IT Master theses
UPTEC F 02 033

Refinement and Evaluation of the Elbow Cache

MATHIAS SPJUTH

Supervisors: Martin Karlsson, Professor Erik Hagersten

Examiner: Professor Erik Hagersten

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

Refinement and Evaluation of the Elbow Cache

MATHIAS SPJUTH

April 2002

INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Master of Science in Technology
at Uppsala University 2002

Refinement and Evaluation of the Elbow Cache

Mathias Spjuth
masp0413@student.uu.se

Information Technology
Department of Computer Systems
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden
http://www.it.uu.se

Uppsala Architecture Research Team
http://www.it.uu.se/research/group/uart/

ISSN 1401-5757
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

During the last 15-20 years there has been an increasing gap between the bandwidth demands of the modern microprocessor and the performance of the memory sub-system. Memory hierarchies, with nested levels of memory buffers for intermediate caching of data, has been devised as the main solution for this problem. Hardware caching has become of fundamental importance for the overall performance of any modern microcomputer. Consequently, much recent research has been focused on different techniques to improve caching.

There are tough demands on a hardware cache. First of all, it needs to be fast to satisfy the CPU's demand for data. It also needs to have good performance in bringing down the total number of data misses. Finally, it must be simple and cheap to implement. In this trade-off between simplicity and performance, cache misses caused by conflicts might become a problem. To mitigate this, different schemes such as victim caches, column-associative caches and skewed-associative caches, have been proposed.

This thesis investigates a novel technique to further improve on skewed-associative caching. The technique, dubbed *elbow caching*, is based on an ability to dynamically move data between alternate positions in the cache. To support this, a new replacement policy based on cache allocation timestamps is suggested. By using trace-driven simulation it is shown that a 2-way elbow cache, using timestamps, has roughly the same miss ratio as a conventional, LRU, 8-way set-associative cache.

Contents

1 Introduction

The main objective of this thesis is to study a new cache-architecture called *elbow cache*. It is a refinement of skewed-associative caching methods meant to reduce mapping conflicts to popular data sets contained in the cache. By dynamically reallocating data, the elbow cache seeks to further reduce conflicts and allow for a wider choice of victims. To benefit from this, it is also necessary to find a better replacement strategy than the ones suggested for skewed caches so far. This is accomplished by tagging each cache block with the timestamp of the last reference. Simulation shows that a 2-way elbow cache perform, on average, similar to a 8-way set-associative cache with regard to the effective miss-ratio.

This thesis is organized as follows; Background and related work is presented in the rest of this section. Section 2 discuses conventional techniques for conflict avoidance. Section 3 and 4 presents the skewed-associative and the elbow caches. Replacement metrics and implementation is discussed in sections 5 and 6. The evaluation methodology is described in section 7 and the results are presented in section 8. The work is summarized in section 9. Finally, section 10 suggests areas for future research.

1.1 Background

Caching has been used as the main solution for bridging the ever-increasing gap between the bandwidth demand of the modern microprocessor and the performance of the memory system. Consequently, well-designed caches are crucial for the performance of the entire system. The main limiting factor in cache performance is the size of the cache, but also architectural limitations in the cache affect cache performance, since they might cause allocation conflicts within an active data set. Therefore, novel architectures that are better at avoiding such conflicts without adding excessive complexity or cost in terms of extra hardware, has become an important research topic.

1.2 Related Work

Some of the earlier work that has addressed the issue of reducing cache conflicts is presented here. Jouppi [5] presented the *victim cache*, primarily for use in direct-mapped caches. Topham and Gonzales [14] studied the use of hash-functions for indexing a cache. Agarwal and Pudar [1] suggested *column-associativity* for improving direct-mapped caches. Seznec and Bodin [11] pioneered the work on *skewed-associative* caches. Further work on skewed caches is presented in some of their later papers [3, 9, 10].

Karlsson and Hagersten [6] showed the usefulness of timestamps as block survival time metric in the RASCAL-cache.

2 Caches and Conflicts

This section presents a brief introduction to the problem of conflicts in caches, how conventional set-associative caches deal with this issue and how misses are classified. Section 3 then presents skewed-associative caches that utilizes a slightly different approach.

2.1 Direct-Mapped Caches

The simplest form of a hardware cache is the direct-mapped cache. In a direct-mapped (or one-way set-associative) cache every address in the computer is associated with a specific position in the cache. Once a data request is made, that position (or cache *block*) is checked to see if it contains the requested data. If not, the data is fetched from memory and put in the cache, replacing any previous data in that block. This simple scheme has the advantage of being not only simple, but also very quick since the requested data, if in the cache, can be found in one specific position only. The problem is that two or more sets of data might compete for the same position, while other positions are left unused.

2.2 Set-Associative Caches

The mapping of memory into the cache might cause an uneven distribution of references to each cache block—i.e, some positions become more 'crowded' than other positions. Such *hot-spots* might cause *conflict* misses—misses that would otherwise never have occurred. By increasing the *associativity* of a cache, some of the irregularities in the access pattern can be removed by joining several cache blocks into *sets*.

In a n-way *set-associative* cache, each memory address can be associated with any block in a set of n equivalent blocks. Thus, there is a choice of which of the n previously cached data items to replace. Additional information stored in each set is used to make this selection. There can now be up to n different data blocks, mapped to the same set, in use at the same time without any conflict occurring.

However, this improvement comes at a price. Since a requested data item can be in one of several different blocks, there is a need for additional hardware to test all these blocks and detect the correct one. The replacement information also needs to be stored. This adds to the cost in terms of extra logic and chip real-estate as well as longer cache access latency. The higher the degree of associativity, the higher is the cost. Also, the potential gain decreases with higher associativity.

Nevertheless, in some cases, conflicts can become a real performance bottleneck. What is needed is a cache that utilizes a low degree of associativity and still reduces conflicts well. Such a cache is the skewed-associative cache described in section 3.

2.3 Classification of Cache Misses

The most used classification scheme for misses that occur in a cache, is the "three C's" model [4]. There, the distinction is made between *Compulsory*, *Capacity* and *Conflict* misses. Compulsory misses are the ones occurring when a block is accessed for the first time. A capacity or conflict miss occurs whenever a previously evicted block is brought back into the cache again. If the block was evicted because of a mapping conflict it is classified as a conflict miss, otherwise it is a capacity miss. In general, it is not possible to classify an individual miss as either capacity or conflict. Instead, the difference in miss rate between the cache in question and a fully associative cache (no conflict misses) of the same size, is used as measure for the conflict miss rate.

The "three C's" model does not cover misses that occur because of the replacement policy used for the cache. A better model, that also considers these, is the OPT model [12]. There, the optimal (OPT) replacement is used as norm and conflict misses are further subdivided into *mapping* and *replacement* misses.

2.4 Different kinds of Conflict Misses

The OPT model [12] can be used to further characterize conflict misses. Assume a cache that has the ability to map any memory address to any cache block. Such a *fully associative* cache has no mapping restrictions and cannot therefore cause any *mapping misses*. Nevertheless, it can cause misses that are neither compulsory nor due to limited capacity. These occur because of limitations on the replacement policy used and are therefore called *replacement misses*. Now, consider a direct-mapped cache. Since there are never any questions about which data to replace (always only one possibility) there can be no replacement misses. But it is already known that direct-mapped caches can suffer heavily from mapping misses instead.

While increasing the associativity in a cache lowers the number of mapping misses, it might increase the number of replacement misses and vice versa. This is because the increased freedom of choosing a victim also increases the possibility of making a bad choice. Usually, the reduction of mapping restrictions that results from higher associativity more than makes up for the potential increase in misses due to a suboptimal replacement strategy. But for some applications—for which the replacement policy used is ill suited—a higher degree of associativity can actually yield worse performance.

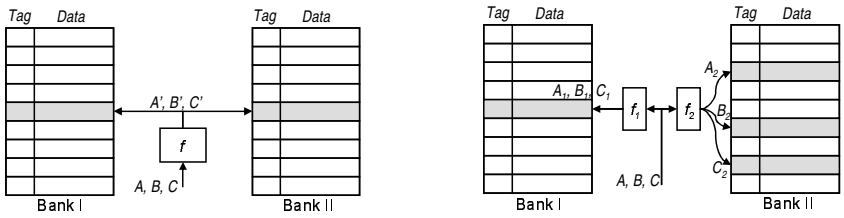
3 Skewed-Associative Caches

The association between the address space and the individual blocks in the cache is done via an indexing function that maps each memory address to its corresponding set in the cache. For a set-associative cache this is usually a trivial mapping, taking the lowest bits of the address and using them as the index. All n alternate positions in an associative set will then have the same index. A cache with a slightly different structure is the *skewed-associative* cache. This cache is the basis for the work in this thesis and is described in more detail below.

3.1 Principle of the Skewed-Associative Cache

Instead of trying to average out hot-spots by joining blocks into associative sets, it is possible to change the mapping to avoid such hot-spots from forming in the first place. *Hashing* has been proposed as a solution to this problem by giving a better statistical distribution of the cache accesses over all blocks in the cache [14]. Nevertheless there is likely to remain some random hot-spots (and *cold-spots*). An improvement to that idea is to organize the cache into two or more *banks* and use different hashing functions for each of the banks. By doing this, accesses that maps into hot areas in one bank have a good chance of hitting a less frequented area in one of the others, thus spreading the accesses more evenly across all the blocks in the cache and thereby reducing the overall number of conflicts.

This is the principle of the *skewed-associative* cache. The difference between a skewed-associative and a set-associative cache is illustrated in Figure 1. Three data items A, B and C, all in active use, are mapped via the indexing functions f and f_1 respectively, into the same position in the first cache bank. In the two-way set-associative cache, all three datums are also mapped into the same position in the second bank, creating a conflict since there are only two possible positions in total. In the skewed cache on the other hand, the hashing- or *skewing*-functions maps each of the items into a different position in the second bank, so they can all be placed without conflict.



(a) Set-associative cache.

(b) Skewed-associative cache.

Figure 1: Two-way set- vs skewed-associative cache.

3.2 Deficiencies of a Skewed Cache

There are a number of problems associated with skewed caches. First of all, the skewing-functions used for the memory-to-cache mapping increases the memory latency for accessing the cache. It might be possible to hide this latency by applying the skewing function during the address computation at an earlier pipeline stage.

Seznec [9] presents efficient skew-function implementations using permutations and XOR-operations. Second, the skewing-functions must make use of a larger part of a memory address for indexing, which makes it difficult to implement caches using *virtual indexing–physical tagging* [4]. *Page coloring* can be used to solve this problem.

Example: Page coloring for skewed cache.

32 KB two-way skewed-associative cache with 64 byte blocks.

Bits needed to identify a byte within a block: 6 ($2^6 = 64$).

Blocks in the cache: 512 blocks divided into two banks.

8 bits needed for a index in one bank ($256 = 2^8$).

8 additional bits needed for the skewing functions.

Total: 22 lowest bits of the address needed for indexing.

Thus to avoid page aliasing pages must be shared with an even 2^{22} = 4 MB-multiple address offset.

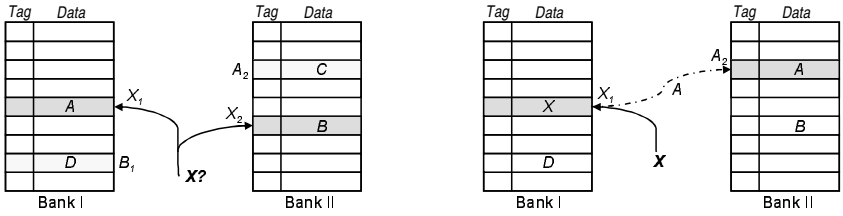
Another problem is to decide which block to replace. In a set-associative cache all potential victims can be found in the same set. To enforce a *least-recently-used* (LRU) policy the cache only needs to keep track of the relative order of the accesses that occurs within each set. In a skewed-associative cache the possible placements for a new cache block in each bank is different for each address due to the separate skewing-functions. To achieve LRU replacement it is therefore necessary to keep a global ordering of accesses to all blocks in the cache. This is infeasible in most cases so some kind of approximative method must be employed instead. Such a method, the NRUE replacement, is discussed in the next section.

3.3 Replacement Policies

Seznec [10] has made an analysis of several different replacement policies for skewed caches. One policy presented there is the *enhanced-not-recently-used* (NRUE). NRUE tries to determine if the cache block has been accessed *recently*, *very recently* or *not recently*, by setting two bits, 'recently' and 'very recently', every memory access. These sets of access-bits are then cleared in a regular manner. For a cache consisting of N blocks the 'very recently' bits are cleared when $N/4$ block have been accessed and the 'recently' bits are cleared when $N/2$ blocks have been accessed. Thus, a block that has the 'very recently' bit set has been accessed more recently than a block with only the 'recently' bit set and, a block with the 'recently' bit set has been accessed more recently than a block without any bit set at all. By testing the bits the cache can make a choice if a block is a good candidate for replacement. The major drawback with this scheme is that just after the access-bits have been cleared, the replacement policy degrades into a simple random replacement.

4 The Elbow Cache

Assume that datum X is to be inserted into a skew-associative cache. X is mapped, by two separate skewing-functions, into positions X_1 and X_2 in bank one and two respectively. This situation is illustrated in Figure 2a. There is a choice of replacing either data item A, currently residing in bank one, or item B in bank two. However, by allowing data to be reallocated between the banks, this choice can be broadened to also include data items C and D. By moving A to its alternate bank-two position, replacing C, space can be freed to insert datum X without evicting A (Figure 2b).



(a) Possible placements for new data item X.

(b) Evicting item C by reallocating item A.

Figure 2: The elbow cache.

Alternatively, if item D is deemed to be a good candidate for replacement, B can be moved to replace D instead. In effect, the degree of freedom in the choice of replacement has increased from two to four. As long as there are no loops formed by the movements (or *transformations*) the data reallocations can continue on until a suitable victim is found. This is the principle of the *elbow* cache.[1]

4.1 Finding the Victim

An elbow cache needs a method for finding a suitable victim. Assuming that there is a way of ranking existing cache blocks, how will the cache find the best block to replace? Two simple methods seem possible. The first is to test the blocks along all possible replacement paths and make a decision based on the information collected. This straightforward scheme gives an optimum placement,[2] but the implementation in hardware is nontrivial at best, impossible at worst. This method may be used during evaluation to estimate an upper bound for the effectiveness of elbow replacement. It will be used as such in this work, and called *lookahead*, since the replacement decision is made after all information is collected.

The other method is to first place the new data in its most optimum immediate position and then test the victim against its possible alternative placements, replacing a suitable new potential victim if such is found. This process can be iterated until there are no good new victims. This method is called *feedback* since it can be implemented by feeding the victim back into an already existing *write buffer*. Note that this method is more restricted then lookahead, since it only makes use of local information that is immediately available. It might also stop prematurely,

[1] A new datum makes room for itself among other, frequently used data blocks, by forcing this data to reallocate instead of replacing it. Metaphorically speaking, it is "using its elbows", pushing other data aside. Hence the name.

[2] By "optimum placement" is meant the best possible placement with regard to the replacement policy used.

before reaching a globally optimal placement, if a local optima is discovered on the replacement path.

4.2 Choosing the Hash Functions

In order to obtain good performance the hashing-functions used to index the cache banks must exhibit certain properties. These criteria are only listed here, for a more thorough discussion see the papers on skewed caching [3, 9, 10, 11]:

- Equitability
- Inter-bank dispersion
- Local dispersion in a single bank
- Simple hardware implementation

To reduce the time needed for running the cache simulations, it would also be of value to have skewing-functions that are easy to compute in software. For this reason, and because the specific choice of functions is of less importance for performance [10], the functions suggested by Bodin and Seznec [3] were used. Consider a cache with up to four distinct cache banks holding 2^n cache blocks of 2^c bytes each. An m-bit memory address A can be represented as a tuple $\{A_3, A_2, A_1, A_0\}$ with $A = A_3 2^{c+2n} + A_2 2^{n+c} + A_1 2^c + A_0$. Let σ be the one bit rotational shift on an n-bit number. The four hashing-functions can now be written[3]

$$\begin{aligned} f_0(A) &= A_1 \oplus A_2 \\ f_1(A) &= \sigma(A_1) \oplus A_2 \\ f_2(A) &= \sigma^2(A_1) \oplus A_2 \\ f_3(A) &= \sigma^3(A_1) \oplus A_2 \end{aligned}$$

for banks 0 to 3 respectively.

Although these functions were chosen for the sake of convenience, the results in this paper are also valid for any other set of functions that fulfill the requirements listed above.

4.3 Reconstructing the Address

In an elbow cache it must be possible to, given a set of data in one bank, calculate the index for that data in another bank. To move data from bank n to m the full address is first recreated by applying the inverse skewing-function f_n^{-1}. Then the new index is found by applying skewing-function f_m. Assume that the address $A = \{A_3, A_2, A_1, A_0\}$ of the data kept at index A_{idx_n} in bank n is to be found. Block offset A_0 is always identical to 0 since it addresses the beginning of a block. The tag field, containing the tuple $\{A_3, A_2\}$, and the index is combined to yield the full address

$$\begin{aligned} A &= f_0^{-1}(A_3, A_2, A_{idx_0}) = \{A_3, A_2, A_{idx_0} \oplus A_2, 0\} \\ A &= f_1^{-1}(A_3, A_2, A_{idx_1}) = \{A_3, A_2, \sigma^{-1}(A_{idx_1}) \oplus A_2, 0\} \\ A &= f_2^{-1}(A_3, A_2, A_{idx_2}) = \{A_3, A_2, \sigma^{-2}(A_{idx_2}) \oplus A_2, 0\} \\ A &= f_3^{-1}(A_3, A_2, A_{idx_3}) = \{A_3, A_2, \sigma^{-3}(A_{idx_3}) \oplus A_2, 0\} \end{aligned}$$

of the data in the respective banks 0 to 3.

[3] In a more general case, two permutation functions ϕ and ϕ' can also be applied to A_1 and A_2 respectively, but these have been omitted here to increase simulation speed.

5 Replacement Metric

The Elbow caches increased ability to select a victim emphasizes the need for a good replacement strategy. *Least-recently-used* (LRU), is commonly considered to be the best general such strategy [4]. But a perfect LRU policy would demand a unreasonable cost in terms of hardware and a NRUE policy, as discussed in section 3.3, do not provide enough information for this purpose. It is therefore necessary to find a technique that approximates LRU better then NRUE does, and is implementable at a low cost. But what makes LRU a good choice in the first place?

5.1 Least-Recently-Used Replacement

An optimal replacement policy (OPT), that gives the lowest possible miss ratio, is the one that replaces the block that *will remain* unreferenced for the longest period of time. LRU instead replaces the cache block that *has been* unreferenced for the longest period of time. But due to the principle of locality, a recently used cache block is more likely to be reused again within a short while than a not recently used block. If the least recently used block is chosen for replacement, the chance of removing a block not recently used is maximized. Note the distinction made here between *least recently* used and *not recently* used.

Assume that the replacement mechanism have to make a choice between two blocks that both have been accessed recently. The LRU policy would replace the block accessed least recently. But in this case, random replacement (RND) might be a better choice since *both blocks were recently used*. The same would be true for two blocks not recently referenced. Their internal access ordering is of less importance, the important fact here is that they are both very 'old' and therefore unlikely to be accessed again in the near future. Once again RND replacement might be preferred over LRU.

The above discussion suggests that a replacement scheme that can differentiate between 'older' and 'younger' cache data might perform as good as or maybe even better than LRU. This is basically what NRUE tries to achieve. Another way is to use timestamps.

5.2 Timestamps

To approximate LRU a scheme that tags each block with a *timestamp* for the latest access made to that block is proposed.

A global *counter* (also called *timer*) is kept inside the cache. An n-bit timestamp field is also associated with every cache block. This timestamp is updated on access with the n most significant bits of the current counter value. By increasing this counter at regular intervals, the timestamps will reflect whether a block has been recently accessed or not. This is done by calculating the *distance* between the timestamp for the block and the 'current time' T_{curr} (the n most significant bits of the global counter). A block with a small distance has been recently accessed and vice versa. Since the global timer has finite number of bits, counter overflows must be taken into account. The distance d for a timestamp T_{st} is then calculated as follows

Algorithm 1 Calculating the timestamp distance.

$$d = \begin{cases} T_{curr} - T_{st} & T_{curr} \geq T_{st} \\ T_{curr} + 2^n - T_{st} & T_{curr} < T_{st} \end{cases}$$

where n is the number of bits in the timestamp.

The counter needs to have a sufficient number of bits to avoid aliasing effects. If a cache block resides unused in the cache long enough it might 'wrap around' and become young again. This happens if the distance becomes greater then 2^n.

One remaining question is what time unit to use as a 'tick' for the global timer. Different applications exhibit different memory access patterns and miss-ratios and any scheme must take that into account. It seems natural to chose *cache allocations* as 'ticks' since:

1. The time between replacements is inversely proportional to the miss ratio.
2. An upper bound for the probability of a cache block surviving unused for a given number of such ticks can be easily estimated by statistical means (Appendix C).

As is shown by Karlsson and Hagersten [6], this normalizes the replacement-behavior between different applications. From the estimates made in Appendix C it can be concluded that the vast majority of all unused cache blocks has been evicted from a N-block cache after about $4N$ replacements, and this is therfore a suitable upper limit for the timer. Very few blocks will then survive unreferenced long enough to experience aliasing. So, the global time counter needs to be $m = log_2(4N)$ bits long. For a n-bit timestamp this results in a *resolution* of 2^{m-n} cache allocations per time unit or tick.

Example Timestamp resolution.

---

16 KB cache with 32-byte blocks, 6-bit timestamp.

$2^{14-5} = 2^9 = 512$ blocks

$m = log_2(4N) = log_2(4 * 512) = 11$ bits

Resolution: $2^{11-6} = 32$ allocations per time unit.

It is believed that such timestamps, with a high enough resolution (enough bits), will approximate a proper LRU-ordering of the cache blocks close enough to be useful as replacement metric in a skewed or elbow cache. A replacement strategy based on timestamps would replace the block with the longest distance. If two or more blocks have the same distance the victim is chosen at random among these.

6 Cache Models

This section presents the actual elbow-cache models evaluated in this thesis.

6.1 n-step Lookahead Cache

The *lookahead* elbow cache selects the most optimal placement possible according to the replacement policy used (i.e. replacing the block with the oldest timestamp). If the cache considers all locations that can be reached by moving data up to n times it is called an *n-step lookahead* elbow cache. By starting at zero and stepwise increasing n, the influence of the increased freedom of replacement on the overall miss-rate, is examined.

A zero-step lookahead cache is identical to an ordinary skewed-associative cache using timestamp replacement. Because the lookahead model uses information gathered earlier on, and utilizes several transformations for each replacement, it may be difficult to implement for two or more steps. However, a one-step lookahead is feasible for a non-blocking, pipelined cache. The timestamps of the alternative victims are read during a second cycle and the data transformation is then made in parallel with the loading of the new data when that is available. This *single-step, pipelined, lookahead cache* is now described. Figure 3 show the constituent parts and the different pipeline stages during a replacement. Item X is to be inserted into the cache. In the first stage the possible placements in each bank are indexed by skewing functions f_1 and f_2 (not shown) into X_{idx_1} and X_{idx_2} respectively. The bank contents in the corresponding positions (A and B) are loaded into two temporary registers R_1 and R_2, and their tags are checked to see if they contain the data requested. Since this is not the case in this example, a request for X is sent to the underlying memory hierarchy. If the cache features prefetching [5], X might also be found in one of the prefetch registers.

In parallel with the cache lookup, the alternative indices (the indices of A and B in their opposite cache banks), are calculated as described in section 4.3. Also, the associated timestamps are loaded into temporary registers T_1 and T_2. The second stage can last for one or more clock cycles depending on the time needed to retrieve X from the memory subsystem. In this stage the timestamps of A and B and the timestamps of their alternative placements C and D, are compared in the combinatoric net N. Once X is available the last stage loads it into place based on the information given from N. At the same time A or B can also be moved to the other bank if either C or D is chosen as victim.

The combinatoric net N has two outputs b (bank) and m (move), that controls the loading of data. b decides in which bank to place the new data and m controls

Victim	b	m	$channel_1$	$channel_2$	$load_1$	$load_2$
A	0	0	X	A	1	0
B	1	0	B	X	0	1
C	0	1	X	A	1	1
D	1	1	B	X	1	1

Table 1: Control of data loading.

whether the old data in that position should be reallocated or not (see Table 1). The new data X_{data} is inserted into one of the caches load channels via multiplexers that also inserts either A_{data} or B_{data} in the channel for the opposite bank. Control signals b and m are combined into $load_1$ and $load_2$ that signals whether or not to load the bank with the data in the corresponding load channel. Also the bank index and the timestamp for each cache block is feed into the load channel.

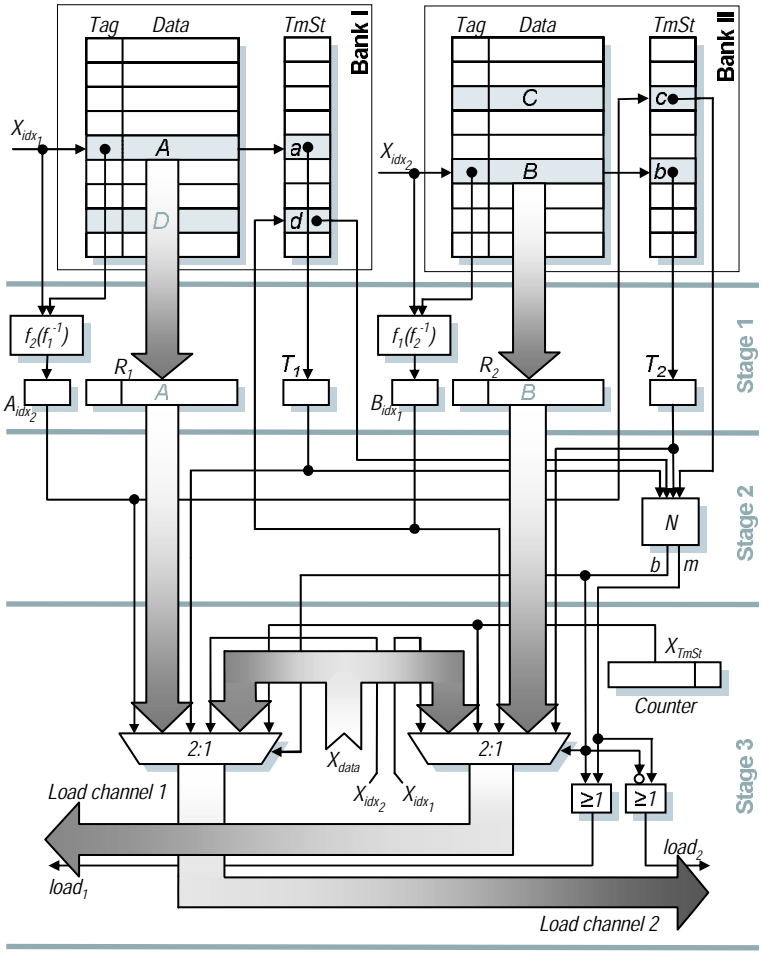


Figure 3: A single-step lookahead elbow cache.

The result is a loading of X and an optional movement of A or B at the same time.

6.2 Feedback Cache Model

In the feedback model the victim is put back into the write buffer and might later be reloaded into another cache bank if deemed valuable enough to keep in the cache. It is necessary to be able to identify an entry in the write buffer as a victim instead of a regular write. This is done with an extra flag bit together with a field marking in which bank the data used to be. Once the write occurs, all the other alternative positions are checked to see if they contain a more suitable victim (i.e with a longer timestamp distance) than the current. If so, the old victim is reloaded into the cache and the new victim is inserted into the write buffer. If not, the entry is discarded.

An interesting side effect of this scheme is that the write buffer will effectively act as a *victim cache* [5].

Two problems might occur: first, the write buffer might be filled to the limit. In this case, flushing one or more victims is a possible solution. The other problem is that loops might form in the replacement path. If so, the original write might be overwritten by a victim. A solution to this is to assure that any victim is always strictly worse, measured in the replacement metric, than the entry to be loaded. This way any loops cannot be formed. Unfortunately, such a restriction

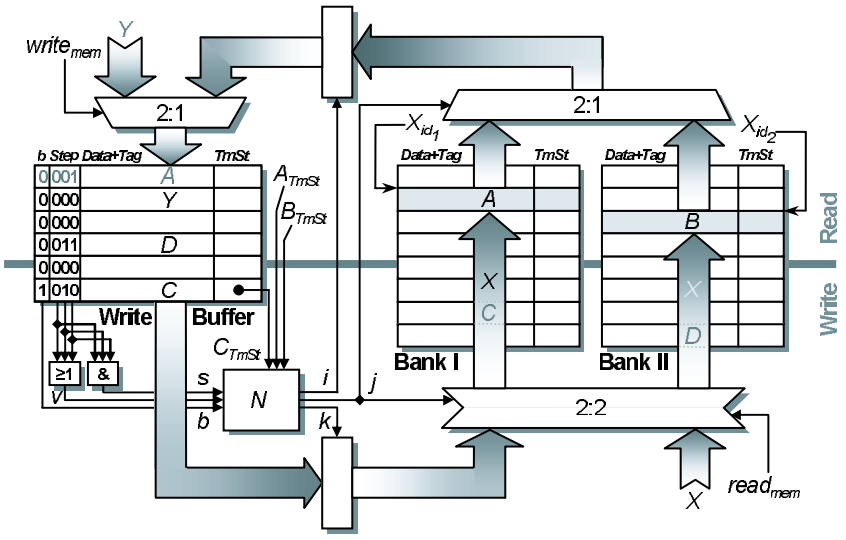


Figure 4: A 7-step feedback elbow cache.

on replacement is likely to give less optimal results. Since the probability of a loop occurring is small, another possibility is to apply the *ostridge algorithm* [13], ignoring the problem altogether. However, in such a case, limiting the maximum number of transformations is necessary to stop any loops from looping forever. Such a limit will also reduce the problem with limited write buffer space. Therefore, an extra *step* field, that contains the number of transformations made so far for this replacement, is kept in the write buffer.

The whole layout is depicted in Figure 4. To the left is the write buffer into which the datum Y has been recently written. A request for item X has missed in both of the cache banks and the datum has just been fetched from the underlying memory hierarchy. X maps into the blocks currently occupied by data A and B in banks one and two respectively. The combinatoric net N reads the timestamps for A and B and makes an allocation decision based on this information. In this case A is selected as victim. Since this is a fresh insertion, A is moved into the write buffer and the b (bank) and *step* fields are set accordingly (0 | 001—victim from bank 1, first transformation step). Note that a victim can be identified by OR:ing all bits in this field, so there is no need for an explicit victim flag in this case. The output of N; i, j and k controls if a datum should be feed back into the write buffer, which cache bank to read from and/or write to, and if an earlier victim should be reinserted into the cache. The next item in line in the write buffer is the datum C. The b and *step* fields show that it is a second order victim, and that C used to reside in bank two (b bit set). This is signaled to N via the b and v (*victim*) signals. When the wite occurs, N will check if the timestamp distance for C is shorter than that for the item currently residing in C's bank-one position. If so, C is moved into the position (indicated in the figure) and, unless the s (stop) signal indicate that the maximum number of transformations (7) has been reached, the new victim is again inserted into the write buffer. If the distance for C is longer then for the potential new victim, C is evicted from the cache.

7 Evaluation Setup

The evaluations were made using the system simulator SIMICS [7], simulating an UltraSPARC-II machine running Solaris 7. This setup allows a full system simulation including the operating system. The benchmarks were started from a standard shell on an otherwise unloaded virtual machine. By the use of SIMICS' trace module, *traces* of the memory accesses made by the benchmark programs were produced. These traces were compressed with a trace compactor program and saved to disk. The traces were then feed into a cache simulator program, simulating the different cache configurations.

This work considers level-1 data caches for uniprocessor systems and assumes a *write back - no allocate on write* strategy and a perfect (infinite) write buffer. This means that writes to the cache can be ignored for all practical purposes.

7.1 The SPLASH-2 Benchmark Suite

Most of the test applications were taken from the Stanford SPLASH-2 benchmark set [8]. Although primarily meant for multiprocessor research, the length of these benchmarks also makes them well suited for computer architecture design in general. It contains a total of 12 kernels and applications and yields 14 benchmarks:

Barnes This is an application that simulates interaction between a system of bodies in three-dimensional space using the Barnes-Hut method.

Cholesky This kernel factors a sparse matrix into a product between a lower triangular matrix and its transpose. It is similar to the LU kernel but operates on sparse matrices.

FFT Fast Fourier Transformation kernel.

FMM Like Barnes but simulates interaction between bodies in two dimensions instead, using Fast Multipole Method.

LU Kernel that factors a dense matrix into the product of a lower and an upper triangular matrix. Contiguous and non-contiguous versions.

Ocean Studies large-scale ocean movements based on eddy and boundary currents. This benchmark exists in two versions, contiguous and non-contiguous where the latter uses a simpler, less efficient implementation.

Radiosity Computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method.

Radix Integer radix sort kernel.

Raytrace Renders a three-dimensional scene using ray-tracing.

Volrend Renders a three-dimensional volume using a ray casting technique.

Water-Nsquared Evaluates forces and potentials that occur during time in a system of water molecules, using an $O(n^2)$ algorithm.

Water-Spatial Solves the same problem as Water-Nsquared but uses a different, $O(n)$, algorithm.

The parameters used and the resulting trace-length for each benchmark is given in Table 2.

Benchmark	Name	Parameters	Read accesses
Barnes	BARNES	< barnes_input	810,144,285
Cholesky	CHOLESKY	CHOLESKY_INPUT	108,297,727
FFT	FFT	-m16	11,464,509
FMM	FMM	< fmm.input.16384	1,729,059,329
LU	LU_C		25,930,229
	LU_NC	(non-contiguous version)	153,827,897
Ocean	OCEAN_C		136,025,825
	OCEAN_NC	(non-contiguous version)	307,465,336
Radiosity	RADIOSITY	-room -ae 5000.0 -en 0.050 -bf 0.10 -batch	625,864,891
Radix	RADIX		53,297,034
Raytrace	RAYTRACE	-m64 -p1 RAYTRACE_car.env	377,501,122
Volrend	VOLREND	1 VOLREND_head	22,530,034
Water-Nsq	WATER_N		169,628,504
Water-Sp	WATER_S	< WATER-SPATIAL_input	162,278,521

Table 2: Parameters used for the SPLASH-2 benchmarks.

7.2 The reduced SPEC 2000 Benchmark Suite

This is a set of benchmark based on the SPEC 2000 benchmark suite, but with much reduced running time to allow them to be used for computer architecture analysis and design [2]. Six of the long-length benchmarks, both integer and floating-point, in this suite were used (the others were to long):

AMMP Computational Chemistry. Floating-point benchmark.

Equake Seismic Wave Propagation Simulation. Floating-point benchmark.

MCF Combinatorial Optimization. Integer benchmark.

Parser Word Processing. Integer benchmark.

VPR FPGA Circuit Placement and Routing. Two sub-benchmarks; VPR_PLACE and VPR_ROUTE. Integer Benchmarks.

The input sets used and the resulting trace lengths are given in Table 3.

Benchmark	Name	Parameters	Read accesses
AMMP	AMMP	< ammp_lgred.in	149,311,204
Equake	EQUAKE	-Q < quake_lgred.in	164,396,815
MCF	MCF	mcf_lgred.in	149,285,010
Parser	PARSER	2.1.dict -batch < parser_lgred.in	693,783,085
VPR	VPR_PLACE	place_lgred.net ref_arc.in	314,304,850
	VPR_ROUTE	route_lgred.net ref_arc.in	220,486,334

Table 3: Input sets for the reduced SPEC 2000 benchmarks.

8 Results

The results obtained from the simulations are presented here. The main results were generated by simulating a 16 KB, level-1 (L1) data cache with 32-byte block length. Today it is not uncommon with caches that uses block length of 64-byte or more. However, for the purpose of this evaluation it is more interesting to study a cache with a larger amount of (smaller) blocks.

Also 8 and 32 KB caches were tested for comparison.

The metric *miss rate reduction* is used to measure the performance of different cache configurations. The reduction norm is taken from a simple two-way set-associative cache of the same size. Miss rate reduction is calculated as shown in Algorithm 2 below.

Algorithm 2 Miss rate reduction.

$$Miss\_rate\_reduction = \frac{Miss\_rate_{norm} - Miss\_rate}{Miss\_rate_{norm}}. \quad (1)$$

Here $Miss\_rate_{norm}$ is the miss rate of the reference cache.

8.1 Lookahead Cache

This model was tested by simulating n-step lookahead caches with n ranging from 0 to 8 transformation steps. This gives a rough indication of the performance increase that can be achieved by reallocating data up to n number of times during each new allocation. From this the performance of a simple skewed cache with timestamp (TS) replacement is found in the 0-step column, and the performance of the simple lookahead cache described above, in the 1-step column (Figure 5). Four different resolutions were used with 8-, 6-, 5-, and 4-bit timestamps, plus perfect LRU for comparison. Both the SPLASH-2 and SPEC 2000 reduced benchmark suites were tested. A typical result is shown in Figure 5—the CHOLESKY benchmark from SPLASH-2.

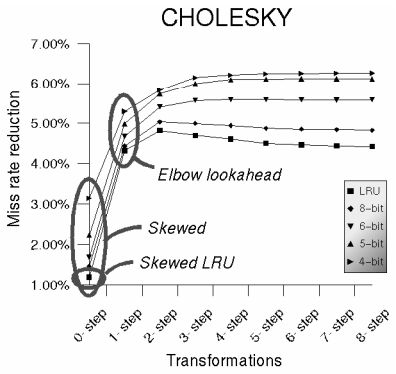
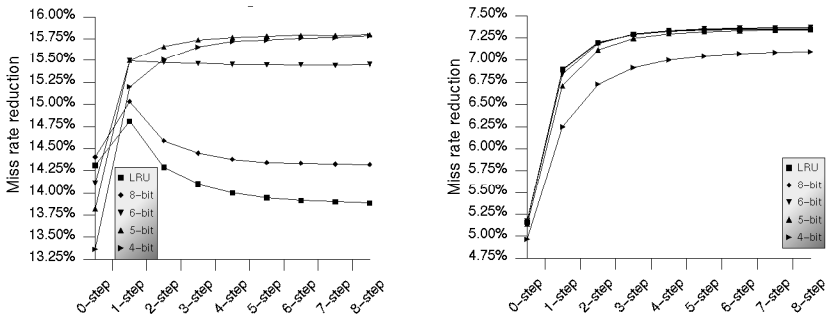


Figure 5: Lookahead cache miss rate reductions for the CHOLESKY benchmark compared with a 2-way set-associative cache.

There is a relatively steep increase in miss ratio reduction for the first few steps, but then the curve soon reaches a plateau. This is just as expected: the cache benefits from the increased freedom of replacement selection, but the probability of finding another even better victim decreases quickly with each additional step.

Note that for this particular benchmark, lower resolution timestamps give better results then higher resolution and LRU. This is not an unusual result and shows the imperfections of LRU as discussed in section 5.1.

The complete set of graphs for the two benchmark suites is found in Appendix A. Averages are shown in Figure 6.



(a) SPLASH-2 (b) Reduced SPEC 2000

Figure 6: Average miss rate reductions vs a 2-way set-associative cache for a lookahead cache and different replacement metrics.

It is clear from these graphs that the timestamps (for this particular cache configuration) needs to be at least 5-bits long, since the performance of the 4-bit length timestamps are noticeably worse for many of the benchmarks. For a block length of 32 bytes, the 5 extra replacement information bits results in a memory overhead of less than 2% .

From the data, the results for a two-way skewed cache with 5-bit timestamp allocation (TS), and a two-way, single-step, lookahead elbow cache (LA), also with 5-bit timestamps are extracted and used for comparison with other caches. To get a good comparison, the results are plotted against miss ratio reductions for other alternative cache configurations with the same cache size and block length; a four-way set-associative, a fully associative, and two skewed (perfect LRU and NRUE replacement) caches.

The results for a 7-step feedback (FB) cache discussed in the next section, is also included. The results are shown in Figure 7 and 8.

As expected the fully associative cache reigns supreme and outperforms all the other caches in 15 out of the 20 benchmarks, with an average miss rate reduction of 14.5% for SPLASH-2 and 9.5% for SPEC 2000. The competition for second place is tougher, but the elbow cache wins the SPLASH-2 race with a margin of around 0.8% over skewed-LRU and 1.1% over skewed-TS (ignoring the feedback cache for now), obtaining a total reduction of 12.3%. For reduced SPEC 2000, the elbow cache wins the silver medal once again with a narrow margin of 6.7% total reduction over the four-way set-associative cache' 6.6%.

Note however, that a few benchmarks, especially LU_NC, gives a disproportionately large contributions to the total average due to some peculiar properties in those benchmarks.[4] A short discussion of some benchmarks is found in section 8.4. The results also show that using timestamps as replacement metric, as suggested in section 5.2, gives good results. In particular, a skewed cache using timestamp replacement with a resolution of only 5-bits, yields a performance very close to that

[4]Specifically, LU_NC is the main reason that the 8-bit and LRU average SPLASH-2 graphs in Figure 6a actually show a decrease in miss ratio reduction for increased number of lookahead steps.

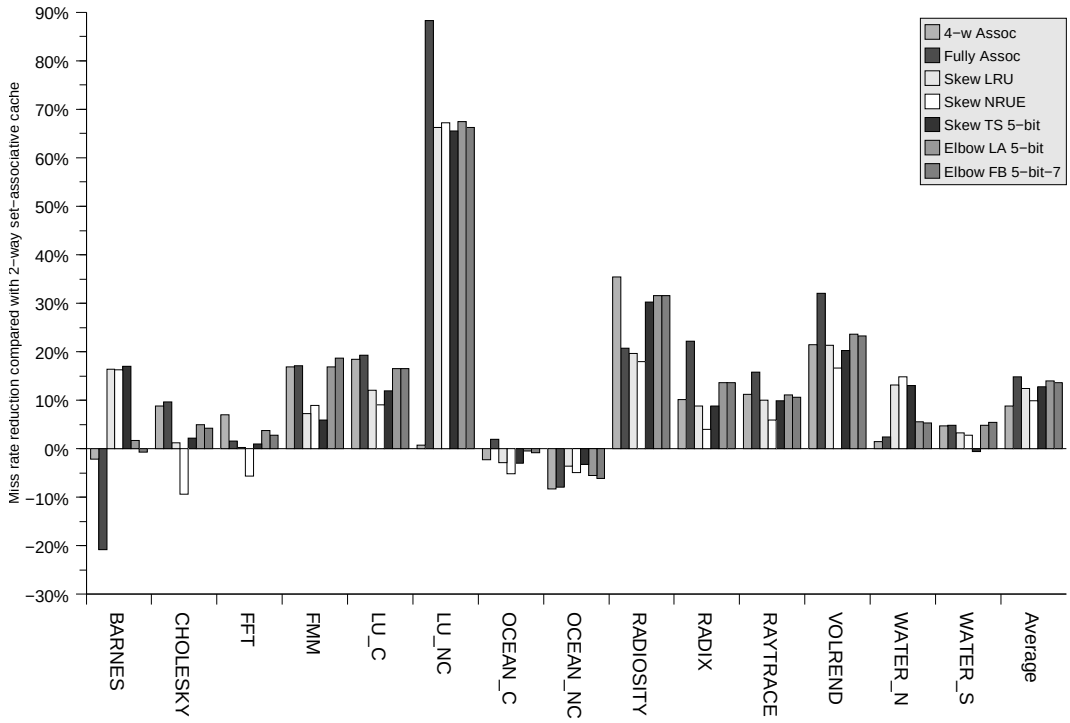


Figure 7: SPLASH-2 suite miss rate reduction results for different 16 KB cache configurations.

of a similar skewed cache using perfect LRU, for almost all of the benchmarks.

8.2 Feedback Cache

The attention is now turned towards the feedback variant of the elbow cache. Unlike the lookahead cache, the feedback cache is easily implementable, even for more than one replacement step. A three bit step counter can record up to seven consecutive transformations during a replacement. This should be enough to catch most potential performance improvements. In fact, a two-bit, three-step maximum variant might do almost just as well. Like before, averages for the two suites are presented (in Figure 9).

As shown in the bar-charts presented in Figure 7 and 8, the overall performance is comparable to the lookahead cache. The average result for the 5-bit, 7-step feedback cache is approximately 11.9% and 6.7% in miss rate reduction, compared with a two-way set-associative cache, for the SPLASH-2 and reduced SPEC 2000 suites respectively.

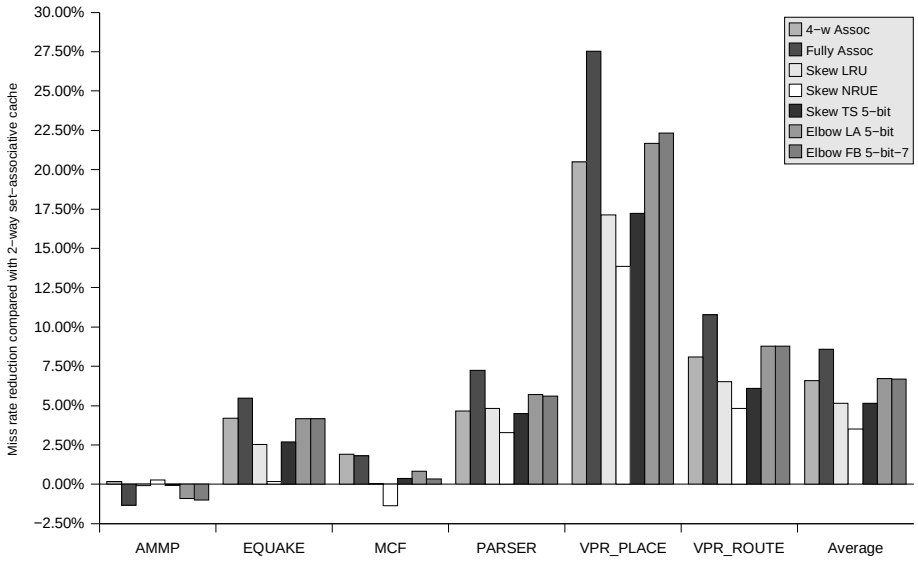
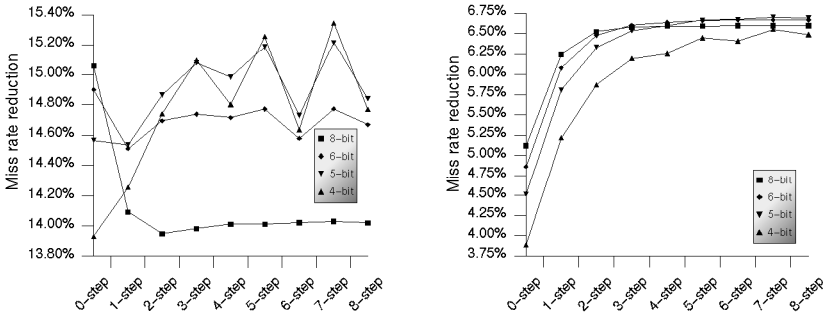


Figure 8: Reduced SPEC 2000 suite miss rate reduction results for different 16 KB cache configurations.



(a) SPLASH-2 (b) Reduced SPEC 2000

Figure 9: Average miss rate reductions vs a 2-way set-associative cache for a feedback cache with different timestamp resolutions.

From the graphs in Appendix A it is apparent that, although the feedback method gives less reduction in miss rate per transformation step, it can reach the same total levels of miss reductions as a single-step lookahead cache by repeatedly applying these transformations.

The full set of results are plotted in Figures 13b and 15, Appendix A. Note the volatile behavior of the low-resolution timestamp traces.[5] This is different from the lookahead cache.

[5]There seems to be some kind of 'sawtooth'-effect evident in some of the results. This is possibly due to loops in the replacement path that cause the newly inserted item to be evicted again immediately, nullifying the effect of the cache. This can easily be avoided by limiting the maximum number of transformation steps to an odd number since such an eviction can only happen after an even number of transformations.

8.3 Different Cache Sizes

It is probable that many of the benchmarks might behave differently if the size of the cache is changed. This is especially true since the measurements, so far, have used a relatively small 16 KB cache. Therefore, the same comparisons as previously, but with cache sizes of 8 and 32 KB, were made. The block length was the same, 32 bytes, as before. Also, the same number of bits was used for the timestamp (5), although this results in a different number of cache allocations per tick (16 and 64 instead of 32, for the 8 KB and 32 KB caches respectively).

For the 8 KB cache there was an average increase in miss ratio reduction compared with the 16 KB case. This is as expected since the smaller size means more potential conflicts that can be avoided by the improved cache architectures.

For the same reason, there is a lower average miss rate reduction in the bigger 32 KB cache. There are also some anomalies for certain benchmarks with regards to their earlier performance. BARNES, for example, suddenly gives large reductions for all architectures, while WATER_N starts to penalize the skewed and elbow caches while still benefiting from increased associativity. Also AMMP from the reduced SPEC 2000 suite starts utilizing the more advanced architectures at 32 KB.

Some of the benchmarks are discussed in the next section. A table summarizing all results can be found in Appendix B.

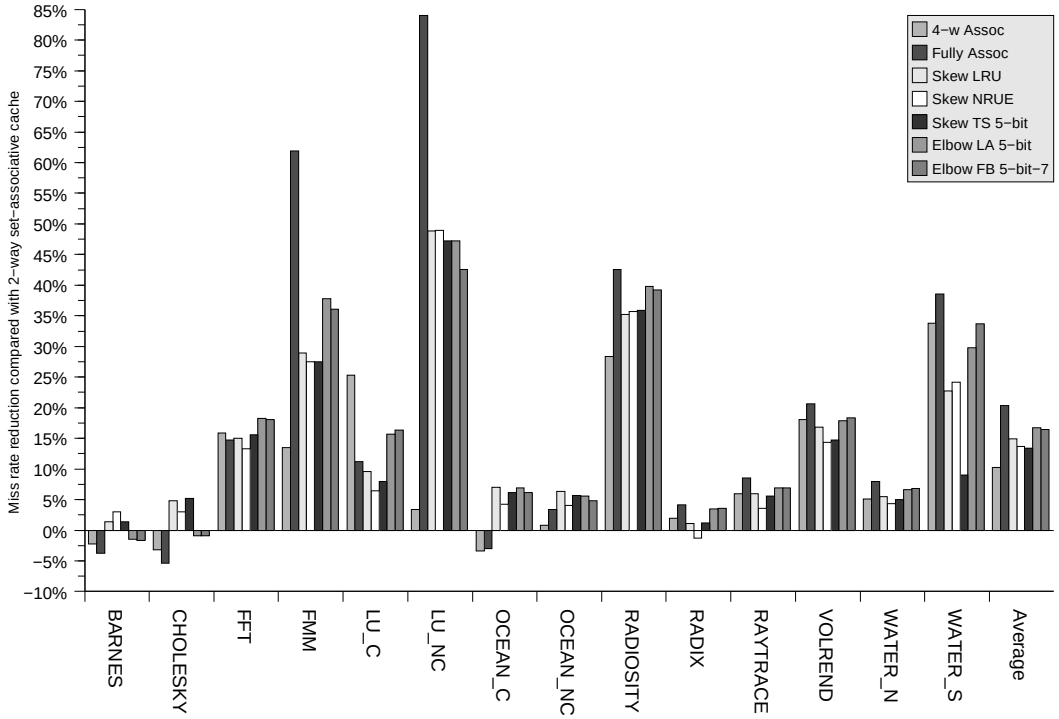
8.4 Benchmark Properties

As was noted in previous sections, some benchmarks exhibit properties that differs from the mainstream behavior. Some of the benchmarks are examined here.

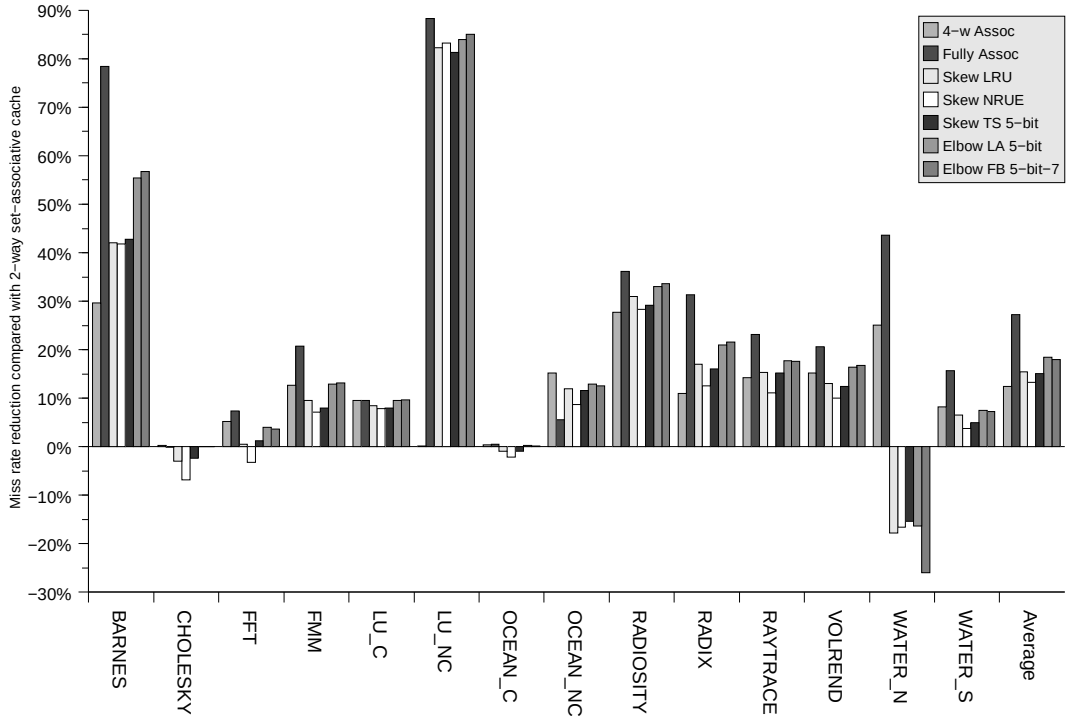
LU_NC From the SPLASH-2 suite. This benchmark seems to suffer heavily from conflict misses, with many blocks competing for only a few sets. The miss rate reduction for this benchmark is therefore very large for the caches that greatly reduces conflicts, like the fully associative cache and skewed caches, while the four-way set-associative cache gains very little. It has a relatively small working set that would mostly fit in the cache if there were no conflicts. LU_NC has a very volatile behavior in some of the tests performed in this thesis.

Barnes For small sizes, the fully associative cache is severely penalized in this benchmark, indicating a very large number of *replacement*-misses. Obviously, LRU is a bad policy to use here. Skewed caches on the other hand, benefits from the reduced number of *mapping* conflicts. Elbow caches increases the degree of replacement freedom and performs worse than a plain skewed cache on this benchmark since it more closely resembles a fully associative LRU cache. Also OCEAN_NC and WATER_N exhibit similar, but less pronounced behavior.

AMMP This benchmark from the reduced SPEC 2000 suite stands out as being the only one where the elbow cache (together with the fully associative cache) is actually the *worst* architecture in the 16 KB cache test. It also has the by far largest overall miss ratio ($\approx$50%) of all the benchmarks. The problem is likely to be the reductions made to shorten the execution time for the benchmark. In fact, *the reduced AMMP benchmark does not seem to compute anything*, only initializing the computation. Thus the access patterns produced is likely to be very different from that of the real AMMP benchmark. Nevertheless, it is interesting as an example of an application that does not benefit from a high degree of associativity and (close to) LRU replacement.

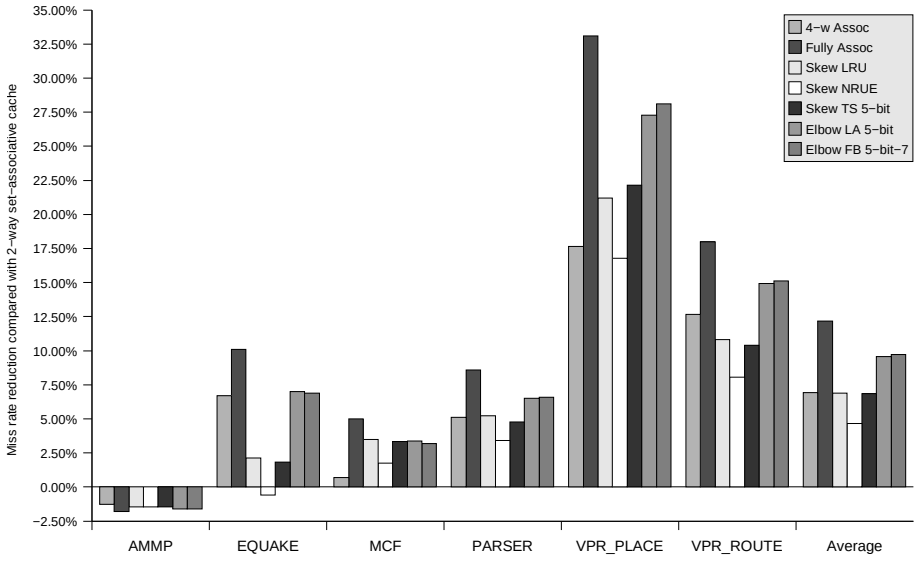


(a) 8 KB cache size

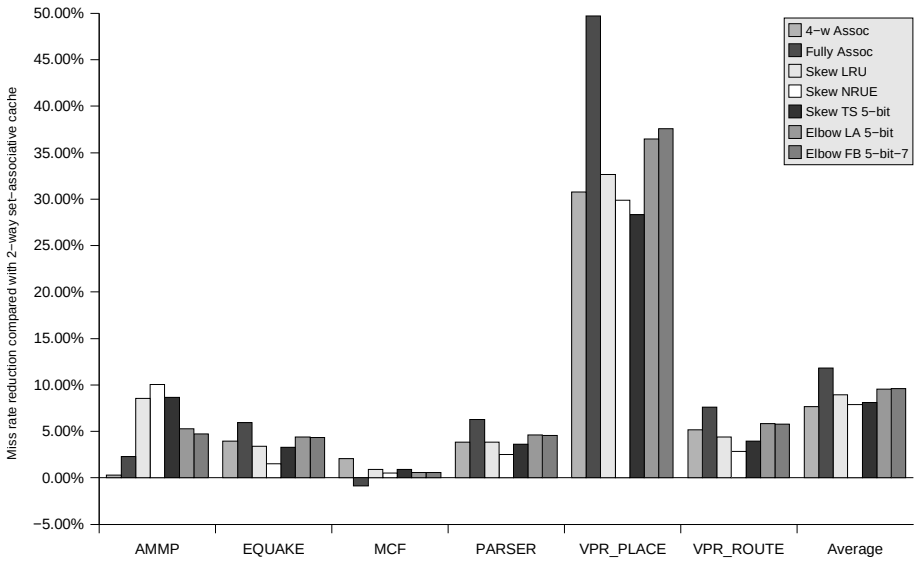


(b) 32 KB cache size

Figure 10: Miss rate reductions for SPLASH-2.



(a) 8 KB cache size



(b) 32 KB cache size

Figure 11: Miss rate reductions for SPEC 2000-reduced.

8.5 Ranking the Cache Configurations

An obvious problem is the large difference in magnitude of the obtained results. Because of this, some benchmarks affects the averages much more than others, somewhat obscuring the relative merits of each cache design. Therefore, as a complement to averaging the miss rate reductions, each cache configuration was also ranked with respect to the others for every benchmark. Table 4 shows the results of this ranking. Ten different configurations were tested and the best configuration

for each benchmark got 10 points, the second best got 9 points and so on down to the worst configuration that only got 1 point. Since there are 20 benchmarks, the maximum possible score is 200 points and the minimum is 20. Half points are awarded for equal results. Figure 12 shows a summary of the average miss rate reductions compared with the average ranking score for each configuration. In the ranking chart, an average score of 10 points means 'always best' (best configuration for all of the benchmarks) and a 1-point average means 'always worst'.

Cache Architecture	8 KB	16 KB	32 KB
2-way set-associative	47.0	53.0	35.5
4-way set-associative	94.0	104.0	106.0
8-way set-associative	110.0	138.0	147.0
16-way set-associative	125.0	147.0	165.0
Fully associative	146.0	155.0	166.0
Skew LRU	112.5	93.5	88.0
Skew NRUE	85.0	74.0	57.0
Skew TS 5-bit	106.5	86.5	80.0
Elbow LA 5-bit	139.0	129.0	129.5
Elbow FB 5-bit 7-step	135.0	120.0	126.0

Table 4: Total ranking scores for 10 different cache architectures applied to 20 benchmarks from the SPLASH-2 and reduced SPEC 2000 suites.

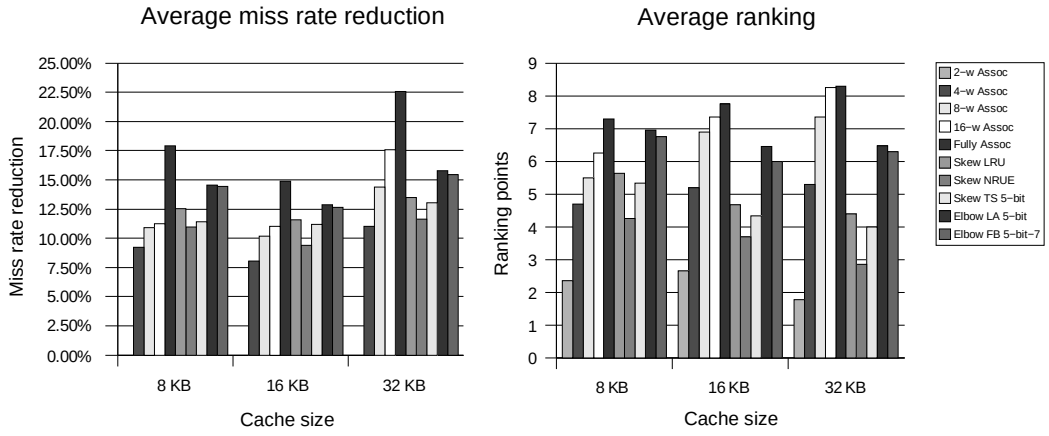


Figure 12: Comparison between miss rate reduction and relative ranking averages for the different cache architectures on the combined SPLASH-2 and reduced SPEC 2000 benchmarks.

The ranking system gives similar results to the previously shown results for the miss rate reductions. Higher associativity gives better performance, the timestamped skewed cache almost reaches similar performance as a LRU skewed cache, and the elbow caches compares very favorably to ordinary skewed and set-associative caches. The only difference is that both 8-way and 16-way set-associativity ranks better than elbow caching for the 16 and 32 KB sized caches.

9 Conclusions

This thesis have introduced the elbow cache, a design to further reduce the number of conflicts in a skewed-associative cache by dynamically reallocating conflicting data.

In order to achieve good performance it is essential to have a better replacement policy than the ones used in skewed caches so far. Therefore, a replacement policy based on timestamps to approximate LRU replacement is suggested. To test whether this idea works or not a timestamp-based cache was compared to caches with traditional replacement policies. This comparison showed that a skewed cache that uses a 5-bit timestamp generally outperforms an otherwise similar skewed cache based on NRUE, and approaches the performance of a skewed cache with full LRU ordering.

The elbow cache comes in two flavors: the lookahead cache and the feedback cache. While both are based on the same idea, the actual way of finding what block to replace is different. The lookahead cache is better in theory, but a realistic implementation can only reallocate one data item each replacement, while the feedback design can apply cache transformations repeatedly by storing temporary victims in the write buffer.

The test results, summarized in the table in Appendix B, compares the two types of elbow caches and a skewed cache using timestamps against other, more traditional, cache architectures. Two benchmark suites; SPLASH-2 and parts of the reduced SPEC 2000 was used as test data. Caches of 8, 16, and 32 KB sizes were simulated.

Both the (single-step) lookahead and the (7-step) feedback caches showed similar performance, with the lookahead having a slight advantage. In the total average, they both outperform all but the fully associative cache for 8 and 16 KB cache sizes, and is beaten only by the fully associative and the 16-way set-associative cache for a cache size of 32 KB.

The lookahead design is slightly more expensive in terms of hardware but gives better performance and does not use up any space in the write buffer. It is also easier to model since the implications of limited buffer space do not need to be considered. A feedback design is cheaper but might loose performance if the number of replacement steps gets reduced because of write buffer overflows. To fully evaluate such design a more fine-grained model that takes the timing of instruction issues, memory latencies and write buffer depth, into account must be employed. A decision which cache design to choose should be based on architectural and implementation considerations as well as expected performance. Issues regarding the resolution and length of the timestamps and what limit to use for the maximum number of transformation steps etc, depends on parameters such as cache size, block length and write buffer depth. These issues should be decided for each configuration based on simulations of a representative set of applications.

The elbow cache suffers some of the same limitations as a standard skewed-associative cache does, but gives better performance at only a slightly higher cost in hardware complexity. A final recommendation must therefore be that the elbow cache should be regarded as a viable alternative whenever a skewed cache is considered for use.

10 Suggestions for Future Work

The work presented in this thesis focuses on evaluating a level-1 data cache typical of many current CPU:s. Though an important special case, it could also be interesting to test this theory against other types of caches like instruction caches, TLB:s and level-2 caches. Also different configurations with regard to cache and block size, associativity and write policy, might be tested as well. An interesting idea is to use the elbow cache on a NUMA-architecture and tag remote memory cached with an extra 'remote'-bit, marking that block as extra precious. This will cause the elbow cache to quickly move all such blocks out of any hot-spots forming.

To use timestamps as a general metric for measuring distance to the last access seems to be useful in many ways such as in the RASCAL algorithm [6]. This thesis has shown that they can be used as a viable alternative to LRU replacement in skewed and elbow caches. It would also be possible to use timestamps in more conventional, highly associative caches.

Acknowledgments

Many thanks go to the people in the UART-group (Uppsala Architecture Research Team) at Uppsala University that helped me with this thesis. Especially my advisors: team-captain Erik Hagersten and Martin Karlsson. Erik Berg helped me get SIMICS up and running and Zoran Radović assisted me solving some LaTeX/LyX issues. Dan Wallin spotted a bug in one of the SPLASH-2 traces and supplied me with an endless stream of irrelevant—but funny—e-mails.

References

[1] A. Agarwal and S. D. Pudar. Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 179–190, May 1993.

[2] AJ KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization*, pages 73–82, 2000.

[3] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. In *IEEE Transactions on Computers*, May 1997.

[4] J. L. Hennessy and D. A. Pattersson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[5] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[6] M. Karlsson and E. Hagersten. Timestamp-based Selective Cache Allocation. In *Proceedings of the Workshop on Memory Performance Issues*, June 2001. held in conjunction with the 28th International Symposium on Computer Architecture (ISCA28).

[7] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and Håkan Grahn. SimICS/sun4m: A Virtual Workstation. In *Proceedings of the Usenix Annual Technical Conference*, pages 119–130, 1998.

[8] S. Woo, M. Ohara, E. Toorie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, 1995.

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

[10] A. Seznec. A new case for skewed-associativity. Internal Publication No 1114, IRISA-INRIA, 1997.

[11] A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE '93, Munich*, pages 305–316, 1993.

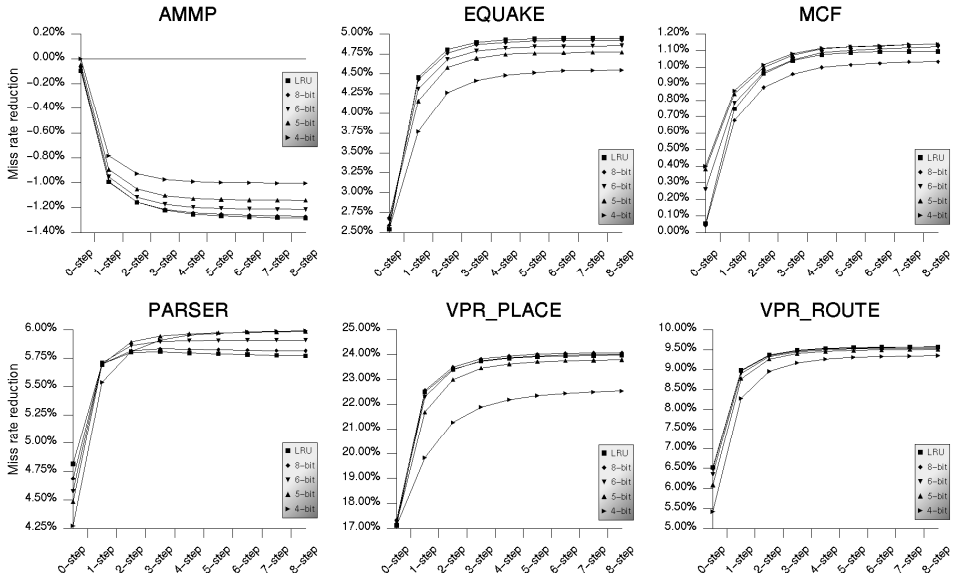
[12] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, 1993.

[13] A. S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, international edition, 1995.

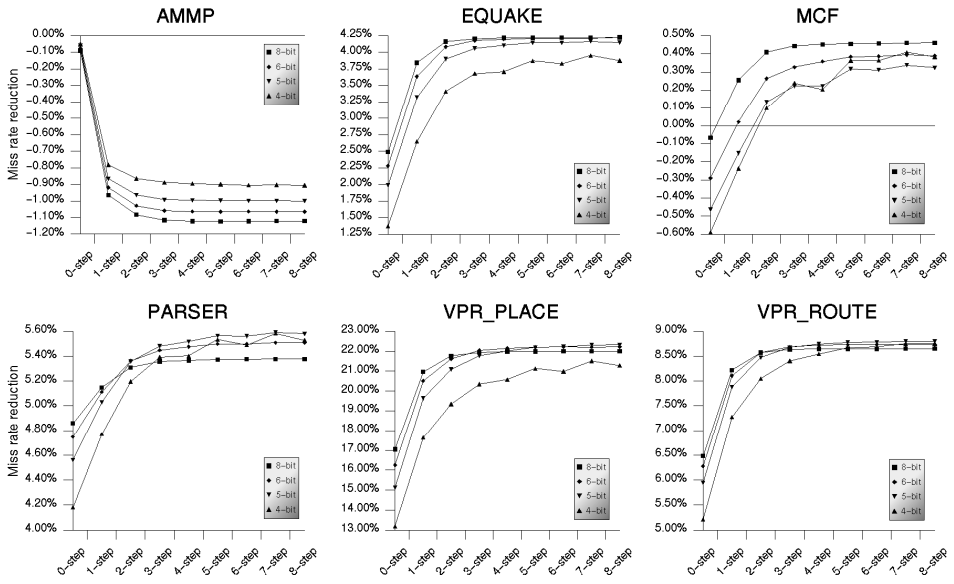
[14] N. Topham and A. Gonzalez. Randomized Cache Placement for Eliminating Conflicts. Technical Report ECS-CSG-37-98, 1998.

A Simulation Results for a 16 KB Cache

Figure 13, 14, and 15 shows the detailed graphs of the 16 KB cache simulations for 0 to 8 transformation steps.



(a) Lookahead cache



(b) Feedback cache

Figure 13: Miss rate reduction of the reduced SPEC 2000 benchmark suite for n-step two-way elbow caches compared with a two-way set-associative cache.

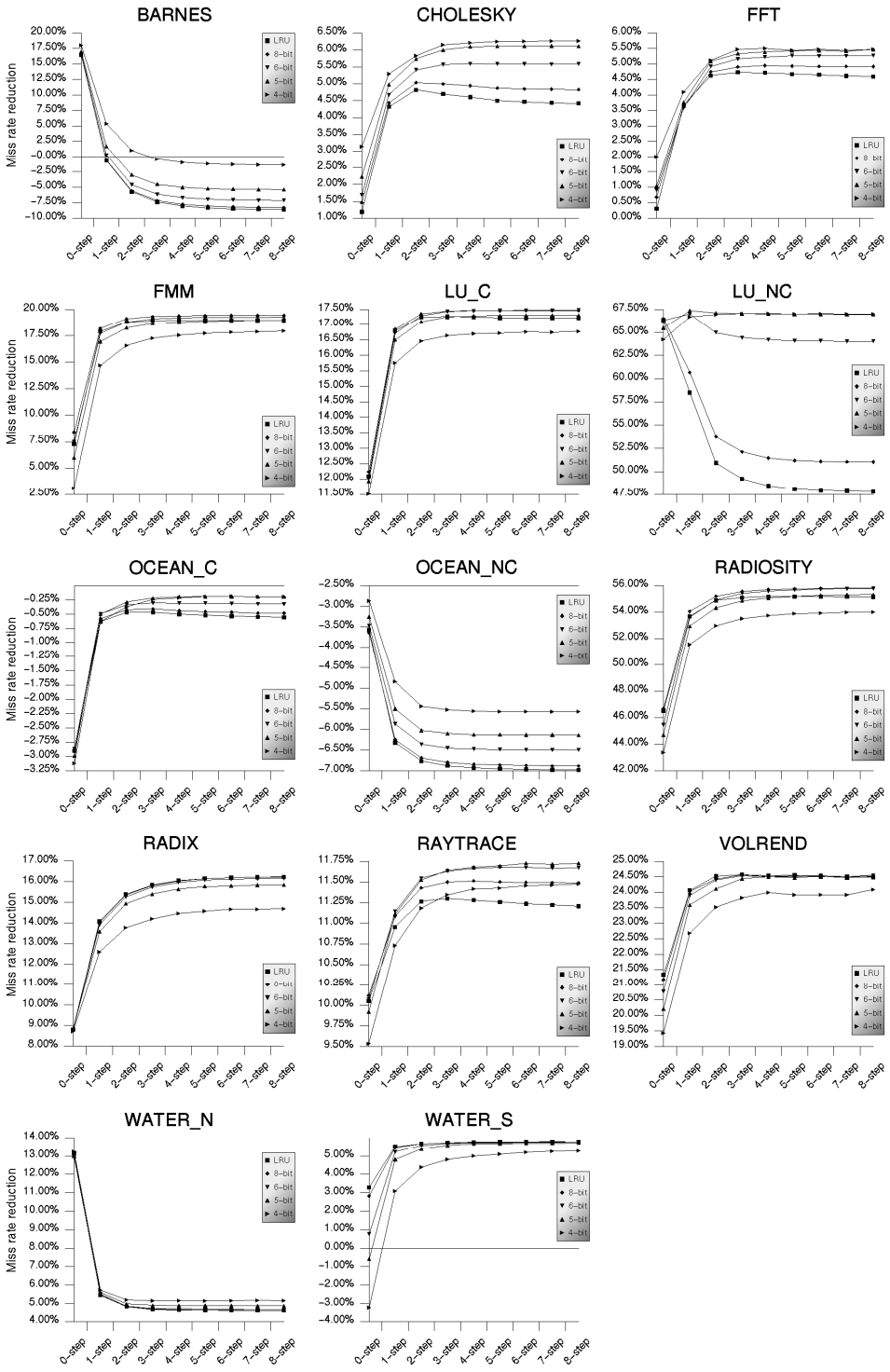


Figure 14: Miss rate reduction of the SPLASH-2 benchmark suite for n-step two-way lookahead elbow cache compared with a two-way set-associative cache.

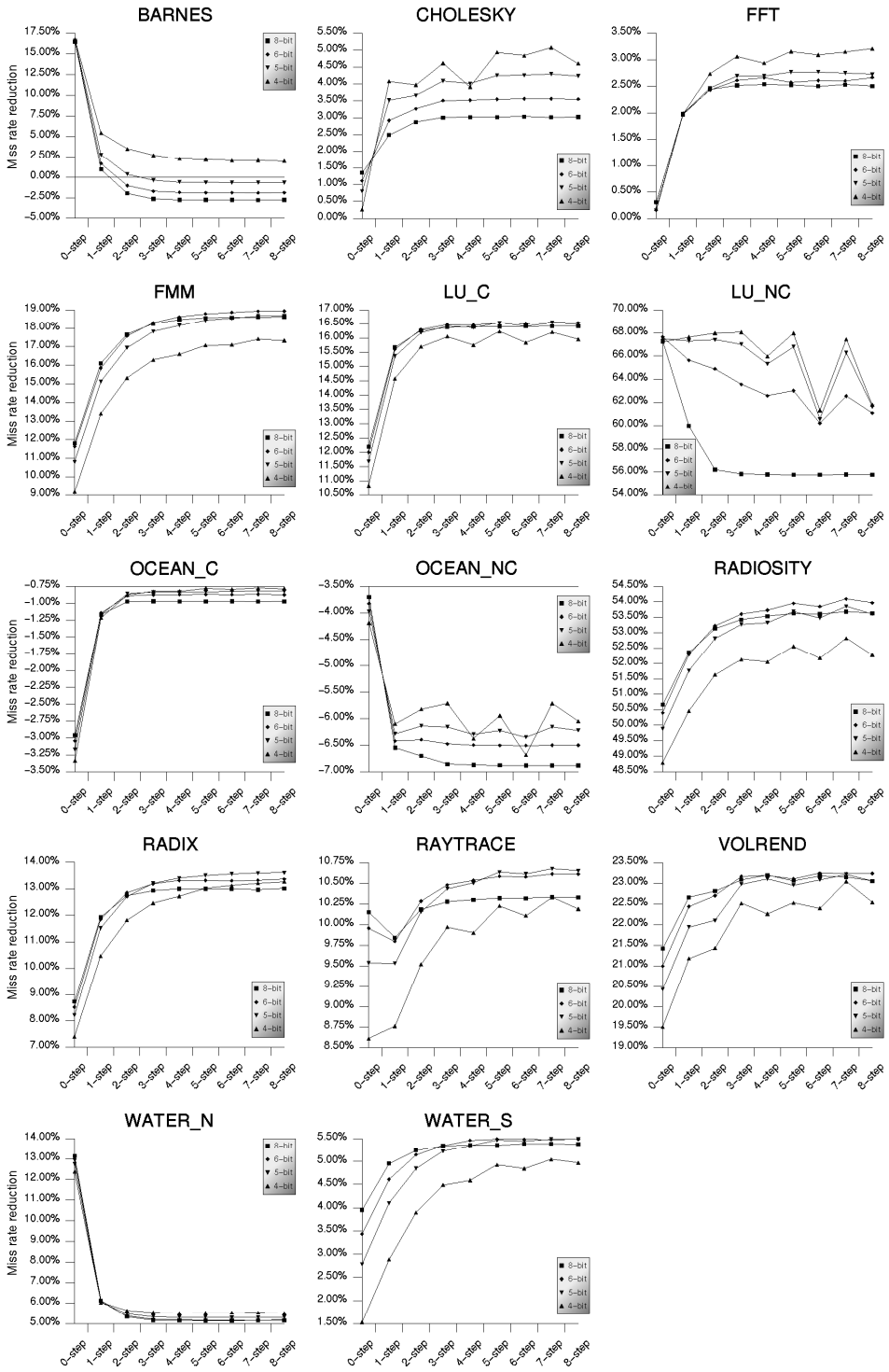


Figure 15: Miss rate reduction of the SPLASH-2 benchmark suite for n-step two-way feedback elbow cache compared with a two-way set-associative cache.

B Results Summary

Benchmark / ***Cache architecture***

Cache architecture			SPLASH−2															SPEC 2000−reduced							
			BARNES	CHOLESKY	FMM	FFT	LU_C	LU_NC	OCEAN_C	OCEAN_NC	RADIOSITY	RADIX	RAYTRACE	VOLREND	WATER_N	WATER_S	Average	AMMP	EQUAKE	MCF	PARSER	VCF_PLACE	VCF_ROUTE	Average	Tot. Average
8 KB	Miss rate	2−w	4.66%	12.37%	1.11%	5.42%	1.14%	12.20%	17.45%	10.87%	2.28%	4.41%	2.86%	0.93%	1.73%	0.56%	5.57%	51.49%	5.55%	30.75%	12.55%	4.84%	9.26%	19.07%	9.62%
	Miss rate reduction	4−w	−2.25%	−3.18%	13.46%	15.83%	25.35%	3.43%	−3.41%	0.81%	28.35%	1.97%	5.98%	18.07%	5.09%	33.81%	10.24%	−1.27%	6.70%	0.70%	5.11%	17.66%	12.65%	6.92%	9.24%
		8−w	−3.21%	−3.38%	12.50%	15.49%	23.59%	3.75%	−3.36%	3.89%	32.19%	3.02%	6.90%	22.27%	6.99%	36.47%	11.22%	−1.69%	8.95%	0.49%	7.30%	29.34%	16.40%	10.13%	10.90%
		16−w	−3.55%	−4.23%	2.83%	15.30%	18.68%	11.38%	−3.20%	3.99%	36.79%	3.78%	8.12%	23.01%	7.50%	38.15%	11.33%	−1.70%	10.06%	0.56%	8.12%	32.03%	17.35%	11.07%	11.25%
		Fully Assoc.	−3.73%	−5.39%	61.93%	14.69%	11.22%	84.01%	−2.97%	3.40%	42.59%	4.12%	8.56%	20.62%	7.94%	38.52%	20.39%	−1.77%	10.12%	5.01%	8.60%	33.09%	17.99%	12.17%	17.93%
		Skewed LRU	1.35%	4.84%	28.89%	15.05%	9.58%	48.81%	6.97%	6.35%	35.26%	1.06%	5.94%	16.87%	5.51%	22.75%	14.95%	−1.47%	2.14%	3.48%	5.22%	21.19%	10.82%	6.90%	12.53%
		Skewed NRUE	2.96%	3.01%	27.48%	13.26%	6.41%	48.94%	4.24%	4.06%	35.72%	−1.24%	3.60%	14.32%	4.30%	24.14%	13.66%	−1.44%	−0.59%	1.74%	3.43%	16.79%	8.06%	4.67%	10.96%
		Skewed TS	1.39%	5.15%	27.49%	15.61%	7.95%	47.23%	6.12%	5.66%	35.89%	1.17%	5.55%	14.70%	4.99%	8.98%	13.42%	−1.47%	1.82%	3.33%	4.77%	22.15%	10.42%	6.84%	11.44%
		Elbow LA 5−bit	−1.43%	−0.85%	37.80%	18.23%	15.71%	47.27%	6.96%	5.54%	39.78%	3.45%	6.94%	17.91%	6.63%	29.80%	16.70%	−1.60%	7.01%	3.40%	6.53%	27.28%	14.93%	9.59%	14.56%
		Elbow FB 5−bit−7−step	−1.66%	−0.86%	36.09%	18.09%	16.33%	42.58%	6.18%	4.83%	39.25%	3.56%	6.92%	18.36%	6.85%	33.70%	16.44%	−1.62%	6.89%	3.17%	6.59%	28.10%	15.13%	9.71%	14.42%
16 KB	Miss rate	2−w	3.40%	7.35%	0.43%	3.41%	0.93%	11.81%	13.60%	8.53%	0.97%	3.16%	2.07%	0.69%	1.57%	0.32%	4.16%	50.74%	4.69%	26.13%	10.38%	2.50%	7.29%	16.95%	8.00%
	Miss rate reduction	4−w	−2.11%	8.89%	16.91%	7.08%	18.48%	0.78%	−2.27%	−8.24%	32.53%	10.17%	11.28%	21.50%	1.48%	4.78%	8.66%	0.19%	4.19%	1.90%	4.67%	20.51%	8.08%	6.59%	8.04%
		8−w	−7.40%	9.50%	18.41%	7.47%	18.70%	1.15%	−0.79%	−10.16%	56.41%	17.09%	13.41%	26.23%	2.06%	5.42%	11.25%	−0.91%	5.20%	2.42%	6.18%	23.73%	9.79%	7.73%	10.20%
		16−w	−12.57%	10.27%	17.82%	7.11%	18.68%	11.13%	−1.03%	−9.56%	58.88%	19.65%	14.30%	27.96%	2.25%	5.50%	12.17%	−1.29%	5.60%	2.87%	6.80%	26.12%	10.31%	8.40%	11.04%
		Fully Assoc.	−20.76%	9.61%	17.19%	1.57%	19.28%	88.24%	1.98%	−7.96%	60.27%	22.15%	15.79%	32.00%	2.46%	4.85%	17.62%	−1.33%	5.47%	1.80%	7.25%	27.54%	10.76%	8.58%	14.91%
		Skewed LRU	16.45%	1.19%	7.31%	0.30%	12.09%	66.31%	−2.90%	−3.57%	46.52%	8.80%	10.05%	21.33%	13.16%	3.29%	14.31%	−0.10%	2.54%	0.06%	4.81%	17.12%	6.52%	5.16%	11.56%
		Skewed NRUE	16.30%	−9.35%	8.94%	−5.59%	9.08%	67.18%	−5.17%	−4.95%	45.83%	4.00%	5.99%	16.62%	14.87%	2.77%	11.89%	0.28%	0.18%	−1.35%	3.29%	13.84%	4.83%	3.51%	9.38%
		Skewed TS	16.99%	2.23%	5.98%	1.04%	11.94%	65.47%	−2.99%	−3.25%	44.69%	8.80%	9.92%	20.21%	12.99%	−0.58%	13.82%	−0.05%	2.69%	0.38%	4.48%	17.22%	6.08%	5.13%	11.21%
		Elbow LA 5−bit	1.70%	4.98%	16.94%	3.77%	16.50%	67.41%	−0.49%	−5.50%	52.94%	13.58%	11.11%	23.60%	5.62%	4.81%	15.50%	−0.89%	4.16%	0.84%	5.69%	21.69%	8.77%	6.71%	12.86%
		Elbow FB 5−bit−7−step	−0.67%	4.29%	18.68%	2.75%	16.55%	66.26%	−0.82%	−6.16%	53.86%	13.58%	10.67%	23.21%	5.29%	5.49%	15.21%	−1.00%	4.16%	0.34%	5.59%	22.31%	8.80%	6.70%	12.66%
32 KB	Miss rate	2−w	1.62%	5.64%	0.28%	2.89%	0.82%	11.67%	13.20%	7.56%	0.46%	2.30%	1.47%	0.49%	0.52%	0.24%	3.51%	50.23%	4.18%	23.73%	8.67%	1.10%	5.78%	15.62%	7.14%
	Miss rate reduction	4−w	29.63%	0.34%	12.67%	5.27%	9.51%	0.11%	0.45%	15.19%	27.75%	10.98%	14.22%	15.25%	25.10%	8.26%	12.48%	0.28%	3.98%	2.07%	3.85%	30.77%	5.20%	7.69%	11.04%
		8−w	51.12%	0.14%	16.60%	7.10%	9.56%	0.06%	0.53%	17.13%	34.04%	14.87%	19.53%	19.20%	22.48%	8.83%	15.80%	0.79%	5.07%	2.79%	5.28%	45.94%	6.83%	11.12%	14.39%
		16−w	70.21%	0.00%	18.95%	7.44%	9.40%	11.30%	0.53%	14.13%	35.96%	27.62%	21.44%	20.43%	33.15%	9.43%	20.00%	2.69%	5.40%	3.30%	5.77%	47.30%	7.31%	11.96%	17.59%
		Fully Assoc.	78.40%	−0.12%	20.75%	7.41%	9.49%	88.23%	0.55%	5.63%	36.15%	31.35%	23.12%	20.60%	43.60%	15.65%	27.20%	2.30%	5.96%	−0.87%	6.29%	49.70%	7.63%	11.83%	22.59%
		Skewed LRU	42.07%	−2.96%	9.54%	0.47%	8.47%	82.22%	−0.94%	11.94%	30.96%	17.01%	15.27%	13.05%	−17.79%	6.58%	15.42%	8.56%	3.38%	0.89%	3.83%	32.64%	4.38%	8.94%	13.48%
		Skewed NRUE	41.81%	−6.87%	7.13%	−3.24%	7.89%	83.20%	−2.12%	8.67%	28.31%	12.61%	11.13%	10.02%	−16.55%	3.72%	13.27%	10.04%	1.53%	0.50%	2.50%	29.86%	2.86%	7.88%	11.65%
		Skewed TS	42.74%	−2.40%	7.94%	1.22%	7.93%	81.32%	−0.90%	11.62%	29.16%	16.05%	15.24%	12.48%	−15.32%	4.97%	15.15%	8.68%	3.30%	0.92%	3.62%	28.31%	3.96%	8.13%	13.04%
		Elbow LA 5−bit	55.45%	0.02%	12.96%	3.99%	9.56%	83.96%	0.26%	12.91%	33.05%	21.05%	17.77%	16.38%	−16.28%	7.48%	18.47%	5.28%	4.39%	0.55%	4.64%	36.47%	5.83%	9.53%	15.79%
		Elbow FB 5−bit−7−step	56.73%	−0.02%	13.15%	3.59%	9.70%	85.03%	0.17%	12.54%	33.56%	21.58%	17.61%	16.83%	−25.96%	7.32%	17.99%	4.75%	4.34%	0.59%	4.58%	37.60%	5.79%	9.61%	15.47%

Table 5: Table of miss rate reductions for different cache architectures and sizes compared with a two-way set-associative cache of similar size.

C Estimating Block Survival Time

A timestamped block that remains unused for a long time might cause aliasing. To avoid this and to find a good trade-off for timestamp resolution, is is necessary to estimate an upper bound for how long a block might reside unused in the cache. Since cache allocations are used as time unit, there is one allocation per timer tick. By assuming that the replacements occur randomly[6] throughout the banks, statistical means can be used for such an estimation.

Using probability theory, the *probability function* is (geometrical distribution)

$$p(k) = q^k p \qquad p = q - 1,$$

where p is the probability for a specific block to be replaced each tick and $p(k)$ is the probability of exactly k number of replacements before this occurs. The *distribution function*

$$F(k) = 1 - \sum_{i=1}^{k} p(i)$$

is then the probability for an unused block to remain in the cache after k number of replacements. For a cache consisting of N blocks the probabilities are

$$p = \frac{1}{N} \qquad q = \frac{N-1}{N}$$

and the distribution function becomes

$$F(k) = 1 - \sum_{i=1}^{k} \frac{(N-1)^i}{N^{i+1}} = 1 - \frac{1}{N} \sum_{i=1}^{k} \left(\frac{N-1}{N}\right)^i .$$

Example: Probability of a block surviving $2N$ replacements.

Assume an $N = 512$ block cache and calculate the probability of a cache block surviving for more than $2N$ replacements;

$$F(2N) = 1 - \frac{1}{N} \sum_{i=1}^{2N} \left(\frac{N-1}{N}\right)^i = 1 - \frac{1}{512} \sum_{i=1}^{1024} \left(\frac{511}{512}\right)^i \approx 13.7\%.$$

To avoid aliasing the timer must contain enough bits. With the formula above it is easily shown that very few cache lines will survive untouched for more then $4N$ ($F(4 * 512) \approx 1.9\%$) replacements.[7] Therefore $log_2(4N)$ bits can be considered enough. In the 512 cache block example this is equivalent to $log_2(2048) = 11$ bits.

[6] This is generally not true in a cache even though the elbow cache, by using hashing functions, comes closer to random. However, since a cache usually is biased towards replacing old blocks and an upper bound for the survival time is to be found, the result from the calculation will probably be overly conservative anyway. The real survival times are likely to be much shorter.

[7] While running applications that have a small working set that fits entirely in the cache - e.g. only conflict misses occurs, blocks may stay unused in the cache for a long period of time experiencing aliasing. This problem is not very severe however, since these blocks are rarely used anyway.

Master theses from the Uppsala Architecture Research Team

UPTEC F 00 093 **Zoran Radović:** ***DSZOOM – Low Latency Software-Based Shared Memory***

UPTEC F 01 017 **Dan Wallin:** ***Performance of a High-Accuracy PDE Solver on a Self-Optimizing NUMA Architecture***

UPTEC F 02 033 **Mathias Spjuth:** ***Refinement and Evaluation of the Elbow Cache***

UPPSALA
UNIVERSITY

Department of Information Technology, Uppsala University, Sweden