Software Assistance for Data Caches *

O. Temam
PRiSM Laboratory
Versailles University
78 Versailles
France
Email: temam@prism.uvsq.fr

N. Drach
LRI
Orsay University
91 Orsay
France
Email:drach@lri.fr

Abstract

Hardware and software cache optimizations are active fields of research, that have yielded powerful but occasionally complex designs and algorithms. The purpose of this paper is to investigate the performance of combined though simple software and hardware optimizations. Because current caches provide little flexibility for exploiting temporal and spatial locality, two hardware modifications are proposed to support these two kinds of locality. Spatial locality is exploited by using large *virtual cache lines* which do not exhibit the performance flaws of large physical cache lines. Temporal locality is exploited by minimizing cache pollution with a *bypass* mechanism that still allows to exploit spatial locality. Subsequently, it is shown that simple software informations on the spatial/temporal locality of array references, as provided by current data locality optimization algorithms, can be used to increase cache performance significantly. The performance and design tradeoffs of the proposed mechanisms are discussed, Software-assisted caches are also shown to provide a very convenient support for further enhancement of data locality optimizations.

Keywords: software-assisted caches, data locality, numerical codes.

1 Introduction

This paper derives from several observations on application codes, cache designs and state-of-the-art compiler-optimizers. Let us first discuss the spatial and temporal locality properties of numerical codes. With respect to temporal reuse, figure 1a shows the reuse distance distribution of the traced memory references for the numerical benchmarks used in this paper (0 corresponds to data referenced only once). First, it appears that a sizable amount of data are used only once or very few times, so that techniques for hiding compulsory misses are required. It also appears that reuse distances are often larger than 1000 references, while for these same traces the average lifetime of a cache line in a 8-kbyte cache with a 32-byte cache line is approximately equal to 2500 references. So, for these codes the temporal reuse is likely to be disrupted by cache pollution. With respect to spatial reuse, figure 1b shows the *average vector length* of requests issued by load/store instructions.[1] This vector length proves to be often larger than the cache line size currently used in small on-chip caches (32 bytes). In other terms, there is

*This work was supported by the Esprit Agency DG XIII under Grant No. APPARC 6634 BRA III.

[1] A vector sequence terminates when the instruction has not been used during more than 500 references, (i.e., a value much smaller than the average lifetime of a cache line,) or when the stride is greater than 32 bytes (i.e., the corresponding spatial locality would not be exploited with a cache line size of 32 bytes).

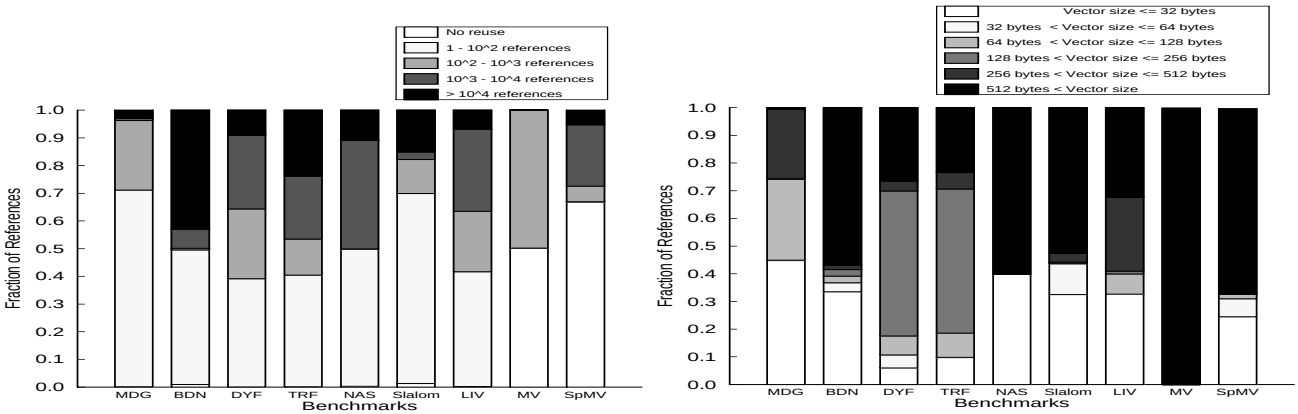

Figure 1a: *Distance of Reuse (in number of references). Distribution of References among these Reuse Distances.*

Figure 1b: *Vector Length (in bytes) of a Reference Stream Issued by Load/Store Instructions. Distribution of References among these Vector Lengths.*

Figure 1: *Temporal and Spatial Reuse in Numerical Codes.*

potential spatial locality to be exploited, and minimizing cache pollution could improve the exploitation of temporal reuse.

Second, current cache designs provide very little, if *none*, flexibility for exploiting spatial and temporal locality. With respect to spatial locality, the same cache line size is used for all codes. It is an optimal value obtained for a *typical* workload (i.e., *numerical* and *non-numerical codes*). With respect to temporal locality, a requested data is always loaded in cache, whether it is reused or not.

Third, compile-time exploitation of temporal and spatial locality is now a well-documented research topic (see [3, 8, 23, 30]). Though the existing algorithms are still unable to manage the cache as a local memory, they are now capable of determining, within a loop nest, whether a reference exhibits significant temporal and/or spatial locality that is worth being exploited.

Consequently, the first goal of this paper is to propose a simple cache design that uses simple software information for better exploiting temporal and spatial locality. The second goal is to show that simple compiler techniques are sufficient to determine whether array references exhibit spatial and/or temporal locality, and by using this information, that it is possible to significantly increase the cache performance of numerical codes. This study is mostly targeted towards small on-chip data caches. The reduced hardware complexity, low cost and low hit time of such caches make them attractive. However, they are particularly sensitive to cache pollution and cannot accommodate large cache lines.

This paper is organized as follows. In section 2, the hardware supports for spatial and temporal locality exploitation are described (sections 2.1 and 2.2), along with the compile-time techniques used to extract the locality information (section 2.3). In section 3, the simulations are described (section 3.1), a performance summary is provided (section 3.2), and further design tradeoffs are discussed. In section 4, it is shown that software-assisted caches provide a convenient support for software optimizations (blocking, copying) and prefetching.

2 A Cache Design for Exploiting Temporal and Spatial Locality

2.1 Hardware Support for Spatial Locality

The simplest way to exploit spatial locality is to use long cache lines. However, figure 8b shows that, for many codes large cache lines are not compatible with small cache sizes. Cache performance is a *balance* between temporal locality and spatial locality exploitation, characterized by the ratio

$\frac{Cache\ Size}{Line\ Size}$. Temporal locality exploitation favors a large ratio (numerous cache entries), while large lines decrease this ratio. Besides, large cache lines breed *additional memory traffic*, since a smaller fraction of words fetched is used because of increased cache conflicts.

The proposed solution is to use a small physical line size (of 32 bytes as for many current caches [7, 11, 17]), and to load a large **virtual line**, simply corresponding to several physical lines, when a reference exhibits spatial locality. Consequently, the cache architecture is hardly modified, except that on a miss request, more than one physical line may have to be placed in cache[2].

Virtual lines should not be confused with *sub-block placement* [14], where long physical lines are used and each physical line is sectored in several smaller sub-lines that can be loaded independently. The purpose of *subblock placement* is to reduce the directory size and memory traffic, but *the physical line size is large*. With *virtual lines*, the physical line size remains small and the directory size does not change. This technique is not innovative in terms of hardware, but it provides the necessary flexibility to support software-assisted spatial locality exploitation.

Several flaws of large physical lines are eliminated with virtual lines. There is no significant additional cache pollution because the purpose of software control is to ensure that most lines loaded are *effectively* used, and that they are used *within a short-time interval* (software instrumentation is described in section 2.3). For the same reason the memory traffic is not significantly increased.

The use of virtual lines raises several hardware issues.

Spatial tags One bit per load/store instruction is necessary to carry the spatial locality information. In RISC processors, address computations are performed by adding an offset to a register (usually the stack pointer [14]). Both the offset and the register are specified in the word of the load/store instruction. Reserving one or more bits of the load/store instruction decreases the maximum value of the offset, but since a 1-word register is used for the address computation, the addressable virtual space is not reduced.

Storing multiple lines In a traditional cache, the target line is selected while the miss request is sent to memory. If the victim line is dirty it is sent to the write buffer. Then, the incoming line is stored in the selected line.

For a virtual line, several physical lines are stored in the cache on a miss request. Therefore, a tag check is normally required, in order to know where an incoming physical line should be placed. However, each tag check would increase the miss penalty by one cycle. To avoid this additional penalty, *the cache locations of the requested physical lines are stored in a buffer*. Assuming the buffer is FIFO and that memory requests are sent back in-order, unstacking the last entry of the buffer provides the cache location of the incoming physical line. Consequently, *the physical lines can be stored in cache at the same pace they arrive from memory*. So, the miss penalty for loading a virtual line corresponding to n physical lines of size L_S, assuming a memory latency of t_{lat} cycles and a bus bandwidth of w_b bytes per cycle is equal to $t_{lat} + \frac{n \times L_S}{w_b}$, i.e., the same as for loading a physical line of size $n \times L_S$, though larger than for a physical line of size L_S. However, the increased miss penalty of virtual lines proved to be a worthwhile tradeoff (see section 3.2).

Note that in the current design, the execution is not resumed until the last physical line has arrived. It might be possible to unstall the processor when the requested line comes back from memory, but the cache would need to remain stalled for reloading the next physical lines of the virtual line. Since superscalar processors can issue up to one request per cycle to the cache, this technique would bring

[2]Note that the words loaded with a virtual line do not necessarily correspond to n physical lines that are *consecutive* to the one requested, but to the words loaded with a physical line of the same size. This is important for page faulting and address translation issues.

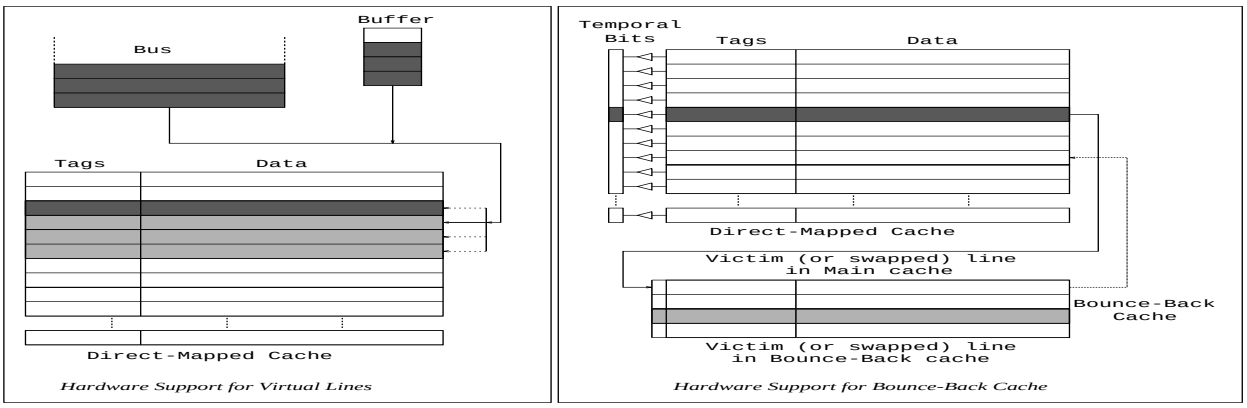

Figure 2: *Virtual Lines and Bounce-Back Cache.*

little improvement for future processors.

The victim lines are transferred to the write buffer if they are dirty, while the missed lines are fetched from memory. However, if the latency is not long enough, the virtual line is large and many target lines are dirty, all the transfer operations may not be hidden and may increase the miss penalty[3].

Coherence Virtual lines also raise coherence issues: while a virtual line may have to be loaded on a miss request, only the presence in cache of the physical line where the miss occurred is checked.

The following solution can be used. The first memory request corresponds to the line where the miss occurs. If a 1-cycle cache access time is assumed, the presence of each subsequent line of the virtual line can be checked in one cycle. Note that address translation need only be performed for the first physical line, because all physical lines of a virtual line belong to the same page by construction. Therefore, while the request for the first line is sent, the next line is checked, then a request is sent, and so on...After each unsuccessful tag check, the cache entry for the corresponding line is stored in the buffer mentioned above. This scheme allows to hide the tag checks of the additional physical lines, so that the miss penalty is not increased. Besides, only the necessary physical lines are fetched from memory. Consequently, additional memory traffic and invalidations in shared-memory multiprocessors are minimized.

Virtual line size Virtual line sizes of 64 bytes to 256 bytes were simulated. Very large virtual line sizes may raise hardware hazards like excessively-long miss penalties and bus occupation (the bus bandwidth is equal to 16 bytes in our case, so that loading a 256-byte virtual line requires 14 more cycles than a 32-byte physical line). A 64-byte physical line proved to be an optimal choice for an 8-kbyte cache.

2.2 Hardware Support for Temporal Locality

The two major cache phenomena that disrupt temporal locality exploitation are *cache interferences* and *cache pollution*. A cache interference is a *mapping* conflict, while cache pollution corresponds more to a *capacity* problem, i.e., cache space is unsufficient to store the reusable as well as the non-reusable data. Distinct solutions have been proposed for both these problems.

Cache Pollution Cache bypassing is the most natural solution for avoiding cache pollution. However, bypassing has a major flaw: spatial locality cannot be exploited for non-reusable data. As a

[3]Transferring a dirty line to the write buffer takes 2 cycles in the current design.

consequence, the performance of cache bypassing is usually poor (see figure 3a). In [5], bypassing has been demonstrated to be efficient, but the cache size used was very small (128 words), so that optimal cache lines were equal to 1 or 2 words. Therefore, spatial locality could not be exploited, and bypassing appeared to perform well. Only vector processors can make an efficient use of bypassing by loading long streams of data without temporal locality in vector registers, and not copying these streams to cache. The Intel i860 [16] implements a form of support for such operations (though without vector registers): data is fetched from memory, used by the processor and only then written to cache. By selectively allowing data to be stored back in cache, a form of bypassing is implemented. Since the memory fetch operation is pipelined, the spatial locality can be exploited in a vector-like manner. However, this mechanism is difficult to exploit because interleaved memory accesses are usually found in numerical loop nests.

Besides, cache bypassing can be harmful. Indeed, compilers may fail to predict and exploit temporal reuse for complex dependences, such as non-uniformly generated dependences (consider references `A(I,J)` and `A(I,K)` for example). By allowing non-temporal data to reside in cache, such locality is naturally exploited.

Cache Interferences Victim caches [19] have been proposed as a solution for cache interferences in direct-mapped caches. However, because of the limited size of such caches, they are not efficient at removing cache pollution, even though they are efficient at reducing cache interferences (see figure 3b). For example, consider the matrix-vector multiply loop below

```
DO j1 = 0,N-1
  reg = Y(j1)
  DO j2 = 0,N-1
    reg += A(j2,j1) * X(j2)
  ENDDO
  Y(j1) = reg
ENDDO
```

If N is large with respect to the cache size C_S, but $N < C_S$, i.e., no capacity miss, then on each iteration of j_1, between $2N - C_S$ and N elements of X are flushed from cache, and reused N iterations later. Consequently, only a large victim cache would be efficient in this case.

A second flaw of victim caches is their high access time, especially with respect to a direct-mapped cache (at least 2-3 cycles). The first reason is that victim caches must be associative. The second reason is that the access to main cache and victim cache cannot occur simultaneously, otherwise a multiplexer needs to be inserted on the critical path, which possibly increases the cache hit time. However, requests to both caches can still be issued at the same time, so that the victim cache request is available after 2 cycles, and can be used at the 3rd cycle or possibly even at the 2nd cycle.

One of the goals of this paper is to design a mechanism that is efficient at removing both cache interferences and cache pollution. The victim cache design can actually be expanded to provide a very convenient hardware support for software-assisted reduction of cache pollution. The main modification is to *bounce back* to main cache data that is about to be discarded from the victim cache, only if it exhibits temporal locality. The victim cache then behaves as a **bounce-back** *buffer* or *cache*. The replacement policy of this bounce-back cache is LRU, as for victim caches.

Consider the behavior of a bounce-back cache for the matrix-vector multiply loop. Array X exhibits temporal locality while array A does not. Elements of X or A that are victim of replacement are stored in the bounce-back cache. When the bounce-back cache is full, only the elements of X start to bounce back to main cache instead of being discarded. When bounced back to main cache, these elements of X mostly flush elements of A (since there are no self-interference or capacity misses) which are probably not used anymore.

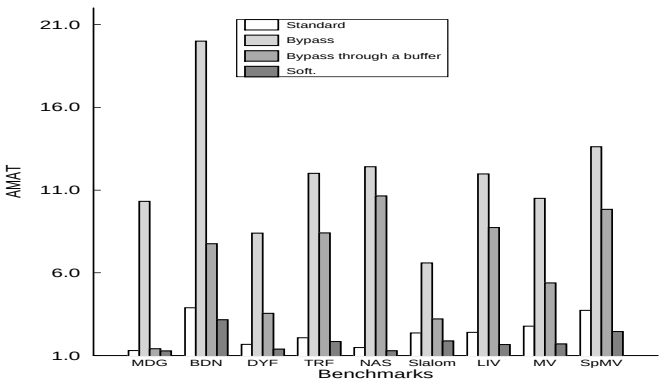

Figure 3a: *Efficiency of Bypassing.*

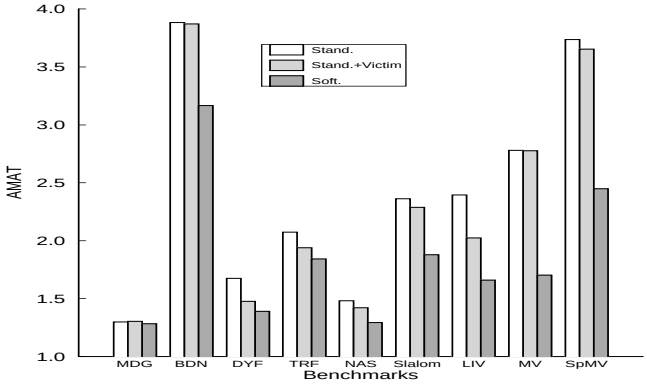

Figure 3b: *Efficiency of Victim Caches.*

Figure 3: *Current Techniques for Reducing Cache Pollution and Cache Interferences.*

Note that the bounce-back technique overcomes a major flaw of LRU replacement strategies as used in set-associative or victim caches. LRU is not well suited for numerical codes, where reuse often occurs in a cyclic way: an element that has not been used for a long time is likely to be reused soon (it is the case for the X vector for instance), i.e., reuse distances are often large. In contrast, with the bounce-back mechanism, a line with temporal locality is sent back to main cache (where it can be accessed faster) instead of being discarded, when it has the lowest LRU priority in the bounce-back cache.

Hiding the Bounce-Back process A data is about to be replaced in the bounce-back cache if a data needs to be transferred from the main cache to the bounce-back cache. This only occurs during miss requests. Otherwise, a hit in the bounce-back cache only results into a swap between the main cache and the bounce-back cache. So, the bounce-back mechanism can take place while awaiting a memory request, and therefore does not increase the stall time as far as the miss penalty is longer than the transfer time.

Dynamic adjustment of the Bounce-Back mechanism *Once a data has been bounced back to main cache, its temporal bit is reset.* It will be set again only if it is referenced by a load/store instruction which temporal tag is set.[4] This is necessary because data are active only during a certain time. Otherwise, the cache could be polluted by "dead" reusable data.

The main cache line victim of the bounce-back process is discarded if it is clean, or it sent to the write buffer otherwise.

If the data bounced back from the bounce-back cache is placed in a target line of the miss request (this situation is unlikely to occur), it will be discarded when the requested line is stored in cache. Therefore, no ping-pong phenomenon can take place. If the bounced-back data is dirty and mapped to the miss target line, it is sent to the write buffer and the bounce-back operation is canceled.

If the data bounced back to cache should erase a dirty line, the transfer is aborted if the write buffer is full.[5]

Temporal tags Each cache line in the main cache and the bounce-back cache has a temporal bit. Besides, each load/store instruction has a temporal tag (one bit) which is used to set the cache line

[4]Note that if the temporal bit is set on a line, and this line is referenced by a load/store with an unset temporal tag, the temporal bit of the line is unchanged.

[5]Also, if the memory latency is relatively low it is better not to bounce back to dirty lines, in order to simplify the process. This is not the solution implemented in the simulator.

temporal bit on both miss and hit requests.

Special support is required for temporal bits if the cache must be able to sustain a high issue rate (as for superscalar processors). Since the temporal bit is modified not only on a miss request, but on any load/store reference, a *write* to the temporal bit must actually be performed on each reference. Write requests typically require two cycles instead of one (one cycle to read the tag and data, and another cycle to get the hit/miss answer [20]). However in this case, *reading* the temporal bit every cycle is not required, since this bit is only useful in the bounce-back cache. Therefore, a solution is to use a separate array for the temporal bits that is accessed one cycle after the corresponding cache line has been accessed. At that time, the hit/miss answer is available, and the temporal bit is changed only in case of a hit. Thanks to this "pipelined" access, it is possible to write every cycle in the temporal bit array.

Using the Bounce-Back Cache as a Victim Cache The on-chip space corresponding to the bounce-back cache is still efficiently used when software control is not active, since the bounce-back cache is then used as a victim cache. The efficiency of victim caches for a large variety of codes has been demonstrated in [19].

Intuitively, only transferring the elements with temporal locality to the bounce-back cache is a more natural idea than transferring any element victim of replacement. However, it was observed that global performance was higher if the bounce-back cache is also used as a victim cache for non-temporal data (probably because of spatial interferences).

Besides, when no load/store is tagged "temporal", the bounce-back cache would be useless and the corresponding on-chip space would be wasted. It could then be used as a victim cache for all references only if a flag is explicitly set each time software control is not active. However, this solution was dropped because of the increased compiler overhead and hardware complexity.

Characteristics of the Bounce-Back cache In several current direct-mapped caches [27, 11], the data is read in 1 cycle, but the hit/miss answer is known in the 2nd cycle. Consequently, the minimum bounce-back cache access time is 2 cycles. In [19], a 2-cycle access time to the victim cache is used. However, because handling a cache miss may require some overhead that can cost at least an additional cycle, a conservative value of 3 cycles has been selected.[6] For a swap operation, the data is also available to the processor after 3 cycles, but both main cache and bounce-back cache are further locked for 2 cycles.

The bounce-back cache should be as large as possible to be efficient as a victim cache. On the other hand, *the smaller the bounce-back cache, the smaller the bounce-back delay* and, therefore, the higher the probability a data victim of pollution is accessed in the main cache (1 cycle) rather than in the bounce-back cache (3 cycles). Because of this tradeoff, small bounce-back caches perform nearly as well as large ones. A bounce-back cache size of 256 bytes (8 cache lines) proved to perform well. The bounce-back cache was considered fully-associative, though experiments showed that a 4-way bounce-back cache would perform reasonably well.

Coherence As for the main cache, virtual lines raise coherence issues for the bounce-back cache (one of the physical lines of the requested virtual line to be loaded can be present in the bounce-back cache). The coherence mechanism of section 2.1 must be modified to accommodate the bounce-back cache. The long access time of this cache does not allow a 1-cycle check. Therefore, the bounce-back cache is checked after the memory requests have been sent. If one of the physical lines is found in the bounce-back cache, the main cache line where it was supposed to be stored is simply tagged *invalid*. However, the memory

[6] Note that this choice decreases the global performance of the mechanism especially when there are numerous hits in the bounce-back cache.

fetch cannot be aborted. Because the bounce-back cache size is small, few such invalidations actually occur.

Combining Virtual Lines and the Bounce-Back cache In the IBM RS6000 [15], where the cache line size is equal to 128 bytes for a 16-byte bus bandwidth, the long miss penalty flaw has been eliminated with a buffer of 128 bytes where the incoming line is stored. Since the requested word is sent first, the processor can resume execution when this first word arrives.

So, it was attempted to use the bounce-back cache as such a buffer. However, this proved to be a poor solution. As already mentioned, it is not possible to achieve the same access time to the bounce-back cache and to the main cache because the main cache is assumed here to be direct-mapped. While for the IBM RS6000, the main cache is 4-way associative and located off-chip, so that checking the buffer can be done in the same time as checking the main cache, without influencing much the hit time.

2.3 Software Informations

Determining whether a reference exhibits temporal or spatial locality, and whether this locality is worth being exploited is a well-documented research topic [3, 8, 23, 30]. However, some of these algorithms are quite complex, and therefore cannot be used in this study because they would lessen our point that simple compiler techniques are sufficient to yield significant performance improvements in combination with the above mentioned hardware supports. Therefore, only elementary techniques derived from the methods presented in [30] (for temporal locality analysis) and [23] (for spatial locality analysis) have been used.

A reference is tagged *spatial* simply if the coefficient of the innermost loop in the reference subscript is smaller than 4 (the cache line size usually considered, i.e., 32 bytes, corresponds to 4 double-precision floating-point data). If the coefficient is a parameter, the reference is not tagged *spatial*.

A reference is tagged *temporal* if it exhibits a *temporal self-dependence* (see `X(J)` in figure 5) or a *uniformly generated temporal group-dependence* (see `B(J,I)`,`B(J,I+1)` in the same example). Finding such dependences amounts to simple subscript analysis. These two simple types of dependences account for a significant fraction of the dependences usually found in numerical codes, as already mentioned in [30]. This is confirmed by the fair amount of trace entries where the temporal bit was set (see figure 4a).

Whenever the loop body contained a `CALL` statement, the temporal/spatial bits of all references were set to 0 (no interprocedural analysis of the locality was performed).

3 Experiments

3.1 Simulations

Traces One of the main difficulties of the experiments was to collect traces. Several solutions were contemplated. The first one would be to collect a standard trace (using the `Spa` [10] tool for instance). Temporal and spatial locality could be detected through analysis of the trace. However, this method would not prove that current compilers are capable of extracting locality properties of references. In another variant, the source code references could be analyzed with a compiler, and a match between source code references and traced references could be made. This task proved to be far too complex, mainly because of assembly code optimizations (generation of several load instructions for a single reference...).

Therefore, it was decided to use *source code tracing*, where for each reference in the source code a call to a tracing subroutine is added in the code (see figure 5). Inserting software information with this technique is simple. On the other hand, source code tracing raises difficult issues: instrumenting all references, and retrieving the time information (i.e., the number of cycles between two references).

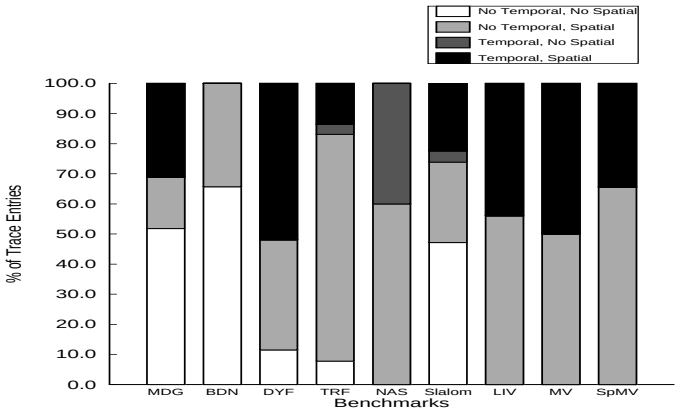

Figure 4a: *Fraction of References with Temporal and/or Spatial Tags.*

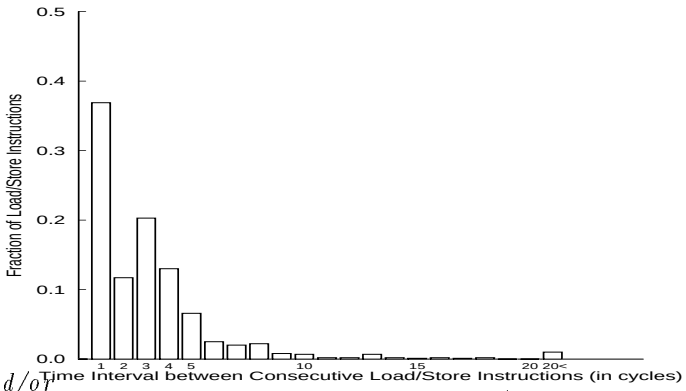

Figure 4b: *Time Distribution of Load/Store Instructions.*

Figure 4: *Software Instrumentation.*

```
Comment: call trace(reference,read/write(1/2),no_temporal/temporal(0/1),no_spatial/spatial(0/1))
   DO I=1,N
     DO J=1,N
         call trace(A(I,J),1,0,0)
         call trace(B(J,I),1,1,0)
         call trace(B(J,I+1),1,1,1)
         call trace(X(J),1,1,1)
         call trace(Y(I),1,1,1)
         call trace(Y(I),2,1,1)
         Y(I) = Y(I) + (A(I,J)+B(J,I)+B(J,I+1))*(X(J)+X(J))
     ENDDO
  ENDDO
```

Figure 5: *Example of Instrumented Loop.*

First, because it is not possible to instrument large benchmarks by hand, the `Sage++` compiler,[7] developed at University of Indiana [12] was used. `Sage++` is a compiler which has been designed for easy analysis and transformations of Fortran, C and C++ codes. The toolbox of subroutines provided a very convenient support for performing source code instrumentation. Thanks to this tool, *all array references* in each code were instrumented.

The second issue of source code tracing is the *time* information. A cache design is sensitive to the processor request issue rate, but the number of cycles between two references cannot be computed at the source code level. For that purpose, all benchmarks were traced with `Spa`. Using these traces, the *average distribution of the time distance between two load/store instructions was evaluated.* This distribution is shown in figure 4b. A pessimistic approach was adopted since each instruction was considered to execute in one cycle (arithmetic, logic, memory or floating-point). During trace extraction, a time distance is randomly generated for each new trace entry, according to that distribution.[8] Thanks to this method,

[7] Previously named `Sigma II`.

[8] Note that this number is generated during program execution, upon storage of a trace entry, not during compiler instrumentation. Because of the loop structures, the latter would have corrupted the time distribution. Note also that the time is recorded as part of the trace entry data, not during the simulation, so that repetitive simulations performed with

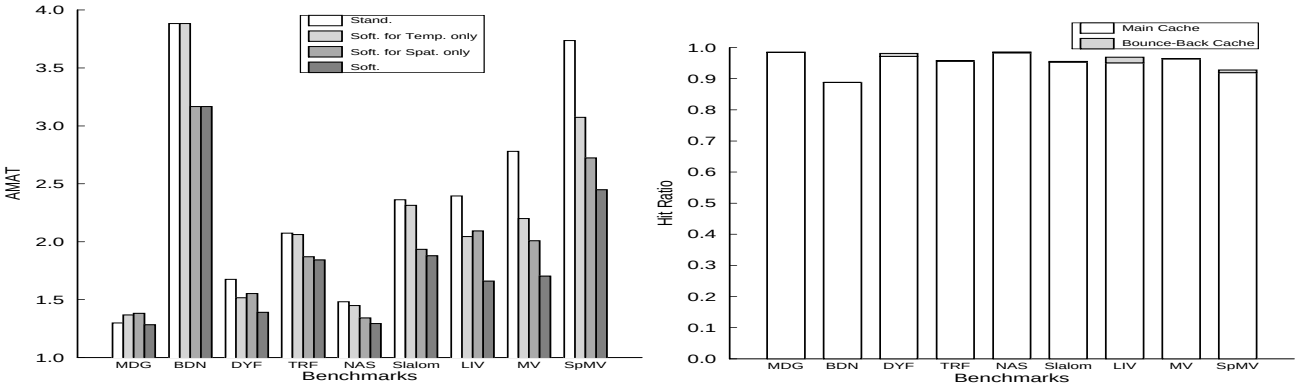

Figure 6a: *Performance of Software Control (AMAT).* Figure 6b: *Repartition of Cache Hits.*

Figure 6: *Performance of Software-Assisted Caches (I).*

a realistic issue rate can be reproduced. Only the global execution time information is incorrect. As a consequence, the AMAT metric (Average Memory Access Time) instead of the CPI metric (Cycles Per Instruction) was used.

Benchmarks The benchmarks used are all numerical codes: real-application benchmarks `ADM, MDG, BDN, DYF, ARC, FLO, TRF` from the Perfect Club Suite [6], the Livermore Loops benchmark `LIV`, the `NAS` and `Slalom` benchmarks, and two numerical primitives Matrix-Vector multiply `MV` and Sparse Matrix-Vector multiply `SpMV`.

Notations and Parameters A baseline cache configuration called *Standard or Stand.* has been used on many graphs. It corresponds to the data cache parameters of the DEC Alpha [27], the MIPS R4000 [11] and the Intel Pentium [17]: Cache Size = 8 kbytes, Line Size = 32 bytes, 1-way. Also, when software control for both temporal and spatial locality is used (i.e., the full mechanism: 256-byte bounce-back cache, 64-byte virtual line, 8-kbyte main cache, 32-byte physical line), the corresponding graph is termed *Soft.*. A memory latency of 20 cycles has been selected because the cost of implementing a new cache design is only profitable when memory access time is a major performance bottleneck. The bus bandwidth used in this paper is 16 bytes (a value found in several current processors like the IBM RS6000 [15]).

3.2 Performance Summary

Figure 7b shows that software-assisted caches exhibit a significantly lower miss ratio than standard caches (up to 62% reduction for `MV`). The reduction in AMAT is nearly the same because most cache hits are *main* cache hits, as shown in figure 6b, thanks to the bounce-back mechanism. Also, note that software-assisted data caches perform better than standard caches in any case, so software-assisted appear to be *safe*.

Though the bounce-back mechanism alone is profitable to several codes (see `DYF, LIV, MV, SpMV` in figure 6a), the virtual line mechanism alone is more efficient (see `BDN, TRF, NAS, Slalom, MV, SpMV`). However, the best performance is always obtained when both mechanisms are combined. The bounce-back mechanism seems to be efficient at hiding the increased cache pollution incurred by virtual lines. Figure 7a shows that virtual lines increase memory traffic, but when both mechanisms are used, memory traffic is barely increased (except for `TRF`).

the same trace are completely identical.

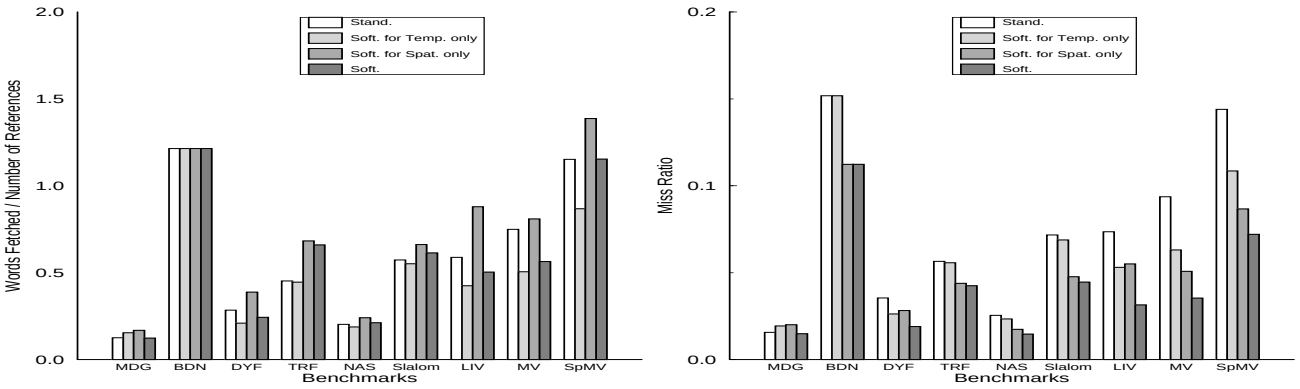

Figure 7a: *Performance of Software Control (Memory Traffic).*

Figure 7b: *Performance of Software Control (Miss Ratio).*

Figure 7: *Performance of Software-Assisted Caches (II).*

The performance increase of Perfect Club codes is smaller than the performance increase of other codes. The first reason is that these codes are known to have small working sets,[9] which imply small vector lengths and little cache pollution, i.e., little potential improvements with software-assisted data caches. This is confirmed by the relatively high percentage of references with no software tags (see figure 4a, `MDG, BDN`): since loop boundaries are small, references located outside loops account for a larger than usual percentage of total references. Also, because many loop bodies within these codes contain a subroutine call, locality analysis could not be performed for a number of loops, so that the spatial and temporal tags of many references were not set. Also, many of the array subscripts are aliases of loop subscripts (this is typical for dusty deck codes). Since subscript expansion was not performed, the locality could not be exploited in these loops. Finally, many loops were also badly ordered, inducing non stride-one references, and preventing the use of virtual lines. However, for references like `A(I,J)`, with `DO J=...` the innermost loop, spatial locality is often disrupted because of cache pollution: the spatial reuse distance is large, and the line can be flushed from cache before it can be reused. Tagging such references `no_spatial` and `temporal` helped exploiting the spatial reuse by limiting the impact of cache pollution.

In order to get a better idea of the software-assisted cache performance if these compiler limitations could be overcome, the most time-consuming subroutines[10] of several Perfect Club codes were manually instrumented and traced alone. As shown in figure 10a, if most references can be instrumented and computational loops account for a larger share of total references, significant further performance improvements could be obtained.

As can be seen in figure 4a, the temporal bit is set in fewer than 30% of the Perfect Club trace entries (except for `DYF` which exhibits the highest relative improvement due to the bounce-back mechanism). Apparently, the temporal reuse detected with our simple techniques accounts for a small share of the total reuse. Within numerical loops, it is actually common to find arrays which are only scanned and not reused inside the loop itself. This is confirmed by the high percentage of trace entries where the spatial bit is set, i.e., 50% and more for three Perfect Club codes. Because spatial locality is heavily exploited, a major share of cache misses removed are compulsory and capacity misses corresponding to

[9] The Perfect Club codes are provided with small test examples in order to use them as benchmarks (large working sets induce excessively long execution time).

[10] These subroutines were obtained by profiling the codes.

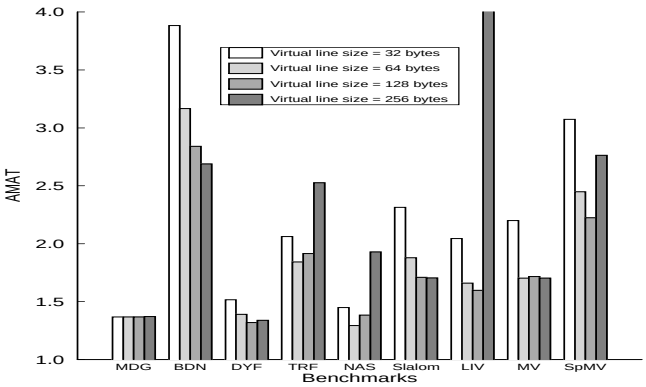

Figure 8a: *Influence of Virtual Line Size.*

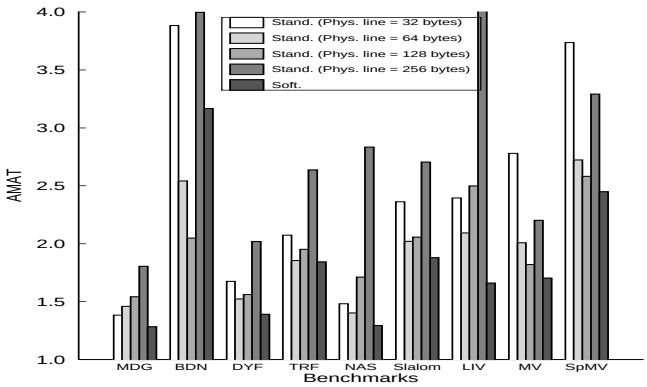

Figure 8b: *Influence of Physical Line Size.*

Figure 8: *Influence of Line Size.*

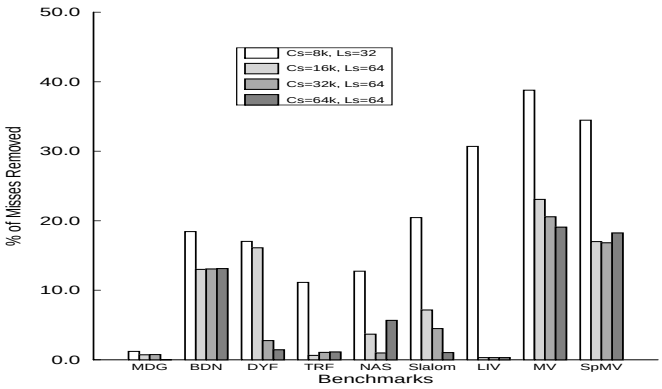

Figure 9a: *Software Control for Large Caches.*

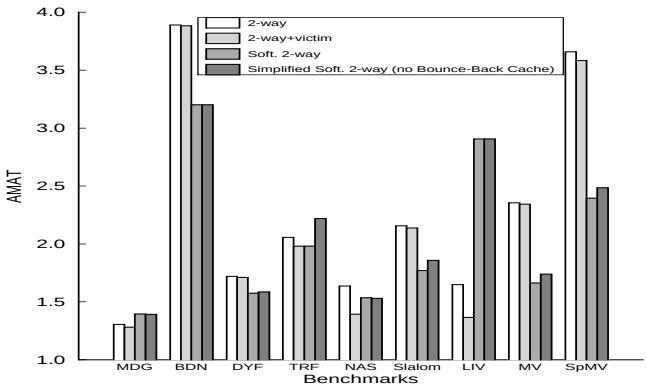

Figure 9b: *Software Control for Set-Associative Caches.*

Figure 9: *Influence of Cache Size and Associativity.*

vector accesses.

Cache Line Size Figure 8b shows that a virtual line of 64 bytes performs usually better than a physical line of 64 bytes and more (except for `BDN`). Besides, figures 8a and 8b show that large virtual lines are much better tolerated than large physical lines. Increasing the virtual line from 64 bytes to 128 bytes is even profitable for several codes (`BDN, DYF, Slalom, SpMV`). Therefore, it is tempting to enhance the design by allowing virtual lines of different lengths. The limitations are the additional bits required for each load/store instruction, and the complexity of the compiler algorithm for determining the amount of spatial locality. Furthermore the physical line size could be reduced (to 16 bytes for instance) in order to improve the performance of codes which favor a large $\frac{\text{Cache Size}}{\text{Line Size}}$ ratio. The performance of software-assisted caches with 16-byte and 32-byte physical lines proved to be similar, so decreasing the physical line size can be tolerated. Reducing the physical line size also has the advantage of slightly decreasing the cache access time by reducing the size of the multiplexor between the cache and the processor (see [26]). Nevertheless, this is in contrast with the current trend of reducing the tag space by using subblock placement, as already implemented in the unified cache of the PowerPC [24] and the instruction cache of the TI Supersparc [29]: a 64-byte physical line with 32-byte subblocks.

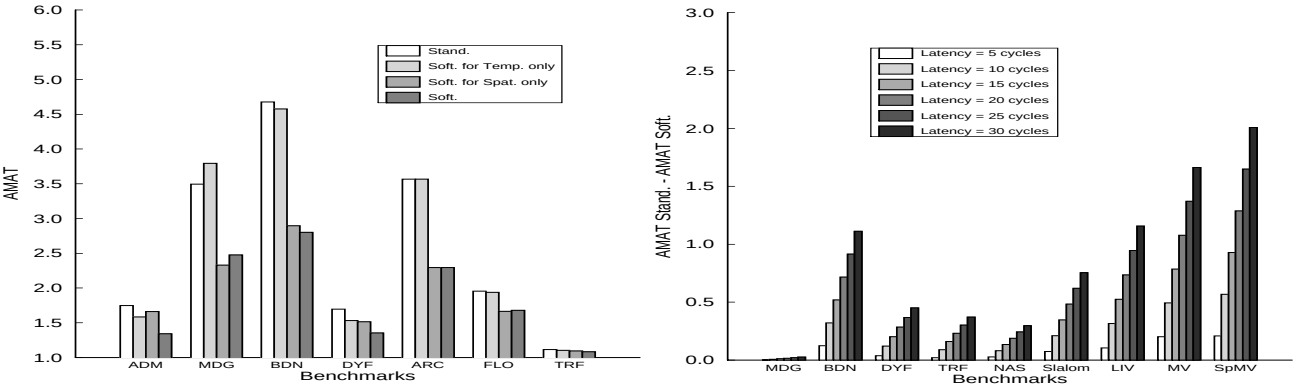

Figure 10a: *Performance of Software Control for the Most Time-Consuming Perfect Club Subroutines.*

Figure 10b: *Influence of Memory Latency.*

Figure 10: *Influence of Software Instrumentation and Memory Latency.*

Set-Associativity Implementing software-assistance for set-associative caches also brings performance improvements for most codes as shown in figure 9b. However, when software-assistance is not active, victim caching and set-associativity are merely redundant (see figure 9b). Besides, as seen in section 3.2, many improvements are due to virtual lines only. So, a more suitable extension of software-assistance to set-associative caches is to *use the temporal information to command replacement.* An LRU policy is still used, but non-temporal data are preferably replaced. This improved software control mechanism is much cheaper since no bounce-back cache is used, but it performs nearly as well (see figure 9b).

Note that this exploitation of temporal locality is close to the techniques proposed in [5] and [13], where data with temporal reuse are tagged with higher priority. On a set-associative cache, this constitutes an efficient implementation of bypassing.

Cache Size As mentioned at the beginning of this paper, the initial purpose of these mechanisms is to assist small direct-mapped caches which cannot use large cache lines and which are sensitive to pollution. However, figure 9a shows that large caches can also benefit from these mechanisms, although the performance improvements are less important. Large caches can accommodate larger cache lines, so a physical line size of 64 bytes was used for the 16,32,64-kbyte cache. Note that the theoretical optimal improvement brought by the virtual line mechanism is then halved.

When the working set fits in the cache (which is obviously the case of `LIV` for a 16-kbyte or larger cache), the mechanism is almost useless. In other cases, the reduction of capacity and compulsory misses corresponding to vector accesses is significant (see `BDN, DYF, NAS, Slalom, MV, SpMV`). Such mechanisms remain useful for large caches because the relative share of compulsory misses increases when the cache size increases, as mentioned in [26].

Memory Latency As can be seen in figure 10b, software-assisted data caches do not perform well for latencies smaller than 10 cycles. Starting at 10 cycles, the performance increases very regularly with the memory latency. Since the main performance tradeoff of software-assisted caches are the additional cycles required to load an extra physical line, this overhead cost becomes negligible when the miss penalty is large.

4 Enhancing Software Optimizations

Software-assisted caches can be used to further increase program performance, because they provide a convenient support for enhancing data locality optimizations. In the sections below several examples

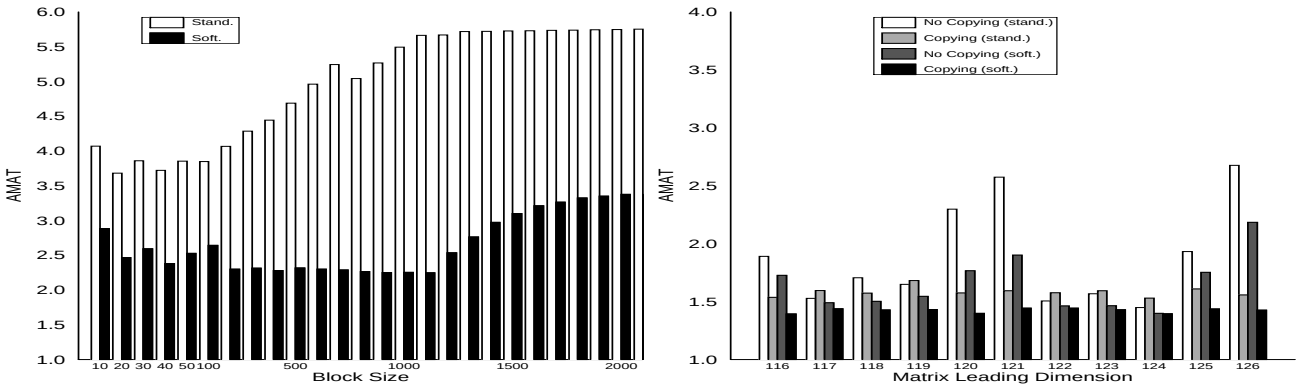

Figure 11a: *Optimal Block Size for Blocked Algorithms (*`Blocked Matrix-Vector Multiply`*).* Figure 11b: *Data Copying (*`Blocked Matrix-Matrix Multiply`*).*

Figure 11: *Performance of Software-Assisted Caches for Data Copying and Blocking.*

of application are provided. Note that, in these sections, the purpose of the examples is to illustrate rather than validate the concepts presented.

4.1 Scarce Locality

A first benefit of software control is to allow the exploitation of scarce locality. For example, in the Sparse Matrix-Vector multiply loop below, the average number of reuses per element is indicated by the average number of non-zero elements per column. Typically it is very small (in average, from 10 to 80 in 3-D problems). Besides, indirect addressing to the vector randomizes accesses and increases the reuse distance, threatening even more temporal locality exploitation. As shown in figure 6a (see `SpMV`), avoiding pollution by the `Matrix` and `Index` arrays (see loop below) significantly contributes to exploiting this locality.

```
DO j1 = 0,N-1
  reg = Y(j1)
  DO j2 = D(j1),D(j1+1)-1
    reg += A(j2) * X(Index(j2))
  ENDDO
  Y(j1) = reg
ENDDO
```

Note that software control needs to be enforced with user directives, since no compiler support exists yet for sparse codes. Note also that, if user directives are supported, software control can be extended to any non-numerical application, i.e., where it can be determined that a data construct exhibits spatial locality or no temporal locality.

4.2 Blocking

For most data locality algorithms, it is assumed that the cache behaves as a local memory and interference and pollution phenomena are ignored. But case-studies like [22] show that block sizes have to be chosen far smaller than the theoretical optimum due to these cache phenomena. By reducing pollution, software control allows to use larger block sizes, yielding higher performance for blocked algorithms, as illustrated on figure 11a, with blocked Matrix-Vector multiply.

4.3 Data Copying

Data copying has been proposed to reduce cache interferences in blocked algorithms [22]. The basic idea is to *effectively* manage the cache as a local memory, through an array which has the size of

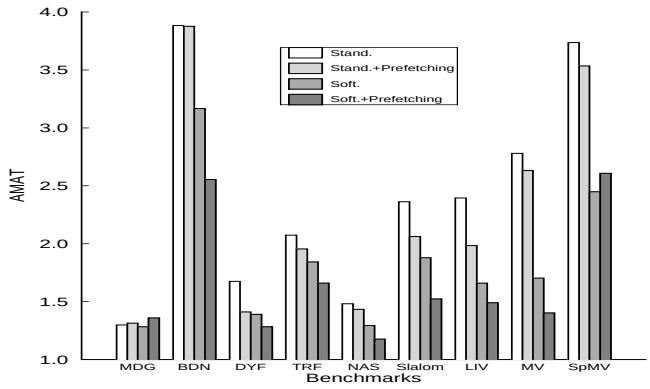

Figure 12: *Prefetching.*

the cache. However, copying is costly and can even degrade loop performance when there are few interferences (see figure 11b).

First, software control can help to enhance data copying by reducing a major overhead: loading the blocks from memory to cache (this overhead is not specific to data copying). By loading the blocks within specific loops (for refilling the *local memory array*), data copying breeds very regular and non-interleaved accesses to memory that can strongly benefit from the virtual line mechanism (ultimately a virtual line size equal to the block size could be employed; it was not the case in the experiments).

Second, during the refill operation, new elements are loaded that pollute the cache, and flush parts of the *local memory array*. In a software-assisted cache, the cost of copying is reduced because *local memory array* elements (which exhibit temporal reuse) are not flushed, as shown in figure 11b.

Performance increases are not very significant for the example provided, but the strong asset of software control in this case is to ensure that data copying is a *safer* technique. It also makes its cost easier to compute (since flushing of the *local memory array is avoided*), which simplifies the decision to copy.

4.4 Prefetching

Software prefetching as proposed in [4] requires hardware support and complex compiler analysis (insertion of prefetching instructions, computation of the prefetch distance...). In [28] software prefetching is combined with a data locality algorithm to avoid prefetching when no capacity miss occurs. The efficiency is improved, but the compiler complexity is even greater.

The two main flaws of *hardware* prefetching techniques are *wrong predictions* and *additional memory traffic*. Hardware schemes [18, 9] have been proposed to reduce these flaws (especially wrong predictions), but their on-chip space cost is very significant.

The mechanisms presented in this paper provide convenient and reasonably cheap hardware support for both *hardware* and *software* prefetching. Though software prefetching is not investigated in details in this paper, it can be noted that the two main hardware supports required, i.e., a prefetch buffer and distinctive load/store instructions, are already part of the current design.

Let us now consider how the mechanisms presented can be used to implement simple and efficient hardware prefetching. First, the bounce-back cache can be used as a prefetch buffer, which is necessary to avoid stalling the main cache, and consequently the processor, by incoming prefetch requests.

Second and most important of all, the software information on spatial locality can be used to assist the prefetch decision. *Prefetching is performed only for references with spatial locality.* As a consequence,

many wrong predictions are avoided.

Third, excessively long memory requests can also be avoided. The mechanism is the following: on a miss request, a virtual line is fetched as usual, and *the consecutive physical line is fetched as well.*[11] This second line is tagged *prefetched* and stored in the bounce-back cache. If there is a hit in the bounce-back cache on a prefetched line, this line is transferred to main cache, and the consecutive physical line is prefetched in turn. Consequently, prefetching is progressive, burst requests are avoided. This mechanism is efficient for latencies up to 25 cycles. Beyond, it becomes worthwhile to increase the prefetch distance by prefetching several physical lines at the same time, at the expense of a higher swap penalty.

Fourth, *prefetch-on-miss*, i.e., prefetching if the data is not already present in cache (as suggested in [21]), is not necessary here, since few useless prefetches will occur due to the software instrumentation that commands prefetching.

Several issues are raised by such a prefetching mechanism. First, prefetched lines can disrupt the bounce-back mechanism. A solution is to restrict the number of prefetched lines that can reside in the bounce-back cache, and to enforce that a prefetched line preferably replaces other prefetched lines (once the maximum number of prefetched lines has been reached). Second, coherence issues arise. On a miss request, the same coherence scheme used for virtual lines can be used for prefetched lines (the problems are exactly the same except that the number of physical lines loaded is more important: 1 virtual line + 1 physical line). If the processor hits on a prefetched line in the bounce-back cache, the problem becomes more difficult. A new prefetch is then initiated, but the prefetched line can already reside in any of the two caches. The solution adopted proved to be a reasonable tradeoff: after swapping occurs, the main cache remains stalled one more cycle, to check for the presence of the prefetched line. The bounce-back cache is also stalled two more cycles, but this second action has no significant impact on performance.

Figure 12 shows the performance of the software-assisted prefetch mechanism as described above. The main advantage of software-assisted caches combined with prefetching over plain software-assisted caches is that compulsory and capacity misses corresponding to vector accesses can be hidden. On the other hand, any kind of prefetching (even combined with software-assisted caches) is sensitive to memory latency: if the latency is low, prefetching is useless, if the latency is high, prefetching requests can not be fetched from memory in time. However, plain software-assisted caches can accommodate and even favor large memory latencies, as mentioned in section 3.2.

5 Other Related Work

Many papers have addressed the problem of optimizing cache performance under numerical workloads.

In [19], a mechanism called *stream buffers* is proposed for avoiding compulsory and capacity misses in regular array references, using hardware prefetching combined with several buffers. The mechanism does not work properly if the number of array references within the loop body, that induce compulsory/capacity misses, is larger than the number of stream buffers.

In [2], a *column-associative* cache reduces conflict misses in direct-mapped caches, by allowing a cache line to reside in several cache entries. Most conflict misses are eliminated. However, the mechanism does not deal with cache pollution. Still, it would be interesting to investigate the scheme performance with large cache lines.

In [23], a mechanism is proposed to dynamically decide whether a data causes conflicts and should be excluded from cache. This technique has been used for instruction caches, but proved to be less

[11] Note that a physical instead of a virtual line is prefetched. The reason is that, on a hit in the bounce-back cache, only a physical line is swapped to main cache. Swapping a virtual line would require to test the bounce-back cache a second time, which nearly doubles the swap penalty.

efficient for data caches which have different reference patterns.

In [1], code profiling shows that few load/store instructions induce many cache misses and it is consequently suggested that *labeled* load/store instructions can be used to optimize the cache behavior.

With respect to virtual lines, the MIPS R3000 maintains per-word tags but fetches four words on a cache miss. However, the MIPS R3000 does not support tagged loads.

Finally, at the time this paper was written and submitted, the authors were not aware of the HP-7200 *Assist Cache* which was announced in August 94 (see [25]). This design also implements a buffer for avoiding cache pollution by non-temporal data (there is a spatial-only bit, i.e., a non-temporal bit, per cache line), except that the buffer is placed before rather than after the main cache (as the bounce-back cache). Also, in the HP design, both the cache and the Assist Cache can be tested in the same time, which we assumed was not possible so as to allow very high clock frequencies (the HP design uses a 170Mhz clock and state-of-the-art circuitry to implement a fast enough fully associative Assist Cache). Finally, there is no mechanism for exploiting spatial locality like virtual lines, but a form of systematic prefetching is implemented that apparently does not use the spatial-only bit.

6 Conclusions

In this paper, a cache design for supporting temporal and spatial locality informations has been proposed. The spatial information is simply exploited by using long virtual lines, which have the main advantages and few of the flaws of long physical lines, except for coherence issues. The concept of *virtual lines* implements the notion of *adjustable line length*. Exploiting spatial locality in numerical codes is particularly profitable because many arrays are accessed with stride-one but are not or cannot be reused, so that compulsory misses can account for a significant share of total misses. The temporal locality is exploited by minimizing cache pollution through a *bounce-back cache*. This mechanism implements a form of *bypassing* that still allows to exploit spatial locality. In numerical loop nests, references with temporal locality are interleaved with references deprived of temporal locality that induce very significant cache pollution and prevent temporal reuse from being exploited.

Simulations proved that significant gains are due to software-assisted spatial locality exploitation for many numerical codes, because arrays without reuse dominate in loop bodies. For several other loops where reusable arrays account for a significant share of total references, eliminating cache pollution breeds substantial benefits. Globally, significant performance improvements were obtained for both the miss ratio and the average execution time, without increasing the memory traffic. Simple and easy to extract software informations proved to be sufficient for obtaining high performance increases. Though more sophisticated techniques might bring further improvements.

Through several examples, it was also illustrated that software-assisted caches provide a convenient support for several data locality optimizations, like blocking and data copying, for codes with scarce locality such as sparse codes and for prefetching.

References

[1] Santosh G. Abraham, Rabin A. Sugumar, B. R. Rau, and Rajiv Gupta. Predicatability of load/store instruction latencies. In *Proceedings of MICRO-27*, 1993.

[2] A. Agarwal and S. Pudar. Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *International Symposium on Computer Architecture*, 1993.

[3] Francois Bodin, Christine Eisenbeis, William Jalby, and Daniel Windheiser. A Quantitative Algorithm for Data Locality Optimization. In *Code Generation-Concepts, Tools, Techniques*. Springer-Verlag, 1992.

[4] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[5] Chi-Hung Chi and Hank Dietz. Unified Management of Registers and Cache Using Liveness and Cache Bypass. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24(7), pages 344–355, June 1989.

[6] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputing Performance Evaluation and the Perfect Benchmarks. In *Supercomputing '90*, pages 254–266, 1990.

[7] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[8] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. On Estimating and Enhancing Cache Effectiveness (Extended Abstract). In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, 1991.

[9] John W. C. Fu, Janak H. Patel, and B. L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO-26*, 1992.

[10] G. Irlam. *SPA package*, 1991.

[11] J. Heinrich G. Kane. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[12] D. Gannnon and al. SIGMA II: A Tool Kit for Building Parallelizing Compiler and Performance Analysis Systems. Technical report, University of Indiana, 1992.

[13] Elana D. Granston and Alexander V. Veidenbaum. An Integrated Hardware/Software Solution for Effective Management of Local Storage in High-Performance Systems. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 83–90, August 1991.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[15] IBM Corporation. *IBM RISC system/6000 technology*, 1990.

[16] Intel Corporation. *Intel i860 reference manual*, 1991.

[17] Intel Corporation. *Intel Pentium reference manual*, 1993.

[18] T-F. Chen J-L. Baer. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of IEEE Supercomputing*, 1991.

[19] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small, Fully-Associative Cache and Prefetch Buffers. In *International Symposium on Computer Architecture*, pages 364–373, May 1990.

[20] Norman P. Jouppi. Cache Write Policies and Performance. In *International Symposium on Computer Architecture*, May 1993.

[21] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *International Symposium on Computer Architecture*, pages 43–53, May 1991.

[22] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance of Blocked Algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[23] Kathryn S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, Technical Report CRPC-TR92214, April 1992.

[24] Motorola. *PowerPC 601, RISC Microprocessor User's Manual*, 1993.

[25] Dick Pountain. A different Kind of RISC. *BYTE*, August 1994.

[26] M. Horowitz S. Przybylski and J. Hennessy. Performance tradeoffs in cache design. In *International Symposium on Computer Architecture*, 1988.

[27] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[28] M. Lam T. Mowry and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, September 1992.

[29] Texas Instrument. *TMS390Z50, Data Sheet*, 1992.

[30] Michael Wolf and Monica Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26(6), pages 30–44, June 1991.