A Locality Sensitive Multi-Module Cache with Explicit Management

Jesús Sánchez and Antonio González

Department of Computer Architecture
Universitat Politècnica de Catalunya
Barcelona - SPAIN

E-mail: {fran,antonio}@ac.upc.es

Abstract

Cache memories are often inefficiently managed, which results in significant memory penalties. An important reason for this poor performance is the homogeneous management of all memory references, even though different memory references may exhibit a very different locality. In this work, we present a novel data cache architecture composed of different modules, each module exploiting a particular type of locality. The information of which module each fetched data is placed on is passed to the hardware by means of a hint encoded in the memory instructions. This hint is set based on a locality analysis that can be performed by the compiler or using profiling data.

The proposed data cache organization exhibits a high-performance for numerical codes, even with low capacity. A 5KB cache has a miss ratio that is about 0.4 times the miss ratio of an 8KB direct-mapped cache and it is very close to that of a 64KB fully-associative cache, when data prefetching is not considered. Furthermore, the locality analysis allows for a selective one-block lookahead prefetch scheme, just for those references that exhibit spatial locality. This prefetch further reduces the miss ratio to one half of that of a 64KB fully-associative cache, with a negligible increase in memory traffic compared with the non-prefetching scheme, due to the locality-driven selective approach.

Although the proposed data cache organization is oriented towards numerical codes, which usually suffer more memory penalties than non-numerical ones, for the latest ones, a very simple profiling analysis results in a performance very close to a conventional cache, in spite of its lower capacity.

1. Introduction

Memory performance has become one of the main bottlenecks for the performance of current microprocessors. Since processor speed improvement is expected to outpace memory bandwidth increase in future generation microprocessors, this problem will further worsen.

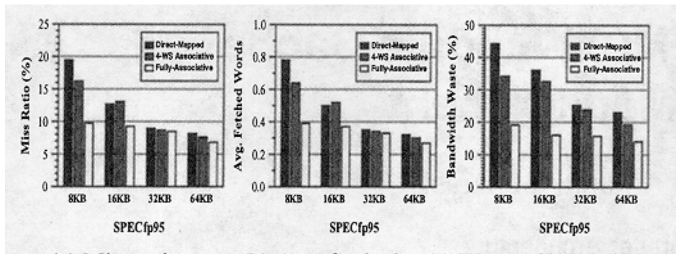
Due to the great impact that cache performance has on the overall processor performance, current processors assign a large portion of its area to implement a first-level cache (typically split into instruction and data caches). In this way, on-chip caches in current processors occupy between 1/3 and 3/4 of the total chip area. However, the performance obtained by these caches can still be insufficient for some applications, mainly numeric applications that require large working sets. For instance, Cvetanovic and Bhandarkar reported that the Alpha 21164 is stalled about 50% of the time for the SPECfp92 and the majority of these stalls are due to memory related issues [5].

Increasing the cache capacity/associativity may help but is not necessarily the most cost-effective solution because both capacity and associativity may increase the cycle time. Furthermore, there are several studies (see [10][2] for instance) that show that the cache memory makes an inefficient use of its storage capability. We claim that this inefficiency comes from the uniform management of all memory references. That is, for every reference which is not in cache, the corresponding block of data is fetched and placed using the same scheme for all references. The cache must then be designed to exploit some average amount of spatial and temporal locality, whereas there are references that may exhibit a very different locality (higher or lower than on average).

In this paper, we propose a new cache architecture that consists of three modules, each of them being specialized to exploit a different type of locality. Unlike previous multi-module proposals, the hardware complexity is kept low since the compiler is responsible for making explicit for each memory instruction the suggested module allocation for the data that it references. The proposed cache architecture is thereby called *Locality Sensitive Multi-module Cache (LSMCache).*

For numerical programs, which are in general more affected by cache hierarchy, the locality of each reference can be estimated quite accurately at compile time using a data locality analysis([6][23][3]). In this paper, we focus on this type of applications. For other type of applications (non-numerical applications, with a lot of pointers and dynamic structures) a static locality analysis can be unfeasible to perform, and thus it is performed with profiling data.

Including some hints in the memory instructions so that the compiler can provide the hardware with relevant information regarding the locality exhibited by each memory instruction is becoming a common practice. For instance, the PA7200 memory instructions have a bit in order to identify references with only spatial locality [4]. The PowerPC provides the possibility of identifying memory instructions that exhibit low locality and thus, to bypass the cache for such instructions [21]. In all these cases, the compiler is responsible for providing the information that is encoded in the memory instruction and that will determine during



(a) Miss ratio (b) Avg. fetched words (c) Waste of bandwidth

Figure 1. Performance of conventional cache architectures averaged for all programs

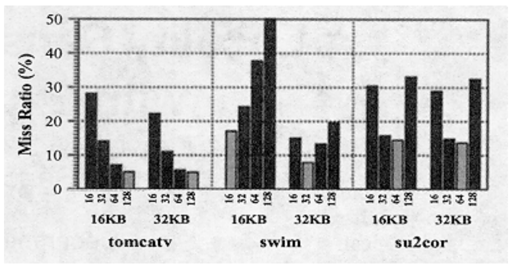


Figure 2. Impact of cache line size on total miss ratio for some SPECfp95 benchmarks

execution the proper action that the hardware must take. A more general approach is taken by the HPL-Playdoh architecture [13]. This architecture emphasizes the philosophy of passing information from the compiler/profiler to the hardware by making it explicit in the ISA. This information may be related to several different issues, such as dependences, speculation, data locality, etc.

The remainder of this paper is organized as follows. Section 2 points out the inefficiency of conventional cache architectures. Section 3 reviews the related work and describes the contributions of this work. The proposed cache architecture is presented in section 4 and some performance figures are shown in section 5 (for numerical codes) and 6 (for non-numerical codes). Finally, section 7 presents a summary and the main conclusions.

2. Motivation

The basic goal of cache memories is to keep the most frequently referenced data near the processor. In order to do that, cache memories exploit an intrinsic characteristic of memory references called *reuse*. Reuse can be of two different types: i) *temporal* (different dynamic instructions access the same data word), or ii) *spatial* (different dynamic instructions access nearby data words). Each one of these two types of reuse can be further characterized as *self-reuse* or *group-reuse*, depending on whether the reuse occurs among different references of the same instruction or among references of different static instructions respectively.

Conventional caches try to exploit spatial reuse by using a *block* (also called *line*) as a transfer unit between the different levels of the hierarchy, and seek to exploit temporal reuse by keeping some recently accessed blocks in the cache memory. All memory references are handled in the same way, that is, they use the same fetch, placement, replacement and write policies [9]. However, this uniform management of all memory references can be very inefficient. In particular, some references can degrade the cache performance by introducing blocks that will not be used in a near future, or blocks where only a small portion of them is used. Such references fetch an unnecessary number of words, wasting memory bandwidth and polluting the cache.

For instance, Figure 1 shows the results of simulating several conventional cache architectures for some SPECfp95 programs (see Section 5.1 for further details about the simulation environment). The graphs show: a) the miss ratio; b) the average number of fetched words (8 bytes) from the next memory level per memory reference (that has a direct relation with the traffic); and c) the percentage of words that are brought into cache but not used before being replaced. Four different capacities are considered: 8KB, 16KB, 32KB and 64KB; as well as three different degrees of associativity: 1 (direct-mapped), 4 and full associativity. All configurations use a typical line size of 32 bytes. The graphs show the results averaged for all the analyzed programs. We can observe that increasing the cache capacity reduces the miss ratio but the benefits are small beyond 32KB (to obtain a further significant improvement a very large capacity is required). Figure 1 also shows that associativity helps but the benefits are more noticeable for small caches. Finally, it can also be observed that for all the configurations there is a high percentage of useless fetched words.

Another important observation is that the spatial locality of each reference may be very different. References with very high spatial locality will benefit from very large cache lines, whereas references with poor spatial locality may favor small cache lines. These differences in spatial locality may be observed if one considers the behavior of different references (or sections) in the same program, or the global behavior of different programs. For instance, Figure 2 shows the miss ratio of some SPECfp95 bechmarks (*tomcatv*, *swim* and *su2cor*) for different direct-mapped caches (16KB and 32KB) when the cache line size varies from 16 to 128 bytes. In the graph, light-grey bars show the best line size for each particular program and cache capacity. It can be seen that there are programs that achieve the best miss ratio by using medium or long lines (such as *tomcatv* and *su2cor*). On the other hand, some other programs work better with shorter lines (such as *swim*). This behavior suggests that a unique line size is not the best solution in order to implement a general-purpose cache.

3. Related Work on Multi-Module Caches

The distinguishing features of the cache architecture proposed in this paper are the classification of memory references into three types, according to their locality, in order to exploit them in three different modules, and the explicit placement of data based on a hint that is set at compile time.

Some other multi-module cache architectures have previously been proposed. The *stream buffers* [12] are FIFO queues added between the L1 and L2 caches. On a memory access, both the data cache and the stream buffers are probed in parallel. If the data cache misses but the stream buffer hits, the block is moved to the data cache and a prefetch to the next block is performed and placed in the buffer. Unlike our proposal, this scheme does not take into account the particular type of reuse exploited by each reference, and thus, unnecessary traffic and low performance can occur for only-temporal and non-strided references.

The *victim cache* [12] has as a primary goal to remove conflict misses. The basic idea is to have a small fully-associative module where blocks discarded from the main cache are placed. If a hit occurs in the victim cache, a swapping of blocks between the victim and the main cache is performed. This scheme also makes a uniform management of the cache architecture since all references are handled in the same way. The main drawback of the victim cache is the "blind" swapping management (in the sense that all replaced lines are moved to the victim cache), in addition to the increase in cache port pressure due to the swapping traffic. A similar cache architecture is the PA-7200 *assist cache* [4]. The management of the two modules is somewhat different, being the software-controlled selective swapping its most important difference. Memory instructions in the PA-7200 have a flag that is set by the compiler for those instructions that are expected to exhibit only spatial reuse. The data accessed by these instructions is brought into the assist cache but is not moved to the main cache.

There are also some works that propose different cache architectures composed of several modules, each one exploiting some particular kind of locality, such as the *dual data cache* [7], the *split temporal/spatial cache* [14], or the *array cache* [8]. All these schemes basically attempt to reduce the negative effects of references that exploit only temporal reuse, by just fetching a single word and allocating it in a special module. However, such schemes do not exploit the fact that some references exhibit just spatial reuse.

Rivers and Davidson proposed the *NTS cache* [16]. This architecture dynamically divides cache blocks into two groups: temporal and non-temporal, based on their past reuse behavior. The decision is made through a *detection unit* indexed by effective address. This architecture has a separate small cache (accessed in parallel with the main cache) where non-temporal blocks brought to cache are placed. The basic goal of this scheme is to reduce conflict misses caused by only-spatial references. In [17], it is proposed a modification of the *CNA cache* presented by Tyson et al [22] that also consists of two cache modules, but in this case the *detection unit* is indexed by program counter.

Another approach of multi-module cache is the one proposed by Johnson and Hwu [11]. This scheme, unlike previous proposals, dynamically divides memory references based on their frequency of reuse. In this case, structures are accessed by effective address, and not by instruction address. Only frequently referenced data are placed on the main cache, whereas the other bypass the main cache and are placed in a small buffer in order to exploit its possible temporal reuse.

Note that an important difference of the *LSMCache* with respect to all these previous proposals is the explicit management of the placement of fetched blocks and the clear differentiation of three types of reuse: only-temporal, low-volume self-spatial and the rest (they are later defined).

4. The LSMCache

The *LSMCache* is a cache architecture composed of three modules, each one exploiting a particular type of locality. The selection of where the data is placed when it is fetched from the next memory level could be done by a static locality analysis or based on an analysis of some profiling data. Then, the information is passed to the hardware by adding a special field or *hint* to the memory instructions. The cache architecture proposed in this work is oriented towards numerical codes, for which module allocation can be completely based on a static locality analysis, due to its high accuracy. However this static analysis is not appropriate for non-numerical codes and a profiled-based analysis may be more effective. In Section 6, some preliminary results for non-numerical applications based on a simple profiled-data analysis are presented.

The working of the *LSMCache* is divided into two parts: (1) the compile-time analysis and tagging of memory instructions, and (2) the run-time behavior. Below we first discuss the hardware architecture of the *LSMCache*. Then, the compile-time support, which basically consists of a static locality analysis, is explained.

4.1. Hardware Architecture

The hardware architecture of the *LSMCache* is shown in Figure 3. It is composed of three modules that are referred to as *spatial (S)*, *temporal (T)* and *spatial-temporal (ST)*. Both modules *S* and *T* are

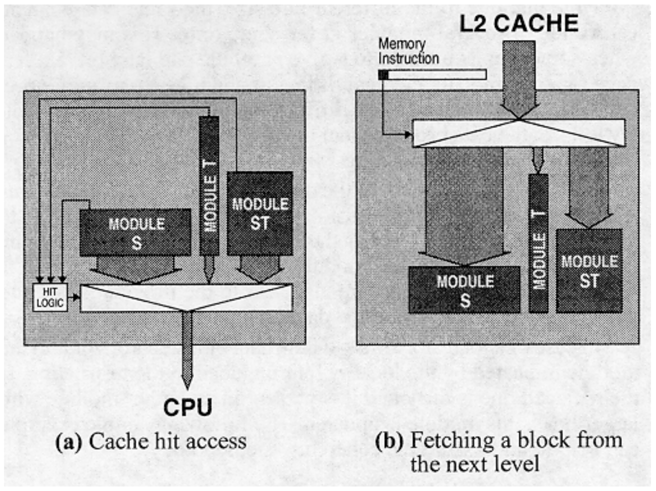


Figure 3. Hardware architecture of the *LSMCache*

small fully-associative buffers, whereas module *ST* is direct-mapped and has larger capacity. The goal of each module is the following:

- *Module S*: it is oriented to exploit *low-volume, self-spatial reuse*. A memory instruction is said to exhibit low-volume, self-spatial reuse if it has self-spatial reuse for a given loop and self-temporal reuse for all inner loops. Intuitively, this is an instruction that has spatial reuse that requires a single cache line to be exploited. For instance, the reference in the next loop:

```
DO I = 1, N, 1
   DO J = 1, M, 1
      ... A(I) ...
   ENDDO
ENDDO
```

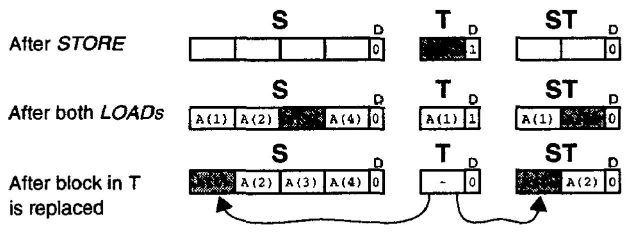
  has spatial reuse in loop `I`, and temporal reuse in loop `J`. Thus, when a line is fetched and placed in the module *S*, all iterations of loop `J` can take advantage of the temporal reuse without increasing the number of fetched lines. Further iterations of loop `I` will exploit the spatial reuse by accessing other elements of the same line.
  Note that either long lines or some simple hardware prefetching technique (prefetching the next or previous block, according to the direction of the stride) may significantly increase the exploitation of the locality exhibited by such references. Both strategies have been evaluated in this work.
- *Module T*: it is oriented to exploit just temporal locality. In a conventional cache, those references with just temporal locality pollute the cache and waste memory traffic because only one word of the entire line is used. This is avoided by using a special module with short lines to keep these references.
- *Module ST*: this module acts as a conventional cache targeted to exploit both temporal and spatial locality. It stores the data not allocated to the previous modules, that is, data referenced by instructions with both spatial and temporal reuse, such that spatial reuse requires a high number of lines to be exploited. Furthermore, it also stores the data referenced by those instructions whose reuse is unknown. This is the case of references outside loops or references for which the locality analysis cannot determine their locality. Finally, this module also caches those references that cannot be placed in the *S* or *T* modules in spite of meeting the reuse requirement, due to capacity constraints.

Note that due to the different line size, the same data element can reside in several modules at the same time. This may happen when some data is brought to a given module and later on, a reference to a nearby data element, brings it again as a part of a larger data block that is placed into a different module. If a copy-back policy is used, the data brought from memory may be stale. Coherency of data is kept in the following way.

For each load instruction, the three modules are checked in parallel. If the data is found in just one module, then it is returned to the processor. In the case that the data is found in more than one module, the data from the module with the smallest line size is returned. Store instructions are also sent to the three modules and those that contain the requested data are updated.

In case of a load/store miss, a new data block is brought into the module indicated by the locality hint included in the instruction. If the replaced line is dirty and it is present in any other module with larger lines, this module is updated. The following simple example can help to understand the coherency methodology:

```
PROGRAM P1         FUNCTION F1(K)    FUNCTION F2(B)
REAL*8 A(N)        REAL*8 K          REAL*8 B(N)
   :                  :                 :
CALL F1(A(1))      STORE K.T         LOAD B(2).ST
CALL F2(A)            :                 :
   :               RETURN            RETURN
LOAD A(3).S        END               END
   :
END
```



In this example each load/store instruction is marked with its hint (*S*, *T* or *ST*), and the **D** bit in a cache line indicates that the line is dirty. First, the store brings `A(1)` to the module *T*. Then, `A(2)` is referenced in function `F2` and a new block is brought into the module *ST* since it is not in cache, but this block contains a stale copy of `A(1)`. Something similar happens to the following load to `A(3)`, in the main program. The two stale copies of `A(1)` can reside in cache together with the updated copy, because the data in the smallest cache line will always be chosen. When the dirty line of the module *T* that contains `A(1)` is replaced, the modules *S* and *ST* are updated.

We will show in section 5 that the number of additional accesses required by the coherency mechanism is very low since, on average, only about 0.5% of dynamic memory references hit in more than one module.

4.2. Selection of the Target Module

This section explains the static locality analysis that is performed in order to set the locality hint of every memory instruction for numerical codes. This analysis is divided into three steps:

1) Choose candidate instructions for each module.
2) Sort the candidates in a priority order.
3) Tag each instruction with the appropriate hint.

The selection of the instructions whose referenced data are candidate to be placed in each module is based on a simple reuse analysis. This analysis is very similar to the one used in [23]. The results of this analysis are two vectors that represent the self-reuse and the group-reuse of each memory instruction. The self-reuse is represented by vector ***SRV***. The dimension of this vector represents the nesting level of the memory instruction (that is, the number of loops that enclose the reference). For each loop *i* (loop *0* represents the outermost), the value of ***SRV(i)*** can be ***N*** (no reuse), ***T*** (self-temporal reuse) or ***S*** (self-spatial reuse). The first step of the algorithm is to determine which memory instructions are candidates to be tagged for each module.

The candidates for module *S* are those instructions that meet the following condition:

$$\exists\, i \mid \boldsymbol{SRV}(i) = \boldsymbol{S}\ \underline{and}\ \forall\, j > i, \boldsymbol{SRV}(j) = \boldsymbol{T}$$

These are those memory instructions that have self-spatial reuse in a loop, and for all their inner loops, if any, they have self-temporal reuse.

The candidates for module *T* are those instructions that meet the following condition:

$$\exists\, i \mid \boldsymbol{SRV}(i) = \boldsymbol{T}\ \underline{and}\ \neg\exists\, j \mid \boldsymbol{SRV}(j) = \boldsymbol{S}$$

These are those references that have self-temporal reuse in one or several loops, but do not have self-spatial reuse for any of the loops where they are enclosed.

The rest of the instructions are candidates for module *ST*, including those instructions that are placed outside loops, or for which the reuse analysis cannot be applied (i.e., non-affine references).

Note that instructions with group reuse among them exhibit the same self-reuse and thus, they are tagged as candidates for the same module.

The second step, which orders the candidates for the same module according to a priority function, is applied to the *S* and *T* candidates. There are three parameters that determine the order among candidates: (i) placement of the memory instruction in the loop nest; (ii) loop where the reuse has to be exploited; and (iii) stride of the access (only for memory instructions with spatial reuse). The ordering of candidates is based on the following heuristics:

1) The number of reuses for each line brought to the module *S* increases as the nesting level of the loop where spatial reuse is exploited decreases.
2) The volume required to exploit a given type of reuse decreases as the nesting level of the loop that generates the reuse increases.
3) The benefits of spatial reuse are higher as the stride is smaller since the number of reuses for each cache block is higher.

In consequence, memory instructions are first sorted according to their location in the loop nest, from innermost to outermost instructions. For all instructions in the same nesting level, a second step of ordering is applied according to the level of the loop where the reuse is exploited (from outermost to innermost loops). If an instruction can exploit reuse in different loops of the same nest, the innermost of such loops is considered. Finally, instructions placed in the same level of the nest and that exploit their reuse in the same loop level are sorted according to their stride, from smaller to larger. For instance, in the next code:

```
                                  SRVs
     DO I = 1, N, 1
❶       A(I)                    (S)
❷       A(I+1)                  (S)
        DO J = 1, M, 1
❸           B(J)                (T , S)
            C(I)                (S , T)
❹           K1                  (T , T)
❺           D(I,J)              (N , N)
❻       ENDDO
        K2                      (T)
❼    ENDDO
```

the allocation of candidates to the different modules and the final ordering of candidates will be as follows:

S Candidates : ❹→❸→❷→❶
T Candidates : ❺→❼
ST Candidates : ❻

The last step of the analysis is the final selection of the tag for each memory instruction. In this step, a volume analysis is applied for both *S* and *T* candidates, in order to determine whether the locality exhibited by each instruction can be exploited given the capacity of the corresponding module. If a reference that was initially candidate for module *S* or *T* does not fit in it, it is finally allocated to module *ST*.

The volume analysis is based on the approach presented in [18], and it is independently applied to both lists of candidates to modules *S* and *T*. For each reference of the list, from highest to lowest priority, the volume in cache lines that this reference contributes to each loop where it is enclosed is computed and added to the accumulated volume so far. If the accumulated volume of the loop where this reference exploits its spatial/temporal reuse (for modules *S* and *T* respectively) does not exceed the capacity of the module, the memory instruction is tagged with the corresponding module (*S* or *T*). Otherwise, it is allocated to the module *ST* and its contribution to the volume of each loop is subtracted from the accumulated volumes.

For instance, the volume analysis for the *S* candidates in the previous code is the next one:

		Accumulated Volume of Loop `I`	Accumulated Volume of Loop `J`	Condition
❹	`C(I)`	1	1	1 ≤ NLINES ?
❸	`B(J)`	1+M/4	2	2 ≤ NLINES ?
❷	`A(I+1)`	2+M/4	2	2+M/4 ≤ NLINES ?
❶	`A(I)`	2+M/4	2	2+M/4 ≤ NLINES ?

In this example, instruction *4* requires just 1 line to exploit its spatial reuse in loop `J`. If this instruction is tagged as *S*, then instruction *3* requires 2 cache lines to exploit its reuse. Since both instructions *4* and *3* have been allocated to module *S*, instruction *2* requires 2+M/4 lines in order to exploit its spatial reuse in loop `I`, which is the volume contributed by instructions *4* and *3*, in addition to the line required by itself. Assuming that this volume is still lower than the module *S* capacity, reference *2* is tagged as *S*, and then, reference *1* is considered. This reference does not contribute any additional volume to any loop, since it reuses the lines referenced by instruction *2*. Therefore, it is also tagged as *S*.

Finally, the result of the whole locality analysis is reflected in each memory instruction by means of one of the following tags (the stride information is relevant just for those schemes that implement prefetching, as discussed in section 5.3):

Hints	Module
00	*S*, positive stride
01	*S*, negative stride
10	*T*
11	*ST*

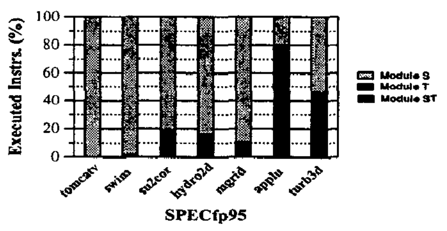


Figure 4. Percentage of dynamic memory instructions allocated to each module

5. Evaluation for Numerical Codes

5.1. Experimental Framework

The previously proposed locality analysis has been implemented in the ICTINEO compiling platform [1]. The programs have been compiled with full optimization (scalar optimizations such as constant propagation, and common subexpressions, deadcode and invariant removal) and the resulting code has been instrumented to generate a trace that feeds a simulator of the cache architecture.

The *LSMCache* has been tested with the following SPECfp95 programs: *tomcatv, swim, su2cor, hydro2d, mgrid, applu* and *turb3d*. The programs have been executed by using the test input data, and were run for the first 1,500 million of memory instructions (except for programs with fewer instructions).

5.2. Schemes without Prefetching

The first experiment evaluates some *LSMCache* architectures that do not incorporate any prefetching scheme. The differences among them are the total size and line size of the module *S*. Table 1 summarizes the evaluated configurations. Modules *S* and *T* have both 16 lines and use an LRU replacement. The label *FA* stands for *fully-associative*, whereas *DM* represents *direct-mapped*.

MODEL	MODULE S			MODULE T			MODULE ST			TOTAL CAPACITY (Kb)
	Total Size	Line Size	Assoc	Total Size	Line Size	Assoc	Total Size	Line Size	Assoc	
LSM-S1	512b	32b	FA	128b	8b	FA	4Kb	32b	DM	4.625
LSM-S2	1Kb	64b	FA	128b	8b	FA	4Kb	32b	DM	5.125

Table 1. Basic *LSMCache* configurations

We have compared the proposed architectures against two conventional caches: (a) a 8KB direct-mapped cache (**8KB-DM**), and (b) a 64KB fully-associative cache (**64KB-FA**) (see Section 2 in order to compare the results with other conventional cache architectures). The *LSM* architectures are comparable in area and access time[1] to the *8KB-DM* cache. A *64KB-FA* cache requires much more area and a much higher access time that the considered *LSMCache* architectures. This organization is used as a reference point of the miss ratio and memory traffic that could be achieved with a very powerful but also very expensive conventional organization.

Figure 4 shows the percentage of dynamic memory instructions tagged as *S*, *T* or *ST* for the *LSM-S1* architecture. We can see that in

1. Fully-associative caches larger than the size of the modules *S* and *T* (also with more lines), and with a one-cycle access time have been implemented in commercial processors. An example is the 2KB (64 lines) assist cache of the PA-7200 [4].

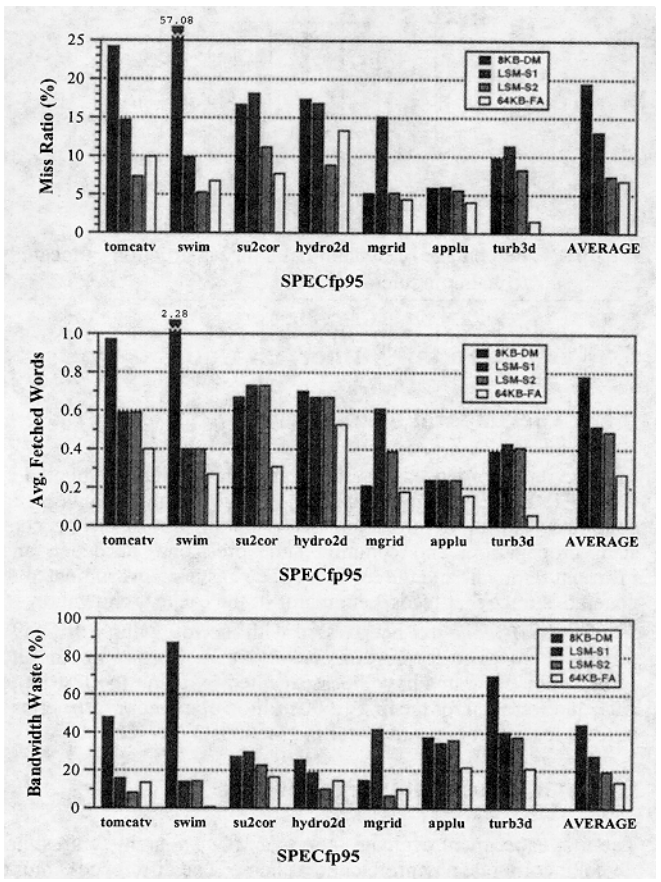


Figure 5. Comparison of *LSMCache* without prefetching against two conventional caches

general, low-volume self-spatial reuse references are the dominant type (these references are allocated to module *S* if there are not volume constraints). However, for some programs, the percentages of references allocated to modules *S* and *ST* are also significant (note that these results are for SPECfp95 programs and, although here the majority of instructions are tagged as *S*, this is not necessary for other codes).

Figure 5 depicts the miss ratio, average number of fetched words per reference, and percentage of unused words brought into cache, for the *LSMCache* and the two conventional caches. The number of fetched words has a direct relation with the traffic generated between L1 and L2 caches. Moreover, the percentage on unused words (or bandwidth waste) denotes the efficiency of this traffic.

On average both *LSMCache* schemes significantly outperform the *8KB-DM* cache for the three performance figures. For instance, the *8KB-DM* miss ratio is about 2.6 times higher than that of the *LSM-S2* cache. Comparing the proposed schemes, *LSM-S2* performs better than *LSM-S1*, mainly in miss ratio. This tendency is quite uniform for each individual program. Moreover, note that for some programs (*tomcatv*, *swim* and *hydro2d*), *LSM-S2* achieves a better miss ratio than the *64KB-FA* cache even though the number of fetched words is higher. This is due to a better usage of the fetched words. Note also that, on average, the miss ratio of the *LSM-S2* is close to that of the *64KB-FA*, in spite of its much smaller capacity. To achieve this performance, *LSM-S2* requires a higher number of fetched words (which is due to its smaller capacity) but the fetch bandwidth efficiency (which is the reciprocal of bandwidth waste) is comparable to that of the *64KB-FA*.

As explained in section 4.1, the *LSMCache* requires some coherency operations when a data element resides in more than one module. Note however that this event is rather infrequent. For the *LSM-S1* architecture, this percentage is 0.43% on average for all the programs (the maximum is 2.50% for *applu*, and the minimum is 0.00% for *tomcatv*, *swim* and *su2cor*). Note that in this architecture a datum can be in module *T*, or just in one of the other two since the line size is the same. Regarding the *LSM-S2* architecture, the average percentage is 0.65% (the maximum is 2.11% for *applu*, and the minimum is 0.00% for *tomcatv*).

5.3. Schemes with Prefetching

Prefetching data is beneficial provided that the cache may anticipate which data will be referenced in the near future. Otherwise, prefetch may harm performance. The detailed characterization of the locality exhibited by each reference allows for an efficient implementation of a prefetch scheme.

We have considered a simple hardware prefetching scheme based on the one-block lookahead schemes (OBL) [19], and extended with a locality analysis. We call the scheme *selective OBL* since the prefetch is performed only for those references that exhibit low-volume, self-spatial reuse (i.e. those allocated to module *S*). Note that this type of locality means that after accessing a data block, it is very likely that the next or the previous block, depending on the direction of the stride, is accessed too. The candidate references for which the prefetch is performed as well as the direction of the stride are provided by the locality analysis described in section 4.2.

Three alternative prefetching schemes, based on those considered in [19], have been implemented:

1) *Always prefetching* (*A*): every time a reference tagged as *S* is performed, a prefetch to the next/previous block is issued.
2) *Prefetching on miss* (*M*): every time a reference tagged as *S* misses (in all modules), a prefetch to the next/previous block is issued as well.
3) *Tagged prefetching* (*T*): every time a reference tagged as *S* accesses a block for the first time since it has been brought to cache, a prefetch to the next/previous block is issued as well.

Note that a prefetch access behaves like an ordinary access, that is, all modules are probed. Table 2 summarizes the different *LSMCache* architectures with prefetching that are considered in this paper.

MODEL	MODULE S				MODULE T			MODULE ST			TOTAL CAPACITY (Kb)
	Total Size	Line Size	Assoc	Pref.	Total Size	Line Size	Assoc	Total Size	Line Size	Assoc	
LSM-PA	1Kb	32	FA	A	128b	8b	FA	4Kb	32b	DM	5.125
LSM-PM				M							
LSM-PT				T							

Table 2. *LSMCache* architectures with prefetching

Both modules *S* and *T* use an LRU replacement. The label *FA* stands for *fully-associative*, whereas *DM* represents *direct-mapped*. Note that the module *S* has 32 lines but the compile-time analysis supposes that it has 16 lines, because some accesses to this module fetch a pair of lines due to prefetching. Thus, the instructions allocated to each module are the same as those in the previous experiments (*LSM-S1* and *LSM-S2*).

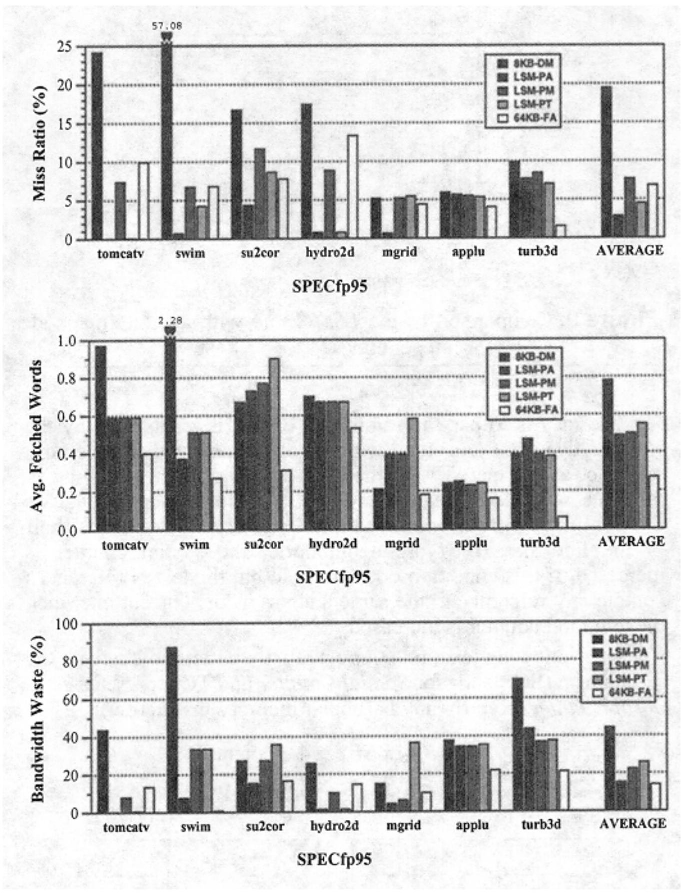


Figure 6. Comparison of *LSMCache* schemes with prefetching against two conventional caches

Figure 6 depicts the miss ratio, average number of fetched words per reference, and percentage of unused words brought into cache, for the *LSMCache* architectures with prefetching, and compares them with the two conventional caches.

Regarding miss ratio, *LSMCache* schemes perform much better than the *8KB-DM* cache, achieving an average reduction in miss ratio by a factor of about 7.0 (*LSM-PA*). Note that for some programs the miss ratio is very close to zero. The average reduction in memory traffic is also significant although for two programs (*su2cor*, and *mgrid*) it is somewhat increased. However, the lines brought into cache are better used, as denotes the bandwidth waste graph. Note that these *LSMCache* architectures achieve a lower miss ratio than a fully-associative cache with a capacity twelve times larger (*LSM-PA* has a miss ratio that is 2.5 times lower than that of the *64KB-FA* cache). This requires an increase in the memory traffic by a factor of about two to compensate for the much smaller capacity, but this traffic is efficiently used since the bandwidth waste is of the same order as that of the *64KB-FA* cache.

Among the different *LSMCache* architectures, the best performance is achieved by *LSM-PA*, that is, by the scheme that always prefetches for those data allocated in the *S* module. This scheme generates about the same traffic as the *LSM-PM* (prefetch on miss) scheme but achieves a miss ratio that is 2.7 times lower. The *LSM-PT* (tagged prefetching) scheme has an intermediate miss ratio but generates slightly more traffic. A positive effect of always prefetching is that prefetched data is kept at the top of the LRU stack, and therefore it is usually not evicted before being used. This is an efficient policy since prefetches are very effective, as shown below, because they are driven by a locality analysis.

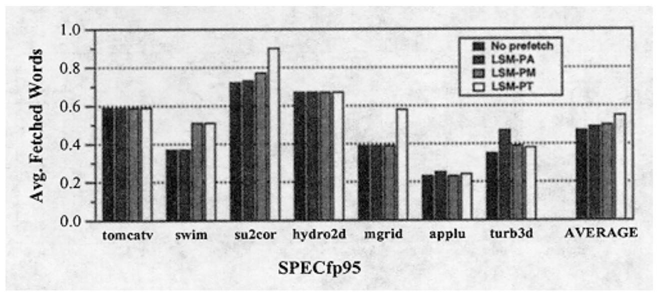


Figure 7. Impact of prefetching on fetched words

The effectiveness of prefetching can be evaluated by measuring the additional traffic that they generate. This is shown in Figure 7, which depicts the average number of fetched words per reference for the previous configurations and the same cache architectures without incorporating prefetch. It can be seen that the increase in memory traffic due to the prefetch schemes is very low.

As for the schemes without prefetching, the percentage of dynamic references that hit in more than one module has been also obtained, being very low for all three prefetching schemes (0.49% for *LSM-PA*, 0.43% for *LSM-PM*, and 0.35% for *LSM-PT*).

Finally, a drawback of prefetching is an increase in the cache port pressure. Each time a prefetch is issued, all the modules in the cache have to be probed. Table 3 shows the average number of cache memory accesses for each dynamic memory reference (without prefetching this number is 1.00):

MODEL	A Always Prefetch	M Prefetch on Miss	T Tagged Prefetch
tomcatv	2.00	1.07	1.22
swim	1.98	1.06	1.08
su2cor	1.81	1.08	1.14
hydro2d	1.84	1.08	1.16
mgrid	1.90	1.05	1.09
applu	1.20	1.00	1.06
turb3d	1.54	1.03	1.23
AVERAGE	1.75	1.05	1.14

Table 3. Averaged number of cache accesses per reference

In this table we can see that the *always prefetch* scheme is the one than achieves the lowest miss ratio but at the expense of a higher pressure on cache ports. If this resource is critical, the tagged prefetch scheme may be the best trade-off when both miss ratio and port pressure are considered.

5.4. Comparison with Other Multi-Module Caches

We have compared the *LSMCache* with three other multi-module schemes: (a) an 8KB direct-mapped cache with a 512B victim-cache (16 lines of 32 bytes each one)(**8KM-VC**); (b) an 8KB direct-mapped cache with 4 stream-buffers (each one with 4 entries of 32 bytes) with the optimizations proposed by Palacharla et al. [15] (**8KB-SB**); and c) an 8KB 4-way set-associative cache (**8KB-4WA**).

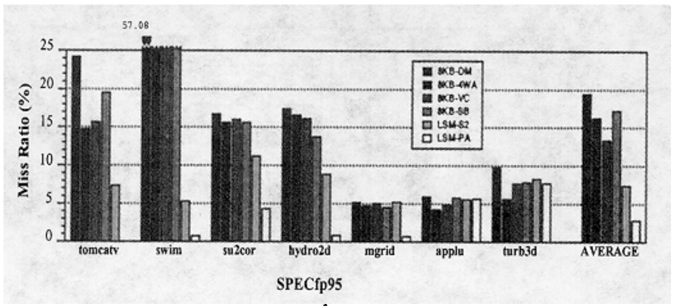


Figure 8. Comparison of the *LSMCache* with other multi-module caches

Figure 8 compares the miss ratio for a direct-mapped cache (**8KB-DM**), three multi-module caches (**8KB-4WA, 8KB-VC** and **8KB-SB**) and two *LSMCache* schemes (**LSM-S2** and **LSM-PA**), without/with prefetching.

We can see in these graphs that on average *LSMCache* schemes perform better than all the other schemes. Looking at individual benchmarks, only for the last two programs (*applu* and *turb3d*), the *LSMCache* architecture is not the best multi-module scheme.

6. Preliminary Results for Non-Numerical Codes

Although the *LSMCache* is oriented towards numerical codes, in which an accurate data locality analysis can be easily performed at compile time, this analysis may be rather inaccurate for non-numerical codes. The dynamic nature of data structures used by non-numerical programs (linked lists, dynamic graphs, etc.), typically written in C language, makes unfeasible a static data locality analysis.

Thus, we propose to use the results of a profiled-data analysis. In this section, we just try to show that with a very simple profiling, the miss ratios obtained by the *LSMCache* of 5 KB are not degraded when compared with a direct-mapped cache of 8KB. Taking into account the large benefits observed for numerical codes, we can conclude that the proposed architecture is an effective alternative. Improving the data locality analysis based on profiled data is by itself a research topic of great importance for many application areas.

6.1. Experimental Framework

The *LSMCache* has been tested with the following SPECint95 programs: *go*, *m88ksim*, *gcc*, *compress*, *li* and *perl*. The programs have been compiled with full optimization (-O4) with the DEC compiler, and the resulting code has been instrumented with ATOM [20]. In order to generate the profiling data, the programs are executed with the *test* input data, and run for the first 1,500 million of memory instructions (except for programs with fewer instructions).

Based on the information obtained by the profiler, each static memory instruction is accordingly tagged. Then, the programs are executed with the *ref* input data, and run for the first 1,500 million of memory instructions (except for programs with fewer instructions).

6.2. Profiled-Data Analysis

The proposed analysis is fed with the addresses of memory references obtained by profiling. For each static memory instruction two counters are allocated, which are referred to as *temporal* and *spatial* counters.

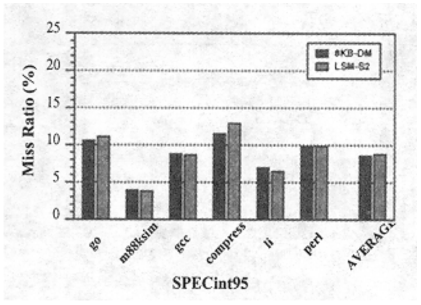
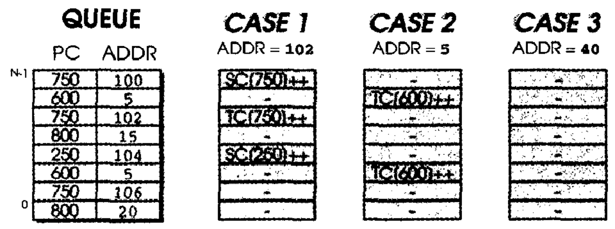


Figure 9. Comparison of the *LSMCache* with a direct-mapped cache for integer codes

The idea is to use the same type of heuristics applied by the static analysis but to a small window of references. Each memory reference is compared with the N previous references. If the N previous references contain any reference to an address that is different from the current one and its difference is less than a half of the line size (they are neighbours), the spatial counter is increased. If this condition does not hold but the last N references contain any reference to the same address as the current one, then the temporal counter is increased.

The following example may help to better understand the mechanism (SC stands for *Spatial Counter* and TC represents *Temporal Counter*, both for any particular memory instruction):

Block Size = 4 elements



The queue shows the previous N addresses referenced by the processor and each case illustrates the actions that are taken when a particular reference follows the N references in the queue. In the first case, address `102` finds in the queue other references to close locations (they differ in less that 2, that is, half of the line size). This happens for addresses `100` and `104`. In this case, the spatial counter of the memory instructions that fetched that addresses are increased. There is also a reference to the same address and thus, its temporal counter is increased. In the second case, address `5` only finds in the queue addresses equal to it (no neighbors), and thus, the temporal counter is increased. Finally, in the last case, address `40` has neither neighbors nor equal addresses in the queue, so the counters of the memory instructions are not modified.

After this process, each static memory instruction will have two values, one for each counter. Then, if the ratio of *temporal* references with respect to the total number of references for this memory instruction is greater than a give threshold t_X, the instruction is tagged as *T*. Likewise, the *spatial* counter is checked against another threshold t_Y to tag the instruction as *S*. In the case that none of the previous conditions are met, the instruction is tagged as *ST*.

6.3. Results

The previous profiled-data analysis has been applied with N=128, t_X=0.75 and t_Y=0.75. This analysis has been used to manage the *LSM-S2* cache of 5KB (as described in Table 1) and the results are compared with an 8KB direct-mapped cache in Figure 9, where the miss ratios for both schemes are shown. We can see that on average the performance is very close. Looking at individual benchmarks, for three programs the behavior of the *LSM-S2* is slightly worse (*go*, *compress* and *perl*), whereas for the other three it is slightly better (*m88ksim*, *gcc* and *li*).

7. Conclusions

This paper proposes a novel cache architecture composed of three modules, each module being specialized to exploit a particular type of locality. The management of the cache is driven by some hints in the memory instructions that are set at compile time to reflect the type of locality that they exhibit. In this paper we propose a static locality analysis and present some preliminary results for a profiled-data analysis.

The main conclusion of this work is that the implementation of smaller caches with a more clever management can be an effective approach to reduce the large area occupied by this component in current microprocessors and its access time.

We have observed that for numerical codes the proposed cache architecture eliminates the majority of cache misses with just 5KB of capacity. It has a miss ratio that is about the same as that of a 12 times larger (64KB) fully-associative cache when data prefetching is not incorporated and 2.5 times lower when data prefetching is added (for the most aggressive prefetching scheme). Prefetching is very effective since it is driven by the locality analysis. We have shown that prefetching hardly increases the memory traffic.

For non-numerical codes, which are much less affected by memory penalties, we have shown that with a very simple profiled-data analysis to tag memory instructions, the proposed cache architecture with a 5KB can achieve about the same performance as a direct-mapped cache of 8KB. This suggests that further research in the area of profiled-data analysis may significantly improve the performance of the proposed cache architecture for this type of codes.

Acknowledgements

This work has been supported by the Spanish Ministry of Education under contract CICYT-TIC 511/98, the ESPRIT Project MHAOTEU (EP24942), and by the Catalan CIRIT under grant 1996FI-3083-APDT.

References

[1] E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero, "ICTINEO: a Tool for Research on ILP", in *Supercomputing'96 Conf. (SC'96), Research Exhibit 'Polaris at Work'*, 1996

[2] D.C. Burger, J.R. Goodman and A. Kägi, "Memory Bandwidth Limitations of Future Microprocessors", in *Procs. of 23th Int. Symp. on Computer Architecture (ISCA'96)*, May 1996

[3] S. Carr, K.S. McKinley and C-W. Tseng, "Compiler Optimizations for Improving Data Locality", in *Procs. of the V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 252-262, Oct. 1994

[4] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher and J. Zheng, "Design of the HP PA 7200 CPU", *Hewlett-Packard Journal*, Feb. 1996

[5] Z. Cvetanovic and D. Bhandarkar, "Performance Characterization of the Alpha 21164 Microprocessor Using TP and SPEC Workloads", in *Procs. of 2nd. Int. Symp. on High-Performance Computer Architecture (HPCA-2)*, pp. 270-280, 1996

[6] D. Gannon, W. Jalby and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations", *Journal of Parallel and Distributed Computing, 5*, pp. 587-616, 1988

[7] A. González, C. Aliagas and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality", in *Procs. of 9th Int. Conf. on Supercomputing (ICS'95)*, pp. 338-347, 1995

[8] S. Hadjiyannis, M. Tomasko and W. Najjar, "An Evaluation of Split Scalar/Array Caches", *Technical Report CS-TR-97-104, CS Department, Colorado State University*, Jan. 1997

[9] J.L. Hennessy and D.A. Patterson, "Computer Architecture. A Quantitative Approach", *Morgan Kaufmann Publishers, 2nd. Edition, San Francisco*, 1996

[10] A.S. Huang and J.P. Shen, "A Limit Study of Local Memory Requirements Using Value Reuse Profiles", in *Procs. of 28th Int. Symp. on Microarchitecture (MICRO-28)*, pp. 71-81, 1995

[11] T. Johnson and W.W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis", in *Procs. of 24th Int. Symp. on Computer Architecture (ISCA-24)*, pp. 315-326, June 1997

[12] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", in *Procs. of 17th Int. Symp. on Computer Architecture (ISCA-17)*, pp. 364-373, 1990

[13] V. Kathail, M. Schlansker and B. Rau, "HPLabs PlayDoh Architecture Specification: Version 1.0", *Technical Report HPL-93-80, Hewlett-Packard Labs.*, March 1994

[14] V. Milutinovic, B. Markovic, M. Tomasevic and M. Tremblay, "The Split Temporal/Spatial Cache: Initial Performance Analysis", in *Procs. of SCIzzL-5 Workshop*, pp. 63-70, March 1996

[15] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", in *Procs. of the 21st Int. Symp. on Computer Architecture (ISCA-21)*, pp. 24-33, Apr. 1994

[16] J.A. Rivers and E.S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with Temporality-Based Design", in *Procs. of 1996 Int. Conf. on Parallel Processing (ICPP'96)*, pp. 93-103, Dec. 1996

[17] J.A. Rivers, S. Tam, G.S. Tyson and E.S. Davidson, "Utilizing Reuse Information in Data Cache Management", in *Procs. of 12th Int. Conf. on Supercomputing (ICS'98), July 1998*

[18] J. Sánchez, A. González and M. Valero, "Static Locality Analysis for Cache Management", in *Procs. of Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'97)*, pp. 261-271, Nov. 1997

[19] A.J.. Smith, "Cache Memories", *Computing Surveys, Vol. 14, No. 3*, pp. 473-530, Sept. 1982

[20] A. Srivastava and A. Eustace, "ATOM: a Flexible Interface for Building High Performance Program Analysis Tools", in *Procs. of Conf. on Programming Language Design and Implementation (PLDI'94)*, pp. 196-205, June 1994

[21] J.M. Stone and R.P. Fitzgerald, "Storage in the PowerPC", *IEEE Micro, vol. 15, no. 2*, pp. 50-58, April 1995

[22] G. Tyson, M. Farrens, J. Matthews and A.R. Pleszkun, "A Modified Approach to Data Cache Management", in *Procs. of 28th Int. Symp. on Microarchitecture (MICRO-28)*, pp. 93-103, 1995

[23] M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm", in *Procs. of Conf. on Programming Languages Design and Implementation (PLDI'91)*, pp. 30-44, 1991