# An optimal replacement algorithm for balancing multi-module caches

Hans Vandierendonck[1] and Koen De Bosschere[1]

[1] Department of Electronics and Information Systems (ELIS)
Ghent University
Ghent — Belgium
Comments and suggestions to: {hvdieren,kdb}@elis.rug.ac.be

*Abstract*—

**Multi-module caches consist of multiple parallel caches. Their goal is to increase level 1 data cache hit ratios by allowing more freedom in placing blocks in the cache. Balancing multi-module caches provide the opportunity to reorganise blocks by moving them between the cache modules, in an attempt to balance the amount of data stored in each module. This capability allows for even higher hit ratios. We present an algorithm to simulate optimal replacement in balancing multi-module caches. This algorithm is optimal since it minimizes the number of cache misses However, it is infeasible to implement it in hardware. We discuss the effects of the set index functions used in the modules and the ways they interact. The algorithm is used to investigate three cache configurations and provide some performance limits for balancing multi-module caches.**

*Keywords*— **cache memory, multi-module cache, optimal replacement algorithm**

## I. Introduction

Current microprocessors use a memory hierarchy to overcome the high latencies of the main memory. A key aspect of any memory hierarchy is that the majority of memory references should hit in the first-level cache. For those references that miss, the delay to the second-level cache is relatively high (e.g. 12 cycles on the UltraSPARC III [HL99]) causing the processor to stall execution. Furthermore, the latency to cache levels further away from the processor core will increase in the future, making the memory hierarchy a key issue in the design of high-performance microprocessors.

The average latency of the memory hierarchy can be reduced by increasing the hit ratio of the level 1 cache. Two techniques to this extent have been proposed which caught our attention: locality-sensitive replacement policies and skewed-associative caches.

Locality-sensitive replacement policies handle temporal and spatial locality separately, without causing any overhead when data exhibit either temporal non-spatial locality or when it has spatial non-temporal locality [GAV95, SG99, RTT+98, TFMP95]. Several of these investigations propose to use a level 1 data cache organisation using multiple, parallel working caches. These cache organisations are termed *multi-lateral* [Tam99] or *multi-module* [SG99] caches. We use the term multi-module cache in this paper. By varying the block size and the degree of associativity, the cache modules are optimised to effectively exploit either temporal or spatial locality or the lack thereof. Often, one of these modules is chosen to be a fully associative cache. This choice clearly has a big impact on the possible performance of multi-module caches. However, little is known about the consequences of this choice. It is also not clear what the effect would be when the fully associativity module would be replaced by a set-associative one, or what the interaction between the set index functions is like. We present a theoretical framework that allows reasoning about these interactions.

The literature also describes the skewed-associative cache [Sez93]. This is another cache structure that holds high promise to reduce conflict misses. A skewed-associative cache is a multi-module cache where each module is direct mapped and has a different index function. Much like other multi-module caches, there is little knowledge about the performance limits of the skewed-associative cache.

The described cache organisations both promise higher hit ratios than traditional set-associative caches. However, a good understanding of their benefits and their limitations is needed before they can profitably be used. The influence of the cache organisation needs to be carefully analysed. This is possible using optimal replacement policies. Furthermore, the skewed-associative cache can provide data reorganisation [BS93], which means that data can be moved between the modules of the cache, however this costs a cache miss. To investigate the limits of this effect, we introduce the notion of a balancing multi-module cache. A balancing multi-module cache can perform data reorganisation without incurring cache misses.

For these reasons, we have constructed an algorithm to perform optimal replacement in balancing multi-module and skewed-associative caches. This algorithm minimizes the number of cache misses incurred and is therefore optimal in the same sense as Belady's algorithm is optimal for fully associative caches [Bel66]. It is different from Belady's algorithm since it takes the restrictions imposed by the index functions into account. Furthermore, our theoretical study of

set index functions reveals several interesting effects that can occur and provides a framework to reason about these effects. Since the algorithm is optimal, it uses future information to guide its decisions. It is thus not intended for hardware implementation. However, it can be used to qualify the performance of feasible replacement algorithms and to compare cache organisations.

The remainder of this paper is organised as follows. In section II, we provide a description of multi-module caches. In section III, we investigate set index functions and their interaction or cooperation in a multi-module cache. The optimal replacement algorithm is presented in section IV. It is applied to three cache configurations in section V.

## II. Multi-module caches

A multi-module cache consists of multiple, traditional (e.g. set-associative) caches, called modules. When a cache block is looked up in the multi-module cache, all modules are probed simultaneously. Each module can have different block sizes, associativity, set index functions or prefetching policies. A multi-module cache is quite different from a multi-level cache hierarchy, since the former consults the caches in parallel and the latter consults them sequentially.

In a balancing multi-module cache, the replacement policy has the capability to selectively move blocks between the modules. This capability should greatly improve the hit ratio. Although it has a high cost and might seem undesirable, its benefits have been repeatedly demonstrated, e.g. in the column-associative cache [AP93] and in the victim buffer-like replacement policy in [RTT+98]. The skewed-associative cache has the same benefits of data reorganisation, but has to incur cache misses to profit from it [BS93].

In this paper, we study balancing multi-module caches with two modules having the same block size. Furthermore, we assume that the modules use the write-allocate policy (i.e. all referenced data has to be placed in the cache). Since we do not optimize the write-back bandwidth, we make no distinction between the write-back and write-through policies.

An example cache is depicted in Figure 1. It has two different index functions, both based on bit selection.

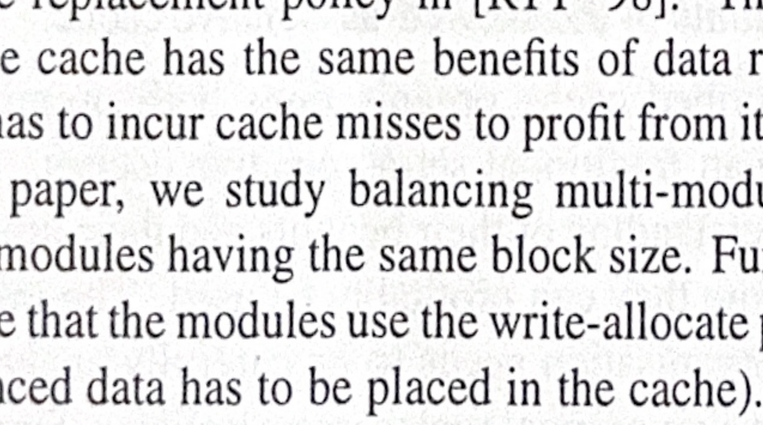


Fig. 1. An example of a multi-module cache.

## III. Properties of set index functions

In a multi-module cache, the set index function in each module can be different, causing the same effects as in the skewed-associative cache to occur. A skewed-associative cache is a multi-module cache. Each module is direct-mapped and has a different set index function with hashing(Figure 2 (left)). As a consequence, if two blocks map to the same frame in one bank of the cache, they do not necessarily map to the same frame in the other banks of the cache (Figure 2 (right)). This effect is called *inter-bank dispersion* or *scattering* of the blocks [Sez93]. Because of the different

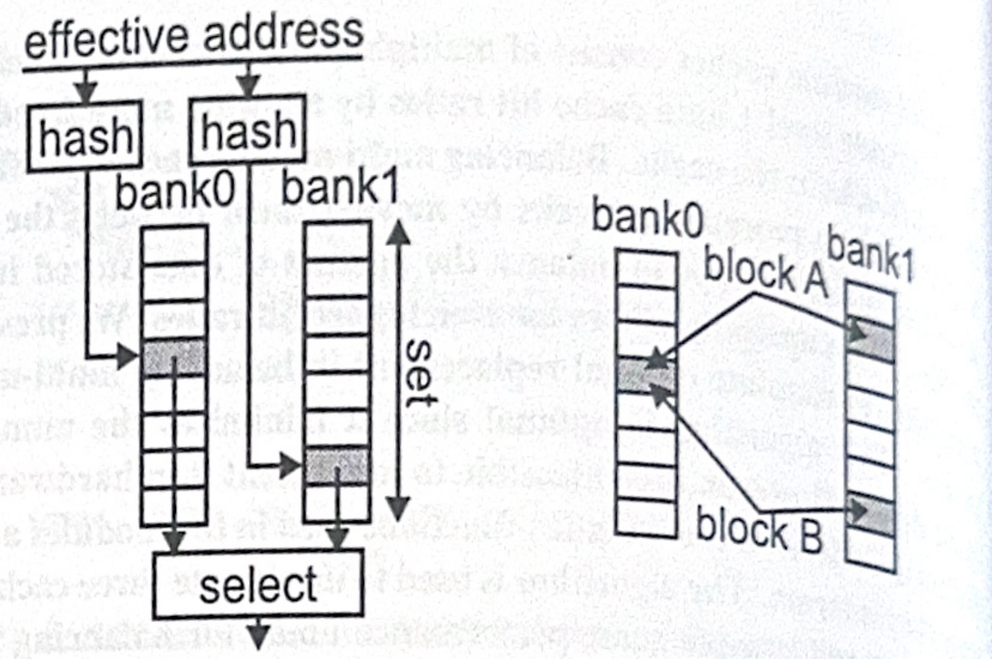


Fig. 2. A skewed-associative cache (left) and conflicts reduction (right).

index functions, a skewed-associative cache is less restrictive than a set-associative cache on the combination of blocks that can be present in the cache at the same time. Clearly, the principles of skewed-associativity also apply when the set index functions are based on bit selection instead of hashing.

An optimal replacement policy for multi-module caches, needs to deal specifically with these complexities. Therefore, we need to have a good understanding of the behaviour of set index functions. In the remainder of this section, we review some known properties of set index functions and introduce some new ones.

### A. Set index functions for set-associative caches

We define a set index function as a mapping from the effective address to a cache set number. A set index function induces a partition on the address space. Two blocks $x$ and $y$ belong to the same partition induced by the set index function $f(.)$ when $f(x) = f(y)$.

For a set-associative cache, all blocks in the same equivalence class map to the same set. These blocks compete for the same cache frames, which results in conflict misses when the associativity is too low. If the cache has $n$ sets, then the distance between two blocks in the same equivalence class is a multiple of $n$ cache blocks.

### B. Set index functions for multi-module caches

When two set index functions are used, the picture is more complicated. First, we inspect an example, namely the multi-module cache presented in section II (Figure 1). Its set index functions are shown in Table I. The part $y$ in the address refers to the block offset and the part $x$ of the address has no relevance to the computed set numbers. The columns labelled $f_{min}$ and $f_{max}$ will be explained below.

TABLE I

SET INDEX FUNCTIONS OF THE EXAMPLE CACHE.

| address | $f_A$ | $f_B$ | $f_{min}$ | $f_{max}$ |
|---|---|---|---|---|
| $x000y$ | 00 | 00 | 000 | 0 |
| $x001y$ | 01 | 00 | 001 | 0 |
| $x010y$ | 10 | 01 | 010 | 1 |
| $x011y$ | 11 | 01 | 011 | 1 |
| $x100y$ | 00 | 10 | 100 | 0 |
| $x101y$ | 01 | 10 | 101 | 0 |
| $x110y$ | 10 | 11 | 110 | 1 |
| $x111y$ | 11 | 11 | 111 | 1 |

The set index functions $f_A$ and $f_B$ enforce more restrictions on the contents of the cache than expected. Through inspection, we see that blocks that can be placed in A-set 0 can only be placed in a limited number of B-sets (namely 0 and 2). Similarly, blocks in B-set 0 can only be placed in two of the four A-sets. Another property is that when a block can be placed in A-set 0, it can never be placed in a B-set where blocks from A-set 2 can only be placed. This is true, since blocks in A-set 0 can only be placed in B-sets 0 and 2 while blocks in A-set 2 can be placed in B-sets 1 and 3. This means that this cache configuration can be broken up in two distinct parts, just like a set-associative cache with 4 sets can broken up in 4 distinct parts. A block can be placed only in one of the parts and there is no interaction between the contents of any of the parts. This break-up of the cache consists of grouping some A-sets and some B-sets together.

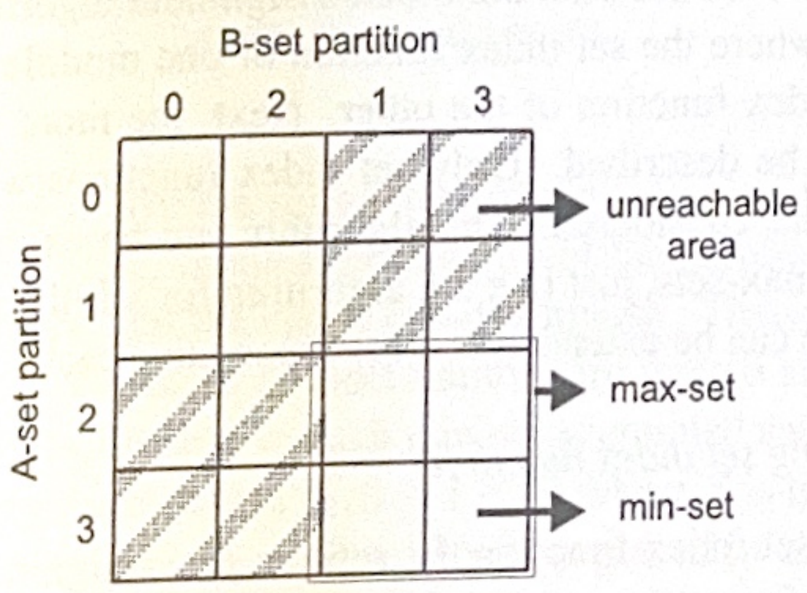


Fig. 3. Address space partition.

It is useful to visualise the partitions of the address space induced by the set index functions. Since there are two set index functions, the equivalence classes are displayed in a two-dimensional picture (Figure 3). The biggest square represents the complete address space. On the vertical axis, the address space is partitioned according to the set index function $f_A$. The horizontal slice of the address space label '0' contains all addresses $a$ for which $f_A(a) = 0$. Using the definition of $f_A$ (Table I), these are all block addresses of the form $0 + 4n$, with $n$ a cardinal number. Likewise, the partition induced by $f_B$ is shown on the horizontal axis. For instance, B-set 2 contains all blocks with address $2 + 8n$ or $2 + 8n + 1$ with $n$ cardinal.

Each cache block is placed in this two-dimensional grid in one A-set and in one B-set, determined by the set index functions $f_A$ and $f_B$. Together, these functions define an even finer partition, namely at their intersections. Two blocks belong to the same (fine) partition when $f_A(x) = f_A(y)$ and $f_B(x) = f_B(y)$. This fine partition is denoted by $f_{min}$. It is significant since it defines the smallest partition of the address space which is noticeable by the multi-module cache. When two blocks $x$ and $y$ have $f_{min}(x) = f_{min}(y)$, inter-bank dispersion or scattering does not occur for these blocks.

When $f_A$ and $f_B$ use bit selection, we define $f_{min}$ as the set index function that selects the union of the bits that are used by $f_A$ and/or $f_B$.

From Table I, we know that not all combinations of A-set numbers and B-set numbers are possible. These areas are shaded in Figure 3. This effect can be seen in the picture as well. Some "fine" sets are grouped together to form "big" sets. Furthermore, each A-set and each B-set is completely contained in exactly one "big" set. These "big" sets are induced by the set index function $f_{max}$. This function selects the bits that are selected by both $f_A$ and $f_B$. The set index function $f_{max}$ tells us the extent to which blocks can be scattered in a multi-module cache. When $f_{max}$ defines only one partition in the address space, then for all A-sets, there exists blocks that can be mapped to any B-set, and vice versa. On the other hand, when $f_{max}$ induces multiple partitions in the address space (as is the case in the example in Table I), only a limited number of combinations between A-sets and B-sets is possible.

### C. Formal definition of $f_{min}$ and $f_{max}$

This section presents the properties of $f_{min}$ and $f_{max}$ for general set index functions, e.g. hashing functions. For this purpose, we use the notion of set-refinement. Set-refinement is described in [HS89] where it is applied to simultaneously simulate multiple cache configurations. A set index function $f_2$ refines $f_1$, when each set induced by $f_1$ is the union of a number of sets induced by $f_2$. Formally, it is required that for all addresses $x$ and $y$, $f_2(x) = f_2(y) \Rightarrow f_1(x) = f_1(y)$.

The set index function $f_{min}$ is the set index function that refines both $f_A$ and $f_B$ and no other set index function refines both $f_A$ and $f_B$ without also refining $f_{min}$. Similarly, the set index function $f_{max}$ has the property that it is refined by both $f_A$ and $f_B$, and that no other set index function has this property, unless it is itself refined by $f_{max}$. The reader should verify for himself that these definitions correspond to the union and intersection of selected bits for set index functions based on bit selection. Furthermore, the space of set index functions has the structure of a lattice with set refinement as partial ordering relation and $f_{min}$ and $f_{max}$ corresponding to meet and join respectively (see e.g. [BB70] on lattices).

For the special case where $f_A$ refines $f_B$, as is the case when the $B$-module is fully associative, $f_{min}$ equals $f_A$ and $f_{max}$ equals $f_B$. In this case, the fully associative cache acts as a spilling buffer for all sets in the $A$-module, providing additional associativity just for those sets that need it.

We note that the above definitions are only correct when two set index functions are regarded as equal iff the partitions they induce on the address space are equal. This is necessary since two index functions that simply rename the sets (e.g. the set numbers 0 and 1 are swapped) should be treated equal.

## IV. Description of the optimal replacement algorithm

The optimal replacement algorithm for balancing multi-module caches presented here is based on the algorithm of Temam [Tem98] for fully associative caches. This algorithm is centred around the concept of locality edges. When a word is referenced at two times $t_1$ and $t_2$ with $t_1 < t_2$, but not in between, then there is a *locality edge* from time $t_1$ to $t_2$. This locality edge expresses the reuse of data. Locality edges can be either selected or not selected. A locality edge is selected iff the block pointed to by the edge is present in the cache at each time $t$ with $t_1 \leq t \leq t_2$. Since there is one cache miss for each unselected edge (the data is not in the cache at the time of the second reference), the miss ratio is minimised when the number of selected edges is maximised. The number of edges that can be selected is limited by the size of the cache. For a fully associative cache with $n$ frames, there can be no more than $n$ edges selected at any one time. Temam showed that an algorithm that selects each selectable edge, when the edges are ordered by increasing time of the second reference, is optimal with respect to minimising the cache miss ratio [Tem98].

Temam's algorithm can also handle spatial edges that occur when the spatial locality is higher than the temporal locality. Spatial edges are not considered here, since their purpose is to load only those words in a cache block which are effectively used. We only need to handle complete cache blocks.

Figure 4 shows an example trace annotated with locality

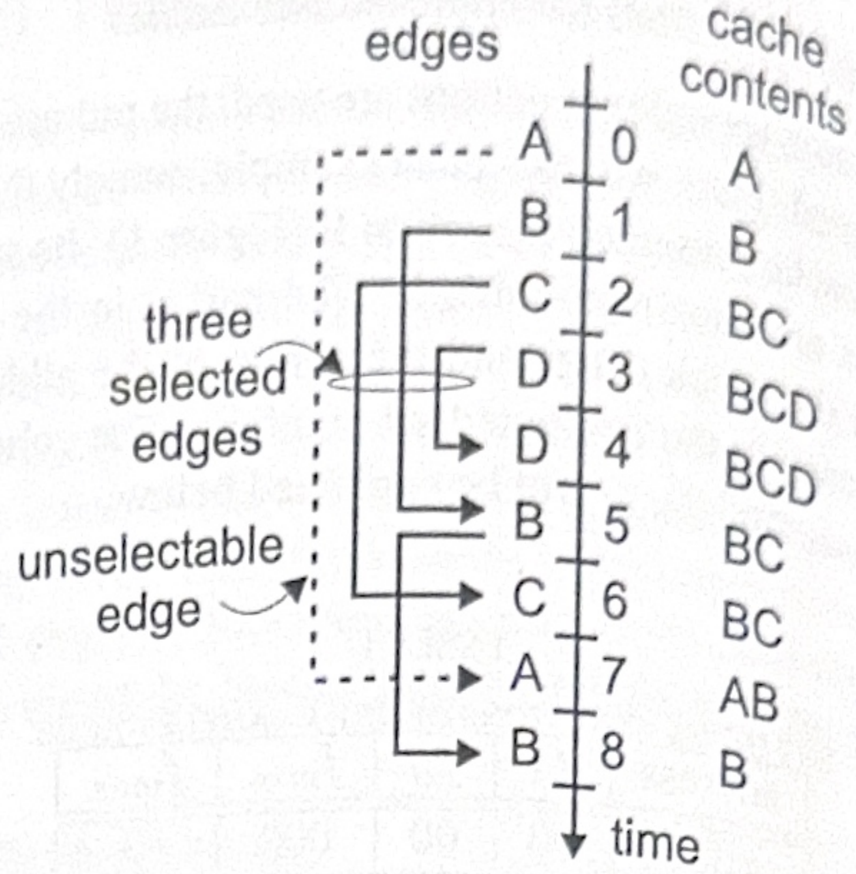


Fig. 4. Memory reference trace with edges. When the cache is assumed to have a size of three cache blocks, the dotted edge cannot be selected.

edges. To the right of the time line, the contents of the cache are shown. Note that the contents of the cache can only be computed when the complete trace is simulated. Assume the simulated cache is fully associative with a capacity of 3 blocks. Then the edge referring to block A at time 7 cannot be selected, because when block C is referenced at time 6, then the edge count at times 3 and 4 is already 3 and the cache is full. Therefore, the edge of block A is unselectable and the second reference to A is necessarily a cache miss.

We extend this optimal algorithm to balancing multi-module caches with two modules by changing the condition under which edges are selectable. An edge is selectable iff for each $t$ ($t_1 \leq t \leq t_2$), the blocks already present in the cache and the currently considered block fit in the cache together, satisfying the constraints imposed by the set index functions. The problem of determining whether a collection of blocks fits in the cache is solved by attempting to assign the blocks to the frames in the cache. If such an assignment can be found, then the blocks fit together in the cache. If not, the currently investigated edge is unselectable.

We will first describe the block assignment algorithm for the case where the set index function of one module refines the set index function of the other. Next, the more general case will be described. Only set index functions with one max-set are considered. The algorithm can be extended to multiple max-sets, just like the algorithm for a fully associative cache can be extended to a set-associative cache.

### A. Refining set index functions

When set index function $f_A$ refines $f_B$, assigning cache blocks to frames is straightforward. Due to the relation between $f_A$ and $f_B$, there is exactly one $B$-set and multiple $A$-sets in one max-set, since $f_{min} = f_A$ and $f_{max} = f_B$.

Assuming there is only one max-set, the frames in module B provide extra associativity to the sets in module A. Therefore, blocks are first assigned to the $A$-module. When an $A$-set is full, blocks for this $A$-set are placed in the $B$-module. When the $B$-module is full as well, not all blocks fit in the cache simultaneously and the currently considered edge cannot be selected.

### B. Non-refining set index functions

The effect of the set index functions on the partition of the address space is illustrated in Figure 5. The left square represents the complete address space; the right square is an enlargement of one max-set. This max-set is partitioned into $A$-sets and $B$-sets. The intersection of an $A$-set with a $B$-set is a min-set. Assigning blocks to cache frames is difficult, since placing one block influences the possible frames where the other blocks can be placed.

To solve this problem, we classify all cache blocks according to their min-set number. The number of blocks in each min-set is then split into the number of blocks that are placed in module A and those placed in module B. These quantities are represented by unknown variables. Then we derive a set of boundary conditions describing the possible values of the unknowns which results in a set of inequalities. The problem of checking whether a set of blocks fit in the cache simultaneously can then be solved by checking if the set of inequalities has a solution.

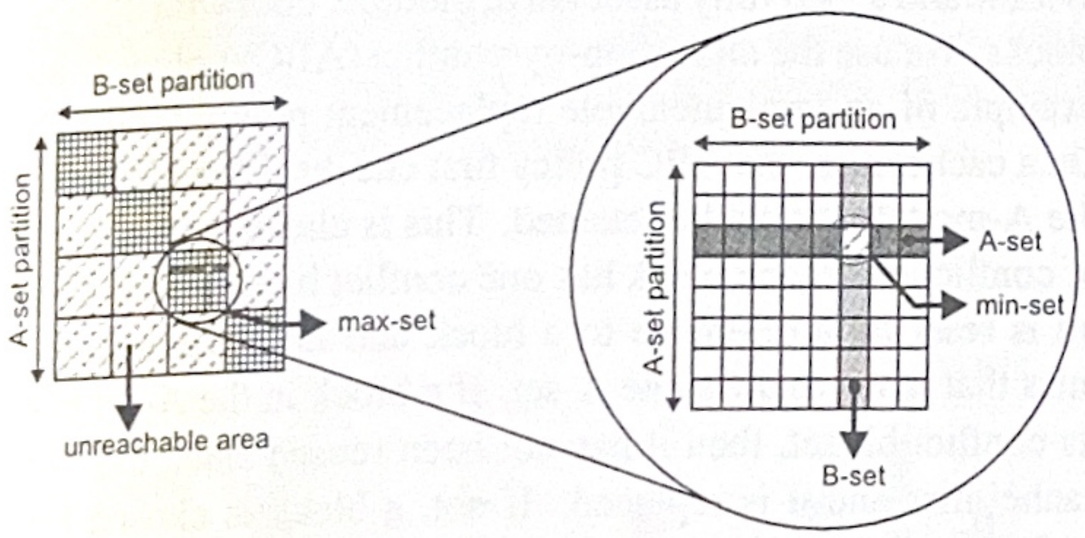


Fig. 5. Partition of the address space into max-sets and of a max-set into $A$-sets, $B$-sets and min-sets.

Now, we derive the set of inequalities. As stated earlier, the min-set is the smallest granularity of the partition of the address space that the cache can distinguish. Therefore, all blocks with the same min-set number are treated similarly. The number of blocks in each min-set is counted and is represented by the numbers $T_l, 0 \leq l < S$, where $S$ is the number of min-sets. Each of these blocks is placed in either the $A$-module or the $B$-module. The number of blocks in each min-set that are placed in the $A$-module is represented by $x_l$. The number of blocks in a min-set, placed in the $B$-module is represented by $y_l$.

It is required that

$$x_l + y_l = T_l, \quad 0 \leq l < S \tag{1}$$

$$0 \leq x_l \leq T_l, \quad 0 \leq l < S \tag{2}$$

$$0 \leq y_l \leq T_l, \quad 0 \leq l < S \tag{3}$$

for $0 \leq l < S$ and the $x_l$ and $y_l$ are cardinal numbers.

A first boundary condition for the values $x_l$ is that the number of blocks in an $A$-set is not higher than the associativity in the $A$-module. The number of blocks placed in an $A$-set is computed by summing all corresponding $x_l$ values. There is a contribution to the number of blocks in an $A$-set by each $B$-set, since there are as many min-sets contributing to this $A$-set as there are $B$-sets (Figure 5). Let $l(i,j)$ be the min-set number corresponding to $A$-set $i$ and $B$-set $j$. The number of blocks in $A$-set $i$ is then given by[1]

$$a_i = \Sigma_{j=0}^{S_B-1} x_{l(i,j)}, \qquad 0 \leq i < S_A$$

The number of blocks in each $A$-set is limited by the associativity $N_A$ of the $A$-module. Therefore it is required that

$$0 \leq a_i \leq N_A, \qquad 0 \leq i < S_A \tag{4}$$

Similarly, the number of blocks assigned to a $B$-set is

$$b_j = \Sigma_{i=0}^{S_A-1} y_{l(i,j)}, \qquad 0 \leq j < S_B$$

and the inequalities

$$0 \leq b_j \leq N_B, \qquad 0 \leq j < S_B \tag{5}$$

should hold.

By solving the equations 1 for $y_l$ and substituting their values into the inequalities 3 and 5, we are left with a set of only inequalities. This set can be solved using Linear Programming [Sch86].

It is instructive to write down the inequalities in a two-dimensional grid with the same dimensions as the grid describing the partitions of the address space (Figure 6). Each $x_{l(i,j)}$ is the number of blocks from min-set $l(i,j)$ that are placed in A-set $i$. The sum of all these blocks over all B-sets $j$ is the total number of blocks placed in A-set $i$. This sum should be less than the number of frames $N_A$ in an A-set. It corresponds to the shaded horizontal bar in the max-set in Figure 5.

### C. Implementation issues

The described algorithm requires long execution times and lots of memory, largely depending on the cache configuration. To obtain the results described in section V, we had to spend a lot of effort in optimising the algorithm.

[1]We use $S_A$ for the number of A-sets and $S_B$ for the number of B-sets. The equality $S = S_A S_B$ holds when there is only 1 max-set.

$$
\begin{array}{ccccc}
x_{I(0,0)} + & \cdots & +x_{I(0,S_B-1)} & \leq N_A \\
+ & & + & \leq N_A \\
x_{I(1,0)} + & \cdots & +x_{I(1,S_B-1)} & \\
+ & & + & \\
\vdots & & \vdots & \vdots \\
+ & & + & \leq N_A \\
x_{I(S_A-1,0)} + & \cdots & +x_{I(S_A-1,S_B-1)} & \\
\geq & \cdots & \geq & \\
-N_B + \sum_i T_{I(i,0)} & \cdots & -N_B + \sum_i T_{I(i,S_B-1)} &
\end{array}
$$

Fig. 6. The set of inequalities

The most important optimisation reduces the number of times the assignment of blocks to frames is done. The trace is divided in *sequences* during processing. A new sequence is started each time a cache miss occurs. The assignment of blocks to frames is only done once for each sequence. This is a valid optimisation since all the blocks in a sequence are necessarily simultaneously present at the time of the cache miss. The blocks involved in deciding whether an edge is selectable at the time of the miss will always be a superset of the blocks involved at later times. Figure 7 shows the possible situations. If an edge crosses the cache miss (edge 1), then it should be involved in the assignment of blocks to frames at the time of the cache miss. If the head and the tail are in the same sequence (edge 2), then there is no miss and thus no replacement in between the two references and the second one is a hit. Another possibility is for an edge to jump over the complete sequence (edge 3). Then the edge is included in the assignment of blocks to frames at the time of the miss.

This optimisation saves time, since the scans along the locality edges is shorter. It also saves space, since the list of blocks present at each time only needs to be stored once for each sequence. This optimisation was also present in Belady's algorithm [Bel66].

## V. Simulation results

We evaluated level 1 data cache organisations using 9 benchmarks chosen from the SPECint95, SPECfp95, task parallel suite [DGO+94] and mediabench [LPMS97] benchmark suites. The SPEC benchmarks were compiled using GCC 2.6.3, retargeted for SimpleScalar (MIPS-like instruction set), with optimisation flags `-O2 -funroll_loops`. For the other benchmarks, we used GCC 2.7.2.3 and the optimisation flags listed in their accompanying makefiles. Traces ranging over 100 million instructions were generated with a SimpleScalar simulator [BAB96], after skipping the first 100 million or 300 million instructions for SPECint and SPECfp benchmarks, respectively. The other benchmarks were executed completely.

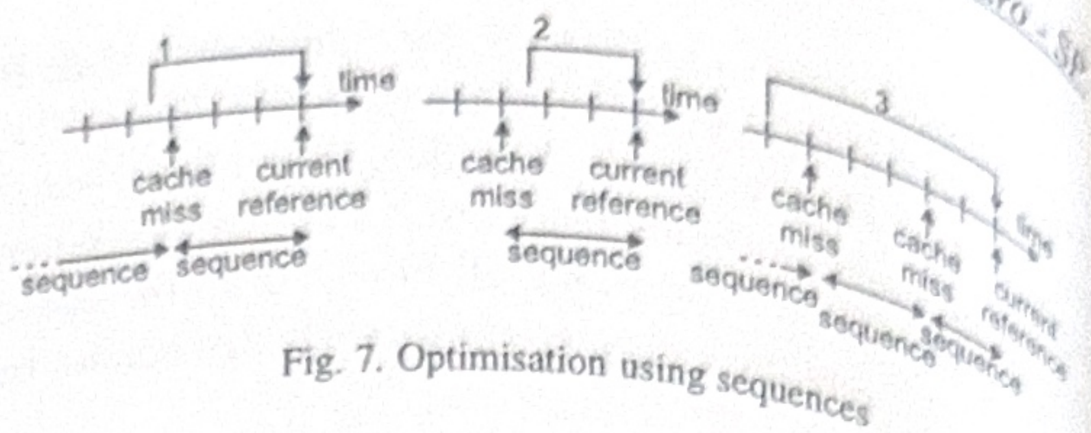


Fig. 7. Optimisation using sequences

TABLE II
DESCRIPTION OF BENCHMARKS. THE COLUMN 'SIZE' LISTS THE NUMBER OF COMPULSORY MISSES.

| Program | Input/flags | Refs | Size |
|---|---|---|---|
| apsi | train | 52.9M | 4.5K |
| mgrd | train | 64.5M | 32.0K |
| swim | train | 32.3M | 427.6K |
| ijpeg | vigo.ppm | 25.7M | 45.7K |
| perl | jumble.in | 40.0M | 306.7K |
| stereo | | 25.1M | 47.8K |
| djpeg | -dct int -gif testorig.jpg | 6.2M | 10.4K |
| epic | test_image.pgm -b 25 | 7.5M | 32.2K |
| pegwit | -e my.pub pgptest.plain out < encryption_junk | 8.3M | 6.1K |

### A. Multi-module cache with fully associative module

The multi-module cache has a 12 kB 3-way set-associative module and a 1 kB fully associative module, both with 32byte blocks. We use the allocation-by-conflict (ABC) policy as an example of an implementable replacement policy [Tam99]. On a cache miss, the ABC policy first checks if the blocks in the A-module should be retained. This is checked by means of conflict bits (each block has one conflict bit). The conflict bit is reset on a reference to a block and is set on a cache miss that maps to the same A-set. If a block in the A-set has its conflict bit set, then it has not been reused since the last cache miss and it is replaced. If not, a block is chosen for replacement from the B-module using the LRU policy.

We also simulated a victim buffer-like replacement policy [RTT+98] (VICLIKE). If the referenced block is not already present in module A, it is placed there. The victim block, selected using LRU, is placed in module B, either replacing the referenced block or generating a new victim block that is removed from the cache.

We show the miss ratio of a 13 kB optimally managed fully associative cache (Belady), our optimal replacement algorithm (OPT) and two implementable replacement algorithms ABC and VICLIKE (Figure 8). For most benchmarks, the restrictions imposed by the limited associativity has no big impact. This is caused by the fully associative module that provides ample associativity for those A-sets that need it. On the other hand, *djpeg* requires more associativity, since its

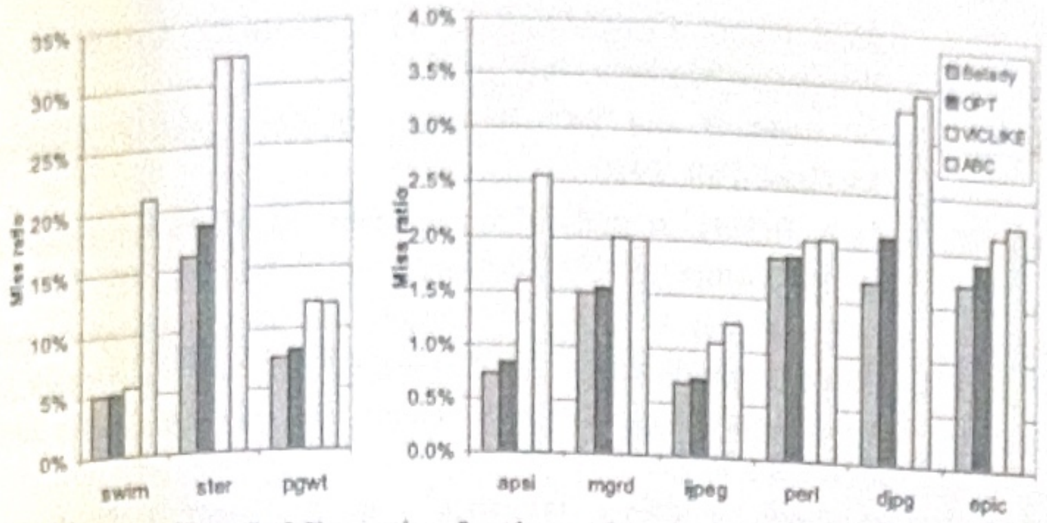


Fig. 8. Miss ratios for the multi-module cache.

OPT-miss ratio is 25% higher than with Belady's algorithm. The geometrical mean of the OPT miss ratios is 9.5% higher than the mean of the fully associative cache.

For some benchmarks, ABC can closely approach the optimal miss ratio. However, the miss ratio is 4 times higher than OPT for *swim* and 3 times higher for *apsi* (74% worse on geometrical mean). The VICLIKE policy usually has lower miss ratios then the ABC policy and it almost equals the OPT policy for *swim* (39% worse on average).

### B. Skewed-associative cache

The second cache organisation is a 16 kB 2-way skewed-associative cache with 32 byte blocks. The index function of module A selects bits 5 to 12 and that of module B selects bit 8 to 15, resulting in 32 max-sets. We study the ENRU replacement policy [Sez97]. It employs usage bits to track the least recently used block. In Figure 9, we also present the miss ratio of a 16 kB fully associative cache (Belady).

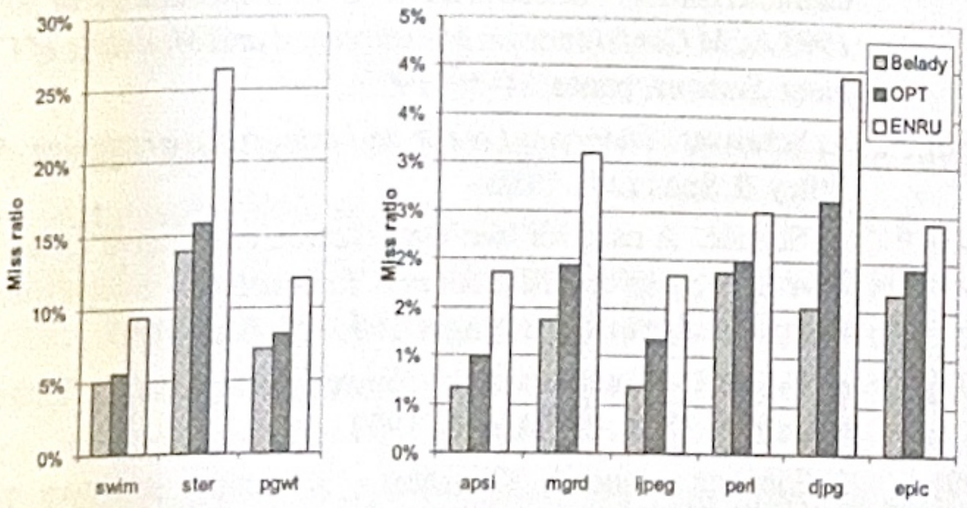


Fig. 9. Miss ratios for the skewed-associative cache.

For the skewed-associative cache, OPT replacement has on average 31% more misses than a fully associative cache (this is only 9% in the case of the multi-module cache). The set index functions of the skewed-associative cache are thus more restrictive than those of the multi-module cache. We also see that the ENRU policy is always at least 25% worse than OPT (e.g. for *perl* and *epic*) but only 53% worse on average.

### C. Skewed-associative cache with hashing index functions

Finally, we studied a 4 kB 2-way skewed-associative cache with hashing index functions as in [BS93]. In this cache, OPT replacement incurs on average only 9.4% more misses than would be the case in a fully associative cache with the same size. The ENRU policy is on average 53% worse than the OPT policy in this cache organisation.

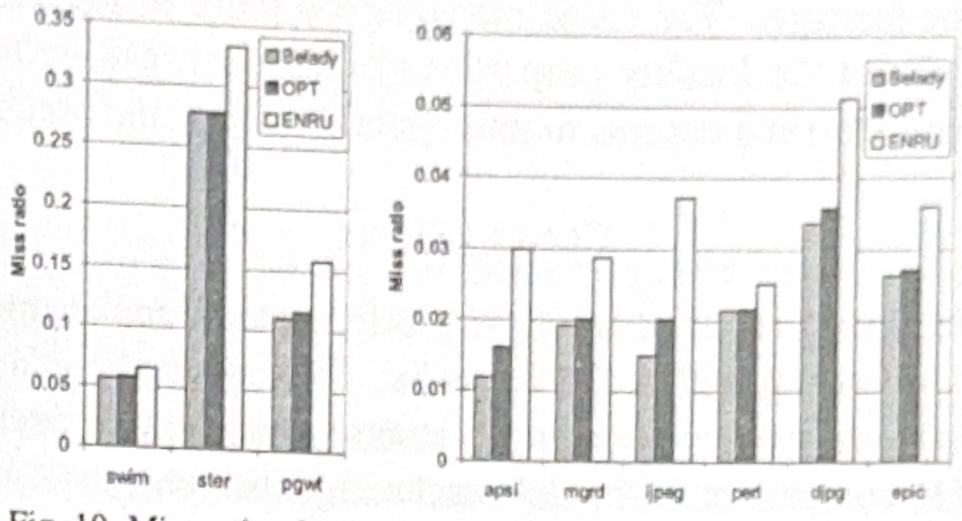


Fig. 10. Miss ratios for the skewed-associative cache with hashing.

In conclusion, we find that the multi-module cache with a fully associative module and the skewed-associative cache with hashing functions both allow to make almost optimal use of the provided cache area. Furthermore, implementable replacement policies come about equally close to what is optimally achievable in those caches. The skewed-associative cache without hashing functions, however, does impose important restrictions on the placement of blocks in the cache.

## VI. Related and Future Work

Many optimal replacement policies have been constructed. Belady's algorithm [Bel66] scans a trace of memory references. For each reference, the decision to keep the data in the cache is postponed until the next reference to the same data. The data is stored in the cache if the cache was never full in the time between the two references.

Mattson developed the notion of stack algorithm [MGST70]. For stack algorithms, the contents of a set-associative cache is always a subset of the contents of a similar cache with higher associativity. It was also shown that each stack algorithm can be implemented by means of a *priority function* which orders the references. Blocks with lower priority are replaced first. As a consequence, Belady's optimal algorithm can be implemented as a two-pass algorithm. In the first pass the priority function for a reference trace is computed and in the second pass the cache is simulated using the computed priority function.

Sugumar and Abraham constructed a one-pass algorithm that computes the priority function for a limited number of references at a time [SA93]. Another version of the optimal replacement policy is discussed by Temam [Tem98]. This algorithm is an extension of Belady's algorithm. Besides ex-

ploiting non-spatial reuse of data, it allows for neighbouring data to be fetched on a cache miss.

A pseudo-optimal replacement policy for multi-module caches with one fully-associative module is presented in [Tam99].

The investigation of a balancing multi-module cache is part of a research project about *active cache management*. An active cache consists of multiple cache modules and a cache manager. The cache manager is a piece of logic that can detect the locality properties of data (or it can be told about them) and use this to intelligently manage the cache.

## Conclusion

We have presented an algorithm for optimal replacement in a balancing multi-module cache. This algorithm can be used to evaluate replacement policies in multi-module caches and to compare multi-module cache organisations.

The algorithm was used to evaluate three cache organisations. The multi-module cache and the skewed-associative cache with hashing functions place few restrictions on the placement of blocks in the cache. On the other hand, a skewed-associative with bit selecting index functions restricts the placement of blocks much more.

We also tested replacement policies for the cache organisations. In both the skewed-associative caches, the average miss ratio of ENRU is 53% higher than the optimal miss ratio, although this policy does not employ balancing. In the multi-module cache, this difference is 48% (VICLIKE) or 74% (ABC). Thus, to closely approximate its optimal performance, the multi-module cache needs to perform balancing, as is exemplified by the VICLIKE policy.

We conclude that both the replacement policy and the index functions of the cache have an important role in determining the performance of the cache. Furthermore, the capability of balancing allows for certain benchmarks to obtain near-optimal miss ratios.

## Acknowledgements

The authors thank Lieven Eeckhout and Kristof Beyls for proof-reading this manuscript. Hans Vandierendonck is supported by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT). Koen De Bosschere is research associate with the Fund for Scientific Research-Flanders.

## References

[AP93] 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 Annual International Symposium on Computer Architecture*, pages 179–190. IEEE, 1993.

[BAB96] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical report, Computer Sciences Department, University of Wisconsin-Madison, July 1996.

[BB70] G. Birkhoff and T.C. Bartee. *Modern Applied Algebra*. McGraw-Hill, 1970.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[BS93] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 265–274, June 1993.

[DGO+94] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994.

[GAV95] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS'95. Proceedings of the 9th ACM International Conference on Supercomputing*, pages 338–347, 1995.

[HL99] T. Horel and G. Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. *IEEE Micro*, 19(3):73–85, May 1999.

[HS89] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Conference on Microprogramming and Microarchitecture*, pages 330–335, December 1997.

[MGST70] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[RTT+98] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Uitilizing reuse information in data cache management. In *ICS'98. Proceedings of the 1998 International Conference on Supercomputing*, pages 449–456, 1998.

[SA93] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *SIGMETRICS'93. Proceedings of the the 1993 ACM Conference on Measurement and Modeling of Computer Systems*, pages 24–35, 1993.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.

[Sez93] A. Seznec. A case for two-way skewed associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, May 1993.

[Sez97] A. Seznec. A new case for skewed-associativity. Technical Report PI-1114, IRISA, July 1997.

[SG99] J. Sánchez, , and A. González. A locality sensitive multi-module cache with explicit management. In *ICS'99. Proceedings of the 1999 International Conference on Supercomputing*, pages 51–59, Rhodes, Greece, June 1999.

[Tam99] E. S. Tam. *Improving Cache Performance Via Active Management*. PhD thesis, University of Michigan, 1999.

[Tem98] O. Temam. Investigating optimal local memory performance. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 218–227, November 1998.

[TFMP95] 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, December 1995.