A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality

Antonio González, Carlos Aliagas and Mateo Valero

Universitat Politècnica de Catalunya
Departamento de Arquitectura de Computadores
Campus Nord - Módulo D6
c/ Gran Capitán, s/n
E-08071 - Barcelona (Spain)

E-mail: antonio@ac.upc.es

Abstract

Current data cache organizations fail to deliver high performance in scalar processors for many vector applications. There are two main reasons for this loss of performance: the use of the same organization for caching both spatial and temporal locality and the “eager” caching policy used by caches. The first issue has led to the well-known trade-off of designing caches with a line size of a few tens of bytes. However, for memory reference patterns with low spatial locality a significant pollution is introduced. On the other hand, when the spatial locality is very high, larger lines could be more convenient. The eager caching policy refers to the fact that data that miss in the cache and is required by the processor is always cached (excepting writes in a no write allocate cache). However, it is common in numerical applications to have large working sets (large vectors, larger than the cache size), that result on a swept of the cache without any opportunity to exploit temporal locality. In addition, they replace some other data that may be required later.

In this paper, a novel data cache organization, called *dual data cache*, is presented. To our knowledge, this is the first time a cache with independent parts for managing spatial and temporal locality is proposed. In this way, both types of locality can be more efficiently exploited. In addition, the dual data cache implements a lazy caching policy, which tries not to cache something until a benefit (in terms of spatial or temporal locality) can be predicted. For both purposes, the dual data cache makes use of a *locality prediction table*, which is a history table with information about the most recently executed load/store instructions. In addition, a simplified implementation of the dual data cache, which is called *selective cache* is also presented.

Keywords

Cache memory, data locality, cache organization.

This paper has been published in the ACM 1995 International Conference on Supercomputing, Barcelona, Spain, ISBN: 0-89791-728-6, pp. 338-347.

1. Introduction

Many real programs, in particular numerical applications, make use of vector data. Vector processors, are aware of this data type and include some instructions that manipulate vectors. However, this is not the case of scalar processors, which usually do not have any specific instruction for vectors. In such processors, operations on vectors are performed by means of a sequence of scalar operations that manipulate individual elements of vectors. Although the computation power of current superscalar processors may be very effective for performing such operations, the processor performance is usually degraded by the low efficiency of the data cache memory.

In the context of this document, a vector is a number of elements identified by its initial address and a constant stride (distance between successive elements of the vector).

The poor performance of current data cache organizations for dealing with vector data is mainly due to the following reasons:

- Vector data usually require very large working sets which result in a swept of the cache without any chance to exploit temporal locality.
- In some cases, vectors are accessed with a stride not equal to one. In those cases, the ability of a cache to exploit spatial locality is significantly reduced. Because conventional cache memories deal with lines of consecutive data, these strides cause a significant cache pollution. This negative effect increases as the stride is higher, and at the worst, it could end up with just one useful data element in every cache line.
- Finally, when the stride measured in cache lines and the number of sets in the cache are not coprime, not all the sets are used to store the elements of the vector, which in turn aggravates the problem caused by large working sets. For instance, suppose a direct mapped cache with 512 lines and a line size equal to the size of a vector element. Then, it can store a vector with stride 3 and length 512. However, if the stride is 2, the length of the vector can not be greater than 256. In other words, some strides make the vector accesses to interfere with themselves, even if the vector is smaller than the cache size.

On the other hand, if the stride is constant, accesses to vector elements are quite easy to predict and therefore it is possible to implement a prefetch mechanism to take advantage of this fact.

We can find in the literature different proposals to deal with some of the above mentioned issues. However, to our knowledge, there is no proposal trying to deal with the four issues (large working sets, strides different to one, interferences and prefetching) at the same time. In this paper, we present a novel cache organization that incorporates some features to exploit the characteristics of vector as well as scalar data. In particular, our cache organization tries to avoid the swept effect of large vectors, reduces the pollution caused by non unit strides, decreases the penalty caused by self-interferences due to strides coprime with the number of lines, and can take advantage of prefetching of vector elements.

The proposed data cache organization, called *dual data cache*, can be used in any (super)scalar processor since it does not require any particular feature in the instruction set architecture. In addition, a simplified implementation of the dual data cache, which is called *selective cache* is also proposed.

The performance of the dual data cache and the selective cache has been evaluated for a set of benchmarks by means of simulation. The results show a significant performance improvement when compared with a conventional cache organization with similar hardware cost.

The rest of this paper is organized as follows. Section 2 summarizes the work related with the topic of this paper. Then, the proposed cache organization is described in section 3. Section 4 presents a qualitative analysis of the performance of the proposed mechanism. Results obtained by simulation are discussed in section 5. Finally, section 6 presents some concluding remarks.

2. Related Work

As described in the previous section, there are four main issues to be considered when designing a data cache for a (super)scalar processor in order to achieve high performance when the processor is using vector data:

- *Large working sets*.
- *Pollution* due to non-unit strides.
- *Interferences* when the stride and the number of sets are not coprime.
- *Prefetching*.

In this section we summarize the most important solutions proposed in the literature to deal with individual or a subset of these four issues.

Block algorithms [12,19] is a technique that tries to reduce the negative effect of large working sets. Block algorithms can be developed by the programmer or, as claimed by several authors ([4] among others), it may be preferable to leave this task to the compiler. In any case, blocking requires a detailed analysis of data dependencies, but in some cases it is very difficult or even impossible to identify some dependencies at compile time. In these cases, the programer or compiler must generate a conservative code that may reduce the possibilities of applying blocking. The cache organization we propose does not exclude the use of block algorithms. Blocking is a software technique that can be used for any cache memory in order to increase the locality of the algorithm, so it can also be applied in the presence of a dual data cache or a selective cache. In addition, the dual data cache implements mechanisms that avoid the swept effect that large working sets would cause when blocking is not satisfactory.

The pollution caused by non-unit strides can be avoided in some cases by means of copying [16]. The main idea of this technique is to copy the elements of non-unit stride vectors into auxiliary vectors with stride one and then performing all the operations using the latter vectors. The main problem with copying is that its overhead may be high when the data is not much reutilized. In addition, the effectiveness of copying is constrained by the fact that the compiler has a limited knowledge of the program.

The performance degradation caused by interferences has been studied by Q. Yang and L.Y. Yang in [20]. However, their proposal focuses on vector processors whereas the present work concentrates on scalar processors. They proposed a direct-mapped cache using Mersenne numbers. A Mersenne number is of the form $2^x - 1$ for some x that makes it a prime. Due to the properties of such numbers, the modulo operation required by the mapping function can be done quite effectively in the context of vector processors. However, it would be very costly if we used it in scalar processors since every cache request would require to perform the modulo operation which requires several additions and therefore would considerably slow down the processor. Many other proposals to reduce the negative effect of interferences have recently appeared in the literature. Some of the most representative are: the victim cache [10], the skewed-associative cache [14], the column-associative cache [2], the dynamic exclusion replacement policy [13], the Half-and-Half cache [17] and the PA7200 Assist cache [11].

Different prefetching techniques have been proposed in the literature. Software techniques are managed by the compiler and usually make use of non-blocking caches in order to issue memory requests in advance. They require very simple hardware support and rely on compile-time information in order to predict future memory requests [6]. Hardware techniques for prefetching vector data require some mechanism to recognize references to vector data. Since the processor does not have vector instructions, this is usually done by means of a history table that keeps informations about load/store instructions. When successive executions of the same load/store instruction are found to generate addresses separated by a constant stride, one can assume that a vector is being accessed. In that case, the computed stride can be used to predict future data accesses to the same vector. The cache is first searched for these data and, in case of miss, a request to the external memory may be issued.

Three of the most significant contributions in the area of hardware prefetching are the *Preloading Scheme* proposed by J-L. Baer and T. F. Chen [3], the *Speculative Prefetching* proposed by Y. Jegou and O. Temam [9], and the *Stride Directed Prefetching* developed by J.W. Fu, J.H. Patel and B.L. Janssens [7].

The Preloading Scheme is based on having two program counters. The first one points to the instruction currently in execution while the other one points to several instructions ahead. When the look-ahead PC finds a load/store instruction, the history table (called reference prediction table by the authors) is checked and the address of this new reference is predicted using past information. If the predicted reference is not in cache, then a request to the external memory is issued. This mechanism requires a considerable amount of hardware since, in addition to the history table, a branch prediction table is needed in order to allow the look-ahead PC to bypass branches. The performance is quite dependent on the branch prediction accuracy which in turn is strongly dependent on the distance between the PC and the look-ahead PC.

The Speculative Prefetching is simpler since it does not make use of any branch prediction mechanism. It uses a history table similar to that of the Preloading Scheme but the prefetching is done at the same time a load/store instruction is executed. That is, when the processor executes a load/store instruction it tries to predict and prefetch, using the history table, the memory reference that will be generated in the next execution (or after *n* executions) of the same instruction. An appropriate choice of the prefetch distance, *n*, is crucial for the performance of the mechanism.

The Stride Directed Prefetching is conceptually similar to the previous schemes. It tries to predict future references to a vector by identifying successive memory references separated by a constant stride.

Another hardware prefetching mechanism is the Stream Buffer [10], which is targeted to benefit from references with unit stride.

A combination of hardware and software support for prefetching has been proposed by other authors, [15, 1, 11] among others.

3. The Dual Data Cache

We assume that the processor, like most current microprocessors, has separate instruction and data caches. Here, we center on the design of the data cache. The proposed data cache organization comes from the observation that different types of data usually exhibit different locality properties when they are referenced. In particular:

- Scalar variables exhibit a moderate degree of spatial locality. Two variables that are stored next to each other in memory are not always referenced close in time. However, its temporal locality usually is high (counters, vector indices, etc., are used repeatedly in many cases). Therefore, for such variables the stress should be put on exploiting temporal locality.
- Vectors with a large stride have a poor spatial locality and the temporal locality may or may not be high. However, since vectors may be very large, temporal locality should be exploited only for those vectors that not exceed the cache storage capacity. Otherwise, the above mentioned swept effect would cause a loss of performance.
- Vectors with a small stride exhibit a very high spatial locality. This spatial locality can be exploited no matter the length of the vector. For the temporal locality the same remarks as in the previous paragraph apply here.
- Random accesses (i.e., sparse matrix computations) do not exhibit in general any type of locality.

In consequence, there is a type of data (scalars and not very large vectors with a large stride) for which it seems most appropriate to spend the hardware resources to exploit just temporal locality, another type of data (not very large vectors with a small stride) for which the hardware should be dedicated to exploit both spatial and temporal locality, a third type of data (very large vectors with a small stride) for which the exploitation of just spatial locality is rewarding, and a last type of data (very large vectors with a large stride and random accesses) for which the most effective decision should be not to cache them.

The proposed data cache, which is called *dual data cache*, consists of two independent memories, or subcaches. One is called the *spatial cache* because it is designed to exploit spatial locality, in addition to temporal locality. The other is called *temporal cache* since it is targeted to exploit just temporal locality. Both subcaches work independently and in parallel. Figure 1 depicts the main parts of the dual data cache.

When the processor issues a memory reference, both caches are looked up at the same time and, depending on the result, one of the following actions is taken:

- If the required data is only in one of the subcaches, the data is read or written in that subcache. This is a cache hit

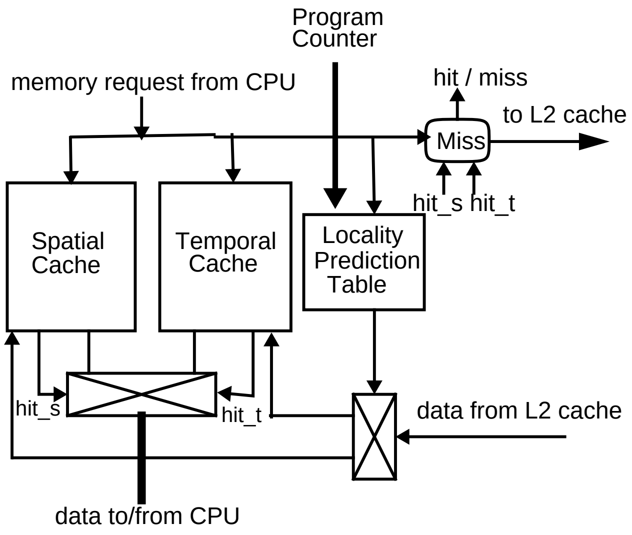


Figure 1: Block diagram of the dual data cache.

- If the required data is found in both subcaches, it is read from the temporal cache or written into both two in parallel. This is again a cache hit.
- If the required data is not in any subcache, a cache miss occurs. In this case, the processor is stalled and the required data is brought from the next level of the memory hierarchy. This data may be placed in just one of the two subcaches or may be not cached anywhere, depending on predicted type of locality for this memory access. This prediction is made by means of a history table, which is called *locality prediction table*, whose operation is explained later on. If spatial locality is predicted for this memory reference, then the missed data is placed in the spatial cache. If temporal locality is predicted and the referenced element is an scalar or if it belongs to a vector that will not cause self-interference in the temporal cache, the data is placed in that cache. Otherwise, it is not cached anywhere. Even if a vector exhibits temporal locality (but not spatial locality), caching it would be detrimental if it interfered with itself.

Both the spatial and temporal caches have the same organization as conventional caches with the characteristic that the line size of the spatial cache is large (several tens of bytes or more) whereas the temporal cache has a small line size (a few bytes). In this way, the spatial cache can exploit both spatial and temporal locality whereas the temporal cache exploits just temporal locality.

The predicted locality for a memory reference is based on guessing whether the accessed data is an scalar, or an element of a vector, and in the latter case, it also depends on the stride and the size of the vector. These attributes are estimated by means of the locality prediction table. The locality prediction table is based on the history table that was proposed by Baer and Chen [3], Fu, Patel and Janssens [7], and Jegou and Temam [9] as part of a hardware mechanism for prefetching vector data. The locality prediction table has a moderate number of entries (it is suggested in [9] that a number about 256 could be sufficient to catch most of the references). Each entry contains information about a recently executed load/store instruction. This information consists of the following fields:

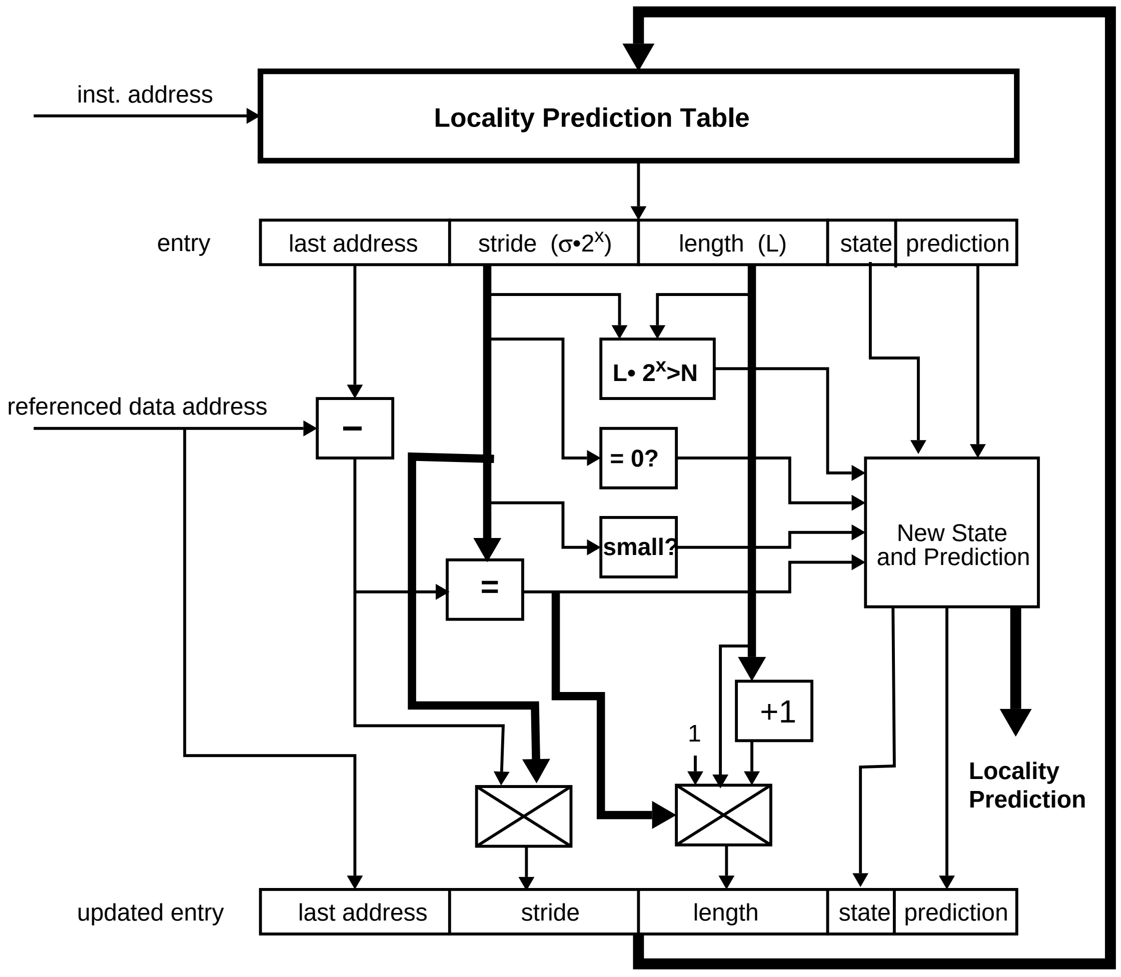


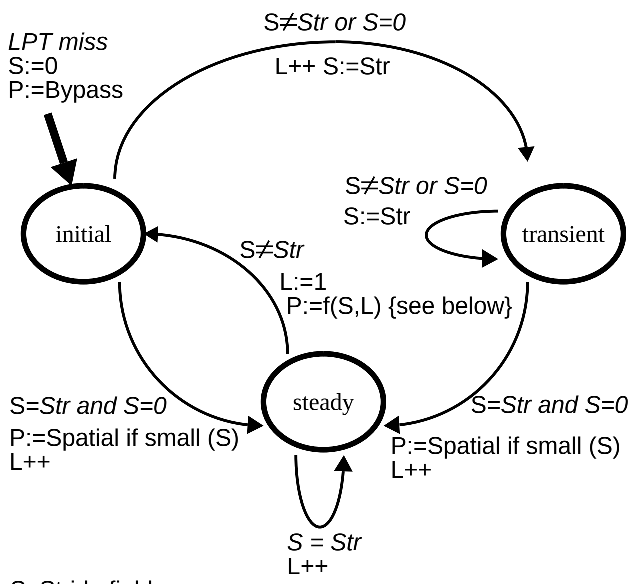
Figure 2: Hardware to update the locality prediction table.

- *Instruction address*: The address of the load/store instruction. This field is the identifier of the entry.
- *Last address*: The last data address referenced by that instruction.
- *Stride*: The difference between the last address and the second last address referenced.
- *Length*: The number of consecutive elements accessed by this load/store instruction with the same stride.
- *State*: This field keeps information about the past behavior of this load/store instruction regarding its stride.
- *Prediction*: This field keeps information about the past behavior of this load/store instructions regarding its prediction.

For every cache miss, the locality prediction table decides where the missed data is cached. The locality prediction table is accessed at the same time as the cache memory. If the load/store instruction that caused the cache miss is not in the locality prediction table, the missed reference is stored in the default subcache (it could be spatial or temporal; we have experimentally checked that the most suitable default value is temporal) once it is brought from the next level of the memory hierarchy. Otherwise, when the load/store instruction is found in the locality prediction table, the information in the state and prediction fields, after being updated, are used to predict the expected type of locality for such reference and therefore, it determines the place where the missing data will be placed. The prediction can be: *spatial* (i.e., in case of miss, the data is placed in the spatial cache), *temporal* (i.e., in case of miss, the data is placed in the temporal cache) or *bypass* (i.e., in case of miss, the data is not cached anywhere).

Figure 2 shows the required hardware to implement the locality prediction table. The hardware mainly consists of a table, two adders, two full comparators, two equality comparators, a shifter and several multiplexers.

The locality prediction table is managed as a cache. It may be direct mapped, set associative or fully associative. Every time a load/store instruction is executed, the locality prediction table is looked up. If the instruction address is not in the table, one of the entries of the table is allocated to this instruction and it is initialized as follows: a) the address of the instruction, b) the address of the



S: Stride field
Str: Difference between current address and previous one
L: Length field
P: Prediction field

```
f(S,L) := if not small(S) then
              if no-self-interference then Temporal
              else Bypass
```

Figure 3: State transitions. In italics the condition that fires the corresponding transition. The rest shows how the LPT entries are updated.

referenced data, c) stride equal to zero, d) length equal to one, e) state equal to initial, and e) prediction equal to bypass. The instruction address will be used as the tag to identify the entry and therefore it will maintain the same value as long as the entry remains in the table. The last address is always updated with the last referenced address. The other fields may vary every time the same load/store instruction is executed, as it is explained below. Figure 3 illustrates how these fields are updated.

The stride field may be either updated with the difference between the previous referenced address and the current one or left unchanged. The length may be incremented by one, reset to 1 or left unchanged.

As it is shown in figure 3, the state field can take one out of three different values:

- *Initial*: This state is reached after the first reference to a vector. In this case the locality prediction is the default value.
- *Transient*: The entry has this state as long as two consecutive references have different stride. In this case the prediction is also the default value.
- *Steady*: This state indicates that successive strides are equal. In this state the prediction is given by the updated value of the prediction field.

Once in the initial state, the update of a locality prediction table entry proceeds as follows. If the current stride (Str in figure 3) is equal to the previous one (S) and they are different from zero, the updated state is steady and the length is incremented. The prediction field becomes spatial if the stride is small (in the experiments presented later, it is assumed that a stride is small when it is lower than the spatial line size); otherwise it maintains the same value. If the current stride is different from the previous one or the latter is equal to zero, the new state is transient, the length is incremented and the stride field is updated.

The entry is kept in the transient state as long as the stride varies or it is equal to zero. In these cases, the stride field is updated for every execution of the corresponding load/store instruction. When the previous conditions does not hold, the state becomes steady. In this case, the length is incremented and the prediction is updated with spatial if the stride is small.

The entry remains in the steady state as long as the stride does not vary. Every time the corresponding load/store instruction is executed, the length is incremented. When eventually the stride varies, the state shifts to initial and the length is reset to one. At this moment, if the stride field is "large" (in the experiments this means greater than the spatial line size), the prediction field is updated depending on the vector length and the family to which the value of the stride field belongs as explained below. The stride family defined by x is the set of strides $\sigma \bullet 2^{x}$ with σ odd. All the strides belonging to the same family (e.g., $12=3 \bullet 2^{2}$ and $20=5 \bullet 2^{2}$ belong to family 2) have the same behavior from the point of view of self-interference [8, 18]. If the value in the stride field is "small" the prediction does not vary (it remains spatial). Otherwise, if the vector length multiplied by 2^{x} is not higher than the size of the temporal cache, the prediction is temporal; if it is higher, the prediction is bypass, since the vector would interfere with itself.

Finally, the proposed scheme could also incorporate speculative prefetching [9], stride directed prefetching [7] or preloading [3] at a very low additional hardware cost, since the reference prediction table required by these schemes is contained in the locality prediction table. Nevertheless, in the figures presented in section 5, a simpler prefetching scheme has been assumed. Whenever there is a cache miss whose prediction is spatial, in addition to the missing line, *p* successive lines are also brought into the spatial cache. The optimal value of *p* has experimentally been determined to be equal to 1.

Note that in some cases the same data element can reside in both caches at the same time. Suppose that a data element *d* is read into the temporal cache and then modified. Due to the copy-back policy 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 subcaches, it is always selected the one from the temporal cache. In addition, whenever a dirty line of the temporal cache is replaced and it is also present in the spatial cache, the spatial cache is updated.

3.1. The Selective Cache

A selective cache is a simplified version of the dual data cache. This cache consists of the same locality prediction table but it only has one memory unit like conventional caches. The locality prediction table behaves in the same way with the difference that now, there is not distinction between spatial and temporal caches. That is, everything that must be cached is brought to the same memory unit.

Notice that a selective cache behaves as a conventional cache with a selective caching policy. Only data with spatial locality or data with temporal locality that is guaranteed not to self-interfere is cached.

4. A Qualitative Analysis of the Performance

In this section, the rationale for the design of the dual data cache is explained. The benefits of the dual data cache are illustrated by analyzing its performance for several usual patterns of memory references:

- References to scalars.
- References to vectors with a constant stride.
- References to “random” elements of a vector.

The objective of this analysis is to justify the chosen design. A detailed performance analysis is presented in the next section.

4.1. Scalars

Scalars are cached in the default cache since the first time they are accessed, they miss in the locality prediction table and therefore the prediction is the one assumed by default. It is not clear whether it is convenient to exploit just temporal or spatial and temporal locality for scalars. If the same scalar is referenced repeatedly the new stride for this load/store will always be zero, and therefore, the corresponding entry in the locality prediction table will remain in the transient state in which the locality prediction is again the default value. As previously mentioned, the simulations showed that the most effective default value is temporal.

4.2. Vectors with Constant Stride

When a load/store instruction is used to access a sequence of elements separated by a constant stride, the first element will be cached in the default cache because it will miss in the locality prediction table. At that time we do not have enough information about the memory reference patterns generated by that instruction.

The reference to the second element will have a stride different from zero and thus, different from the previous stride which was initialized to zero. Then, the state of the corresponding entry shifts from initial to transient and therefore the locality prediction is again given by the default value. At that time it is not known whether the stride found between the first two accesses is going to be constant or this load/store instruction is accessing “random” elements.

The third and following accesses will find a constant stride. If the stride is “small” they will be cached in the spatial cache in order to exploit spatial locality and eventually temporal locality. If the stride is “large” they will not be cached anywhere. If the same vector is referenced again, the prediction will remain spatial if the stride is small, and it will depend on the vector length and the stride family otherwise. If it can be assured that the vector is not going to interfere with itself (i.e., the number of elements times 2^{family} is not higher than the number of temporal cache lines) the prediction will become temporal. If not, it will not be cached anywhere, since it does not exhibit spatial locality and temporal locality cannot be exploited due to self-interference.

Note that delaying the caching of a vector with poor spatial locality until the second time it is accessed has the additional benefit of avoiding the pollution of cache memory for vectors that are accessed only once (e.g., the matrix in a matrix by vector algorithm, if it is traversed with a large stride). On the other hand, it has the disadvantage that if the vector is used several times and it fits into the cache, every element will be brought from memory twice. The negative effect of this additional penalty decreases as the number of times the vector is reused increases.

4.3. Random Accesses

This third scenario refers to the case when successive executions of the same load/store instruction reference memory locations that are not at a constant distance. This happens for instant in the algorithm that multiplies a sparse matrix by a vector. The accesses to the vector elements do not follow any pattern.

Consecutive random accesses will usually have a variable stride and therefore, the corresponding entry will remain in the transient state where the prediction is given by the default value.

5. Performance Evaluation

The performance of the dual data cache and the selective cache has been evaluated for a set of benchmarks and compared with the performance of a conventional cache with similar hardware cost. The benchmarks were compiled and executed on a DEC workstation with an Alpha 21064 microprocessor. Those benchmarks written in Fortran were compiled with the DEC Fortran 77 compiler using the most aggressive optimizations (-O5). Those written in C were compiled with the DEC C compiler again using all the optimizations (-O5). A trace of the execution were obtained using the *atom* instrumentation tool. This trace was fed into a simulator of a conventional data cache memory, a simulator of a dual data cache and a simulator of a selective cache.

In addition to compute the miss ratio, the simulators also keep track of the delay caused by data memory references which allows us to compute the execution time of the simulated programs and the average memory access time.

5.1. Simulation Parameters

All the caches are assumed to be direct-mapped with a copy-back and write allocate policy. The cache memory is connected to the next level of the memory hierarchy by means of a 8 byte bus. The first word (8 bytes) of a request to the next memory level arrives after 10 cycles and then, successive words arrive at the rate of one per cycle.

In order to compare different configurations with similar hardware cost, a conventional cache of size S is compared with a dual data cache with a temporal cache of size S/4 and a spatial cache of size S/2. After taking into account the amount of memory required by the tags, the remaining amount of storage (less than S/4) is dedicated to the locality prediction table. In consequence, depending on the value of S and the line size, a different number of entries has been assumed for such table (between 128 and 512 for the results presented later on). The selective cache is assumed to have the same size and configuration as a conventional cache and in addition it has a very small locality prediction table (16 entries for a 16 Kbyte cache memory and 32 entries for a 32 Kbyte cache memory). Unlike the dual data cache, in this case the increase in hardware required by the locality prediction table is not compensated with a reduction in the cache capacity because it is much simpler than that of the dual data cache. This should be taken into account when comparing the performance results.

For the dual data cache, the line size of the temporal cache is 8 bytes and the line size of the spatial cache is either 16 or 32 bytes.

5.2. Benchmarks

The performance of the dual data cache has been evaluated for both vectorizable and non-vectorizable code. Three different types of benchmarks have been used: synthetic benchmarks, kernels and some benchmarks from the SPEC 92 suite. For the latter benchmarks, 5 million of memory references were simulated after discarding the first 10 million.

5.2.1. Synthetic benchmarks

The synthetic benchmarks consist of a set of loops that deal with vectors of different size and stride. Each loop makes use of either one or two vectors and traverse it/them ten times incrementing every element at each iteration. The objective is to consider all possible scenarios regarding temporal and spatial locality and interferences for a data cache of 32 Kbytes. Table 1 lists the characteristics of the benchmarks that use only one vector and table 2 shows the same information for the benchmarks that use two vectors.

The columns of the tables have the following meaning:

- *Length*: The number of elements of the vector that are referenced.
- *Stride*: The stride when accessing the vector.
- *S-Loc*: The amount of spatial locality that will be exploited by the conventional cache. A value of *x/y* means that *x* out of *y* loads will result in a hit due to spatial locality.
- *SS-Loc*: Idem for the spatial subcache of the dual data cache.
- *T-Loc*: The amount of temporal locality that will be exploited by the conventional cache. A value of *x* means that *x%* of the vector elements will remain in cache memory (not replaced by further references) once they have been accessed by the first time.
- *TS-Loc*: Idem for the spatial subcache of the dual data cache.
- *TT-Loc*: Idem for the temporal subcache of the dual data cache.

Bench.	Length	Stride	S-Loc	T-Loc	SS-Loc	TS-Loc	TT-Loc
v1	2000	1	3/4	100	3/4	100	-
v2	4000	1	3/4	100	3/4	2	-
v3	15000	1	3/4	0	3/4	0	-
v4	1000	5	0	100	-	-	100
v5	1000	15	0	13	-	-	100
v6	2500	6	0	0	-	-	0
v7	1000	12	0	100	-	-	0

Table 1: Synthetic benchmarks dealing with one vector

In table 2, every benchmark has two values for each entry. Each of these values correspond to one of the two vectors used by the benchmark.

Note that in these benchmarks each load is followed by a store to the same location. This store will always result in a cache hit in the conventional cache due to temporal locality. However, this is not the case of the dual data cache because some data may be not cached. In other words, this sequence of load-store will benefit most to the conventional cache.

5.2.2. Kernels

The kernels are a set of routines frequently found in numerical applications:

- *fft*: A Fast Fourier Transform of a 2^{19} element input vector. This is a vectorizable code with strides power of 2.
- *mm*: A matrix by matrix multiplication. A vectorizable code with unit and non-unit strides
- *mmb*: A block algorithm for matrix by matrix multiplication, optimized to make an optimal use of a 16 Kbyte conventional cache.
- *smv*: A sparse matrix by vector multiplication. A non-vectorizable code.
- *tri*: A triangular matrix multiplication.

Benchm.	Length	Stride	S-Loc	T-Loc	SS-Loc	TS-Loc	TT-Loc
v11	1000	1	3/4	100	3/4	100	-
	1000	1	3/4	100	3/4	100	-
v22	2000	1	3/4	50	3/4	2	-
	2000	1	3/4	50	3/4	2	-
v23	2000	1	3/4	14	3/4	100	-
	1000	5	0	52	-	-	100
v24	2000	1	3/4	0	3/4	0	-
	15000	1	3/4	0	3/4	0	-
v25	2000	1	3/4	0	3/4	100	-
	2200	7	0	0	-	-	0
v33	500	15	0	22	-	-	100
	500	15	0	22	-	-	100
v34	1000	5	0	0	-	-	100
	15000	1	3/4	0	3/4	0	-
v35	1000	5	0	0	-	-	100
	2200	7	0	0	-	-	0
v44	15000	1	3/4	0	3/4	0	-
	15000	1	3/4	0	3/4	0	-
v45	15000	1	3/4	0	3/4	0	-
	2200	7	0	0	-	-	0
v55	2200	7	0	0	-	-	0
	2200	7	0	0	-	-	0

Table 2: Synthetic benchmarks dealing with two vectors

5.2.3. SPEC 92 benchmarks

The following benchmarks from the SPEC 92 benchmark suite have also been used: *compress, eqntott* and *espresso* (from SPECint92) and *alvinn, hydro2d, mdljdp2, mdljsp2* and *swm256* (from SPECfp92). We will refer to them as *com, eqn, esp, alv, hyd, mdd, mds* and *swm* respectively.

5.3. Performance figures

5.3.1. Line Size of the Conventional Cache

Figure 4 shows the average memory access time in cycles of a conventional cache with line size of 32 and 64 bytes. Two different cache sizes have been considered: 16 and 32 Kbytes. From these graphs we can conclude that in general a 32 byte line size is more efficient. In consequence, in the following experiments, a dual data cache and a selective cache will be compared with a conventional cache with such line size.

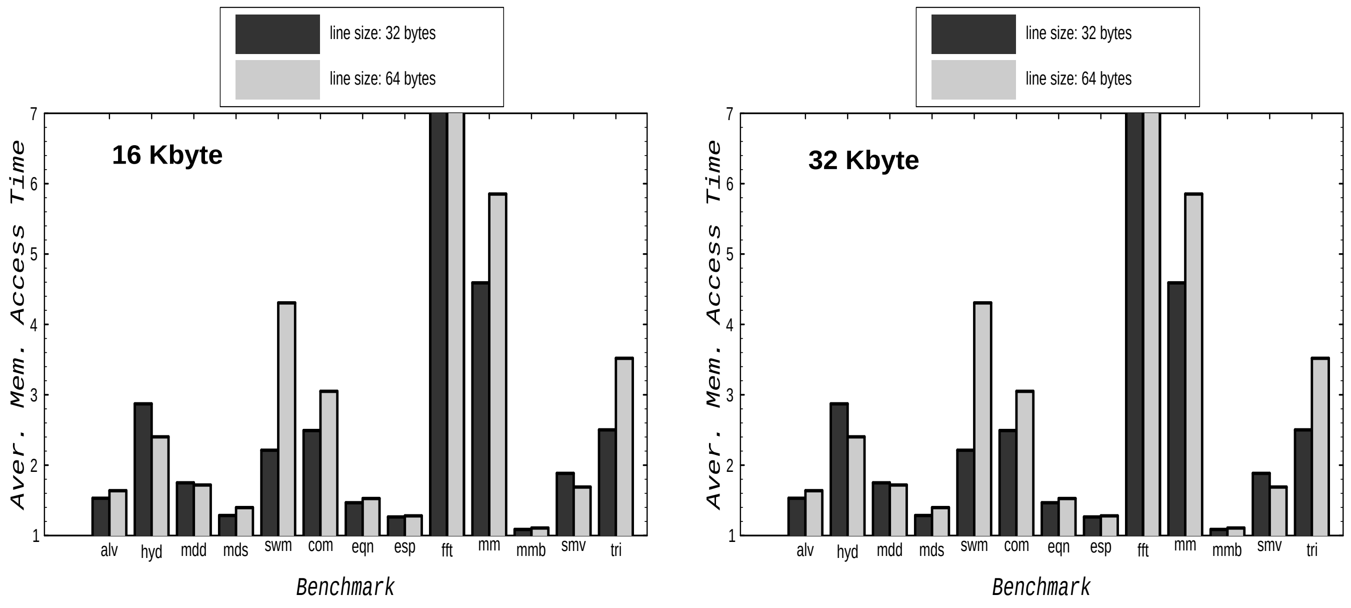


Figure 4: Performance of a 16/32 Kbyte conventional cache for different values of the line size. The average memory access time corresponding to *fft* is 17.5 for a line of 32 bytes and 23 cycles for a line of 64 bytes.

5.3.2. Synthetic Benchmarks

Figure 5 compares the average memory access time of a conventional cache, a selective cache and a dual data cache for the synthetic benchmarks. The dual data cache is assumed to have 1024 lines of 8 bytes in the temporal cache and 512 of 32 bytes in the spatial cache. The conventional and selective caches have 1024 lines of 32 bytes.

The performance figures show that the dual data cache significantly outperforms the conventional cache for almost all the benchmarks. For the benchmarks listed in tables 1 and 2, the dual data cache has a worse performance only in two cases (*v7* and *v2*). For *v7*, the vector is traversed with a stride 12 elements, that is 3 lines. Since 3 belongs to the family 0, the conventional cache does not have any self-interference as long as the number of referenced elements is not higher than the number of lines. In this case, the number of elements is 1000 and the number of lines is 1024, so the vector fits in the conventional cache. The dual data cache does not cache this vector in any of its two subcaches. This is because the stride is large which makes it a good candidate for the temporal subcache. However, the stride 24 belongs to the family 3 and the temporal cache has 1024 lines. This implies that if the vector has more than $1024/2^3=256$ elements, it will interfere with itself. Because this is the case, the dual data cache decides not to cache this vector.

In the case of *v2*, this is a vector that exhibits temporal and spatial locality so, it is placed in the spatial cache. However, it fits into the conventional cache but not into the spatial cache of the dual data cache since its size has been halved.

Notice that in many of the remaining cases, the dual data cache is much better than the conventional cache. One of the highest differences is for *v33*. In this case, the program traverses two vectors with stride 15. Although only 500 elements of each vector are used, which is quite less than the capacity of the cache, the conventional cache can only exploit a small percentage of the temporal locality due to the pollution introduced by a stride greater than one. In this case, every cache line has only one useful element. Spatial locality cannot be exploited neither because the stride is greater than the line size. On the other hand, the dual data cache can exploit temporal locality placing both vectors in the temporal cache. Since the line size of this cache is one element, there is no pollution. In addition, there are not interferences because the stride belongs to the family 0.

Finally, it can be observed that the selective cache has a performance similar to the dual data cache except in two cases: *v4* and *v33*. The reason is that in these two scenarios the selective cache decides not to cache the vectors since it assumes that they would cause self-interferences in the cache. However, the dual data cache can cache the vectors in the temporal cache.

5.3.3. Kernels and SPEC Benchmarks

Figures 6 and 7 depict the performance of the conventional cache, the selective cache and the dual data cache for the kernels and SPEC benchmarks.

We can observe that for those cases when the conventional cache is quite good (see *mmb*, *esp*, *mds*, *alv*), the dual data cache and the selective cache are also very effective, in most cases they are even better than the conventional cache. *bmm* deserves a special mention. This is a program that has been designed in order to exploit the locality of the algorithm as much as possible using a conventional cache of 16 Kbytes. This required some transformations of the original matrix multiply program (blocking and copying). Nowadays there is a huge effort in investigating how these transformations can be automated and applied by the compiler. However, the effectiveness of such transformations depends on the characteristics of the algorithm. It could be relatively straightforward for perfect loop nests with uniform dependencies, but rather limited in other cases. The selective cache behaves not very well for a 16 Kbyte cache. This is due to the pollution introduced by prefetching. This pollution does not degrade performance with a 32 Kbyte cache since the block size has been designed in order to fit into 16 Kbytes and therefore there is much space left in cache.

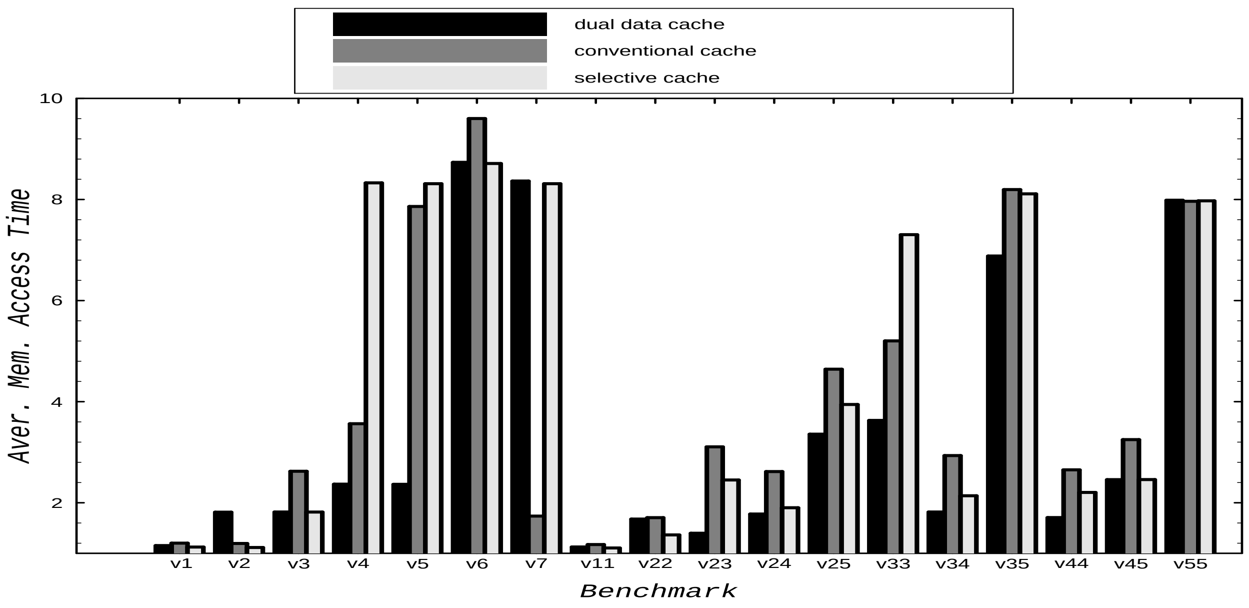


Figure 5: Synthetic benchmarks with a 32 Kbyte cache.

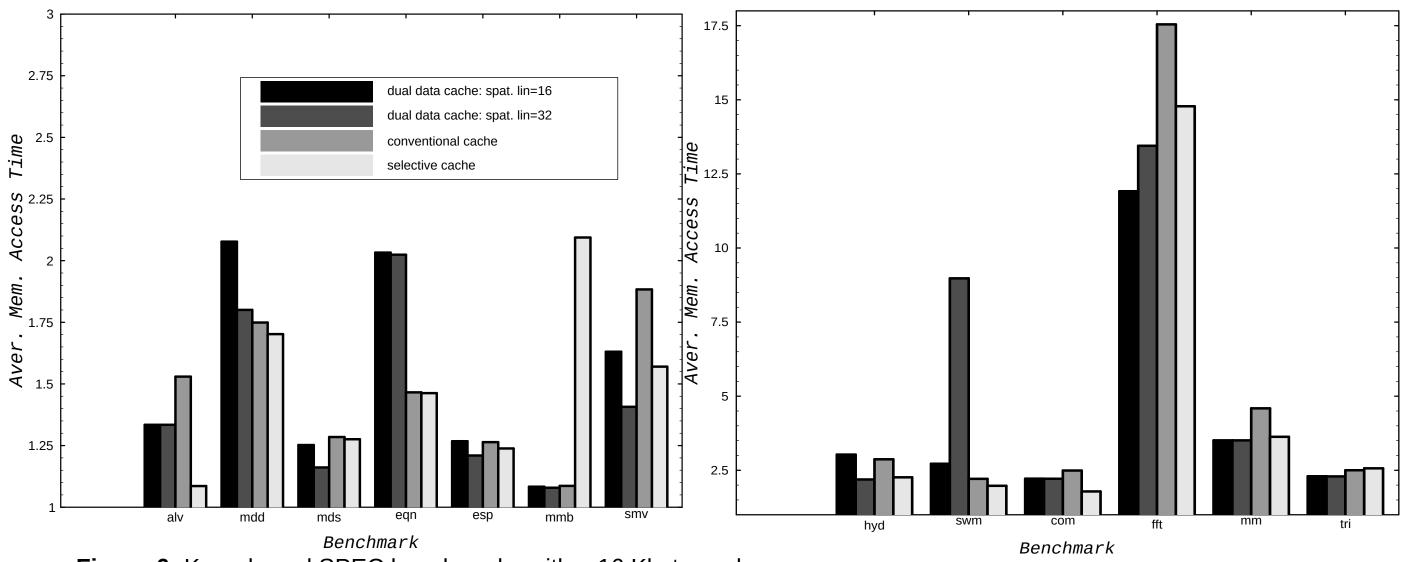


Figure 6: Kernels and SPEC benchmarks with a 16 Kbyte cache.

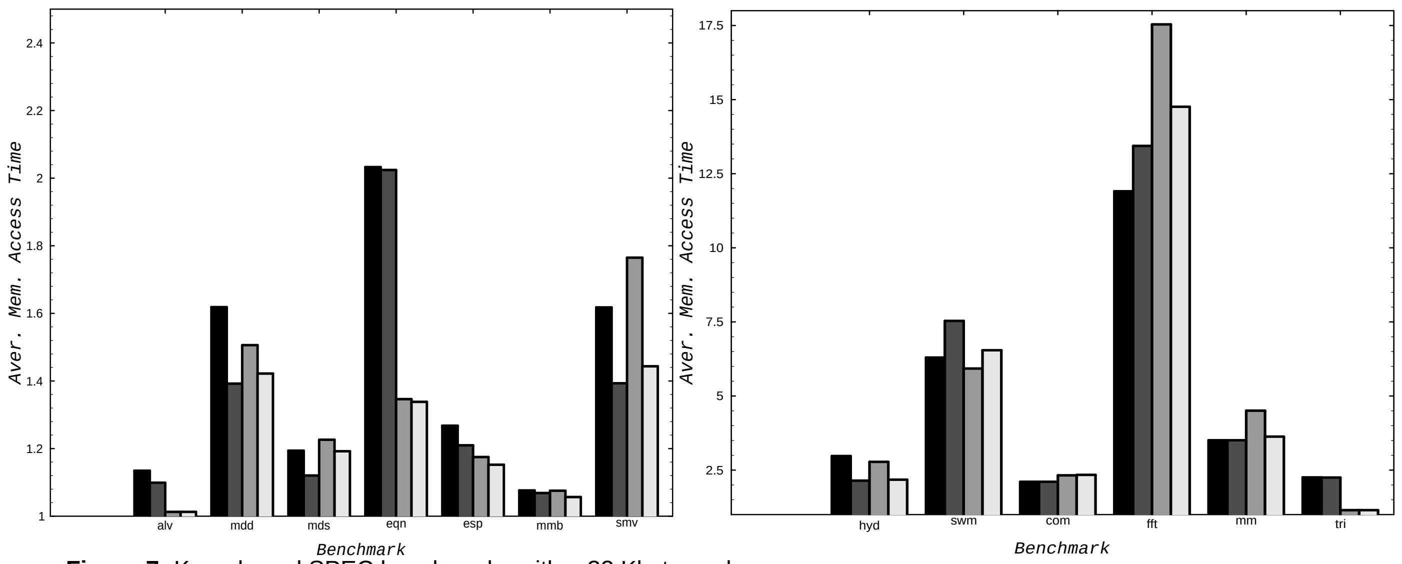


Figure 7: Kernels and SPEC benchmarks with a 32 Kbyte cache.

For the *FFT* benchmark, the dual data cache has much better performance than a conventional cache. This benchmark makes vector accesses with strides that are powers of 2. If the stride is equal to 2^s, a cache with capacity for 2^c elements will cause self-interferences for any vector with more than 2^{c-s} elements. In other words, a power-of-two stride reduces the effective capacity of the cache in a factor equal to the stride. Because many of the vectors are quite large, the conventional cache undergoes many interferences. On the other hand, the dual data cache do not cache those vectors that are going to interfere and then it benefits from the shorter miss penalty due to the shorter line size.

The dual data cache may have a worse performance than a conventional cache for some memory reference patterns. This happens when almost all memory references exhibit spatial locality. In this case, almost all the data is cached into the spatial cache. Since it has a lower number of lines than the conventional cache, more interferences will occur and then the performance will be degraded. This is the case of the *swm* benchmark.

We can also see in figures 6 and 7 that for the non-vectorizable benchmarks (*com* and *smv*), the dual data cache and the selective cache have a slightly better performance than a conventional cache.

In some other cases, the dual data cache is a little bit worse than the conventional cache (*eqntott* always; *alv, mdd* and *esp* for some configurations). However, it is remarkable the good performance of the selective cache, which is always better than the conventional cache with a very few exceptions.

6. CONCLUSIONS

A novel data cache design for superscalar processors has been presented. The proposed data cache, called dual data cache, has two independent parts: the spatial cache, which has been designed to exploit both spatial and temporal locality, with the emphasis in the former, and the temporal cache which exploits only temporal locality. In other words, the proposed scheme is a way of having large lines, which are convenient to exploit spatial locality, and small lines, which are more suitable if spatial locality is not present, in the same cache memory.

A simplified version of the dual data cache, called the selective cache has also been presented.

The performance figures obtained for a set of benchmarks show that the dual data cache outperforms in many cases a conventional cache. It is even more remarkable the good performance of the selective cache, which consists of an ordinary cache plus a locality prediction table with a very few entries.

We have also shown that for those cases where software techniques can be applied to improve the locality of a given algorithm (blocking and copying), the performance of the dual data cache is not degraded. For instance, in the case of matrix multiply with blocking and copying, the performance of the dual cache is about the same as that of a conventional cache.

Acknowledgments

This work has been supported by the Ministry of Education and Science of Spain (CICYT TIC-95/0429, TIC-92/880 and TIC-91/1036).

7. REFERENCES

[1] S.G. Abraham et al.
Predictability of Load/Store Instruction Latencies
in *Proc. of MICRO-26*, pp. 139-152, 1993

[2] A. Agarwal and S.D. Pudar
Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches
in *Proc of the 20th. Int. Symp. Comp. Architecture,* pp. 179-190, 1993.

[3] J-L. Baer and T-F. Chen
An Effective On-Chip Preloading Scheme to Reduce Data Access penalty,
in *Proc of Supercomputing'91 Conference,* pp. 176-186, 1991.

[4] S. Carr and K. Kennedy,
Compiler Blockability of Numerical Algorithms,
in *Proc of the Supercomputing'92 Conference,* pp. 114-124, 1992.

[5] A. Chang,
Application of Sparse Matrix Methods in Electric Power System Analysis,
in *Proc. of the Symp. on Sparse Matrices and their Applications,* pp. 113-122, 1969.

[6] T-F. Chen and J-L. Baer,
A Performance Study of Software and Hardware Data Prefetching Schemes,
in *Proc of the 21th. Int. Symp. Comp. Architecture,* pp. 223-232, 1994.

[7] J.W. Fu, J.H. Patel and B.L. Janssens,
Stride Directed Prefetching in Scalar Processors,
in *Proc of the 25th. Int. Symp. on Microarchitecture (MICRO-25)*, pp. 102-110, 1992.

[8] D.T. Harper III and D.A. Linebarger,
A Dynamic Storage Scheme for Conflict Free Vector Access,
in *Proc of the 14th. Int. Symp. Comp. Architecture,* pp. 72-77, 1987.

[9] Y. Jegou and O. Temam,
Speculative Prefetching
in *Proc. of the 1993 Int. Conf. on Supercomputing,* pp. 57-66, 1993.

[10] N.P. Jouppi,
Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers
in *Proc. of the 17th. Symp. Comp. Architecture,* pp. 364-373, 1990.

[11] G. Kurpanek et al.
PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface
in *Proc. of CompCon94*, pp. 375-382, 1994.

[12] M.S. Lam, E.E. Rothberg and M.E. Wolf,
The Cache Performance and Optimization of Blocked Algorithms
in *Proc. of ASPLOS 1991*, pp. 67-74, 1991.

[13] S. McFarling
Cache Replacement with Dynamic Exclusion
in *Proc of the 19th. Int. Symp. Comp. Architecture,* pp. 191-200, 1992.

[14] A. Seznec
A Case for Two-Way Skewed-Associative Caches
in *Proc of the 20th. Int. Symp. Comp. Architecture,* pp. 169-178, 1993

[15] O. Temam and N. Drach,
Software Assistance for Data Caches
in *Proc. of the 1st. Int. Symp. on High-Performance Computer Architecture*, pp. 154-163, 1995.

[16] O. Temam, E.D. Granston, W. Jalby,
To Copy or not to Copy: A Compile-time Technique for Assesing when Data Copying Should be Used to Eliminate Cache Conflicts,
in *Proc. of Supercomputing'93 Conference,* pp. 410-419, 1993.

[17] K.B. Theobald, H.J. Hum and G.R. Gao
A Design Framework for Hybrid-Access Caches
in *Proc. of the 1st. Int. Symp. on High-Performance Computer Architecture*, pp. 144-153, 1995.

[18] M. Valero et al.
Increasing the Number of Strides for Conflic-Free Vector Access,
in *Proc of the 19th. Int. Symp. Comp. Architecture,* pp. 372-381, 1992.

[19] M. Wolfe,
Iteration Space Tiling for Memory Hierarchies,
in *Proc. of the Third SIAM Conference on Parallel Processing for Scientific Computing*, Dec. 1987.

[20] Q. Yang and L.W. Yang,
A Novel Cache Design for Vector Processing
in *Proc of the 19th. Int. Symp. Comp. Architecture,* pp. 362-371, 1992.