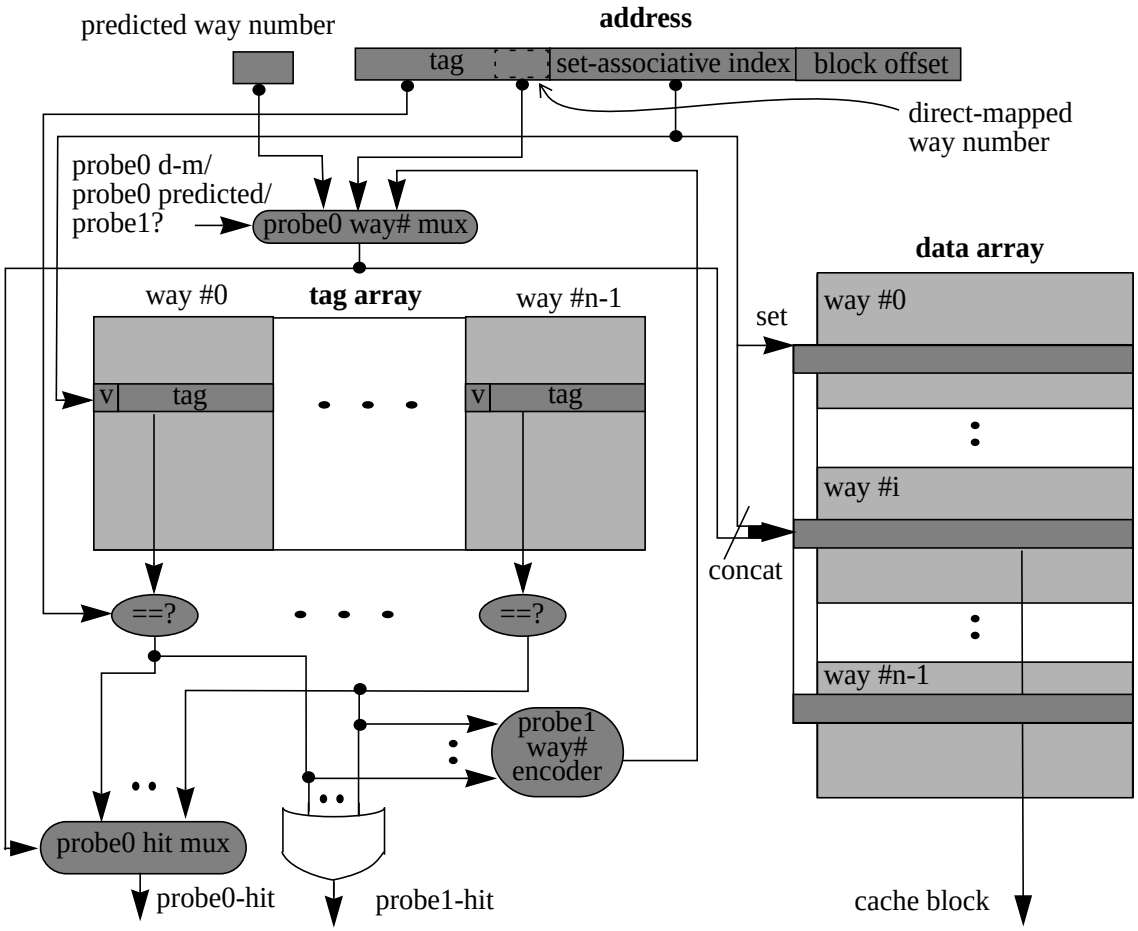

Fig. 2.1. The reactive-associative cache.

end of probe0, just like in a conventional cache). Of course, if the block is not in the cache at all, then this additional probe is unnecessary, and we can proceed immediately with an L2 cache access.

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 with the correct position in probe1.

2.1.1 Data side

The data side of a conventional n-way set-associative cache (Figure 1.2) has n way banks, each of which is a separate array. The r-a cache uses one data bank, similar to a

direct-mapped cache (Figure 1.1). This single data array may be thought of as containing the traditional data bank for each set-associative way, placed one below another, sharing bit lines (see Figure 1.3). Thus, the blocks belonging to a set in the r-a cache are spread out in the data array, placed at a stride equal to the number of sets in a traditional set-associative cache (Figure 2.1).

To index into the data array for probe0 direct-mapped accesses, the index bits corresponding to a conventional set-associative cache are concatenated with a few lower order bits from the conventional set-associative tag field, called the *direct-mapped way number* for the address. For probe0 set-associative accesses, the conventional set-associative index is concatenated with the predicted way number provided by the way-prediction mechanism. For probe1 accesses, the conventional set-associative index is concatenated with the correct way-number obtained during probe0. Thus, the set is identified by the conventional set-associative index, but the block within the set is determined by the way-number output of the multiplexor *probe0 way# mux*, as shown in Figure 2.1. The select input to the multiplexor is set to probe0 direct-mapped, probe0 set-associative, or probe1. The select for *probe0 way# mux* is set to probe0 direct-mapped or probe0 set-associative by the way prediction hardware prior to the initial probe. If probe0 fails (but there is a tag match in an alternate way during probe0), then the mux is set to probe1 for the subsequent cache access, so that the correct way will be supplied to index the data array.

2.1.2 Tag side

The r-a cache tag side is similar to that of a set-associative cache (Figure 1.2 & Figure 1.3). Much like conventional set-associative caches, the r-a cache uses as many tag banks in the tag array as the associativity of the cache, with each entry containing the usual valid and other state bits and the conventional set-associative tag. The tag array is accessed using the conventional set-associative index, probing all the banks in parallel. The difference between the r-a tag side and the set-associative tag side is in the Hit/Miss signals that are generated. Both will generate an overall hit signal (that will indicate that the cache block is resident in any of the n ways), but the r-a cache also generates a probe0 hit signal. The probe0 hit signal indicates that the cache block is indeed in the cache, and

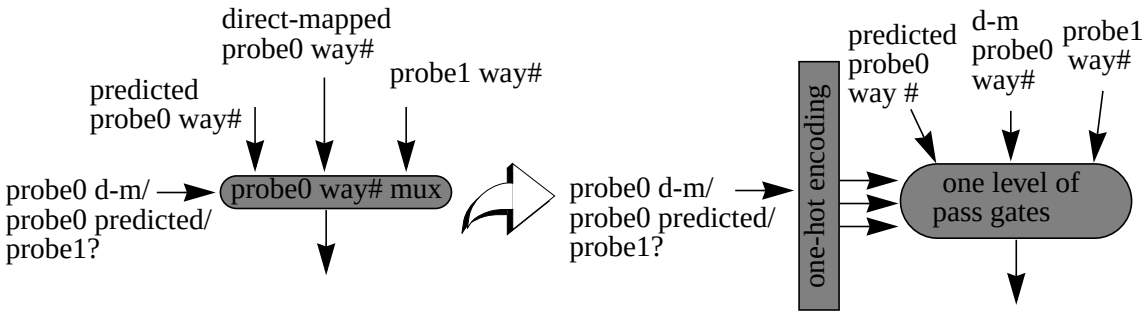

Fig. 2.2. Implementation of probe0 way# multiplexor.

the initial probe prediction was accurate. If the probe0 hit signal is low, and the overall hit signal is high, then a probe1 must be performed. The probe0 hit signal is created by multiplexing (by *probe0 hit mux*) the tag match signal generated by the comparator for the way number of the initial data probe, as in Figure 2.1.

Although the r-a cache uses prediction to probe the data array, data is returned to the processor only after ascertaining that the prediction is correct—so there is no increase in speculation over a system with a direct-mapped cache. However, the r-a cache does not preclude optimistically returning the data before the tag match is confirmed [11], which is not possible in a conventional set-associative cache.

2.1.3 Cache miss

On an overall miss, the cache block is retrieved from the L2 cache, and usually is placed in its direct-mapped location in L1. However, if the cache block address is identified as a *victim* (a persistently conflicting cache block), then it will be placed into an associative position. The associative position can be selected using any of the conventional replacement algorithms such as LRU, random, etc.

2.1.4 Probe0 hit latency

As discussed in Section 1.1, set-associative caches are fundamentally slower than direct-mapped caches. However, our critical path analysis indicates that a reactive associative cache is comparable in speed to a direct-mapped cache for the initial probe, if we assume that the prediction information is available prior to the cache access. In Section 2.2, we show how way-prediction can be done before the data address is available.

Compared to a conventional direct-mapped cache of equal size, the r-a cache introduces the extra multiplexor *probe0 way# mux* in the data array index path (Figure 2.1). 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 to choose between the probe0 direct-mapped, probe0 way-predicted, and probe1 way numbers is available earlier than the data address because the signal comes from the way-prediction mechanism. A circuit-level optimization to exploit this fact is to generate a one-hot encoding of the select signal to set up the multiplexor select path well before the data address is available, so that the multiplexor adds only its propagation delay, and not any delay due to the select signal itself, to the data array index path.

Probe0 way# mux can be implemented as a single level of pass gates using the one-hot encoding of the select signal, as shown in Figure 2.2. In the case of direct-mapped probe0, the data array index path propagates through one extra pass gate, compared to that of a conventional direct-mapped cache. In the case of way-predicted probe0, because the predicted way number is available earlier than the address, the data array index path also propagates through one extra pass gate, compared to that of a conventional direct-mapped cache. Some or all of this pass gate delay may be further folded into the data array address decode tree or the address-compute pipeline stage. Probe1 may incur a whole extra cycle to account for the *probe1 way# encoder* (Figure 2.1) and the pass gate.

The tag side 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 2.1). Using the multiplexor *probe0 hit mux* (Figure 2.1), the probe0 hit signal is generated by selecting the tag match for the probe0 way-number from among the tag matches from all the banks of the tag array. 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. A one-hot encoding of this select input to *probe0 hit mux* allows the multiplexor select path to be set up well before the tag match signals reach *probe0 hit mux*. Thus, the probe0-hit signal incurs extra delay of only one pass gate, compared to a conventional direct-

mapped cache hit signal. Assuming that the select lines for *probe0 hit mux* & *probe0 way# mux* are set up early, as we mentioned earlier, there are no additional transistor switch delays in the critical path of the probe0 access in excess of those in a direct-mapped cache.

The overall-hit signal (probe1 hit signal) incurs extra delay of one OR gate, compared to a conventional direct-mapped cache hit signal.

2.2 Way Prediction

For each access to an r-a cache, there must be a prediction as to whether it is a direct-mapped or a set-associative access. If it is a set-associative access, we also must predict the associative way that the block is in. Perhaps confusingly, we refer to both the above predictions as being performed by the way prediction hardware. In the hit time discussion above (Section 2.1.4), we indicated that the r-a cache has a direct-mapped hit latency if the way predictions are available prior to the cache access. Clearly, this strict timing constraint limits the possible prediction criteria to those available prior to generation of the cache block address. Icache way-prediction techniques can be combined with branch prediction, but dcaches do not interact directly with control flow, so those techniques cannot be used directly.

We examine two handles that can be used to perform way prediction: instruction PC of the memory reference; and XOR, which is an approximation of the data address formed by XORing the register value with the instruction offset (proposed in [10], and used in [8]). These two handles represent the two extremes of the trade-off between prediction accuracy and early availability in the pipeline. PC is available much earlier than the XOR approximation but the XOR approximation is more accurate because it is impossible for PC to disambiguate different data addresses touched by the same instruction. Other sources for way-prediction are possible, some of which are explored in [8].

Figure 2.3 shows a generic out-of-order processor pipeline and the pipeline timing of both PC-based and XOR-based way-prediction with respect to cache access. Because the instruction PC is available early in the pipeline, 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 com-

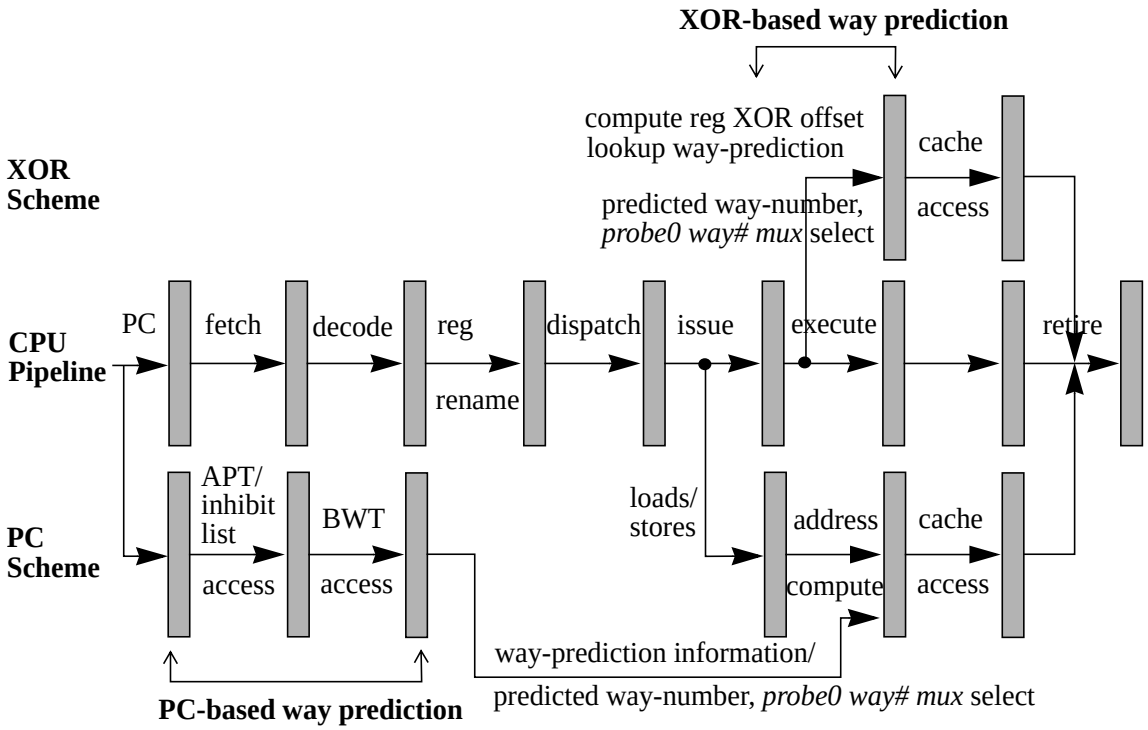

Fig. 2.3. Pipeline timing of way-prediction.

puted. The XOR scheme would require obtaining the contents of a physical register (often obtained from a register-forwarding path), performing the bitwise XOR operation on that value with the offset, and using the result to index into a prediction table. The prediction table entry will then provide the way prediction for cache access. This entire algorithm must complete within the time to compute a full add (for the address computation) to avoid delaying cache access. Note that the prediction table must have more entries that the L1 dcache, or must be highly associative, since the XORed values will experience the same line contention problems in the prediction table as their counterparts do in the dcache. Thus, the prediction table alone will have a significant access time.

2.2.1 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 (e.g., register $29 in MIPS, register $30 in Alpha), (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.

2.2.2 Access-prediction table and block way-number table

Figure 2.4 depicts the way-prediction scheme showing all the structures used by the mechanism. At this point in the discussion, only the access-prediction table (APT) and the block way-number table (BWT) are of interest, as they perform the actual predictions.

A naive implementation of a PC-based prediction might use a single array, indexed by PC, which provides a way prediction for that instruction. However, we found that a direct correlation from PC to way is problematic (as we describe below) and yields poor prediction accuracy. We solve the major problems with a level of indirection through the APT.

When a block is displaced to a set-associative position, it may cause further conflicts, in which case it is replaced from the cache. The r-a cache places a block replaced from a set-associative position into 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 another, 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, problems due to block transit are exacerbated because each of these instructions incurs a misprediction. Because way-

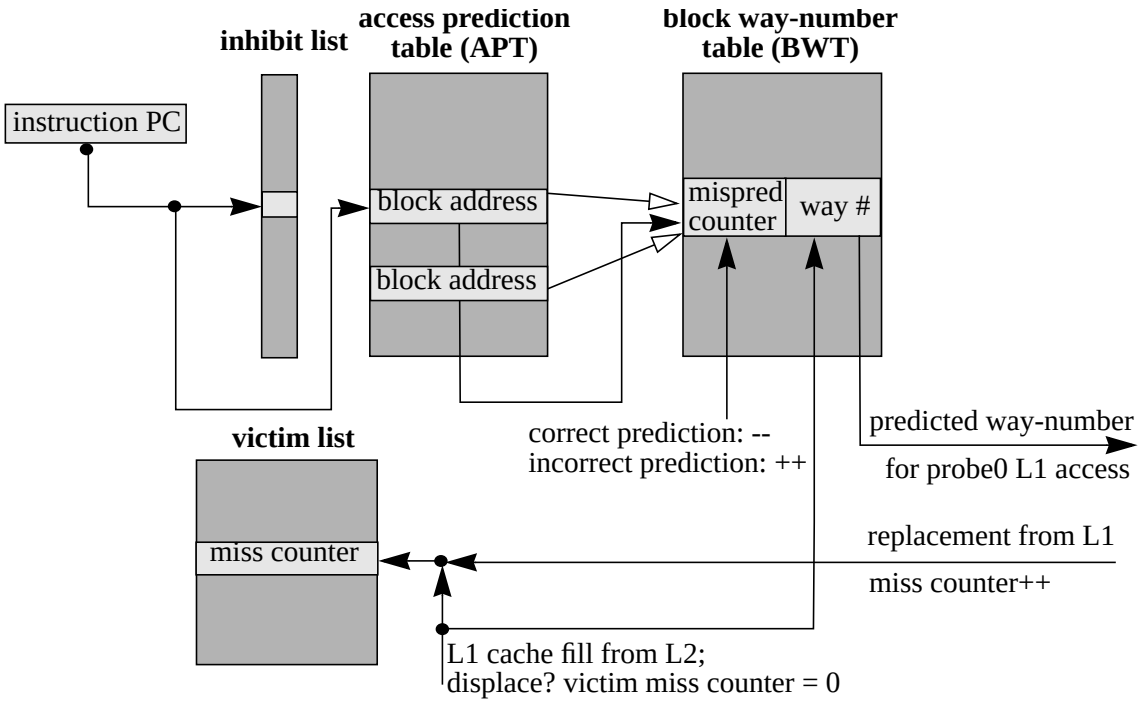

Fig. 2.4. PC-based way-prediction structures.

prediction maps PCs to way-numbers, and not the other way, updating way-number is difficult, since there is no inverse mapping into the array.

Adding one level of indirection solves the block transit problem. The instruction 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 of reference 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 instruction PC is not found in the APT, or if the block address supplied by the APT is not found in the BWT, then the access is predicted to be direct-mapped. If the appropriate entries are found in the APT and the BWT, then the way-number from the BWT entry is sent to the cache as the predicted

probe0 way-number. 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 has to hold only the instructions that access displaced blocks, small size may suffice. However, the table should be associative, so that we don't transfer instruction line contention problems to data cache performance. Similar to the APT, the BWT has to hold the way-numbers of only the displaced blocks. Because displaced blocks, by definition, conflict in the cache and the BWT uses block addresses to index, the BWT may need to be large or highly associative (or use a skewing function to index). Conflicts in the BWT cause mispredictions because way-numbers of displaced blocks are lost when BWT entries get replaced, and when the APT does not find a matching BWT entry, the APT predicts a direct-mapped access, which results in mispredictions for truly displaced blocks. But because both the APT and the BWT access are well ahead of the cache access in the pipeline (Figure 2.3), high associativity of the APT or BWT is not likely to delay probe0 initiation.

Much like a cache, each APT and BWT entry requires a tag to identify the matching PC and block address, respectively. The tags can be compressed to reduce the tag storage overhead. This compression involves performing a simple, bit-wise logical function on the original tag bits, allowing for high accuracy, but with modest tag overhead.

2.2.3 XOR-based way-prediction

XOR-based way prediction is similar, in flavor, to the idea proposed in [10], and subsequently used for way-prediction in the PSA cache. As mentioned earlier, XOR finds a way prediction by performing a bitwise XOR on the offset and source register value and using the result to index into a prediction table, all during the address computation. 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.

Because the XOR scheme does not use instructions as the prediction handle, the scheme does not need a level of indirection used in the PC-based scheme, and hence, does

not use the APT. The XOR scheme uses the BWT indexed by the XOR value, instead of the block address supplied by the APT of the PC scheme. 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 may be more accurate than the PC scheme, this late access may cause the timing problems discussed in Section 2.2, further exacerbated because of BWT's high associativity or large size requirements (the PSA paper suggests 1024 entries for 256 blocks in the L1 cache), as discussed in the previous section. Otherwise, the XOR scheme's BWT behaves identical to the PC scheme's BWT.

2.3 Selective Displacement and Feedback

While displacing conflicting blocks reduces overall misses, the initial-probe miss rate typically suffers due to increased pressure on the way-prediction mechanism. Because mispredictions result in a second data array access (if the block is in the cache), overall hit latency and bandwidth to the processor may be noticeably deteriorated if the initial-probe miss rate is significantly higher than the direct-mapped miss rate. The main reason for this degradation 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-processor bandwidth.

Because way-prediction has to use inexact information due to pipeline timing constraints, it is difficult to build a perfect (100%) way predictor. Therefore, the r-a cache attempts to reduce the number of accesses it predicts, so that either data is in a direct-mapped position or is highly-predictable in a set-associative position, keeping the number of mispredictions to an acceptable amount. Regardless of whether the PC-based or XOR-based way-prediction scheme is used, the r-a cache (1) selectively 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 redisplaced from set-associative to direct-mapped positions, avoiding repeated mispre-

dictions and (3) disallows unpredictable blocks from being displaced to set-associative positions using a feedback mechanism. Thus, the r-a cache achieves performance robustness by trading-off overall hit rate for first probe hit rate, and lowering bandwidth demand.

2.3.1 Victim list: selective 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. Each victim list entry consists of a block address and a saturating counter. The block address of a missing block is inserted in the victim list and the corresponding counter, which counts the number of misses the block has incurred in the past, is incremented. After the block is 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. Thus, the victim list approximates identification of conflicting blocks, enabling the r-a cache to displace only those blocks, irrespective of the prediction scheme used.

The victim list needs to be a high-associative structure to avoid conflicts in the victim list itself. The victim list is indexed using block addresses whenever a block is replaced from the cache and whenever a L1 cache fill is encountered, as shown in Figure 2.4. L1 cache misses insert the missing block address into the victim list and L1 cache fills lookup the victim list to determine whether the missing block should be displaced to a set-associative position. 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 will not impact hit latency. The victim list need not be implemented as a separate structure, the miss counters could be placed in the L2 tag array; this implementation would increase the L2 size by less than 0.5%.

2.3.2 Feedback with misprediction counters and the inhibit list

There are some memory instructions that have unpredictable data access behavior, regardless of whether the PC-based or XOR-based scheme is used. If the cache blocks that these unpredictable instructions act on are permitted to reside in associative positions, then they will incur many initial probe misses. A high initial probe miss-rate causes high bandwidth utilization, which can actually hurt system performance on a bandwidth limited system. The solution to this problem is to track the way prediction accuracy of instructions that act on associative data, and to disable associative displacement for these instructions.

Way prediction accuracy is tracked using saturating misprediction counters, resident in each entry of the BWT (Figure 2.4). When the BWT provides a way-prediction for an associative access, the misprediction counter is incremented or decremented for a correct or incorrect prediction, respectively. When accesses to a particular cache block produce repeated initial probe misses due to mispredictions, the BWT misprediction counter for that block saturates at a threshold, referred to as the *inhibit threshold*. At this point, any instruction that acts on this cache block is labeled as unpredictable. Here we see another benefit to the level of indirection through the APT for PC-based prediction (see Section 2.2.2), namely that all instructions that act on a single datum will share a single misprediction counter. Since the misprediction counter is incremented on a misprediction and decremented on a correct prediction, the group prediction behavior of all the instructions is measured. This prevents one unpredictable instruction from limiting associative displacement when several other instructions are able to predictably access the cache block.

When an instruction or a group of instructions are labeled as unpredictable (i.e., when the BWT misprediction counter reaches the *inhibit threshold*), we proceed to inhibit those instructions from further associative displacement. This is done by setting an inhibit bit for each unpredictable instruction in the inhibit list (Figure 2.4). Though we discuss it as a separate structure, the inhibit list naturally lends itself well to being combined with the L1 instruction cache. Such an arrangement would require only one extra bit of state information per instruction in the icache. Once an instruction is inhibited, it will henceforth always access the direct-mapped position on the initial probe into the dcache (the

inhibit list is accessed with the instruction is decoded, for both the PC and XOR schemes). Furthermore, if an inhibited instruction encounters a cache block in a set-associative position, the cache block will be evicted from the cache. When the evicted block is brought back in from L2, it will be placed in the direct-mapped position. Inhibit information spreads through the BWT to all other instructions that share cache blocks with an inhibited instruction, and then to the BWT entries of all the other cache blocks that those instructions act on. This plague-like algorithm eliminates repeated cache block evictions when a cache block is shared by inhibited and uninhibited instructions. Since the feedback mechanism will often favor direct-mapped line contention over repeated probe0 misses, it trades-off overall hit rate for probe0 hit rate. This is the main reason that a r-a cache will have a higher overall miss rate, in general, than the prior multi-probe schemes. However, feedback, along with selective displacement, will allow a reactive-associative cache to have a low bandwidth utilization, and to perform well (compared to a direct-mapped cache and prior schemes) regardless of the application.

2.3.3 Uninhibiting to accommodate dynamic program behavior

We found that the predictability of instructions and cache block addresses varies in different phases of program execution. For example, in the initialization phase of a program, or in the first few iterations of a large loop, instructions may be labeled as unpredictable, when in actuality, they become predictable at some later time.

To optimize for the dynamic nature of predictability, it is necessary to occasionally clear the inhibit list, as well as the misprediction counters in the BWT. Several methods were explored for doing this, but it turns out that simpler is better. One method, called periodic clearing, will clear the feedback information at regular intervals of (measured in number of cache accesses). For example, the inhibit list and misprediction counters can be cleared every 100,000 cache accesses. Although a generic interval (50,000-200,000 accesses) can be used successfully for most benchmarks, the precise interval to maximize performance under periodic clearing is application dependant.

Another possibility is to use a scheme we refer to as TLB clearing. This scheme is motivated by the observation that data and instruction TLB misses correspond to entrance

into new program phases. On a data TLB miss, we clear the inhibit list (which is indexed by instruction PC), with the hope that the new data will be more predictable by currently inhibited instructions than the old data. On an instruction TLB miss, we are entering a new code phase of the program, so we anticipate that cache blocks that are currently unpredictable (indicated by having saturated misprediction counters in their respective BWT entries) may be more predictable by the new instructions. Therefore, on an instruction TLB miss, we clear the BWT misprediction counters. After clearing the inhibit list and/or the BWT misprediction counters, instructions are free to access displaced cache blocks.

Although periodic clearing will perform better than TLB clearing if the interval is pre-selected per application, we recommend the TLB scheme, as it is more robust and does not require per-application tweaking. We tried more complex dynamic schemes, but none of these would out-perform periodic or TLB clearing.

3. QUALITATIVE COMPARISON

There have been several prior proposals for low-latency implementations of associative caches. Since the lowest-latency cache organization is direct-mapped, most of the research has surrounded performing successive probes into a direct-mapped type cache. These multi-probe caches are classified by more parameters than a traditional set-associative cache, and henceforth require further examination. In this section, we explore the more popular multi-probe cache schemes and qualitatively compare them with the organization we propose here, the reactive-associativity cache. In Section 3.1, we examine the issues that are common to all multi-probe caches. In Section 3.2 and Section 3.3, we discuss statically and dynamically probed caches, respectively. In Section 3.3.2, we revisit the reactive-associative cache in the context of the prior multi-probe cache proposals.

3.1 Issues Common to Multi-Probe Caches

As discussed in Section 1.1, the primary reason that a set-associative cache has a higher hit time than a direct-mapped cache is the late way multiplexing on the data path. To avoid this delay, it is necessary to use a direct-mapped type data path, in which there is no way muxing at the output. Associativity can still be supported, as mentioned earlier, by performing multiple probes into the data array, with each using a different indexing function. All of the multi-probe caches that we examine use what we refer to as a direct-mapped data path. Though the data path must be direct-mapped, our analysis indicates that the tag path may be organized like that of a set-associative cache, without impacting hit latency. Therefore, the organization of the tag side is a qualitative parameter for multi-probe caches.

Since only one line can be accessed at a time, and it is desirable to only do one probe in the common case, the initial probe location is another major issue facing multi-

probe caches. We pay particular attention to the distinction between a statically-probed and dynamically-probed organization, as this property guides many of the other characteristics of multi-probe caches. Statically-probed organizations rely on a fixed initial probe location, while dynamically-probed caches use way-prediction to access any cache line on the initial probe. The maximum number of additional probes required, the probe sequence, and the actual mapping functions for subsequent probes are other parameters for multi-probe caches.

3.1.1 Complications due to pipeline timing

All multi-probe caches can present problems for processor pipelines that have previously not had to contend with the notion of a variable hit latency. Even though modern processors are dynamically scheduled, it is still common to use fixed latencies to reserve data buses, to vacate items from the load/store queue, etc. If it is common for the L1 dcache to perform many second probes, it is possible that pipeline scheduling could become difficult and wasteful. Although these potential problems are implementation specific, they are an important considerations in guiding the research in this area. Clearly, it would be better from a scheduling standpoint if additional probes were unnecessary or uncommon. Furthermore, it would be beneficial if the processor knew by the end of the initial probe the number of additional cycles it would take to service a cache hit (if there was a probe0 miss), or if it will be an overall cache miss. This is a primary motivation for using a set-associative tag path for a multi-probe cache, as used in reactive-associative.

Also, the more deterministic the delay through the dcache, the easier it will be to pipeline the structure, as many modern L1 dcaches are pipelined. Wave-pipelined caches, like those that appear in the Alpha 21264, would be difficult to make into multi-probe caches (especially those than have high probe0 miss rates), as they have limited ability to support cache pipeline stalls.

3.1.2 Performance metrics

Architects are used to discussing the performance of caches using terms like hit latency and miss rate. However, the performance metrics for multi-probe organizations are

more complex than for traditional caches. Clearly, average memory access time is an appropriate measure of performance of any memory subsystem, but this is difficult to quantify in an out-of-order superscalar processor, and is too blunt to use for comparison purposes. We discuss the performance of multi-probe caches using: probe0 hit latency, probe0 miss rate, additional probe hit latency, overall miss rate, and bandwidth utilization. Bandwidth utilization is usually discussed with respect to the bandwidth demand of a direct-mapped cache, and is a qualitative measure of cache throughput. Modern processors are remarkably adept at overlapping and hiding latencies, but are fundamentally limited by bandwidth, of the processor as a whole and of the memory system in particular. Therefore, we pay close attention to the bandwidth utilization of each cache organization.

3.2 Statically-Probed Caches

Statically-probed caches are multi-probe organizations that have a fixed initial probe into the data array for a given cache block address. If the initial probe misses, the cache may make one or more additional probes into the data array according to a set algorithm. Usually the fixed initial location is to the direct-mapped position for a cache block, so we use the terms direct-mapped location, probe0 location, and initial probe location interchangeably for statically-probed caches. Clearly, these caches can never achieve a probe0 miss rate lower than that of a direct-mapped cache (of the same size). In actuality, the probe0 miss rate is considerably higher than the direct-mapped miss rate, unless aggressive techniques are used to guarantee that most accesses are to the direct-mapped location. This is insured by using cache block swapping, wherein a cache block found on a subsequent probe is swapped with the cache block in the direct-mapped location. Cache block swapping relies on block-level locality of reference, and allows for initial-probe miss rates only modestly higher than direct-mapped miss rates. Unfortunately, dedicated circuitry to support cache block swapping is impractical for caches, which use regular and highly optimized layouts. A modest implementation of cache block swapping would require two reads and two writes (using an intermediate cache block buffer), which increases average hit time and increases the bandwidth utilization. Furthermore, frequent cache block swapping exacerbates the scheduling problems discussed in Section 3.1.1. All

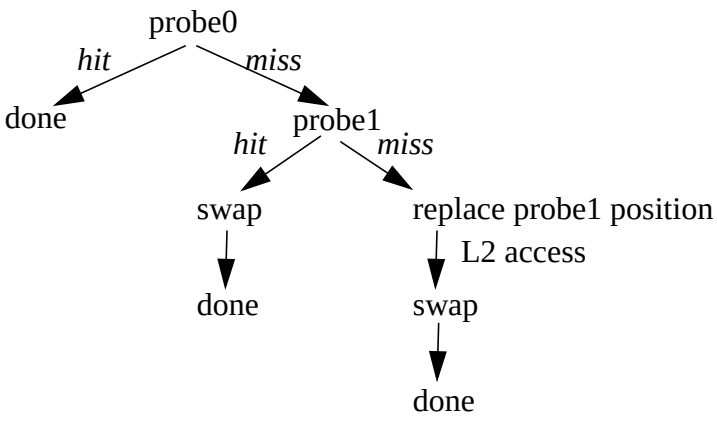

Fig. 3.1. Decision tree for the hash-rehash algorithm

of the statically probed caches use cache block swapping, which is a fundamental problem that will undoubtedly prohibit any statically-probed cache scheme from gaining industry acceptance. However, there have been many interesting architectural innovations developed in the context of statically-probed caches, so it is still beneficial to examine them.

3.2.1 Hash-rehash cache

The hash-rehash cache was initially proposed in [4], as a two-way associative organization. It is indexed like a direct-mapped cache for the initial probe, but if the block is not found, a second probe is performed using a different hash function. If the block is found on the second probe, a swap is performed so that the next access to that address will find the cache block in the direct-mapped location. If the block is not found in the alternate position (which would be an overall cache miss), then the alternate location is used as the replacement position when the cache block is retrieved from L2. Once the cache block is placed in the probe1 position, a swap is performed between the probe1 and the probe0 lines, so that the next access to this block will find it in the direct-mapped location. Figure 3.1 shows the decision tree for the hash-rehash algorithm. Probe0 is to the direct-mapped location, while probe1 is to the alternate location for a given cache block address. In Figure 3.1, the term *swap* indicates that a cache block swap of the probe0 and probe1 locations is performed. The hash-rehash cache will always probe both locations, even if the block is not in the cache.

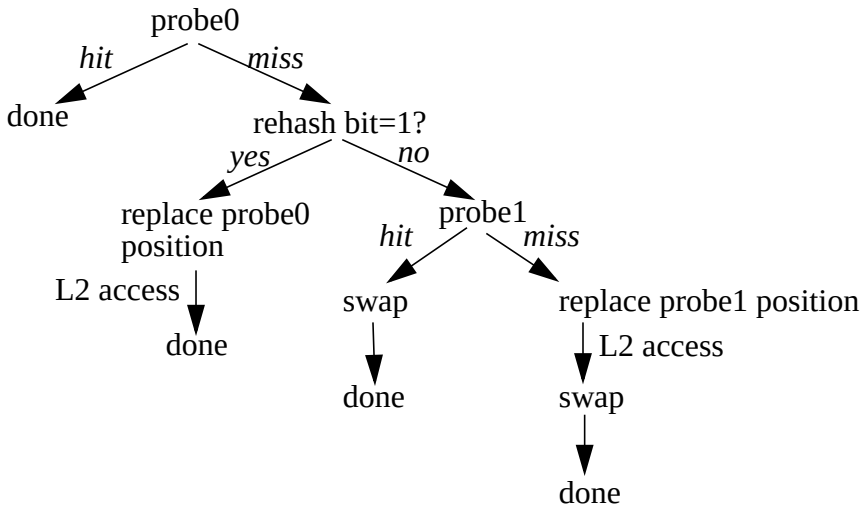

Fig. 3.2. Decision tree for the column-associative algorithm

3.2.2 Column-associative cache

The column-associative cache [5] augmented the hash-rehash cache with an improved decision tree (Figure 3.2). The goal of the column-associative cache (c-a cache) is to reduce the number of second probes that are performed. This is accomplished by affixing a *rehash bit* to each cache line. This bit indicates whether or not the cache line contains an out of position block (i.e., the cache block is accessed through the probe1 indexing algorithm). On a direct-mapped access to a given line, if the line's rehash bit is high, then it is not possible that an additional probe will find the cache block. It is not possible specifically because the rehash bit indicates that the alternate (or probe1) position contains direct-mapped data. This rehash bit eliminates many of the superfluous second probes that existed in the hash-rehash cache. Often, though, it still takes as many probes as the associativity of the cache (in this case, two) to determine that the block is not in the cache at all, which delays initiation of an L2 access until all the probes complete. The replacement algorithm is also improved over that of the hash-rehash cache, as the rehash bit indicates whether it is more prudent to replace the block in the probe0 or probe1 location (column-associative approximates an LRU replacement policy). Figure 3.2 illustrates the decision tree for the c-a cache, using the same action semantics as in Figure 3.1.

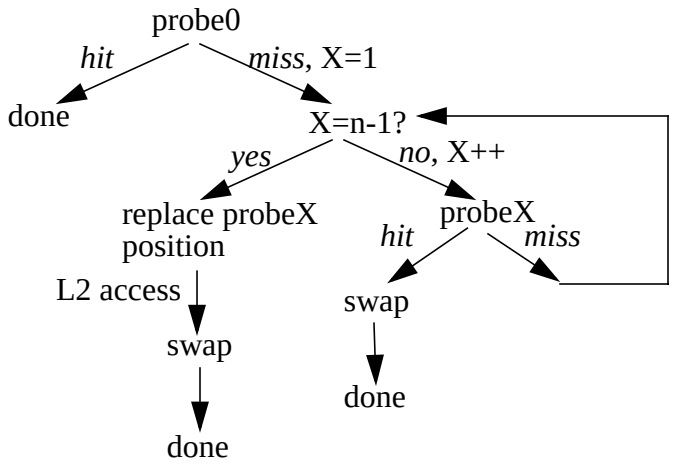

Fig. 3.3. Decision tree for a n-way sequential multicolumn cache (SMC)

3.2.3 Sequential multicolumn cache

The sequential multicolumn cache (SMC) was introduced in [6], and basically augmented and updated the earlier MRU cache schemes [12] [13]. We do not examine the MRU cache implementations, as they are not attempts at a low-latency cache, since both versions of the MRU cache ([12] [13]) are technically incapable of approaching a direct-mapped cache in terms of hit latency. The initial idea with the MRU cache was to track the most recently used ways in a set, so as to optimize the sequential search path through a multi-probe cache. The sequential multicolumn cache extended this idea with what the authors refer to as the *multiple MRU block technique*. The multiple MRU block technique basically forces the initial probe to the direct-mapped location for a given cache block address (as in hash-rehash and column-associative), as opposed to a MRU location per set (as in earlier implementations of the MRU cache). Subsequent probes are performed according to MRU information that is tracked separately for each cache line. Cache block swapping is used to guarantee that the most recently accessed block is always in its major location. Figure 3.3 illustrates the decision tree for a sequential multicolumn cache. Once again, probe0 is the direct-mapped location, and probes 1 through n-1 (for a n-way associative cache) are ordered according to the MRU information that is recorded in a separate data structure. The MRU search algorithm allows SMC to generalize multi-probe caches

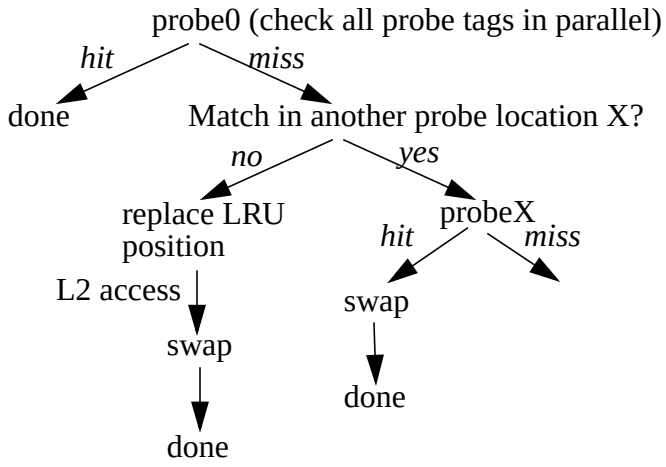

Fig. 3.4. Decision tree for a n-way parallel multicolumn cache (PMC)

for n-way associativity, in a relatively efficient manner. However, SMC still requires cache block swapping, and is therefore subject to the same bandwidth problems as column-associative (and all statically probed caches).

3.2.4 Parallel multicolumn cache

The parallel multicolumn cache [6] introduced the idea of a decoupled tag and data organization, with an associative tag side and a direct-mapped data side. The initial probe is to the direct-mapped location, and the subsequent probe is determined by tag match info from the associative tag side. This is similar to the organization that we use in reactive-associative, to ease pipeline timing, and to allow for at most one additional probe on a probe0 miss. Cache block swapping is used to force the most recently used cache block into the direct-mapped location, and therefore to improve the probe0 miss rate. Figure 3.4 shows the access algorithm for the parallel multicolumn cache, using the same semantics as in the prior cache schemes. PMC contributes the concept of an associative tag side and a direct-mapped data side to the research of cache design, but the requirement for swapping limits its performance, and will undoubtedly limit its acceptance by industry. The sequential and parallel multicolumn caches are the first to make a legitimate attempt to provide a low-latency cache with levels of associativity higher than 2-way.

3.2.5 Group-associative cache

The newly proposed group-associative cache [7] also requires swapping of entire cache blocks. However, group-associative (g-a cache) did introduce some interesting concepts to the research community. The g-a cache maintains a list of the most recently accessed cache blocks, which it uses to control associative displacement. The g-a cache reasons that if a cache block has been recently accessed, then it would be a desirable to keep it nearby. Therefore, if a cache block from L2 is being placed into a line in L1 that currently contains a recently accessed block, then the recently accessed block is moved to an alternate location in the cache. Cache blocks that have not been recently accessed are replaced as in a direct-mapped cache. This algorithm implements a limited notion of selective displacement.

As mentioned earlier, recently accessed blocks are displaced into alternate locations in the cache. The alternate location is selected by an algorithm which detects under utilized cache lines, of which there are many in a direct-mapped cache. This concept of detecting unused cache lines, or holes, is an important contribution to multi-probe cache research. When a cache block is placed in an alternative (or out of position) cache line, its tag information must be recorded in a special data structure referred to as the OUT directory. The set of accesses that map to the same direct-mapped location, but are displaced as indicated in the OUT directory, are referred to as an associative group, and hence the name group-associative.

On a cache access, the initial probe is to the direct-mapped location. While the direct-mapped probe is being performed, the OUT directory is checked for a matching tag (which would indicate that the block is out of position). The OUT directory must be an associative structure (or else conflicts from the dcache would be amplified there), and in [7] it is recommended that it be fully-associative. Furthermore, since the OUT directory must be checked on every access (to insure correctness), and the previous lookup must complete before the next cache access, its access time must be smaller than that of the dcache or the OUT directory will determine the hit time of the cache. However, our analysis using CACTI [2] indicates that a fully-associative cache of the size recommended by the group associative paper (64 entries for a 256 line dcache) is 48% slower than the

direct-mapped cache itself. Even if the OUT directory is only 2-way set associative, the hit time of a reactive-associative cache would be at least 21% higher than that of a direct-mapped cache, and undoubtedly the overall miss rate would much worse than if the OUT directory were fully-associative. Therefore, the group-associative cache is fundamentally prohibited from having a hit latency comparable to that of a direct-mapped cache. This combined with the requirement for cache block swapping makes the group-associative cache impractical for implementation, and we don't consider it a viable multi-probe organization.

3.3 Dynamically-Probed Caches

The term *dynamically-probed* indicates that the initial probe into the cache is not to a fixed location, but rather to any line in the cache. This is accomplished with the use of a way-predictor (which, intuitively, predicts the way of the data array index, as discussed in Section 2.2). With dynamically-probed caches, we can no longer use the terms probe0 location and direct-mapped location interchangeably. When discussing dynamically-probed organizations, the direct-mapped location refers to the direct-mapped way, and the probe0 location is the predicted way. A major motivation for using a dynamically-probed arrangement would be the ability to access displaced cache blocks on the initial cache probe, and therefore reduce probe0 miss rates without cache block swapping. The only prior attempt at a dynamically-probed (low-latency) dcache is the predictive sequential associative cache, which we discuss in Section 3.3.1. The reactive associative cache, introduced in this thesis, is also a dynamically-probed arrangement. We discussed the r-a cache organization thoroughly in Section 2, and we will contrast it with the prior cache schemes in Section 3.3.2.

3.3.1 Predictive sequential-associative cache

To avoid cache block swapping, the predictive sequential associative cache [8] proposed way prediction to access any way, as opposed to only the direct-mapped way, on the initial probe into the cache. The way prediction is accomplished via a data structure (called the steering bit table (SBT)) that is indexed with a prediction handle available ear-

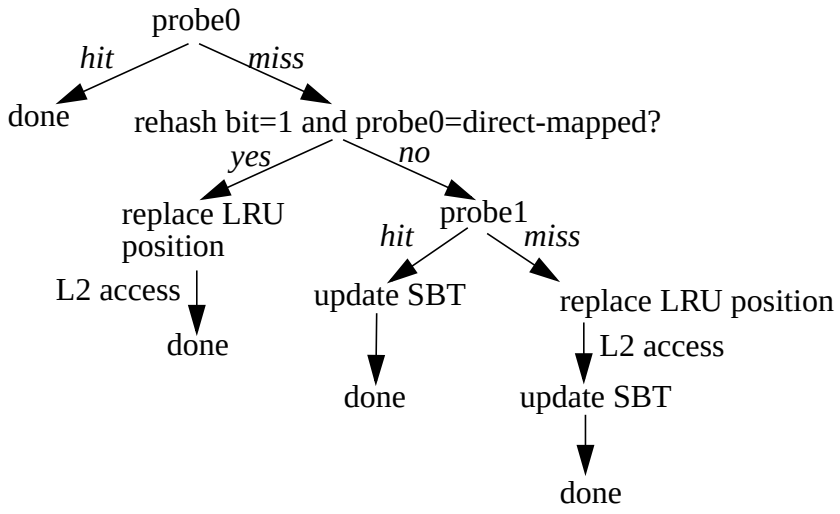

Fig. 3.5. Decision tree for a 2-way predictive sequential associative cache (PSA)

lier than the cache block address itself. As with reactive associativity, there are many possible prediction handles, and several were explored in [8]. The PSA paper suggests the XOR scheme because of its high accuracy. As mentioned in Section 2.2.3, the XOR scheme requires an XOR operation on a value, often obtained from a register-forwarding path, followed by a lookup of a table with many entries (1024 way-prediction entries for a 256-block cache), all *within* the time to perform a full add. In many pipelines this scheme is likely to increase cache access initiation time [14]. Furthermore, if the XOR scheme was indeed feasible, than it would certainly be possible to simply access a direct-mapped cache a cycle earlier with the XOR approximation of the memory address, which would be a so-called *zero-cycle load*, as proposed in [10]. The XOR scheme is an accurate predictor, but even with XOR-based prediction, PSA's way prediction accuracy is low. This is because all accesses are predicted, even accesses that would hit in a direct-mapped cache. The low way prediction accuracy yields poor probe0 miss rates. PSA does achieve low overall miss rates; but, because there are many second probes, average hit latency and L1 dcache bandwidth utilization are increased. Under realistic bandwidth limitations, this increase often offsets the low overall miss rate and leads to degraded system performance. PSA uses a direct-mapped tag-side, and therefore must perform additional probes into the cache until the block is found or it is determined that the block is not in the cache. Since

the probe sequence is not optimized, as in SMC (Section 3.2.3), this limits the scalability of PSA to higher levels of associativity.

3.3.2 Reactive-associative cache

Table 3.1 qualitatively compares the prior proposals with the reactive-associative cache. Like the PSA cache (Section 3.3.1), the reactive associative cache uses way prediction to allow for a dynamically determined probe0 location. However, unlike PSA, the r-a cache uses a set-associative tag side (similar to the parallel multicolumn cache discussed in Section 3.2.4), which simplifies pipeline timing and allows r-a to scale to higher levels of associativity.

Way prediction accuracy of the reactive-associative cache is much higher than PSA due to the selective displacement and feedback mechanisms (Section 2.3). With higher way prediction, the r-a cache is able to have a low probe0 miss rate, and low bandwidth utilization. Selective displacement refers to the idea of displacing only those cache blocks for which there will most likely be a clear benefit to having cached. Group-associative (Section 3.2.5) had a limited concept of selective displacement (by only displacing recently used blocks), but reactive-associative goes further by seeking to displace only those cache blocks which cause direct-mapped line contention. Feedback refers to the concept of measuring the performance of the cache, and restricting displacement when necessary. Reactive associative is the only cache organization that supports a form of feedback, in which the prediction accuracy is measured and associative displacement is inhibited on a per-instruction basis. Because the way-prediction accuracy of the r-a cache is high, it is able to use more realistic prediction handles (like instruction PC) than PSA, which only performs well under the XOR scheme. As mentioned earlier, if the XOR scheme was practical to implement, then zero-cycle loads [10] could also be implemented. Interestingly, the r-a cache using PC prediction could also use zero-cycle loads (could issue cache accesses early). However, to avoid confusing the advantages of the r-a cache and zero-cycle loads, we do not perform zero-cycle load optimizations in this thesis.

Like PSA, the r-a cache does not require cache block swapping. The statically-probed implementations do require cache block swapping, however, which will cause all

of them to have increased bandwidth demand, and will, in general, limit their performance gains. Also, cache block swapping is undesirable to implement from a circuit perspective.

Table 3.1
Qualitative comparison of previous schemes and the r-a cache.

	static probe0 to d-m location?	tag array/ data array	probe0 miss rate w.r.t. d-m miss rate/ reason	overall miss rate/ probe0-hit latency	L1 band-width demand w.r.t. d-m / reason	controlled or restricted displacement?
column-associative	yes, needs swap	d-m/ d-m	higher/ static first probe	~ 2-way/ ~ d-m	higher/ swapping	no
group-associative	yes, needs swap	d-m/ d-m	higher/ static first probe	fully-a/ fully-a	higher/ swapping	yes, selective displacement
sequential multicolumn	yes, needs swap	d-m/ d-m	higher/ static first probe	n-way/ ~ d-m	higher/ swapping	no
parallel multicolumn	yes, needs swap	s-a/ d-m	higher/ static first probe	n-way/ ~ d-m	higher/ swapping	no
predictive sequential associative	no, uses way-prediction	d-m/ d-m	higher/ high first-probe miss rate	~ 2-way/ ~ d-m	higher/ high first-probe miss rate	no, and also predicts all accesses
reactive-associative (PC scheme)	no, uses way-prediction	s-a/ d-m	lower or near/ low first probe miss rate	> 2-way/ ~ d-m	comparable/ low first probe miss rate	yes, selective displacement & feedback

Unfortunately, the low probe0 miss rate and low bandwidth utilization of a r-a cache come at a cost. The reactive associative cache trades-off overall hit rate for a lower probe0 miss rate. However, for realistic pipelines, it is better to maximize bandwidth through the L1 dcache than it is to minimize the overall miss rate of the dcache. In Section 4, we will show execution time results (as well as miss rates) that bear out this fact.

3.4 Related Work

There have been several ideas to use program characteristics to improve memory system performance. Farrens and Tyson [16] detect loads that miss often and mark such loads to bypass the cache. Farrens and Tyson do not track data usage behavior or alter the organization of the cache to enable better management. Johnson and Hwu [17] use cache

block reuse behavior based on data addresses to guide placement of data in a victim cache as opposed to the cache itself. Johnson and Hwu do not use instructions to guide data placement but focus only on data addresses. Informing memory operations [18] is a technique to provide notification of a load miss and to inform applications about cache misses. Application-level restructuring of code to improve cache performance has been explored in [1]. Compiler optimizations to reduce conflicts have been explored in [19] [20].

4. QUANTITATIVE ANALYSIS

4.1 Hit time of the reactive-associative cache

The r-a cache uses a unique organization utilizing a set-associative tag side and a direct-mapped data side. The r-a cache also requires some additional logic to support the ability to perform both direct-mapped accesses and set-associative accesses (with the aid of a way-prediction). We argue that this additional logic, as well as the inclusion of a set-associative tag side, will not significantly increase the hit time of a r-a cache over that of a direct-mapped cache (see Section 2.1.4). To justify this assertion, we use the analytical cache model CACTI [2], configured for 0.18 micron technology. It is desirable to separately analyze the speed of the tag side and the data side of the cache, both measured from the time when the cache block address is available. We quantify the delay of the tag side as being the total time it takes to generate the hit/miss signal. The delay of the data side is considered to be the time to produce a cache block on the output bus. The hit latency of the cache is the longer of the tag side delay and the data side delay. For further discussion on the components of delay for direct-mapped and associative caches, please refer to Section 1.1.

In Table 4.1, we present the tag side (not including the OR gate in the hit signal path in set-associative caches) and data side latencies for direct-mapped through 8-way conventional set-associative caches. Probe0 timings for a 4-way r-a cache can be derived from the direct-mapped data side and the conventional 4-way set-associative tag array timings. We added a 8-ps delay, obtained by Hspice simulations, for each of the pass gate multiplexors in the probe0 hit signal path and the data array index path, as discussed in Section 2.1.4. 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. As

mentioned earlier, the total hit time is the larger 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 comparable in speed to the direct-mapped data array. Thus, a 4-way r-a probe0 critical path (for both data and tag) is similar to the critical path of a direct-mapped cache. For an 8 KB cache, the 4-way r-a cache is about 4.9% slower than direct-mapped, but still 47% faster than a 2-way cache, and 61% faster than a 4-way set-associative cache. For cache sizes greater than 8 KB, however, a 4-way associative tag array is no longer any slower than a direct-mapped tag array, and a 4-way r-a cache is less than 0.8% slower than a direct-mapped cache. For sizes of 32 KB and greater, an 8-way r-a cache is less than 0.7% slower than a direct-mapped cache. This implies that the hit latency of a r-a cache maintains direct-mapped equivalence with increasing associativity. We examine an 8-KB cache only because using a larger cache results in negligible miss rates for the SPEC95 benchmarks, but most modern microprocessors use L1 dcaches of size 16 KB or larger, so replacing a direct-mapped cache with a r-a cache should not affect the clock rate.

Table 4.1
Cache hit times (ns).

	Direct-mapped conventional	2-way conventional	4-way conventional	8-way conventional	4-way r-a
8KB Data out	0.816	1.252	1.374	1.642	0.824
8KB hit signal	0.798	0.796	0.844	0.962	0.852
8KB total latency	0.816	1.252	1.374	1.642	0.852
Increase over DM	0%	53.5%	68.4%	101.2%	4.5%
16KB Data out	0.992	1.365	1.466	1.723	1.000
16KB hit signal	0.874	0.880	0.902	1.006	0.910
16KB total latency	0.992	1.365	1.466	1.723	1.000
Increase over DM	0%	37.6%	47.8%	73.7%	0.8%

4.2 Methodology

We modified the Simplescalar3.0 simulator [15] to model the L1 D-cache as a r-a cache. Table 4.2 shows the base 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 the SPEC95 benchmarks. 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 4.1. 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). We assume 12 cycles for L2 cache hits to model the wire delays, and not necessarily the L2 hit time, involved in accessing an on-chip L2 cache which is usually far from the L1 cache (e.g., Alpha 21264).

Table 4.2
Hardware parameters for base system.

Component	**Description**
Processor	8-way out-of-order issue, 64-entry reorder buffer, 32-entry load/store queue
Branch prediction	combines a bimodal predictor using 4096 entries and gshare using a 10-bit history
L1 I-cache	16 Kbytes, 2-way associative, 32 byte blocks, 1 cycle hit, lock-up free
L1 D-cache	8 Kbytes, 32 byte blocks, 1 cycle probe0 hit, 3 cycles probe1 hit, lock-up free
L2 cache	256 Kbytes, 8-way associative, 64 byte blocks, 12 cycle hit, pipelined
Memory Bus	Split transaction, 32 bytes per bus cycle transfer
Main memory	Infinite capacity, 60 cycle latency
Way-prediction resources	**Reactive-associative, PC**:128-entry APT & BWT each with compressed tags, 2048-bit inhibit list (in I-cache), and 256-entry victim list; inhibit threshold: 3 and victim threshold: 5; (total 1184 bytes). **Reactive-associative, XOR**: 1024-entry BWT, 2048-bit inhibit list (in I-cache), and 256-entry victim list; inhibit threshold: 3 and victim threshold: 2.

For our experiments, we choose those benchmark/input combinations from the SPEC95 suite that do not require prohibitively long simulation runs. Table 4.3 presents the SPEC95 benchmarks and their inputs used in this study. Apart from the SPEC95 programs, we also use troff as one of our benchmarks. The benchmarks were compiled for a Compaq Alpha AXP-21164 using the Compaq C and Fortran compilers under -O4 -ifo

optimization flags. Most of the simulations are run to completion, but in the cases where the runs are inordinately long, we halt the simulation at 1 billion instructions.

Table 4.3
Benchmarks and inputs.

Benchmark first group	**Input**	**#instructions simulated**	**Benchmark second group**	**Input**	**#instructions simulated**
vortex	ref	1 billion	go	9stone21.in	1 billion
gcc	1recog	347 million	troff	paper.me	70 million
li	train	365 million	m88ksim	train	171 million
perl	jumble	1 billion	swim	train	430 million
			fpppp	train	235 million

4.3 Base performance of the reactive-associative cache

In this section, we present the base 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 hit cache as a reference point. Figure 4.1 shows the speedups of processors using various cache configurations normalized to the performance of a system with a direct-mapped cache. 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.

From Figure 4.1 we can see that 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%-9% with the ideal, 2-way cache. *Swim* has a pathological mapping problem which causes the 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. The r-a cache dynamically adjusts to the associativity requirements of the application, providing the first group with low degrees of associativity, and the second group with high degrees of associativity—but at

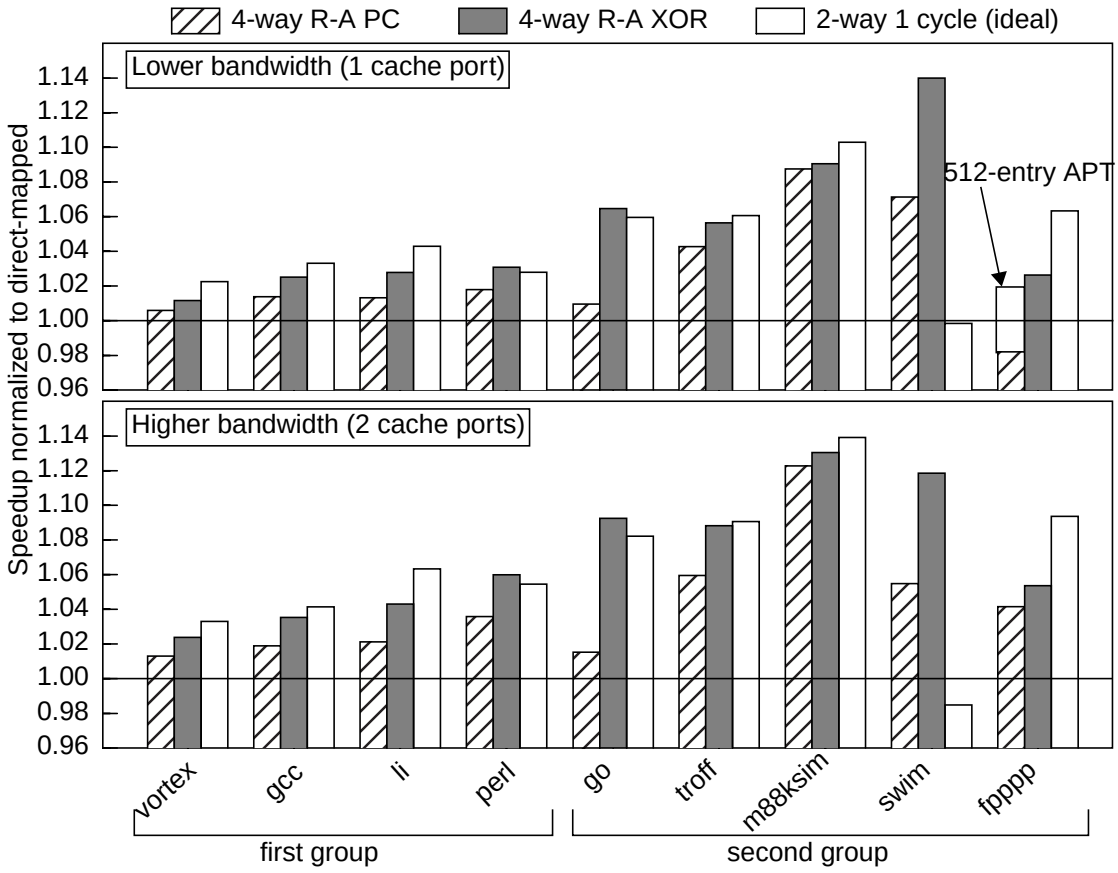

Fig. 4.1. Base performance of the reactive-associative cache.

direct-mapped speeds. For the second group, the 1-port r-a achieves 1%-9% speedups using PC, and 2%-14% speedups using XOR, approaching the ideal, 2-way cache for many benchmarks. The 2-port case follows similar trends.

The r-a cache's only slow-down is for *fpppp* under the PC scheme using 1 port due to *fpppp's* poor instruction locality, disrupting the PC-indexed APT, which causes high probe0 miss rates and wastes L1 cache bandwidth. In the 2-port case, this bandwidth pressure is absorbed by the extra port, resulting in a 4% speedup. As indicated in Figure 4.1, the slowdown for the 1-port case turns into a 2% speedup when the APT size is increased to 512 entries. For both the 1-port and the 2-port cases, the performance of the PC scheme approaches that of the XOR scheme, even though the XOR predictor is potentially more difficult to implement.

In Table 4.4, we compare the miss rates for the r-a cache with those of direct-mapped and 2-way caches. The probe0 miss rates of both the PC and XOR r-a caches are less than or comparable to the miss rates of a direct-mapped cache for all benchmarks except for *vortex* and *fpppp*, indicating the success of our way-prediction scheme. *Vortex's* probe0 miss rate decreases to 5.3% when the inhibit list is cleared less frequently, indicating that vortex has many unpredictable instructions. *Fpppp's* large instruction footprint causes thrashing in the APT, resulting in a high probe0 miss rate when using the PC scheme, even though the overall miss rate approaches that of 2-way. The final row of Table 4.4 contains the geometric means for each column.

Table 4.4
Reactive-associative miss rates compared to set associative caches

	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	pred. accuracy	probe0 miss rate	overall miss rate	pred. 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

The prediction accuracy columns for PC and XOR schemes show the performance of the way predictor. Note that unpredictable instructions are tracked and blocks that they touch are prohibited from being displaced, which enables high prediction accuracies for those blocks that remain. This policy achieves high accuracy, which conserves valuable L1 bandwidth, but at the cost of higher overall miss rate compared to a 2-way set-associative cache because unpredictable blocks are not displaced to set-associative positions even if they conflict. 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

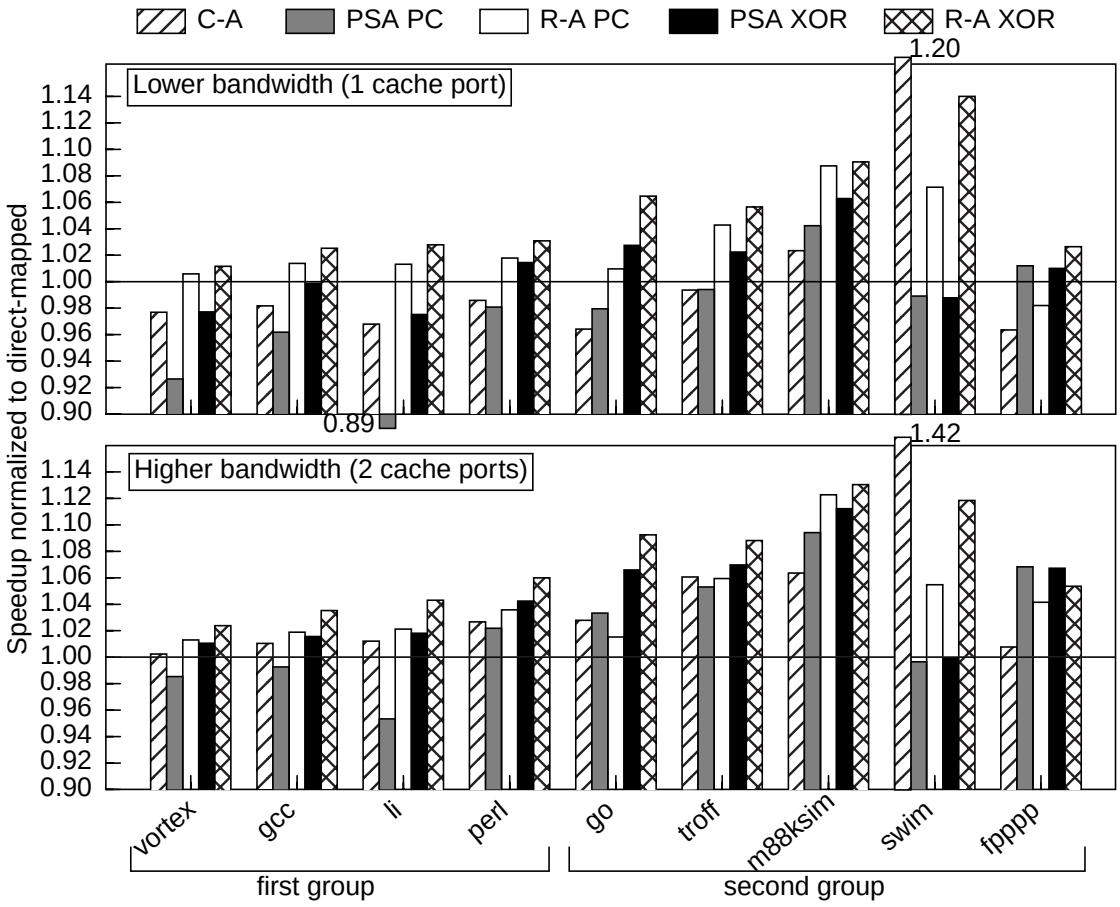

Fig. 4.2. Comparison of reactive associativity to previous schemes

relaxation results in lower probe0 and overall miss rates for XOR However, in some cases this relaxation of parameters causes the XOR scheme to actually have worse way-prediction accuracy than with the PC scheme.

4.4 Comparison between the reactive-associative and 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. We do not compare against group-associative because of the fundamental circuit problems that we outlined in Chapter 3, which prohibit a group-associative cache from having a direct-mapped hit latency, and also because it requires 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 [8]), with 1 hash-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 authors of the PSA paper recommend the XOR scheme but do not recommend using the PC scheme. We include the PC scheme here for comparison purposes only. In Figure 4.2, we compare speedups (over a system with a direct-mapped cache) of the 4-way r-a cache against those of the column-associative and PSA caches.

For the 1-port (lower bandwidth) case in Figure 4.2 (the upper graph), PSA using PC and XOR schemes and column-associative (c-a) suffer slow-downs on many benchmarks due to their high bandwidth demand, 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. The r-a cache using PC or XOR performs better than the corresponding PSA, and better than c-a, because r-a has a lower bandwidth demand. Column associative has large speedups for *swim* because of *swim*'s pathological set-associative mapping problem. Since column-associative uses a skewing hash function (i.e. the complement of the index bits), it alleviates this problem and achieves massive speedups. Similarly, if the APT and BWT sizes are increased significantly, the 4-way r-a cache does achieve similar speedups for *swim*.

For the 2-port (higher bandwidth) case in Figure 4.2 (the lower graph), column associative and the predictive sequential caches achieve somewhat higher speedups because the extra port absorbs the bandwidth pressure, though there are still a few cases of slow-down. The r-a caches using PC and XOR continue to outperform 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 2-port case. The major exception is *fpppp*, whose large instruction footprint wreaks havoc with the PC scheme for an r-a cache.

Table 4.5
Initial probe (p0) and overall (ov) miss rates for various cache 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
Bench	ov	p0	p0	p0	p0	p0	ov	ov	ov	ov	ov
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 4.5 shows the first probe (indicated by p0) and overall (indicated by ov) miss rates for the 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 the 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 (due to selective displacement and feedback), the r-a cache performs better than PSA or c-a, which validates the concept of balancing overall miss rate and probe0 miss rate for optimal performance in the face of bandwidth limitations.

4.5 Effect of filtering and feedback on probe0 and overall miss rates

To isolate the effects of selective displacement and feedback on the r-a cache, in Table 4.6 we present probe0 and overall miss rates using the PC scheme without selective displacement (i.e., no victim list), and without feedback (i.e., no mispredict counters in the BWT, and no inhibit list). In the case of no selective displacement, probe0 miss rate suffers 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 with selective displacement, but the overall miss rate approaches those achieved by PSA or c-a because the cache displaces all cache blocks regardless of predictability. These increased probe0 miss rates would, in general, cause an increased average cache hit time and a higher bandwidth utilization. Depending on the L1 cache bandwidth of a particular system, the designer may choose to be more or less strict about which blocks to displace. The degree of tolerance to unpredictable instructions can be adjusted by changing the inhibit threshold—which makes the r-a cache easily adapted to maximize performance across systems with varying memory bandwidths. Furthermore, if the prediction handle being used is naturally more accurate than instruction PC, the architect may choose to lower the victim list threshold (which we set to 5 for PC, and 2 for XOR), and therefore encourage more displacement.

Table 4.6
Effect of filtering and feedback on probe0 and overall miss rates

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

5. CONCLUSIONS

In this thesis, we propose the reactive-associative cache (r-a cache) based on the key observation that even for applications that benefit from increased associativity, the common case for a direct-mapped cache is a hit. This observation implies 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 keeps as much data in direct-mapped positions as possible, and displaces only conflicting blocks to set-associative positions. The r-a cache therefore provides flexible associativity that increases or decreases depending on application characteristics.

The reactive-associative cache is a dynamically-probed multi-probe organization which has a direct-mapped hit latency for the initial probe. For a given cache block address, the initial probe can be to the direct-mapped location or to a set-associative position, with the use of a way-prediction. Therefore, all direct-mapped and correctly predicted set-associative accesses can complete in the initial probe. Incorrect way-predictions will require a single additional probe into the data array. The ability to service mispredictions in a single extra probe (even for high associativities) is accomplished through the use of a unique organization in which a set-associative tag side is joined with a direct-mapped data side. This arrangement allows all the tags in a set to be checked in parallel with the initial probe in the data side. If the initial probe prediction was incorrect, the correct way is known from the tag match information acquired during the first probe. We have shown that using a set-associative tag side does not significantly increase the hit latency over that of a direct-mapped cache.

Prior statically-probed multi-probe caches (such as hash-rehash, column associative, and group-associative) have required swapping of entire cache blocks to improve their initial-probe miss rates. But cache block swapping degrades both latency and band-

width. The previously-proposed predictive sequential associative cache avoids swapping via way prediction, but incurs initial-probe miss rates much worse than direct-mapped miss rates. This is because it displaces, and subsequently predicts, all cache blocks, which places a strain on the prediction resources. The poor first-probe miss rates result in the depletion of valuable L1 bandwidth. In contrast, the r-a cache selectively displaces, and consequently predicts, only conflicting blocks. In addition, the r-a cache employs a feed-back mechanism on way prediction to prevent unpredictable blocks from being displaced, achieving initial-probe miss rates often lower than (or at least comparable to) direct-mapped miss rates and conserving L1 bandwidth.

The r-a cache delivers performance robustness by trading-off overall hit rate for first-probe hit rate, achieving initial-probe miss rates often lower than (or at least comparable to) direct-mapped miss rates, and conserving L1 bandwidth. Simulations using some of the SPEC95 benchmarks show that a 4-way associative r-a cache, using 1184 bytes of prediction storage, achieves 1%-11% and 1%-7% improvements over a direct-mapped cache and a predictive sequential associative cache, respectively, with one L1 port.

The reactive-associative cache is a low-latency associative organization that has a bandwidth demand comparable to a system with a direct-mapped cache, and therefore would be ideal as the L1 dcache in a memory bandwidth-limited system.

LIST OF REFERENCES

[1] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, Oct. 1994.

[2] S. J. E. Wilson and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, July 1994.

[3] 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. Association for Computing Machinery, Oct. 1996.

[4] 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.

[5] 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. Association for Computing Machinery, May 1993.

[6] C. Zhang, X. Zhang, and Y. Yan. Two Fast and High-Associativity Cache Schemes. *IEEE Micro*, Vol. 17, No. 5, September/October 1997.

[7] 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 240–250, Oct. 1998.

[8] 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.

[9] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *32nd International Symposium on Microarchitecture (MICRO-32)*, pages 248-259, November 1999.

[10] T. Austin and G. Sohi. Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. MICRO-28, November 1995.

[11] N. P. Jouppi. Architectural and organizational tradeoffs in the design of the MulitTitan cpu. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 281-289. Association for Computing Machinery, May 1988.

[12] J. H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS System/370. In *14th Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 208-213. IEEE, June 1987.

[13] 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.

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

[15] 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.

[16] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103. Association for Computing Machinery, Dec. 1995.

[17] T. Johnson and W.-M. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[18] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, May 1996.

[19] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 139–149. Association for Computing Machinery, Oct. 1998.

[20] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. Association for Computing Machinery, May 1999.