Accurate data distribution into blocks may boost cache performance

Dan Nam Truong
dtruong@irisa.fr

François Bodin
bodin@irisa.fr
IRISA/INRIA
Campus de Beaulieu
35042 Rennes, Cedex
France

André Seznec
seznec@irisa.fr

Abstract

Applications often under-utilize cache space and there are no locality optimization techniques available for non-scientific applications. DWe propose that data redistribution in memory be used to modify reference patterns to improve locality of references.

To understand the potential of such an approach and to explain where gains come from, we introduce distribution misses, and define a correlation measure to evaluate spatial locality.

Data distribution can help reduce capacity and conflict misses in regular caches, as our experimental results to show. We use as example a profile-based scalar data layout heuristic, which was able to remove up to 76% of the direct-mapped cache miss ratio on some benchmark traces.

Keywords: *Cache memories, Distribution miss, Data Layout, Optimization, Locality, Non-scientific code*

1 Introduction

Modern non-scientific applications tend to under-exploit cache space [BGK95] . Working sets become so large that caches are not able to take advantage of locality with irregular reference patterns. Since cache sizes grow too slowly [BGK96], it is important to regularize the cache reference patterns. A software optimization approach that we call data distribution could be used to adapt applications to the memory hierarchy. It consists in modifying the address of data-elements in memory so that data used simultaneously fits in the cache. This approach is orthogonal to the loop-transformation approach used for arrays.

As a first step towards data-distribution solutions, we propose in this paper a formalism to explain the impact of data distribution. It explains in particular how data distribution can impact on capacity misses. We also present preliminary experimental results, to show that data distribution is a very promizing approach for non-scientific applications optimization.

Regular array accesses beneficiate from efficient loop transformation algorithms. They are source code transformations which modify the reference patterns but do not modify the placement of the data in memory. This approach cannot be used for irregular memory references generated by non-array data: address computations are not index based. Therefore we consider a dual approach consisting in changing the relative position of data elements in memory to improve spatial locality. We call this data-layout optimization approach data distribution, in contrast to loop-transformation techniques.

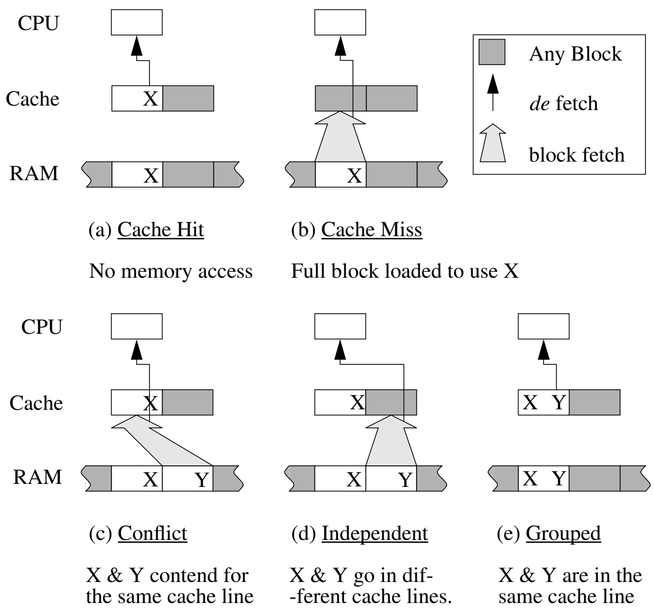

Figure 1: Locality impacts on cache behavior

Let's analyse how data must laid out in the cache to improve locality[1] of references, by illustrating some simple layout alternatives, as shown in figure 1. Let's look in more detail into reasons of a hit or a miss (figure 1(a,b)): A cache manipulates blocks of memory: A reference to a data-element is a reference to the block holding the data element. Therefore, if we consider two data-elements X and Y with interleaved references, their distribution among the blocks impacts on the cache miss rate. Three possibilities arise.

- Figure 1(c): X and Y are stored in distinct memory blocks which compete for the same cache line. Each reference generates a conflict miss. Temporal locality is counter productive.
- Figure 1(d): X and Y are stored in distinct blocks mapping to different lines. Temporal locality is beneficial. A block-layout optimization must prevent (c) to get (d) to remove conflict misses.
- Figure 1(e): X and Y are stored in the same block. Temporal locality is beneficial, and only a single line is used: The cache benefits from spatial locality. This solution is the best: fewer blocks are used used to handle this reference pattern. Fewer blocks to store in the cache simultaneously mean fewer cold and capacity misses, and fewer chances of having conflict miss hazards.

With these simple remarks, we see the importance of good spatial locality and proper data-element distribution among blocks. Therefore, data distribution optimization for set-associative caches must be a two steps process. Data elements used simultaneously must be redistributed among blocks to improve spatial locality, and blocks used simultaneously must be laid in memory to improve block temporal locality. Block layout uses the cache address mapping properties of set-associative caches to ensure that blocks used simultaneously load in different lines.

Standard arrray transformations attempt to improve spatial locality by making stride 1 accesses to arrays. Data-element distribution into blocks was not considered because array elements have a fixed position in the array. If elements are referenced directly, their relative position in memory does not impact on program correctness, so they can be redistributed in memory to improve spatial locality. Redistribution has just to conform to the few layout constraints imposed by compilers (keeping static, stack, heap allocated data respectively together).

In section 2 we present related work. We introduce data element block *distribution misses* in section 3 to account for the impact of data layout on the capacity miss ratio, and define a lower-bound for cache miss-ratio. In section 4 we provide two metrics *Activity* and *Correlation* to quantify locality of reference. We used these metrics to develop a profile based heuristic for data-elements distribution and block layout, and used it to evaluate the potential of such an approach. Our experimental results, shown in section 5, show a great potential, with miss rates dropping by as much as 76%. We conclude in section 6 and propose some reseach directions to exploit this great potential.

[1] We say that a memory location **benefits from temporal locality** in the cache when it is accessed more than once while it is held in the cache. Furthermore, we say that a memory block**benefits from spatial locality** when many of its data-elements are accessed while it is held in the cache.

2 Related work

Many hardware solutions are available to address all aspects of cache performance [CB92, FJ94, DSW95, BS95, Jou90, AP93, JW94]. However, Burger et al. argue that applications often do not use more than 10% of the cache space, and a software managed local memory might be a better solution [BGK95]. However static compiler analysis alone may also be insufficient to take fully advantage of the local memory, and necessitates extra instruction overhead to handle it. We argue that with data distribution, it is possible to combine beneifits of software management of local memories with the dynamic cache adaptation to program behaviour, while saving on the extra overhead incured by explicit local memory management.

Much work has been done to analyze cache behavior, but most works measure evolution of the miss ratio for different cache parameters [Prz90, Smi82, Smi86]. To our knowledge, besides [BGK95], little work has been done to evaluate the the impact of data layout on locality, and measure the ratio of cache space effectively used. This paper proposes distribution misses to take into account spatial locality, and proposes data-distribution to address the problem. Grimsrud et al. express locality in a reference trace as $Addr(T+\delta_t) = Addr(T) + \Delta_{Addr}$, and uses this to provide a 3D graph, $reference = \mathcal{F}(stride, time)$, to view locality patterns [GAFN94], but do not consider layout optimization. The locality measures we provide in this paper are adapted to data-layout optimization.

Profile based instruction layout in memory to improve I-cache performance has already been explored in many papers [HC89, PH90, Hei94, CG94, TXD95].

The approach is similar in its principles to data-distribution, and efficient layout techniques have already been proposed. Pettis & Hansen [PH90] propose basic-block layout heuristics (they pack frequently used looping sequences) as well as procedure layout (they lay together procedures calling each other to minimize cache interference risks). They also prevent I-cache polution by removing unused basic blocks from the main instruction stream. Profile-based instruction layout provides about 10% overall speedup by removing I-cache misses, virtual page faults and improving branch prediction. With proper training sets (they use profiling), speedups can be held for any inputs. Instructions are a special case of irregularly referenced data (for the I-cache) and we believe that efficient layout optimization can also be achieved for data caches.

To hide memory latency and to keep the CPU busy, non-blocking caches and software [CKP91] or hardware [Jou90, BC91, FPJ92, PK94] data prefetching techniques have been proposed. Prefetching tends to consume extra memory bandwidth, which in many applications is a critical resource, while non-blocking caches consume register space. Data layout optimization can reduce distribution and conflict misses, reducing at the same time the number of prefetches or pending loads necessary. Furthermore, the locality metrics we propose can be used by software prefetching techniques to select which blocks to prefetch.

Scientific programs have benefited from much more extensive analysis than non-scientific codes. Software array access optimization research has been very active for some years. With loop transformations like tiling [BGS94, CKT94], it is possible to prevent most interference misses [TGJ93] on regular array access patterns. These techniques are sometimes insufficient because interferences can still appear for some array leading sizes [BS95]. Modifying access patterns is not sufficient to eliminate interferences so it can be combined with array layout (distribution) optimizations. It is quite limited: Only array leading-size padding and inter-array padding can be used. Padding has been used to align arrays on multiprocessors [GMB95, TLH90, Por89] but has also been used to align arrays in caches [BCJ94]. For non-scientific applications, however, code transformations like tiling are irrelevant, since complex structures are not accessed using indexes. The orthogonal approach, data-distribution is better suited.

Although scalars benefit from efficient register allocation techniques [ASU86, CCK90], and new dynamic allocation libraries take advantage of temporal locality by reallocating recently freed blocks [GZH93, BZ93], this is still insufficient to reap full processing power from a modern CPU with non-scientific applications, and new approaches must be developped to improve memory hierarchy performance. This paper attempts to show that data-distribution is a very promizing approach for non-scientific code optimization that will have to be investigated thoroughly.

3 Characterizing cache use

We recall the Three Cs miss classification [Hil87] using Sugumar's definition which use a perfect[2] cache [Sug93] and update it to handle data distribution. Data-distribution impacts not only on block layout in memory, but also on the distribution of data-elements among the memory blocks. This changes block reference patterns, and impacts on the capacity miss ratio. Therefore we provide an extention to the classification, data elements block *distribution misses*. They account for sub-optimal data-element distribution, just like conflict misses account for sub-optimal block replacements. This allows us to evaluate the lower miss ratios boundary attainable with a perfect data distribution and a perfect cache of given cache and line size.

3.1 Sugumar and Hill's classifications

To analyze cache hardware block replacement policies, Hill classified cache misses. We use Sugumar's definitions which are finer, and recall them since they are not as widely used.

Definition 1 *A* **cold miss** *or* **compulsory miss** *is the first load of a block in the cache, due to the first reference to it since the beginning of the program.*

This condition looks at block being loaded. Subsequent characterizations look at the block replaced.

Definition 2 *A memory block is* **dead** *at a given time if it will never be referenced by the program again, otherwise it is* **live**.

Belady [Bel66] has shown that there exists at least one optimal block replacement policy denoted *Min*, which yields the smallest miss ratio possible for a fully-associative cache. On a miss, *Min* replaces any dead cache block, or if there are none, the live block farthest referenced into the future. Therefore, a fully-associative cache using *Min* policy is perfect.

[2]We call perfect a fully-associative cache using Belady's optimal *Min* block replacement policy [Bel66]

Definition 3 *A* **capacity miss** *is the replacement of the block in the cache that will be referenced the farthest in time*[3].

Definition 4 *A* **conflict miss**[4] *or collision miss is the replacement of a live block which is not the farthest one referenced in time.*

A suboptimal block replacement can happen with suboptimal hardware replacement policies like LRU, or with address mapping. To differentiate these two cases in *set-associative caches*, Sugumar [Sug93] further differentiates conflict misses:

Definition 5 *A* **Mapping miss** *is the replacement of the farthest referenced block in a set, while the farthest referenced block in the cache is not in that set.*

Definition 6 *A* **Replacement miss** *is the replacement of a block that is not the farthest one referenced in the set.*

A mapping miss is a loss of efficiency because the cache is split into sets, replacement misses are due to a bad replacement policy. All conflict misses in a *direct-mapped cache* are mapping misses, exept when cache bypass is allowed: Min policy could choose not to load the block in the cache but few processors actually implement a bypass instruction.

The above classsification only considers cache blocks, and mapping misses account only for block layout in memory. With data-layout optimization, it is necessary to consider the impact on cache performance of data element distribution into blocks.

3.2 Distribution misses

If we redistribute the data-elements among the different blocks, the reference patterns to the memory blocks will change. Therefore, Min generates different replacement choices, and the miss ratio of a perfect cache can change. We define *distribution misses* to account for this and compute a lower bound to the miss ratio of a perfect cache of a given size.

A data-layout is the address mapping of data elements in memory. Shuffling addresses of data elements modifies the data layout. Of all the possible data layouts, there is at least one that generates the lowest miss ratio for a perfect cache (the number of permutations is finite).

Definition 7 *An optimal* **Data-layout** *is a data-layout for which a perfect cache yields the lowest miss ratio possible.*

Note: the address offset of a data element within its block does not change block reference patterns, so it does not affect the miss ratio – assuming data elements do not overlap two blocks. The offset within the block will therefore not be considered.

We propose a new miss characterization, which takes into account how data-elements are distributed among the blocks, similar to the definition of conflict misses.

Definition 8 *A data-element block* **distribution miss** *is a capacity miss that would not have happened with an optimal data-element distribution into memory blocks.*

Looking back at figure 1(c,d,e), we see that misses generated by (c,d) can be prevented by changing the data-elements distribution to get case (e). The misses generated by (c,d) are distribution misses.

We cannot provide a method for computing exactly distribution misses because we do not know how to easily find an optimal data-element distribution into blocks (we would have to try all address permutations). However, we can easilly compute a lower bound for the miss ratio of a perfect cache with an optimal distribution, so we will use this to evaluate the quality of a distribution. Data elements cannot be smaller than a byte, so a perfect cache of size C with 1 byte per cache lines generates a minimal miss ratio $M1$ independant of data distribution.

Property 1 *A perfect cache of size C and line size L generates a miss ratio M greater or equal to $M1/L$, even with a perfect layout.*

If we approximate the miss ratio of a perfect cache with optimal data distribution to $M1/L$, we can split the capacity miss ratio according to our data distribution framework. The capacity miss ratio is the sum of the distribution miss ratio and the fundamental miss ratio:

Definition 9 *We call* **Fundamental miss** *ratio $M1/L$. It is not possible with a given cache size and line size L to generate fewer misses.*

Definition 10 *We assimilate $M - (M1/L)$ to the* **Distribution miss** *ratio.*

[3] Hill's original classification defines as a capacity miss a miss occuring in a fully-associative cache of the same size with the same replacement policy as the one studied – usually LRU. However, Sugumar showed that this definition is too approximative, since it can yield negative conflict miss ratios [Sug93] and using a perfect cache is a better alternative.

[4] Sometimes also called an **interference miss**. This is misleading since cross or self interference misses usually denote conflict and capacity misses within loop nests which do not fully exploit temporal locality of references to an array.

We can make this approximation for two reasons. First, $M1/L$ is a lower miss ratio bound, so it is not possible to get negative distribution miss rates[5]. Secondly, current caches are quite large, and with larger caches $M1/L$ will be closer to the real miss ratio of the perfect cache with optimal layout. An extreme case is when the whole working set fits in the cache: data distribution does not matter any more, since, no matter the reference pattern, the data will be in the cache.

The miss classification becomes:

$$total\ miss \begin{cases} Cold \\ Capacity \begin{cases} Fundamental \\ Distribution \end{cases} \\ Conflict \begin{cases} Replacement \\ Mapping \end{cases} \end{cases}$$

Data-elements distribution into the blocks is an important factor, it changes the distribution miss ratio, impacting on all cache configurations, from direct-mapped to fully-associative caches. For direct-mapped and set-associative caches, block layout in memory also affects performance by changing mapping miss ratios. Since block layout impact is already covered by Sugumar's mapping-conflict miss definition, the miss classification needs no more refinement to evaluate data distribution optimizations. Data distribution and block mapping effects can combine to provide a great optimization potential. Only fundamental misses cannot be removed.

4 Irregularly referenced data-layout

We have seen that data layout affects the cache miss ratio, and we have defined a miss classification to explain the impact of data layout. However, this does not help to evaluate the quality of a data layout, a priori. In section 4.1, we propose metrics which measure the quality of a data layout, to evaluate the impact of a data distribution on locality properties of memory reference patterns. We use these metrics in the scalar data-layout heuristic shown in section 4.2, that we developped to generate the preliminary experimental results.

4.1 Quantifying locality of references

To get good cache performance, it is important that memory reference patterns display very good locality properties. Data elements with interleaved references must be redistributed into the same blocks to reduce distribution misses, and blocks with frequently interleaved references must not contend for the same cache lines, to remove mapping misses hazards. We introduce some notions needed to quantify formally these aspect of locality of references.

Caches use spatial and temporal locality properties to attempt to keep simultaneously variables that are "*used at the same time*" (interleaved, repeated references). To measure this effect, it is necessary to have a notion of time.

Min, measures time as the count of references separating 2 references to an element. However, Belady notes that replacing erroneously a block is less disastrous if the next reference to it is far in time than if it is very close [Bel66]: until its next reference, it won't interfere with references to other block. Analyzing an unknown number of references into the future is not practical to define global metrics, and we are not constraining ourselves to finding the optimal layout solution, but only to finding a good layout at the lowest expense. Therefore, we introduce a time window to account for this approximation.

Definition 11 *We define a* **time reference window** *W as a time interval of size $||W||$. Two references are said to be* **close** *in time if the time interval between them is smaller than $||W||$.*

The choice of the window size is a sampling problem: if the window is too small, some interleaved reference patterns, considered too far, will not be seen. If the window is too big we will fall back to the complexity of Min measures.

Data elements that are unused while their block is in the cache waste cache space used to store them as well as memory bandwidth used to load them. Elements referenced farther apart than $||W||$ have a higher likelihood of necessitating a reloading of the block, so the block should not be considered as benefiting of spatial locality between them.

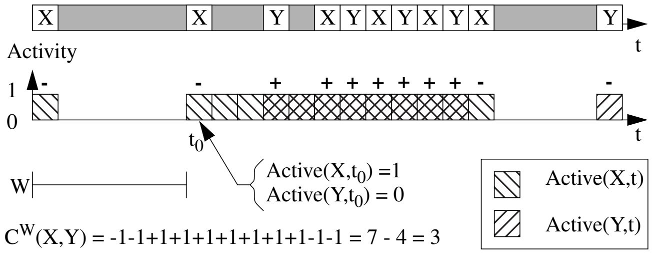

Figure 2: Computing *Correlation* between 2 data-elements

[5]This happenned to Sugumar, and led him to reconsider the definition of the 3C's miss rates using Belady's Min policy

To depict the fact that an element X of memory is referenced or not within $||W||$, we define its activity (as shown in figure 2).

Definition 12 *We note $Ref(X,t)$ the predicate that is 1 if X is referenced at time t and 0 otherwise.*

Definition 13 *An element X of memory is* **active** *at a time t, if t is between two references to X that are not separated by a time interval greater than $||W||$.*
$A^W(X,t) = 1$ *if* $\exists(t', t") \mid (t \in [t', t"]) \wedge (Ref(X,t') = 1, Ref(X,t") = 1) \wedge (|t" - t'| \leq ||W||)$, *and* 0 *otherwise.*

Now we can provide a correlation measure based on activity, to indicate if 2 elements have often closely interleaved references (see figure 2), and are rarely referenced independantly. Correlation is a global measure over the whole program (time independant) to evaluate the relevance of regroupping two scalars for spatial locality improvments.

Definition 14 *The* **correlation** $C^W()$ *of two data-elements is the count of all references done while both are active minus all the references to each while the other element is inactive.*
$C^W(X,Y) = \sum_{t=0}^{t_{End}}((A^W(X,t) \times Ref(Y,t)) + (Ref(X,t) \times A^W(Y,t)) - ((1-A^W(X,t)) \times Ref(Y,t)) - (Ref(X,t) \times (1 - A^W(Y,t)))$

An optimal layout is therefore a layout for which highly correlated data-elements are stored in the same block, and for which highly correlated blocks map to distinct cache lines.

4.2 A direct-mapped cache data-layout heuristic

With the *Activity* and *Correlation* metrics, we developped a profile-based scalar data-elements layout optimization heuristic for direct-mapped caches. We used this heuristic to provide preliminary data-layout results to show the potential gains of data layout.

The algorithm, described in figure 3, is decomposed into four steps:

- *Computing activity* of each data-element at each time t it is referenced during profiling.
- *Redistribution of correlated data-elements into the same blocks.* We repeatedly insert in a block the data element correlating most with it, and recompute block's activity, until it is full. Data elements are not allowed to overlap 2 blocks.
- *Distribution of correlated blocks into distinct cache lines.* We seek the block correlating the most with the blocks already laid-out. We then assign it to the cache line that correlates the least with it. We update cache and line activities and lay another block.
- *Memory layout* is done by assigning blocks any free memory address that maps into the selected cache line (we assume a direct-mapping). We also pick the block offset of data-elements arbitrarily – it does not affect miss ratios.

We obtain a layout that uses a minimum number of memory blocks to satisfy a given number of memory references. The layout also reduces the risks of having blocks used simultaneously in the cache conflicting for the same cache line.

This heuristic always finds an efficient layout solution. However it is still quite costly and cannot be used as-is in an optimizing compiler:

- Profiling needs to store activity information for each data-element over the whole execution trace, this can be too costly for huge programs. Data-elements placement needs $O(n^2)$ correlation computations and blocks layout needs $O(b^2)$, n the number of data-elements, and b the number of blocks. This is too expensive if n and b are big (i.e. a whole program's data).
- This heuristic applies only for scalars, and must be adapted to handle complex data structures.

Taking into account data structuring and compiler data layout constraints necessitates a more complex analysis. However, it can also help the scalar data layout: constraints reduce the number of elements that can be redistributed together, reducing n greatly (for example, redistribution of fields within a C structure). These layout constraints reduce the layout possibilities, and could reduce data layout gains. However, current caches are large, providing a high tolerance on layout. Furthermore, program and compiler layout constraints often reflect locality in data usage. It is locality generated by these data organisation constraints that caches capture. They are however too weak to provide good locality as application data sets grow larger. Data distribution can therefore provide the extra layout constraints needed based on locality properties. More work will be necessary to verify this, however.

To show that distribution heuristics can be developped, we wrote one which lays scalars using profiling data. In the next section we use it to show that indeed, miss rates are improved.

5 Experimentation

We provide experimental results to show the potential of data distribution. Fundamental, distribution, capacity and conflict miss ratios are given for direct-mapped, 2-way and 4-way set associative caches. The capacity miss ratio is the miss ratio of a perfect cache. The fundamental miss ratio is computed using the miss ratio of a perfect cache with 1 byte per line. Cold misses are counted as capacity misses. Results are given for the original and improved layouts.

5.1 Experimental set-up

Experimental results were obtained using traces generated by *Spy*, a tracing tool working under SunOS-4 [Irl92]. The cache simulated is a 16 KB cache with 32 bytes per lines, and true LRU is used for associative caches. We developped a range of tools to gather the miss ratios, figure 4.

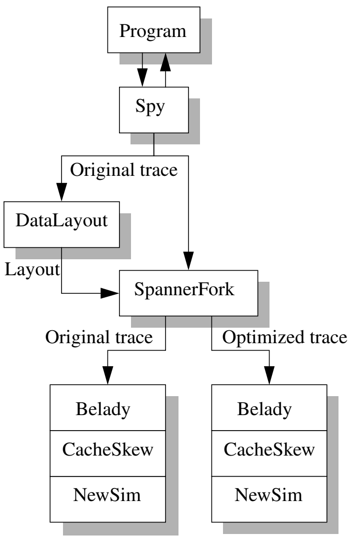

Figure 4: Simulation tools used

- *Belady* provides the miss ratio of a perfect cache (a fully-associative cache using Belady's Min replacement policy). It is used to get the capacity miss rates, as well as the fundamental miss rate (we simulate 1 byte lines and divide the miss rate obtained by the line size, 32). *Belady* does not simulate set-associative caches, so we do not present mapping and replacement miss ratios. We do not present cold miss ratio either, since it is irrelevant for our study (divide the number of bytes used by the line size).
- The $CacheSkew$ and $NewSim$ cache simulators provide direct and set-associative cache miss rates.
- $DataLayout$ is the program which optimizes the data layout. it generates a layout information file decoded by $SpannerFork$. $SpannerFork$ outputs an optimized layout trace by replacing data addresses with optimized addresses.

A trace simulation approach was chosen, because our heuristic can only layout scalars, and in a trace, all data references are seen as scalar references. To handle data structuring constraints, more work will be necessary. Our goal is not to provide a finished product, but to show the potential behind data distribution. The downside is a slow simulation process since we have to compute correlation between all the scalars (with compiler layout constraints we would have smaller sets of scalars to correlate), so traces are only a few million references wide.

The $DataLayout$ program implements a simplified version[6] of the heuristic presented in section 4.2:

- We used a fixed instead of a sliding time window. This cuts time space into $||W||$ sized chunks. References in distinct windows are considered far apart, although they might be close if we used a sliding window (for example $t_i \in W_1$ and $t_j \in W_2$ and $t_i = t_j + 1$).
- One bit is used to mark up if a data-element is referenced in a window. This reduces storage needs of the reference/activity array to 1 bit per de per W_i. Of 2 elements active in a window, we cannot tell which one is most referenced. if W is too large, our program will find that all elements correlate. $A^W(de,t) = A^W(de,W_i) = 1 \ if \ (t \in W_i) \ \wedge \ (\ \exists \ t' \in W_i \mid Ref(de,t') = 1)$
- Correlation is simplified to a sum of AND-ed bits. $C^W(X,Y) = \sum_{W_i=0}^{W_{End}} (A^W(X,W_i) \wedge A^W(Y,W_i))$
- to eliminate block activity recomputations during data-elements distribution, the block's activity is approximated to the activity of the first element stored in the block – it is the most referenced element available when the block is created.

[6] Our goal is not to provide the most efficient layout, but only to show that better layouts can provide low miss rates

5.2 Tracing “ls”

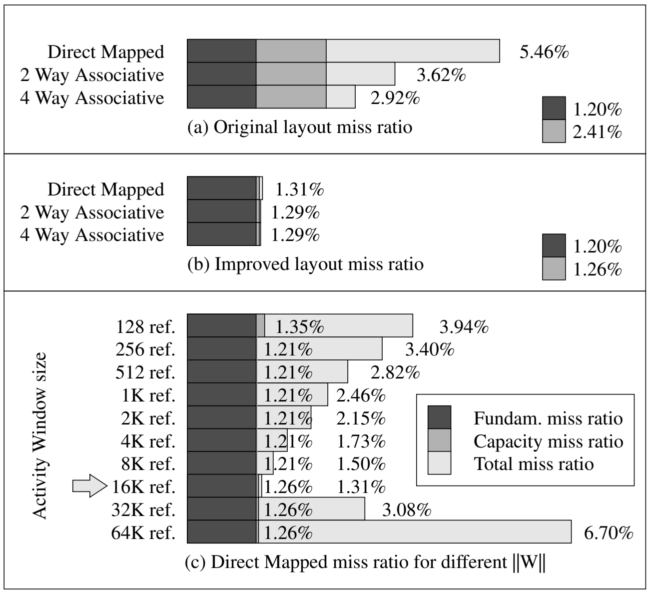

Figure 5: Miss ratios for “ls”

We first attempted our optimization on a very small trace of 102,614 memory references generated by the Unix *ls* command. It references 16,149 distinct data-elements for a total of 39,471 bytes. These preliminary results (figure 5) already give some insights:

- The original capacity miss rate of 2.41%, figure 5(a), is reduced to 1.26% with improved data-element distribution, figure 5(b), a 48% gain. This is due to a 95% reduction of the distribution miss rate (from 1.21% to 0.06%). Note that fewer capacity misses means fewer block replacements, therefore fewer chances of making a wrong block replacement choice (i.e. conflict miss).

- The direct-mapped cache miss ratio drops from 5.46% (a) to 1.31% (b), a 76% gain. The effects of our data-distribution and block layout optimizations combine to provide a total miss rate which is actually lower than the capacity miss rate of the original layout alone. This shows the dramatic impact of data distribution on cache performance: we even bested the capacity miss ratio.
  Contrarily to hardware cache optimizations, with data-layout optimizations, the miss ratio obtained with a perfect cache is not sufficient to evaluate the lowest miss ratio attainable with a real cache.

- Associative caches also benefits from the improved layout. The total miss ratio of the 4 way set-associatif cache drops from 2.92% to 1.29%, a 56% gain. For this example, data distribution is so performant that using an associative cache yields little gain over a direct-mapped cache. In fact, the direct-mapped cache miss ratio (1.31%) is already close to the fundamental miss ratio (1.20%), so even a fully-associative cache could not provide any gain (in fact, it could even degrade performance by generating replacement misses, since LRU is not optimal).

- The heuristic we used to improve data-layout uses a time window to measure data correlation. The choice of the window impacts on the layouts obtained. We tried a range of time windows to see the impact of this choice on miss rates obtained. For the capacity miss, the variation induced is small (c). The window must be large enough (over 256 references) to find data-elements to regroup into blocks, and small enough (less than 8K references) to differentiate between elements that should not be in the cache simultaneously. Since the cache is quite large there is a high tolerance on the distribution of the data elements into blocks. The window impacts much more on block layout. If the window is too small, mapping conflicts arise because blocks are not correlated sufficiently well (repetitive, but far appart reference patterns are not seen). If the window is too large, data-elements distribution becomes too laxist to allow efficient block mapping, and the heuristic also becomes incapable of discriminating between weak and strong correlations (because our implementation uses only a single bit to measure activity). Since our heuristic uses the same window size $||W||$ to distribute data-elements and perform block layout, a window size of 16K references seems the best compromize. We will use this value for all the other experiments.

An optimal data layout can reduce the direct mapped cache miss ratio by up to 78% (by leaving only 1.20% fundamental misses). Our heuristic achieves a 76% reduction, far better than what set-associativity alone can achieve with the original layout.

5.2.1 Tracing “f77” compiler

We traced 2 million memory references of the f77 compiler under Sun-OS 4 while it was compiling 078.swm256. We first discarded 2 million references to remove program startup effects. The fundamental miss rate is 1.35% ($\frac{907,507}{32} = 28,360$ misses).

Table 1: F77 Miss ratios

Cache	DM	2-SA	4-SA
Fundamental	**1.35%**	1.35%	1.35%
Distribution	**1.38%**	1.38%	1.38%
Capacity	2.73%	2.73%	2.73%
Conflict	2.95%	1.78%	1.42%
Total **Original**	**5.68%**	**4.51%**	**4.15%**
Fundamental	1.35%	1.35%	1.35%
Distribution	**0.65%**	0.65%	0.65%
Capacity	2.00%	2.00%	2.00%
Conflict	1.83%	1.26%	1.01%
Total **Improved**	**3.83%**	**3.26%**	**3.01%**

Table 2: 078.swm256 miss ratios

Cache	DM	2-SA	4-SA
Fundamental	**0.242%**	0.242%	0.242%
Distribution	**0.001%**	0.001%	0.001%
Capacity	0.243%	0.243%	0.243%
Conflict	0.095%	0.001%	0.002%
Total	**0.338%**	**0.244%**	**0.245%**
Fundamental	0.242%	0.242%	0.242%
Distribution	**0.000%**	0.000%	0.000%
Capacity	0.242%	0.242%	0.242%
Conflict	0.010%	0.010%	0.010%
Total	**0.252%**	**0.252%**	**0.252%**

In the original layout, 51% of the capacity misses are distribution misses which could be removed. Our layout heuristic removes 53% of the distribution misses (from 1.38% to 0.65%), reducing the capacity miss rate by 27%, table 1. Furthermore we remove 38% of the conflict misses for the direct mapped cache (2.95% to 1.83%). After layout optimization, there are still conflict misses. An associative cache can remove some more conflicts (1.01% left for a 4-way associative cache).

A perfect data layout can reduce the original direct-mapped cache miss ratio by up to 76% (from 5.68% downto 1.35%). Our layout heuristic manages a 33% reduction (down to 3.83%). By cumulating set-associativity and data distribution, gains reach 47%.

5.2.2 Tracing "078.swm256" Specfp-92 benchmark

After discarding the first million references, we traced 5 million references of 078.swm256 (compiled under Sun-OS 4 using -O4 flag). This program accesses arrays sequentially. The instrinsic miss rate is 0.242% ($404,424/32 = 12,639$ misses).

We show with this example, that a well behaved scientific application makes almost perfect usage of the cache line space, there are almost no distribution misses, table 2.

Some conflicts appear in the direct mapped cache, easily removed by set-associative caches, or data-distribution. The only data-distribution alternative for scientific codes, padding, may not suffice. Contrarily to non-scientific applications, loop-transformations can be used efficiently to generate tiled or stride-one accesses.

5.2.3 Tracing "gzip"

We traced 2 million references of gzip while it was compressing a trace. We first discarded 2 million references to remove program startup effects. The fundamental miss rate is 0.71% ($476,185/32 = 14,881$ misses). The 141,379 data-elements referenced represent 213KB of data.

Table 3: GZip Miss ratios

Cache	DM	2-SA	4-SA
Fundamental	**0.71%**	0.71%	0.71%
Distribution	**8.96%**	8.96%	8.96%
Capacity	9.67%	9.67%	9.67%
Conflict	11.54%	9.27%	8.27%
Total **original**	**21.21%**	**18.94%**	**17.94%**
Fundamental	0.71%	0.71%	0.71%
Distribution	**2.00%**	2.00%	2.00%
Capacity	2.71%	2.71%	2.71%
Conflict	6.63%	4.54%	3.78%
Total **Improved**	**9.34%**	**7.25%**	**6.49%**

Although the 0.71% fundamental miss rate is quite low, GZip 9.57% original capacity miss ratio is extremely high, due to distribution misses, table 3. Our heuristic removes 78% of the distribution misses (from 8.96% to 2%), a 72% reduction of capacity misses.

Improved block mapping reduces conflicts by 43% for the direct-mapped cache (from 11.54% to 6.63%).

The total miss rate is reduced by 56% (from 21.21% to 9.34%). It again beats the original capacity miss rate. Associativity also combines well with data distribution to remove conflicts. The 4 way set-associative cache miss ratio drops to 6.49%.

5.2.4 Experimental summary

We defined distribution misses and divided capacity misses into distribution and fundamental misses. The fundamental miss ratio is usually quite low (less than 1.35% in all our experiments), leaving much room for cache miss rates improvment. Capacity miss rates are usually quite high due to distribution misses (up to 9.67% in gzip). Data-distribution heuristics can be used effectively to reduce distribution misses.

Layout also impacts on conflict misses. Better data-distribution reduce conflict miss hazards, while improved block layout into memory reduce mapping miss rates. By choosing block addresses in memory, we actually provide compile time hints on the future to the hardware block replacement policy.

We used a simple layout optimization heuristic to show that it is possible to design a tool capable of finding improved layouts. Layout choices made by our heuristic usually provided significant capacity miss reductions of up to 71%, and direct-maped cache miss reductions of up to 76%.

6 Conclusion

In many applications, cache space is under-utilized, yielding excessive miss ratios. Loop transformations can be used to modify array reference patterns, but non-scientific applications tend to use other data types. We show that data-distribution is an interesting approach to provide software locality optimization techniques for the other data types.

Data distribution consists in choosing the relative address locations of data elements in memory. We provide a formalism for data distribution. Since data-distribution optimizations impact on capacity miss ratios, we define distribution misses to account for poor data-element distribution into blocks and compute a lower miss ratio boundary, the fundamental miss ratio. We also define correlation between data elements to evaluate spatial locality. It facilitates the development of data distribution heuristics.

Data distribution for caches is a two faceted problem. Memory blocks layout in memory must be selected according to the cache mapping policy and block reference patterns to reduce mapping-conflict misses. More importantly for non-scientific applications, we have shown that proper data-elements redistribution into blocks impacts significantly on the capacity miss rate (we removed up to 71% of the capacity misses). Data distribution can count on a great potential for miss reduction, since limit is only the fundamental miss ratio. The data distribution heuristic we develloped was not able to reach this limit, but already managed significant gains for direct mapped caches, of up to 76%, sometimes even beating the capacity miss ratio of the original layout.

This study is a first step to evaluate the potential of data-layout for cache optimization of non-scientific programs. These first results hint that it is important and possible for compilers to pay attention to how data is placed in memory, and that doing so can remove a significant amount of capacity and conflict misses. Currently compiler data layout constraints are weak, and are never based on data locality considerations. We feel that one of the most promizing aspects of data layout optimization for non-scientific codes, is the optimisation of the layout of complex data structures allocated dynamically (C struct an C++ classes). We are currently developping a new allocation library which will handle data-layout optimizations. Other interesting aspects of data-distribution are fast scalar layout, array padding for caches and instruction basic block layout and prefetching.

Table 4: Total miss counts

Cache	Perfect	DM	2-SA	4-SA
f77				
Fundamental	28360			
Original	57211	119177	94678	87213
Improved	41950	80217	68262	63106
078.swm256				
Fundamental	12639			
Original	12655	17646	12721	12787
Improved	12640	13149	13147	13147
gzip				
Fundamental	14881			
Original	202886	444839	397174	374833
Improved	56867	195872	151970	136205

Acknowledgements

We would like to thank Richard Uhlig and Olivier Michel for proofreading this article and G. Irlam for developping the spy package.

References

[AP93] A. Agarwal and S. D. Pudar. Column associative caches: A technique for reducing the miss rate of dm caches. *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, Techniques and Tools*. Addison Wesley, 86.

[BC91] J. L. Baer and T-F. Chen. An effective on chip preloading scheme to reduce data access penalty. *Proceedings of Supercomputing 91*, 1991.

[BCJ94] D. F. Bacon, J-H. Chow, and D. R. Ju. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. *Proceedings of CASCON'94*, November 1994.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM System Journal*, page 24, 1966.

[BGK95] D. C. Burger, J. R. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general purpose multiprocessor. Technical report, University of Winsconsin, 1261, 1995.

[BGK96] Doug Burger, James R. Goodman, and Alain Kagi. Memory bandwidth limitations of future processors. In *23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[BGS94] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[BS95] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. *Proceedings of the 22nd International Symposium on Computer Architecture*, 23(2):265, May 1995.

[BZ93] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, 28(6):187–196, July 1993.

[CB92] T. F. Chen and J. L. Baer. Reducing memory latency via non blocking and prefetching caches. *5th symposium on Architectural Support for Programming Languages and Operating Systems*, 1992.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

[CG94] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, November 1994.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *4th symposium on Architectural Support for Programming Languages and Operating Systems*, 1991.

[CKT94] S. Carr, K. S. Mc Kinley, and C-W. Tseng. Compiler optimisation for improving data locality. *proceedings of the 4th symposium on Architectural Support for Programming Languages and Operating Systems*, page 11, 94.

[DSW95] N. Drach, A. Seznec, and D. Windheiser. Direct-mapped versus set associative pipelined caches. *Proceedings of PACT'95*, 1995.

[FJ94] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st International Symposium on Computer Architecture*, page 12, 1994.

[FPJ92] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. *Proceedings of Micro-25*, 1992.

[GAFN94] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. In Springer-Verlag, editor, *7th International Conference on Computer Performance Evaluation*, volume 794 of *Lecture notes in computer science*, pages 369–388, May 1994.

[GMB95] Elana D. Granston, Thierry Montaut, and F. Bodin. Loop transformations to prevent false sharing. *International Journal of Parallel Programming*, 1995.

[GZH93] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allcation. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, 28(6):177, July 1993.

[HC89] W-M Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251, May 1989.

[Hei94] R. R> Heisch. Trace-directed program restructuring for aix executables. *IBM Journal of Research and Development*, pages 595–603, September 1994.

[Hil87] M. D. Hill. Aspects of cache memory and instruction buffer performance. Technical Report UCB//CSD-87-381, University of California, Berkeley, 1987.

[Irl92] G. Irlam. "spa" personal communication. the Spa package is available from gordoni@cs.adelaide.edu.au, 1992.

[Jou90] N. P. Jouppi. Improving direct mapped cache performance by the addition of a small fa cache and prefetch buffers. *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[JW94] N. P. Jouppi and S. E. Wilton. Tradeoffs in two level on-chip caching. *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positionning. In *ACM SIGPLAN'90 Conference on Programming Languages and Design Implementation*, pages 16–27, June 1990.

[PK94] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. *Proceedings of the 21st International Symposium on Computer Architecture*, page 10, 1994.

[Por89] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, CRPC, 1989.

[Prz90] S. A. Przybylski. *Cache and Memory Hierarchy Design.* Morgan Kaufmann, 1990.

[Smi82] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi86] A. J. Smith. Bibliography and readings on cpu cache memories and related topics. *Computer Architecture News*, pages 22–42, January 1986.

[Sug93] R. A. Sugumar. *Multi Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs.* PhD thesis, University of Michigan, 1993.

[TGJ93] O. Temam, E. D. Grandston, and W. Jalby. To copy or not to copy: A compile time technique for assessing when data copying should be used to eliminate cache conflicts. *Proceedings of Supercomputing'93*, 1993.

[TLH90] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared data placement optimisation to reduce multiprocessor cache miss rates. *Proceedings of the International Conference on Parallel Processing*, 1990.

[TXD95] J. Torrellas, Chun Xia, and Russel Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *First Symposium on High-Performance Computer Architecture*, page 360369, January 1995.

$NewBlock(), NewLine(), NewLineList()$: Creates a new block, line or list of lines, and resets its *Activity*.
$SizeUsed()$: returns the memory space used up by a given element to store the de(s).
$Correlate()$: Computes the *Correlation* of 2 elements or $\int_t A^W dt$ of an element if the other is $\emptyset$.

```
BlockList DataElementLayoutInBlocks(deList)
    BlockList = ∅      % List of blocks with data-elements layed out
    While deList ≠ ∅
        B = NewBlock()
        BlockList = BlockList ∪ {B}
        While SizeUsed(B) < BlockSize
            Bestde = ∅
            For all de ∈ deList
                If Correlate(de, B) ≥ Correlate(Bestde, B)
                    Bestde = de
                End If
            Loop
            B = B ∪ {Bestde}      % This modifies the Activity of B
            deList = deList − {Bestde}
        Loop
    Loop
return BlockList

LineList BlockLayoutInLines(BlockList)
    LineList = NewLineList()      % List of lists of blocks mapping into the same cache lines
    For i = 1 to CacheLineCount
        L = NewLine()      % Creates a list of blocks mapping to the same cache line
        LineList = LineList ∪ {L}
    Loop
    While BlockList ≠ ∅
        BestBlock = ∅
        For all B ∈ BlockList
            If Correlate(B, LineList) ≥ Correlate(BestBlock, LineList)
                BestBlock = B
            End If
        Loop
        BestLine = ∅
        For all L ∈ LineList
            If Correlate(L, BestBlock) ≤ Correlate(BestLine, BestBlock)
                BestLine = L
            End If
        Loop
        BestLine = BestLine ∪ {BestBlock}      % This modifies the Activity of BestLine and LineList
        BlockList = BlockList − {BestBlock}
    Loop
return LineList

Compute Addresses using LineList de and block layout constraints.
```

Figure 3: Data-layout Heuristic