Static Locality Analysis for Cache Management

F. Jesús Sánchez, Antonio González and Mateo Valero

Universitat Politècnica de Catalunya
Department of Computer Architecture
c./ Jordi Girona, 1-3 - Mòdul D6
08034 - Barcelona (SPAIN)
E-mail: {fran,antonio,mateo}@ac.upc.es

Abstract

Most memory references in numerical codes correspond to array references whose indices are affine functions of surrounding loop indices. These array references follow a regular predictable memory pattern that can be analyzed at compile time. This analysis can provide valuable information like the locality exhibited by the program, which can be used to implement a more intelligent caching strategy.

In this paper we propose a static locality analysis oriented to the management of data caches. We show that previous proposals on locality analysis are not appropriate when the programs have a high conflict miss ratio. This paper extends those proposals by introducing a compile-time interference analysis that significantly improve the performance of them.

We first show how this analysis can be used to characterize the dynamic locality properties of numerical codes. This evaluation show for instance that a large percentage of references exhibit only temporal locality and another significant percentage does not exhibit any type of locality. This motivates the use of a dual data cache, which has a module specialized to exploit temporal locality, and a selective cache respectively. Then, the performance provided by these two cache organizations is evaluated. In both organizations, the static locality analysis is responsible for tagging each memory instruction accordingly to the particular type(s) of locality that it exhibits.

1. Introduction

Current high-performance microprocessors rely on a efficient cache memory organization to mitigate the increasing gap between processor and memory speed. Despite of the tremendous research effort that has been devoted to this topic, current cache organizations make a poor use of the cache capacity. For instance, it is shown in [10] that most programs require considerably less cache memory than what a typical superscalar processor has.

One of the drawbacks of conventional cache organizations is that they perform a blind management of all memory references, that is, all of them are handled in the same way: if the reference misses, a new block is brought into cache at the expense of replacing another.

When a reference does not exhibit any type of locality, this results in cache pollution and memory bandwidth waste. The pollution is due to the placement in cache of a non-reusable block whereas the memory bandwidth waste is caused by the additional data brought from L2 cache to L1 cache in the same block as the requested data. To cope with this issue, some current microprocessors provide memory reference instructions that can bypass the cache.

When a reference has only temporal locality (i.e., only one data element of each cache block referenced by it is used by itself or any other instruction), it also results in cache pollution and memory bandwidth waste since only one element of the new block will be used. To overcome this problem, a cache could provide an additional module to store those data elements with just temporal locality. This was for instance proposed in the dual data cache organization [7].

In this paper we propose a static locality analysis to manage both selective data cache and dual data cache organizations. The locality analysis is inspired in the proposals presented in [18], [14] and [3]. However, these proposals do not consider conflict misses when performing the locality analysis. This may cause the analysis to be very inaccurate for programs with a high conflict miss ratio. To overcome this problem, we propose to extend those previous proposals for locality analysis with an interference analysis module. We show that very simple interference analysis schemes may deliver a quite accurate estimation of the locality of most references.

The paper provides quantitative statistics about the different types of locality exhibited by the nested loops of the

0-8186-8090-3/97 $10.00 © 1997 IEEE

SpecFP95 benchmarks. We find for instance that a substantial percentage of references do not exhibit locality due to cache conflicts for some programs and that a significant percentage of references exhibit just temporal locality. These observations motivate the use of selective and dual data caches.

The rest of the paper is organized as follows. Section 2 reviews the related work. The static locality analysis is presented in section 3. The experimental framework is described in section 4. Section 5 analyses the locality exhibited by loop nests of numerical codes. The application of the locality analysis to the selective cache and dual data cache is discussed in section 6. Finally, the main conclusions of this work are summarized in section 7.

2. Related work

Selective caching (also called cache bypassing) is a feature of current microprocessors like the PowerPC [15]. In the literature there are a number of proposals on that topic for both instruction and data caches. Some remarkable works for data caches are [6], [1], [7] and [17]. The scheme proposed in [6] is based on a compile-time estimation of data lifetimes. The mechanism proposed in [1] identifies non-cacheable data by means of profiling. The scheme proposed in [7] is based on a run-time managed history table of the most recent load/store instructions. The approaches proposed in [17] are either hardware-based or make use of simple schemes based on profiling.

The selective cache considered in this paper is like a conventional cache in which all the memory instructions have an additional bit that is set up by the compiler. In case of a cache miss, this bit controls whether a new block is brought from L2 cache and placed in L1 or just the missing data is requested from L2 and it bypasses L1 cache. We assume a 64-bit data bus between L1 and L2, thus, this is the bandwidth spent by any bypassing request regardless of the actual size of the required data.

The dual data cache was proposed in [7]. It is composed of two modules, called temporal and spatial. The former is targeted to exploit just temporal locality. The latter is designed to exploit spatial locality, in addition to temporal locality if a reference exhibits both types of locality. In consequence, the temporal module has very short blocks (one 64-bit word is assumed in this study) and the spatial cache has larger blocks (32 bytes per block is assumed here). Figure 1 shows the basic block diagram of the dual data cache. In this case, the compiler sets up a two-bit field of each memory instructions that indicates one of three possible actions in case of a cache miss: a) bring a new long block (32 bytes) and place it in the spatial module; b) bring a new short block (8 bytes) and place it in the temporal module; and c) bring just the requested data, which requires

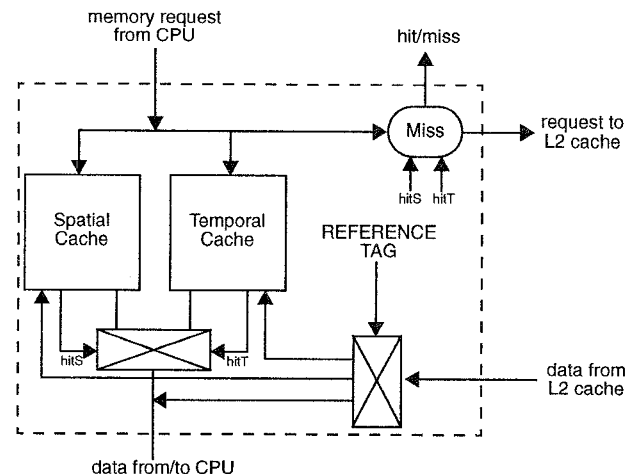

Figure 1. Block diagram of the dual data cache

one 64-bit word transaction due to the assumed bus width, and do not place it in any module.

The main difference between the dual data cache architectures considered here and that proposed in [7] is that in the latter, memory references were tagged at execution time using an additional hardware called *locality prediction table*.

An architecture with some similarities with the dual data cache has been recently proposed in [12]. Like the dual data cache, it has different modules for different types of locality. However, the allocation of data to the modules is done initially by a very simple heuristic based on the data type and then it may be changed by profiling or dynamically by means of a hardware that monitors their behavior.

A software managed data cache is provided in the HP PA-7200 [5]. In this machine, every memory instruction includes a hint called *spatial locality only* that indicates that the data referenced by that instruction exhibits spatial locality but not temporal locality. The first level of the memory hierarchy of the PA-7200 consists of two modules: the assist cache and the off-chip cache. The former stores all the data referenced by any instruction while the latter stores the data replaced in the assist cache if the spatial locality hint is not set. In consequence, the assist cache is targeted to any type of reference while the off-chip cache is targeted to store all the data except that with just spatial locality.

Static locality analysis has been previously used for different purposes. The most remarkable proposals are targeted to: a) improve the locality of loop nests by transformations like interchange, reversal, skewing and tiling [18]; and b) software prefetching schemes [13], [14], [3]. However, to our knowledge, this is the first study that addresses the use of a static locality analysis oriented to the manage-

ment of selective and dual data caches. In addition, our proposal extends previous schemes by incorporating an interference analysis. This analysis is crucial for the accuracy of the locality analysis for those cases in which conflict misses are the main source of cache misses, which usually is the case for small cache memories. For instance, more than half of the cache misses of the SpecFP95 benchmarks for a 8 Kbyte 2-way associative cache are conflict misses [8]. An interference analysis was proposed in [16] in order to estimate the number of conflict misses at compile time. That analysis is different and much more complex than the one proposed in this paper since it had a different objective. They tried to quantify at compile time the number of conflict misses whereas our objective is just to identify those cases in which cache interference will inhibit completely the exploitation of locality.

Regarding the quantitative evaluation of the locality exhibited by loop nests of numerical codes, this has been the focus of a recent paper [11]. This paper improves that work mainly in the following two ways. First, for each memory reference we consider all types of reuse that it exhibits (see section 3.1 for a definition of reuse) whereas that work only considered the last one in the order given by the source code. For each type of reuse, our analysis quantifies the amount of cache volume required to exploit it. Second, we consider any loop nest whereas that analysis only considered a subset of them with some particular features: at most 3 deep and with only one loop at each level.

3. Static locality analysis

The static locality analysis consists of three steps that are described below: reuse analysis, interference analysis and volume analysis.

We restrict the locality analysis to references inside loops, which represent the majority of references. The locality analysis estimates the type of locality for both scalar and vector references. For the latter, the locality analysis is performed just for array references where the array indices are affine (i.e., linear) functions of surrounding loop indices. In the analyzed benchmarks, the references that were handled by the analysis represent about 90% of the total. For the remaining references, it is assumed that they exhibit spatial and temporal locality.

3.1. Reuse analysis

The locality analysis starts by computing the *reuse* properties of each load/store instruction as proposed in [18]. An instruction has *self-temporal reuse* if the same data is referenced by at least two different iterations of the loop. It exhibits *self-spatial reuse* if the same data block is referenced by at least two different iterations. Likewise, different instructions may access the same locations or the same data blocks. In these cases, it is said that the instructions have *group-temporal reuse* and *group-spatial reuse* respectively. For group reuse, it is distinguished which reference access first the common data/block. This is called the *leading* reference whereas the other one, which is the one that can benefit from reuse, is called the *trailing* reference. Obviously, a reference may have several types of reuse.

Reuse is a measure that is inherent in a given program and does not depend on the order in which instructions are later executed. For instance, it is the same for both an in-order execution and an out-of-order execution processors. Besides, it is almost independent of the particular cache organization. In particular, the temporal reuse is completely independent whereas the spatial reuse just depends on the cache block size.

The reuse of each memory instruction is computed following the methodology described in [18]. The results are represented as a vector space that identifies the loops in which reuse is found (each dimension corresponds to a loop). We distinguish between two types of temporal and spatial reuse:

1) *Unitary*: the vector has only one element different from zero, that is, vector $(0,...,0,n_i,0,...,0)$ indicates that this reference has reuse after n_i iterations of loop i.
2) *Combined*: the vector has more than one elements different from zero, that is, vector $(0,...,0,n_i,n_{i+1},...,n_N)$ indicates that this reference has reuse after n_i iterations of loop i, n_{i+1} iterations of loop $i+1$ and so on.

The result of this phase is a list of the different reuses exhibited for each reference indicating the loop(s) for which each reuse holds. For instance, the reuse analysis of the sample code of Figure 2 will produce the following result:

Reference	Reuse in J	Reuse in I
`A(J)`	self-spatial	N.A.
`B(I,J)`	no reuse	self-spatial
`C(I,J)`	no reuse	group-temporal (trailing) with `C(I+1,J)` and self-spatial
`C(I+1,J)`	no reuse	self-spatial
`D(I,J)`	no reuse	self-spatial
`E(1,J)`	no reuse	self-temporal

3.2. Interference analysis

Whereas the reuse analysis is almost independent of any particular cache organization, the interference analysis is not. For the interference analysis, we assume in this

```
DO J = 1, 10, 1
  A(J) = 0.D0
  DO I = 1, 1000, 1
    B(I,J) = C(I,J) + C(I+1,J)
    D(I,J) = E(1,J)
  ENDDO
ENDDO
```

Figure 2. Sample code.

paper a direct-mapped organization for the selective and the spatial module of the dual data cache. The extension for other organizations such as set-associative caches is possible, but it is beyond the scope of this paper.

This analysis tries to identify groups of memory instructions that will repeatedly produce conflict misses due to interferences among them. There are two types of interferences: self-interferences and group-interferences. The former are those caused by different executions of the same instruction. The latter are produced by different instructions that reference either the same of different memory structures.

Interferences prevent the exploitation of the reuse exhibited by the interfering instructions.

Traditionally the effect of interferences have been taken into account by setting the "effective" cache size to be a fraction of the actual cache size [13]. This simple scheme does not consider the reference characteristics at all and may result in most cases in either an overestimation or an underestimation of the effect of interferences. Besides, interferences are not uniformly distributed over all memory references and therefore, their contribution should be measured for each reference independently. The effect of memory conflicts may be very high for some programs as it has been previously reported. Therefore, a more accurate estimation is crucial for the performance of the locality analysis.

3.2.1. Self-interferences. Self-interferences occur when different data blocks referenced by the same instruction are mapped onto the same cache location.

An affine array reference will generate sequences of memory references at addresses separated by a constant stride. The self-interference analysis only considers those instructions with a stride larger than or equal to the block size. Otherwise, the instruction exhibits self-spatial reuse that can be exploited before any potential self-interference happens.

If the stride is multiple of the block size, self-interferences will occur in a direct-mapped cache if the number of blocks of the cache is lower than the length of the sequence multiplied by $2^{\text{stride_family}}$. The stride family defined by x is the set of strides $\sigma \cdot 2^x$ with σ odd [9]. All the strides belonging to the same family (e.g., $12=3 \cdot 2^2$ and $20=5 \cdot 2^2$ belong to family 2) have the same behavior from the point of view of self-interference. Therefore, to approximate the effect of self-interferences, the volume of cache (see section 3.3) consumed by a reference is multiplied by $2^{\text{stride_family}}$.

If the stride is not multiple of the block size, the stride is rounded up to the next multiple of the block size and the above scheme is applied.

3.2.2. Cross-interferences. The cross-interference analysis focus on identifying those groups of references that reference different data blocks that map onto the same cache location for every iteration of the loop. These interferences will inhibit completely the exploitation of any reuse exhibited by the interfering instructions. This analysis is only applied for references whose variables are allocated at compile time, that is, those variables whose base address and size of every dimension is statically known. We have measured that more than 75% of dynamic references for the benchmarks considered in this paper meet these conditions. Extending the analysis to deal with other types of interferences is an future extension of this work.

The interference analysis is applied between the reuse and the volume phases, because its result can modify the volume of data fetched by each loop. The analysis consists of the following steps:

1) For each affine array reference, compute an expression that determines the effective memory address as a function of the initial address and the loop indices:

$$\text{EffAddress} = \text{IniAddress} + \left(r_0 + \sum_{i=1}^{N} r_i \cdot I_i \right) \cdot \text{ElementSize}$$

where I_i is the index variable of loop i in a nest of depth N.

2) Build an interference graph for each basic block. A conflict between two references R_1 and R_2 is assumed if they are mapped onto cache at a distance lower than the block size:

$$|R_1 \bmod \text{CacheSize} - R_2 \bmod \text{CacheSize}| < \mathit{Block}\text{Size}$$

Potential conflicts are analyzed for each pair of references and they are identified in the interference graph by means of an edge.

3) Remove interferences. If two instructions with some type of reuse interfere, their respective reuse cannot be exploited since the block brought in cache by any of them will be evicted immediately by the other, before it is reused. The objective of this step is to tag some of the interfering instructions as non-cacheable so that the remaining instructions do not interfere and therefore their reuse can be exploited.

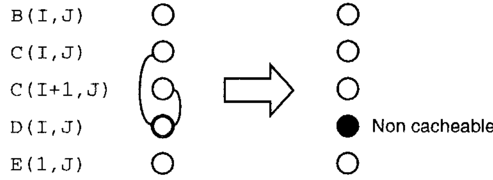

Figure 3. Interference analysis for code of Figure 2

The algorithm works as follows: in the interference graph, the node with the maximum number of edges is chosen. This reference is labeled as non-cacheable, and its edges are removed. Then, the process is repeated until the graph has no edges.

If we apply this analysis to the example of Figure 2, the results are shown in Figure 3. We have supposed that the initial interference graph is the one on the left. The selected reference is `D(I,J)`. Thus, this reference will be tagged as non-cacheable and it will not be cached despite of having reuse. However, the reuse exhibited by `C(I,J)` and `C(I+1,J)` can be exploited.

3.3. Volume analysis

Another factor that may inhibit the exploitation of reuse is the limited storage of cache memory. In other words, if the amount of different data blocks that are referenced between two consecutive reuses of the same block is higher than the cache capacity, this reuse cannot be exploited[1]. The resulting cache miss is usually called a capacity miss.

This requires to determine the amount of data that is used by each reference in each loop. This amount of data depends on:

(a) *Type of reuse*: calculated in the previous step.

(b) *Loop bounds*: if they are unknown at compile-time, they are estimated using the approximation proposed by D. Bernstein et al. [3]: each memory reference R is represented in the following way:

$$R \equiv c_1 + \sum_{j=2}^{M} c_j \cdot \prod_{k=1}^{j-1} D_k = r_0 + \sum_{i=1}^{N} r_i \cdot I_i$$

where M is the number of dimensions of the variable, D_k represents the size of dimension k, and c_j represents the subexpression in dimension j.

Then, the last sum is sorted by order of decreasing magnitude of coefficients r_i. The estimation is based on the assumption that a well behaved vector reference will access different locations for different values of the loop indices appearing in the expression. The estimated loop bounds are computed as follows:

$B_i = r_{i-1} / r_i$, if $i>1$

$B_i = \text{ArraySize}/r_1$, if $i=1$

(a default value is used if the array size is unknown at compile-time).

We use a simplification if the reference expression has only one loop index. In this case the estimation is based on the assumption that a vector subindex do not exceed the corresponding dimension ($c_i<D_i$).

1.This is true for LRU replacement. For other replacement policies, this is just an approximation.

Unknown:

$$V(R,i) = V(R,i+1)\ B_i$$

None:

$$V(R,i) = V(R,i+1)\ B_i\ 2^{\text{stride_family}}$$

Unitary temporal:

$$V(R,i) = V(R,i+1)$$

Unitary spatial:

$$V(R,i) = V(R,i+1) \left\lceil B_i \frac{\text{stride}}{\text{CacheBlockSize}} \right\rceil$$

Combined temporal or spatial:

$$V(R,i) = V(R,i+1)\ B_i \left(1 - \prod_{k=i}^{N} \frac{B_k - |n_k|}{B_k} \right)$$

Group trailing:

$$V(R,i) = V_L(R,i)\ \Upsilon_{R,i}\ , \text{where } \Upsilon_{R,i} = \frac{\left|r^L_0 - r^T_0\right| / \left|r_i\right|}{B_i}$$

Figure 4. Contributed volume of a reference to a loop. r^L and r^T are the coefficients of the leading and trailing references respectively and B_i is the upper bound of loop *i*.

The analysis follows these steps:

- Calculate for each memory reference the number of cache blocks that are accessed in one iteration of each loop. Figure 4 shows how the contributed volume of a reference R to a loop i, which is denoted by $V(R,i)$, is computed according to the type of reuse (note that $V(R,N) = 1$):
  - *Unknown*: we suppose the worst case, that is, every access uses a new cache block.
  - *None*: every access uses a new cache block too, but the volume is modified by the stride family in order to take into account the effect of self-interferences. In the case of the selective and the dual data caches, this kind of accesses are bypassed and, therefore, do not affect the computed volume.
  - *Unitary temporal*: the loop i accesses to the same data set as loop $i+1$ and thus, the volume is not increased.

- *Unitary spatial*: the volume is increased according to the number of elements that fit in a cache block. This number is given by the stride of the reference sequence divided by the cache block size.
- *Combined temporal or spatial*: the volume is computed multiplying the previous volume by a factor that represents the iterations where there is no reuse and thus, a new cache block is needed. Each loop with reuse contributes to the expression by means of the percentage of the total iterations that exploit reuse.
- *Group trailing*: the volume is computed as the volume of its leading reference multiplied by a factor $\Upsilon_{R,i}$ that represents the percentage of iterations needed to exploit the group reuse with respect to the total number of iterations of the loop. Since the trailing and leading references only differ on the independent term r_0, this factor is computed by the difference between independent terms divided by the coefficient that affect the current loop index (r_i) and by the loop count (B_i).

- A reuse in a loop b can be exploited if $\sum_{\forall R} V(R, b)$ is not higher than the cache size. Otherwise, each attempt to reuse a data element will result in a capacity miss.

If we apply this analysis to our example of Figure 2 the results are the following, assuming that the block size is 4 data elements and the cache has 256 blocks:

Reference	Contributed volume to loop I	Contributed volume to loop J
`B(I,J)`	1	250
`C(I,J)`	1	250
`C(I+1,J)`	0	0
`D(I,J)`	1	250
`E(1,J)`	1	10
Total	4	760
`A(J)`	-	1
Total	4	761

Consequently, only reuse across loop `I` can be exploited. Therefore, the reference `A(J)` will result in repetitive cache misses even though it has spatial reuse.

After the locality analysis is done, each memory instruction is tagged accordingly: references with no reuse are tagged as *bypass*, and the rest as *cacheable* in the selective cache and as *temporal* or *spatial* in the dual data cache. If the reference only can exploit temporal reuse, it is tagged as temporal and it is tagged as spatial otherwise. An additional constraint in the dual data cache is that the references that exhibit group locality have to be allocated to the same module.

4. Experimental framework

The cache experiments presented in this paper have been performed using the following SpecFP95 benchmarks: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu* and *turb3d*. All of them are written in Fortran language.

The locality analysis has been implemented using the ICTINEO toolset [2]. ICTINEO is a source to source translator that produces a code in which each sentence has a semantics similar to that of current machine instructions. Currently, ICTINEO assumes an infinite number of registers and thus, the references produced by spill code are not considered in this work. Optimizations usually applied by current compilers are implemented in ICTINEO and are applied to the resulting code. In this way, the resulting code is very similar to the code generated by a production compiler.

Memory references are instrumented according to the locality analysis results, and the trace obtained from the execution of instrumented code feeds a cache simulator of a selective and a dual data cache. A conventional cache is also simulated for comparison. The results presented in this paper correspond to the execution of the first 100 million of memory instructions of each benchmark.

5. Statistics of loop nest locality

The locality analysis previously presented can be used to obtain quantitative measures of the locality exhibited by loop nests.

We are interested in all types of reuse exhibited by each single memory reference. Consider for instance the following code:

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

Our analysis will conclude that for loop I, the first and third references exhibit group-temporal reuse. Group-temporal reuse is also exhibited by the second and first references (in this case the reuse is after one iteration). Besides, each reference exhibits self-spatial reuse. Now, considering loop J, we have that the three references exhibit self temporal reuse and any pair of references exhibits group-temporal reuse. Assuming that the interference analysis does not detect any interference and that the size of vector A is not higher than the cache capacity, the analysis will conclude

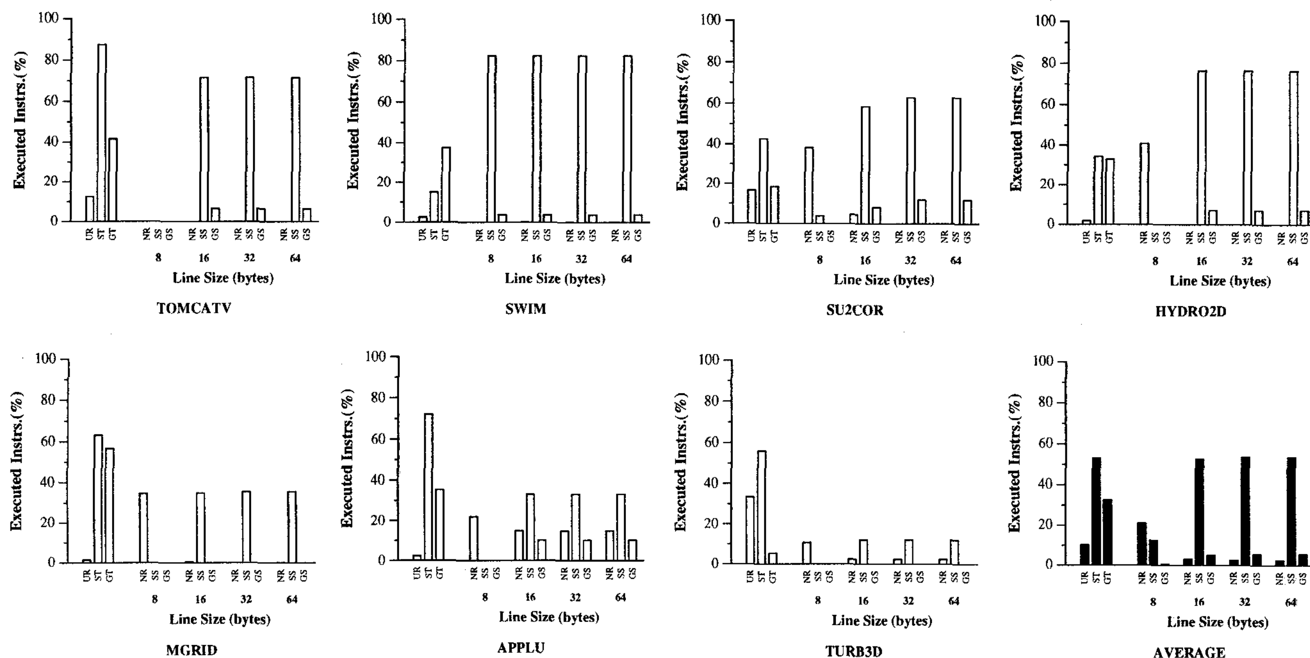

Figure 5. Reuse statistics. The different types of reuse are denoted by: UR (unknown reuse), NR (no reuse), ST (self-temporal), GT (group-temporal), SS (self-spatial) and GS (group-spatial).

that all the reuse can be exploited (the reuse across loop J requires a larger volume that the reuse across loop I, but it is still into the limits of the cache size).

Considering only the last type of reuse in program order as proposed in [11], the analysis would detect only a subset of the different reuses[1]. In particular, it would observe just group spatial reuse for every memory reference. This could suggest that for the above code it is not worthwhile to exploit temporal locality, whereas this is not the case.

Figure 5 shows the reuse statistics for the loop nests of the considered benchmarks. Each bar corresponds to the dynamic frequency of a different type of reuse. Since spatial reuse depends on the cache block size, different bars are drawn for a block size ranging from 8 to 64 bytes. The figure shows the reuse characteristics of each benchmark and the average among them. Notice that the sum of the frequencies of the different types of reuse may be greater than 1 since a reference may exhibit more than one type of reuse.

Several conclusions can be drawn from Figure 5. First, we can see that in average, self-temporal and self-spatial reuse are the most frequent and none of them is dominant. Group temporal reuse is also quite common whereas group spatial reuse is relatively infrequent. As expected, this results differ from those presented in [11], where it was reported for instance that self-temporal reuse was the least frequent type of reuse[2]. The dominant type of reuse varies significantly for the different benchmarks. Self-temporal is dominant for *tomcatv, applu* and *turb3d.* Self-temporal and group-temporal are the most frequent for *mgrid.* Self-spatial is dominant for *swim, su2cor* and *hydro2d.* Group spatial is always the least common type of reuse. Notice also that in average, the locality analysis can determine the reuse exhibited by about 90% of the executed instructions. Finally, it can be observed that almost all the references exhibit some type of reuse.

Figure 6 shows the percentage of executed instructions that exhibit just one type of reuse, either spatial or temporal. From now on, the figures present just average statistics over the different benchmarks. From this graph it can be concluded that temporal reuse is the most common type of single reuse, which may suggest the inclusion of a module

1.In [11], what we call reuse is referred as to locality. However, to be consistent with the rest of this paper, we have changed their terminology according to our definition.

2.Another reason for the discrepancy is the different benchmark suite.

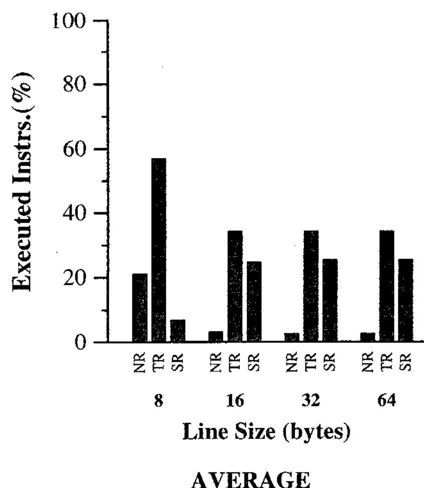

Figure 6. Percentage of instructions with just one type of reuse: no reuse (NR), temporal (TR) or spatial (SR).

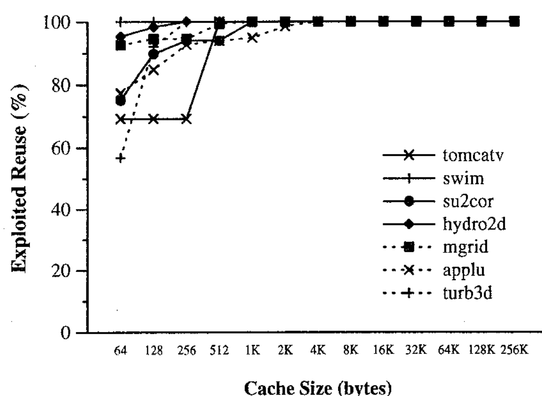

Figure 7. Exploiting temporal reuse only

specialized to exploit temporal locality as it is the case of the dual data cache.

Figure 7 shows the percentage of temporal reuse that can be exploited with a fully-associative cache that is used only for references that exhibit just temporal reuse (here a fully-associative cache is modelled by just not considering cache interferences). In this case, the line size is 8 bytes (one double precision float) since a larger line does not make sense because spatial reuse is not present. From this figure we can conclude that a 16 line (128 byte) temporal module is enough to exploit most of the single temporal reuse. As we have seen in Figure 6, these references represent about 35% of the total. Thus, this will be the size of the temporal module of the dual data cache for the experiments of the next section.

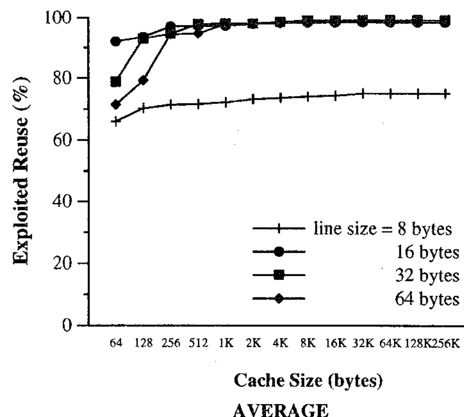

Figure 8. Percentage of reuse exploited with a varying cache size without interferences.

As pointed out above, a given instruction can have several types of reuse. Given a particular cache organization, we define the percentage of reuse that is exploited as the number of executed instructions that can exploit at least one type of reuse divided by the number of executed instructions that have at least one type of reuse.

Figure 8 shows the percentage of reuse that can be exploited for a varying cache size with a line size ranging from 8 to 64 bytes and neglecting the effect of cross-interferences. It can be seen that a cache size of about 1 Kbyte with lines greater than 8 bytes can capture some reuse for practically all the instructions of the analyzed programs with some reuse

Since almost all the references exhibit some type of reuse (as it has been shown in Figure 5) and this reuse can be exploited with a relatively small volume, a locality analysis that did not include a interference analysis would incorrectly conclude that it is worthwhile to cache almost all memory references. The percentage of reuse that would be exploited by this approach would be significantly lower than expected due to interferences. This is shown in Figure 9 for a varying cache capacity and line size. For instance, comparing the graphs of Figure 8 and Figure 9 for a 8 Kbytes capacity and 32-byte line size, it can be observed that without interferences nearly 100% of the reuse can be exploited but only 80% of it is actually exploited when considering the effect of interferences. For some programs with a high conflict miss ratio, the effect of interferences is even much more noticeable. This is the case for instance of *tomcatv*. Figure 10 compares the percentage of reuse that can be exploited with a varying cache size and a line size of 32 bytes. Whereas 1 Kbyte is enough to exploit all the reuse if there were not interferences, when considering interferences the reuse exploited with a 8 Kbyte cache is just 28%.

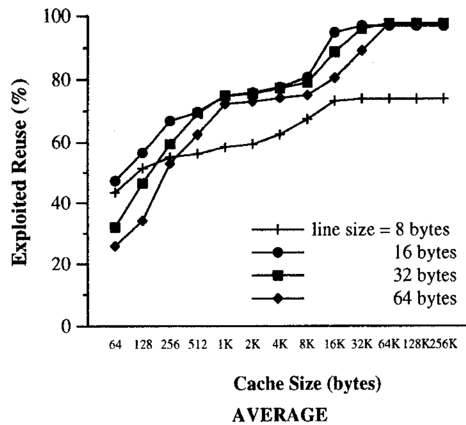

Figure 9. Percentage of reuse exploited with a varying cache size considering interferences.

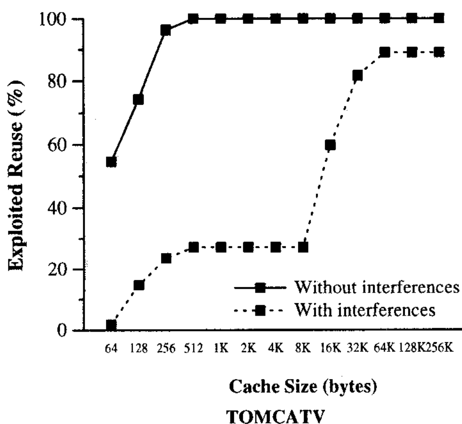

Figure 10. Percentage of reuse exploited with a varying cache size with/without interferences for tomcatv.

Selective caching can play an important role to reduce the negative effect of interferences. Applying the interference analysis presented in section 3.2, reuse can be exploited more effectively as it is shown in Figure 11. For instance, a 4 Kbytes selective cache can exploit more reuse than a 8 Kbyte conventional cache. The differences observed in Figure 11 are much higher for programs with many interferences (*tomcatv* and *swim*).

6. Applying the locality analysis to the selective and dual data caches

In this section, we present two types of results: first the accuracy of the locality analysis is evaluated, and then the

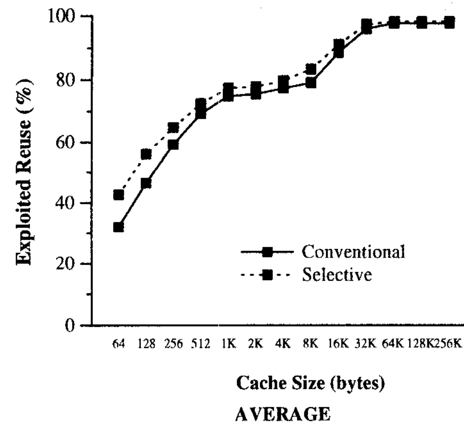

Figure 11. Percentage of reuse exploited with a selective cache, varying the cache size and compared with a conventional cache.

performance of the selective and dual data caches are compared against that of a conventional cache.

For the latter, it is assumed that the cache memory is connected to the next level of the memory hierarchy by means of a 8 byte bus. The latency of the next memory level is assumed to be 5 cycles plus an extra cycle per 64-bit word. The conventional and selective caches are direct-mapped, write-allocate and copy-back. Cache size is 8 Kbytes and block size is 32 bytes. The spatial module of the dual data cache is like a conventional cache. The temporal module is a very small (16 words) fully-associative buffer. This size has been proved to be sufficient to store practically all memory references that exhibit only temporal locality (see section 5).

Table 1 shows the results of the locality analysis applied to the selective cache.

Benchmark	%Bypass	%C-Hit	%B-Miss
tomcatv	42.94	71.18	84.37
swim	57.32	89.09	82.06
su2cor	0.06	93.11	0.83
hydro2d	0.05	84.44	69.15
mgrid	0.04	97.19	38.62
applu	1.92	94.51	9.67
turb3d	5.68	96.73	38.71

Table 1. Locality results for the selective cache

The first column indicates the percentage of memory references that are bypassed. The second column lists the hit ratio for the references that are cached. The last column shows the miss ratio of bypass references on a conventional

cache, which caches all references. The two last columns provide an evaluation of the locality analysis. An accurate locality analysis should result in a high hit ratio for cached data and in a high miss ratio for non-cached data. One can see in Table 1 that the hit ratio of cached references is near or above 90% for most programs. On the other hand, the miss ratio of bypassed references on a conventional cache is high excepting some cases in which the percentage of bypass references is very low and therefore the results are not significant (*su2cor, mgrid, applu* and *turb3d*).

Table 2 shows similar results for the locality analysis applied to the dual data cache. The second and third columns show the dynamic percentage of references labeled respectively as temporal or spatial by the locality analysis.The columns labeled as T-Hit and S-Hit list the hit ratio in the temporal and spatial modules respectively. The relatively high hit ratio of cached references prove again the accuracy of the locality analysis.

Benchmark	%Bypass	%Temp	%Spat	%T-Hit	%S-Hit
tomcatv	42.93	19.10	37.97	100.0	57.31
swim	57.32	21.34	21.34	100.0	79.48
su2cor	0.06	41.86	58.08	99.99	88.98
hydro2d	0.02	17.34	82.64	99.94	81.95
mgrid	0.03	59.70	40.27	100.0	94.10
applu	4.53	41.00	54.47	94.25	92.23
turb3d	5.38	79.29	15.33	99.90	80.38

Table 2. Locality results for the dual data cache

Figure 12 shows a comparison among conventional, selective and dual data caches in terms of hit ratio, average memory access time and average number of words fetched from the next memory level per memory reference. These figures are divided in programs with low locality (*tomcatv* and *swim*) and high locality (the others).

Figure 12 shows that the selective cache and the dual data cache provide a significant improvement in the first group of benchmarks. Compared with a conventional cache, they reduce the average memory access time in about 25% and the amount of data fetches in about 65%. Notice that this latter benefit may be very effective to reduce memory bandwidth, which is expected to be an important limitation for future microprocessors [4]. In the second group of benchmarks, where the memory behavior on a conventional cache was already good (see Figure 12b), the new cache architectures slightly improve the performance except for one benchmark (*applu*) which experiences a small increase in average memory access time.

The dual data cache provides very little improvement compared with the selective cache. This lack of significant enhancement may be due to the small number of cache

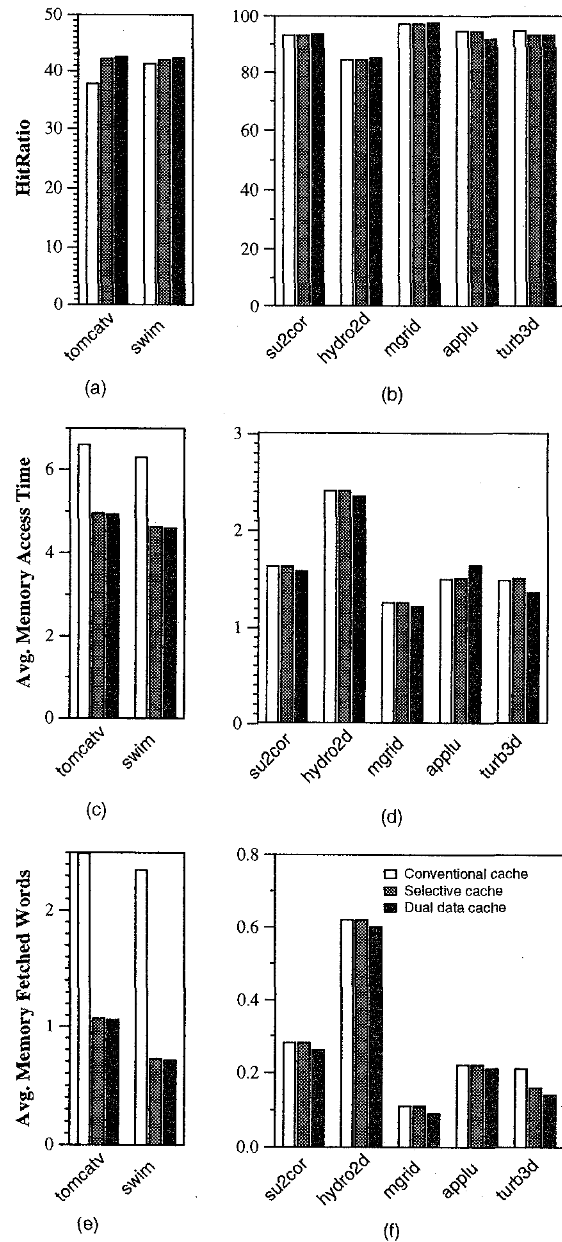

Figure 12. Comparison among conventional, selective and dual data caches

entries required to exploit temporal locality. Because of that, they cause few interferences in the selective cache.

The extra bit (or two bits for the dual data cache) used to manage the cache do not come free. The most obvious implementation would reduce the range of the constant displacement in memory instructions by a factor of two or four. If the displacement field has 16 bits (which is typical for current architectures) and can be used to address 64KB of data, in the modified instruction set we have that value reduced to 32KB or 16KB. This may incur in extra instruc-

tions if the addressed data is larger. We have measured for the benchmarks considered in this paper that only 2.41% of dynamic memory instructions executed need extra instructions (compared with memory instructions that has a displacement of 16 bits) using 14 bits of displacement, and 1.32% using 15 bits, which confirms that the penalty introduced by these additional instructions is negligible.

7. Conclusions

A static locality analysis oriented to the management of cache memories have been presented. The analysis presented extends previous proposals with an interference analysis step, which have been shown to be crucial for the accuracy of the analysis in some programs where conflict misses are dominant.

The analysis has been used to characterize the locality exhibited by loop nests of numerical codes. It has been shown that self-temporal and self-spatial reuse are dominant, closely followed by group-temporal reuse. It has been measured that about 35% of the references exhibit only temporal reuse and that this reuse can be exploited with a very small fully-associative buffer, which motivates the use of the dual data cache.

It has also been shown that interferences cause a significant degradation of cache memory. Interferences cause a large increase in the volume required to exploit a given percentage of reuse. This negative effect can be significantly reduced by a selective caching strategy.

Applying the locality analysis to the management of a selective cache and a dual data cache, it has been observed that these cache architectures provide a significant reduction in average memory access time and amount of data fetched from the next memory level, especially for programs with a poor locality, when compared with a conventional cache.

Acknowledgments

This work has been supported by the Spanish Ministry of Education under contract CICYT TIC 429/95 and by the Catalan CIRIT under grant FI-DT/96-3.083.

References

[1] S.G. Abraham, R.S. Sugumar, B.R. Rau and R. Gupta "Predictability of Load/Store Instruction Latencies" in *Proc. of MICRO-26,* pp. 139-152, 1993

[2] 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, Reseach Exhibit "Polaris at Work"*

[3] D. Bernstein, D. Cohen, A. Freund and D.E. Maydan "Compiler Techniques for Data Prefetching on the PowerPC" in *Proc. of PACT 95*, pp. 19-26, 1995

[4] D. Burguer, J.R. Goodman and A. Kägi "Memory Bandwidth Limitations of Future Microprocessors" in *Proc. of ISCA 96*, pp. 78-89, May 1996

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

[6] C-H. Chi and H. Dietz "Unified Management of Registers and Cache Using Liveness and Cache Bypass" in *Proc. of PLDI 89,* pp. 344-355, June 1989

[7] A. González, C. Aliagas and M. Valero "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality" in *Proc. of ICS 95*, pp. 338-347, 1995

[8] A. González, M. Valero, N. Topham and J.M. Parcerisa "Eliminating Cache Conflict Misses Through XOR-Based Placement Functions" in *Proc. of ICS 97,* 1997

[9] D.T. Harper III and D.A. Linebarger, "A Dynamic Storage Scheme for Conflict Free Vector Access" in *Proc of the 14th. ISCA,* pp. 72-77, 1987.

[10] A.S. Huang and J.P. Shen "A Limit Study of Local Memory Requirements Using Value Reuse Profiles" in *Proc. of MICRO-28*, pp. 71-81, Dec. 1995

[11] K. McKinley and O. Temam "A Quantitative Analysis of Loop Nest Locality" in *Proc of ASPLOS-VII*, pp. 94-104, Oct. 1996

[12] V. Milutinovic, B. Markovic, M. Tomasevic and M. Tremblay "The Split Temporal/Spatial Cache: Initial Performance Analysis" in *Proc. of SCIzzL-5*, March 1996

[13] T.C. Mowry, M.S. Lam and A. Gupta "Design and Evaluation of a Compiler Algorithm for Prefetching" in *Proc. of ASPLOS-V*, pp. 62-73, Oct. 1992

[14] T.C. Mowry "Tolerating Latency Through Software-Controlled Data Prefetching" *PhD Thesis*, Stanford University, CSL-TR-94-628, 1994

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

[16] O. Temam, C. Fricker and W. Jalby "Cache Interference Phenomena" in *Proc. of SIGMETRICS 94*, pp. 261-271,

[17] G. Tyson, M. Farrens, J. Matthews and A. Pleszkun "A Modified Approach to Data Cache Management" in *Proc. of MICRO-28,* pp. 93-103, Dec. 1995

[18] M.E. Wolf and M.S. Lam "A Data Locality Optimizing Algorithm" in *Proc. of PLDI 91*, pp. 30-44, 1991