Software Management of Selective and Dual Data Caches

F. Jesús Sánchez, Antonio González and Mateo Valero

Department of Computer Architecture
Universitat Politècnica de Catalunya
Campus Nord - c./ Jordi Girona, 1-3
08034 - Barcelona (SPAIN)

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

Abstract

The performance of cache memories relies on the locality exhibited by programs. Traditionally this locality has been divided into temporal and spatial, and conventional caches exploit both types of locality. For the sake of simplicity, cache memories manage all memory references in the same way regardless of their locality. This homogeneous treatment has an important drawback: references either with only temporal locality or without locality bring useless data into cache. This results in an unnecessary increase in memory access time and memory bandwidth, in addition to the penalties introduced by potentially removing useful data from the cache.

In a previous paper, we proposed two novel organizations called Selective Cache and Dual Data Cache as an attempt to avoid this loss of performance by managing each memory reference in a different way depending on its type of locality. In those proposals, the locality of each reference was dynamically estimated at run-time based on its history. In this paper, we present a novel design of the Selective and Dual Data Caches in which the locality estimation is based on a locality analysis performed at compile-time. We show that previous proposals on locality analysis are not appropriate when the programs have a high conflict miss ratio. This paper extends this locality analysis by introducing a compile-time interference analysis that significantly improve the performance of the system.

1. Introduction

Cache memories play an important role to handle the increasing gap between processor and memory speed. However, 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. Many new organizations have been proposed in order to improve cache performance: victim [11], assist [6], skewed-associative [15], and column-associative [2] caches are some relevant examples. In almost all of them, all memory references are handled in the same way: if the reference misses, a new block is brought into cache at the expense of replacing another. They do not provide a different treatment depending on the type of locality.

This approach is targeted to exploit both temporal and spatial locality. However, when spatial or both types of locality are not exhibited by a reference, it may result in a significant performance degradation. This loss of performance is due to the following issues:

- If the reference only has temporal locality, the memory bandwidth and memory storage consumed by the remaining data in the cache block is wasted.
- If the reference does not have any type of locality, in addition to the previous problem, the memory storage consumed by the data itself is also wasted.

This causes an unnecessary movement of data among the levels of the memory hierarchy that results in an increase in the miss ratio and average memory access time.

In a previous paper we proposed two novel organizations called Selective Cache and Dual Data Cache [8] that try to overcome these problems by handling each memory reference in a way that depends on its type of locality. In these designs, the estimation of the locality of memory references was performed at run-time based on their history. This approach has three main drawbacks: a) the amount of hardware required by the locality prediction scheme is high, b) the mechanism is unable to detect conflict misses caused by interferences among different memory instruc-
tions and c) does not identify reuse among several instructions (group reuse).

In this paper we propose an alternative design for the Selective Cache and Dual Data Cache that is based on a compile-time locality analysis. This locality analysis is inspired in the proposals presented in [17] and [14]. However, these proposals do not consider conflict misses when performing the locality analysis. This may be cause the analysis to be very inaccurate for programs with a high conflict miss ratio. To overcome this problem, we propose to extend the 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 rest of the paper is organized as follows. Section 2 reviews the most relevant related work. In section 3, the Selective and Dual Data Caches are presented. Section 4 describes the compile-time locality analysis. Section 5 evaluates the proposed cache architectures, and the main conclusions are drawn in section 6.

2. Related work

Selective caching (also called cache bypassing) has also been proposed by other authors for both instruction and data caches. Some remarkable works for data caches are [7], [1] and [16]. The scheme proposed in [7] is based on a compile-time estimation of data lifetimes. The mechanism proposed in [1] identifies non-cacheable data by means of profiling. The approaches proposed in [16] are either hardware-based or make use of simple schemes based on profiling.

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 [6]. In this machine, every memory instruction includes a hint called spatial locality only that indicates that data exhibits spatial locality but not temporal locality. A similar hint is used by the cache architectures proposed in this paper. The first level of the memory hierarchy of the PA-7200 consists of two modules, like the Dual Data Cache. These two modules are 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. On the other hand, the Dual Data Cache has one module for data with only temporal locality and another for data with spatial or spatial and temporal locality.

3. Selective and Dual Data Caches

Caching a data without any type of locality has a twofold negative effect:

- Replace a block that could be reused by a non-reusable one, which results in additional cache misses.
- Fetch into cache an entire block (of which only one word is useful), increasing the memory access time and the traffic with the next memory level.

The Selective Cache tries to avoid these drawbacks using a bypass strategy. With this aim, memory instructions are tagged at compile-time as cacheable or bypass references. All references check the data cache to seek for the required data. If a cacheable reference misses, it brings to cache a new block containing the data. However, if a bypass reference misses, the data is not brought into cache. In this case, only the missing word is requested to the next memory level and it is delivered to the processor.

The objective of the Selective Cache is to reduce the penalty caused by data that do not exhibit any type of locality. The Dual Data Cache, in addition, tries to reduce the penalty caused by data with only temporal locality.

A Dual Data Cache is composed of two modules, called temporal and spatial. The former is targeted to exploit just temporal locality and the latter is designed to exploit spatial locality, in addition to temporal 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 cache architecture, memory instructions are tagged at compile-time with one of the following three values: spatial, temporal or bypass. All references check both modules in parallel to seek for the required data. If a reference with a spatial or temporal tag misses in both modules, a new block is brought into the module indicated by the tag. Instructions tagged as bypass are handled like in the Selective Cache.

Note that in some cases, the same data element can reside in both modules at the same time. Suppose that a data element d is read into the temporal cache and then modified. If a copy-back policy is used, the higher memory level is not immediately updated. Suppose that later on, a nearby data element that causes a cache miss is referenced. If the prediction for such reference is spatial, it may happen that an old copy of d is brought into the spatial cache. To solve this consistency problem, when a data is found in
both modules, it is always selected the one from the temporal module. In addition, whenever a dirty block of the temporal module is replaced and it is also present in the spatial module, the spatial module is updated.

The main difference between the cache architectures presented here and those proposed in [8] is that in the latter paper the memory references were tagged at execution time using an additional hardware called locality prediction table.

4. Locality analysis at compile-time

The goal of the locality analysis is to determine statically the type of locality exhibited by memory references in order to tag them adequately according to the cache organization being used. This tag will define some hint bits in the memory instruction (one bit in the Selective Cache and two bits in the Dual Data Cache) that determine, at execution-time, the caching strategy.

We restrict the locality analysis to references inside loops, which represent the majority of references. This analysis is performed just for array references where the array indices are affine (i.e., linear) functions of surrounding loop indices. The remaining references are tagged with a default value that corresponds to the assumption that they exhibit spatial and temporal locality. In the analyzed benchmarks, these references represent a small percentage.

The locality analysis that we use has many common points with other previous proposals with different goals: improve data locality [17] and software prefetching [4] [13] [14]. These proposals divide the locality analysis into two steps: (i) reuse analysis and (ii) volume analysis. The main extensions that we propose are the addition of a cross-interference analysis step oriented to anticipate conflict misses and the enhancement of the volume analysis in order to consider auto-interferences.

Below we first summarize the reuse and volume analysis. Then, the interference analysis is described.

4.1. Nomenclature

The following nomenclature will be used in the rest of this paper.

We consider references inside loop nests not necessarily perfectly nested. Given a loop nest of depth \( N \), \( I_i \) \((i = 1, ..., N)\) identifies index of loop \( i \). Before the locality analysis, the loops are normalized (i.e., \( I_1 \) starts from 1 and is updated with step 1). \( B_i \) identifies the upper bound of loop \( i \).

For each affine array reference, \( M \) represents the number of dimensions of the array; \( D_j \) \((j = 1, ..., M)\) denotes the number of elements of dimension \( j \); \( c_j \) denotes the expression (linear function of loop indices) that is used to index dimension \( j \).

We will use this Fortran code as a sample program to illustrate each step:

```fortran
DO J = 1, 10, 1
  A(J) = 0.0
ENDO
DO I = 1, 1000, 1
  B(I,J) = C(I,J) + C(I+1,J)
  D(I,J) = E(1,J)
ENDDO
ENDDO
```

4.2. Reuse analysis

The goal of this analysis is to determine the intrinsic reuse for each memory reference, assuming an infinitely large cache.

This analysis was proposed by M.E. Wolf et al. [17]. Each vector reference is represented by means of a matrix notation which maps \( n \) loops indices onto \( m \) array indices. This notation is made up of a matrix (it represents the loop indices that appear in the reference) and a constant vector. The reuse type is determined using linear algebra operations over this elements such as nullspace computation, comparison, etc. (see [17] for details).

For each reference, it obtains the reuse that the reference exhibits across each loop. Five types of possible reuse are distinguished:

- **Unknown**: memory instructions outside loops or with irregular patterns. By default these references are always cached for the Selective Cache and allocated to the spatial module for the Dual Data Cache.
- **Temporal**: when an instruction accesses the same location in multiple iterations.
• **Spatial**: when an instruction accesses near locations in multiple iterations.

• **Group** (temporal or spatial): when different instructions access near locations (or the same) in multiple iterations. The instruction that first access each location is called leading instruction and the remaining are referred as to trailing instructions.

• **No-Reuse**: in any other case. This references are always tagged as bypass.

In our example the reuse analysis will produce the following results:

- **A(J)**: Spatial reuse in loop J
- **B(I,J)**: No reuse in loop J
- **C(I,J)**: Group reuse with C(I+1,J) in loop I
- **D(I,J)**: No reuse in loop J
- **E(I,J)**: No reuse in loop J
- **F(J)**: Temporal reuse in loop I

The results are represented as a vector space that identifies the loops in which reuse is found (a dimension corresponds to a loop). We distinguish between two types of temporal or 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 to 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 reuse analysis step defines the tags for those references with either unknown or no reuse. For the rest of the references, their tag is computed in later steps.

### 4.3. Volume analysis

A reference with some type of reuse exhibits locality if the reuse of data occurs before the data is displaced from the cache by other memory instructions. This is estimated by computing the volume of data that is referenced between two consecutive reuses.

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

- **Type of reuse**: calculated in the previous step.
- **Loop bounds**: they may be unknown at compile-time.

In this case we need to estimate it. We use the approximation proposed by D. Bernstein et al. ([4]): each memory reference \(R\) is represented in the following way:

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

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:

\[
B_i = \frac{r_{i-1}}{r_i}, \quad \text{if } i > 1
\]

\[
B_i = \frac{\text{ArraySize}}{r_1}, \quad \text{if } i = 1
\]

(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 \((c_i < D_i)\).

(c) **Cache features**: in particular, the volume analysis requires to know the number of blocks and the block size.

The analysis follows these steps:

- Calculate for each memory reference the number of cache blocks that are accessed in one iteration of every loop. Figure 2 shows the contributed volume of a reference \(R\) to a loop \(i\), which is denoted by \(V(R,i)\) (note that \(V(R,N) = 1\)). If the reference does not have any type of reuse it is not cached, and therefore it does not contribute to the volume.

**Figure 2.** Contributed volume of a reference to a loop. \(r^l\) and \(r^T\) are the coefficients of the leading and trailing references respectively.
• According to its reuse type and the loop volume in which this it is exploited, the reuse becomes locality in a loop \( b \) if \( \sum_{v \in R} (b, r) \leq \text{NumberOfBlocks} \).

Notice that some other authors like [14] propose to account for cache conflicts by setting the “effective” cache size to be a fraction of the actual cache size. 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 conflicts effect. Besides, conflicts are not uniformly distributed over all memory references and therefore, their contribution should be measure for each reference independently. The effect of memory conflicts may be very high for some programs as it is shown later in this paper. Therefore, a more accurate estimation is crucial for the performance of the locality analysis.

To account for the effect of conflicts in a more suitable way, we have included in the volume contribution of each reference a factor that accounts for the effect of using a stride that do not make use of all the cache blocks. This factor is \( 2^{\text{stride family}} \) in Figure 2. In this context, the stride is defined as difference between two consecutive effective addresses generated by the same memory instruction and it is measured in number of block units. The stride family is the family to which the stride belongs. The stride family defined by \( x \) is the set of strides \( \sigma \cdot 2^x \) with \( \sigma \) odd. 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. In addition, we have included in the locality analysis an interference analysis step as it is described in the next section.

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Contributed volume to loop ( I )</th>
<th>Contributed volume to loop ( J )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B(I, J) )</td>
<td>1</td>
<td>250</td>
</tr>
<tr>
<td>( C(I, J) )</td>
<td>1</td>
<td>250</td>
</tr>
<tr>
<td>( C(I+1, J) )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( D(I, J) )</td>
<td>1</td>
<td>250</td>
</tr>
<tr>
<td>( E(I, J) )</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>4</td>
<td>760</td>
</tr>
<tr>
<td>( A(J) )</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>4</td>
<td>761</td>
</tr>
</tbody>
</table>

Consequently, only reuse across loop \( I \) is translated into locality. Therefore, \( A(J) \) has no locality.

After this analysis is done, the locality for each reference in each loop is determined. The references with no locality are tagged as bypass, and the rest as cacheable in the Selective Cache and as temporal or spatial in the Dual Data Cache. An additional constraint in the Dual Data Cache is that the references that exhibit group locality have to be allocated to the same module.

### 4.4. Interference analysis

Traditionally three types of cache misses are distinguished: compulsory, capacity and conflict. The volume analysis tries to avoid capacity misses using the amount of data accessed by every memory reference and checking if the cache size is sufficient to store it.

Conflict misses occurs when different data is mapped onto the same cache block. These references can not exploit its reuse because other references will replace the data.

Conflict misses are the most important source of cache misses for some programs. For instance, for a 8Kb direct-mapped data cache with a block size of 32 bytes, we have measured a conflict miss ratio of 41% and 48% for the tomtomatv and swim respectively (from the SPEC95). We have applied a locality analysis based just on the reuse and volume phases to this program. The result is that it can only identify less than 0.1% of the memory instructions as instructions without locality, while around one half of the instructions causes cache misses. Therefore, any decision based on this locality analysis may be very misleading. An interference analysis may significantly increase the performance for this type of programs.

The interference analysis that we propose is based on knowing at compile-time the base memory address of global and local variables, using the symbol table of the program. Extending this information to other type of variables dynamically allocated is a 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. We assume here a direct-mapped organization for the conventional, Selective and spatial module of the Dual Data Cache. The extension for other organizations is straightforward. 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:

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

2) Build an interference graph for each basic block. Only references that satisfy the following three conditions are considered:

• They have information in the symbol table (the variable is not a parameter)
• They have some type of reuse.
For the Dual Data Cache, only references with spatial reuse are considered since, as we will see later, the temporal module is a small fully-associative buffer.

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:

\[
| R_1 \mod \text{CacheSize} - R_2 \mod \text{CacheSize} | < \text{BlockSize}
\]

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

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

If we apply this analysis to our example the results are as follows:

\[
B(I,J) \quad \square \quad \bigcirc
\]

\[
C(I,J) \quad \square \quad \bigcirc
\]

\[
C(I+1,J) \quad \bigcirc \quad \rightarrow \quad \text{Non cacheable}
\]

\[
D(I,J) \quad \square \quad \bigcirc
\]

\[
E(I,J) \quad \square \quad \bigcirc
\]

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 later tagged as bypass and will not be cached despite of having reuse.

5. Performance results

5.1. Experimental framework

The cache architectures presented in this paper have been evaluated for 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 [3]. 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. 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.

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 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 (up to 16 words) fully-associative buffer. This size has been proved to be sufficient to store practically all memory references that exhibit only temporal locality. Figure 3 shows the hit ratio of this module for a varying number of blocks. It can be observed that 16 blocks are enough to capture the majority of temporal-only locality.

5.2. Performance evaluation

In this section, we present two types of results: first the accuracy of the locality analysis is evaluated, and then the performance of the Selective and Dual Data Caches are compared against that of a conventional cache.

Table 1 shows the results of the locality analysis applied to the Selective Cache. The first column indicates the percentage of memory references that are bypassed (tagged as bypass by the locality analysis). The second column lists the hit ratio for those references that are cached. The last column shows the miss ratio of bypass references on a conventional cache. The two last columns provide an evaluation of the locality analysis. An accurate locality analysis will 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 nearly or above 90%. On the other hand, the miss ratio of bypass tagged 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 and applu).
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.

Notice that a high percentage of memory references exhibit only temporal locality but this locality can be captured with a very small buffer as shown in Figure 3. The relatively high hit ratio of cached references prove again the accuracy of the locality analysis.

Figure 4 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 4 shows that the Selective Cache and the Dual Data Cache provide a significant improvement in 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 [5]. In the second group of benchmarks, where the memory behavior on a conventional cache was already good, the new cache architectures slightly improve the performance except for one benchmark (applu) which experiments 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 entries required to capture temporal locality. Because of that, they cause few interferences in the Selective Cache.
6. Conclusions

Two novel software-managed cache architectures have been proposed, which are called Selective and Dual Data Caches. Their management is based on a compile-time locality analysis. The proposed locality analysis is an extension of previous proposals that includes, as a main contribution, an interference analysis targeted to identify conflict misses.

The simulations results show that the locality analysis is accurate for 80-90% of the memory references and that the interference analysis plays a crucial role for programs with a high conflict miss ratio.

When compared with a conventional cache, the new 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.

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


[8] 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


