Implications of Memory Mappings on Cache Misses

Daniela Genius Jörn Eisenbiegler
Institut für Programmstrukturen und Datenorganisation
Zirkel 2
Fakultät für Informatik
Universität Karlsruhe, 76128 Karlsruhe,Germany
e-mail:{genius|eisen}@ipd.info.uni-karlsruhe.de

June 29, 1998

Abstract

This paper proposes an optimization by an alternative approach to memory mapping. Low set associativity allows representing cache lines by corresponding memory areas. With the help of the notion of temporal reuse in the innermost loop, the behaviour of values in the cache is modelled. Combining these values into cache lines so that spatial reuse is considered demands an alternative memory mapping. Memory mappings with a low expectation of conflicts are achieved by the random placement of arrays in memory. Significant increase of cache misses for a worst case placement is shown by experiments, as well as cache miss reduction achieved by improving reuse.

1 Introduction

Scientific applications are particularly sensitive wrt. cache performance as large amounts of data are regularly accessed in nested loops. On the other hand, a lot can be gained if a compiler makes use of this regularity. A source of inefficiency often neglected is the competition for a single physical cache line. This might lead to cache thrashing when in turn data is evicted that will shortly be reloaded into the same cache line. First level caches almost always have low associativity. Typically, two (Intel Pentium) or four sets or —more seldom— direct mappings (DEC Alpha) are implemented. Fully associative caches are expensive and thus seldom used, as vendors prefer economical solutions. When high performance is the issue, the problem to make good use of the cache mostly remains with the programmer. This typically takes a lot of hand-optimization and should better be accomplished by the compiler.

Compiler-controlled optimization techniques exploit compile-time information such as loop boundaries and variable life ranges. Classical compiler optimizations [WL91] [MCT96] work on the model of a fully associative cache. They focus on locality improvement, i.e., they try to prevent data from exceeding cache size. Up to now, the problem of improving behaviour for caches with less flexible mappings has very seldom been tackled. Caches with low set associativity (including direct mapping as an extreme case) have a property that can be used for optimization: They allow, in contrast to fully associative caches, the drawing of conclusions from the memory layout back to cache behaviour as the cache line resp. set a memory location maps to is known. A goal that has also often been neglected is to aggressively achieve so-called spatial locality. Cache lines are often rather large, so that for regular accesses to small data items, as is typical for scientific codes, a constant improvement is achieved.

The present paper proposes a profound change of the memory mapping. A reuse-based strategy is proposed that enables to capture the behaviour of cache lines more exactly than so far. Reuse of data is considered by two stages: firstly, representing life times of values in the cache captures temporal reuse. Secondly, combining such values into cache lines for multiple accesses aggressively improves spatial reuse. The knowledge gained about the location of a virtual cache line in memory allows to detect potentiality for conflicts. The code remains unchanged except index transformations. By experiment, the impact of the optimizations is validated for typical scientific computing benchmarks. Negative effects from cache thrashing reach from ca. 20% to 800%.

Basic terminology can be found in section 2. Section 3 contains the assumptions made. The optimization algorithm is presented in section 4. Results of preliminary measurements are shown in section 5. In section 6, recent related work is summed up and distinguished from the approach taken here. In the final section, an overview is given on directions of further work.

2 Basic Notions

Figure 1 shows three levels of a (possibly larger) memory hierarchy. Low associativity is assumed, i.e. memory areas bearing the same colour map to one cache line (resp. set of cache lines) bearing the corresponding colour (thin arrows)[1]. The fat arrows represent the actions by the compiler. The notion of *cache line*, usually denoting a physical cache line,

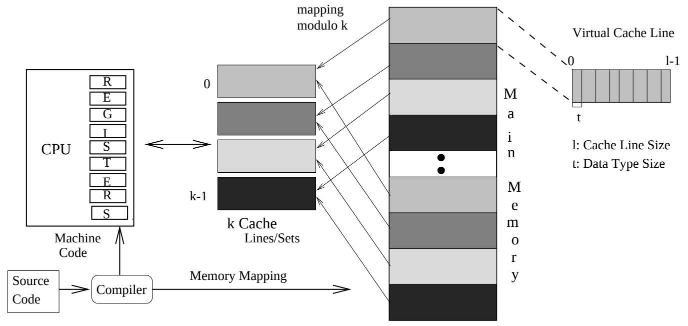

Figure 1: Setting, Terminolgy for Low-associativity Caches

needs to be defined more precisely. A cache consists of k physical cache lines of a length of l bytes each. A variable of t bytes length can be considered a *cache value* in memory. In the context of this paper, variables are array elements. The *cache value life time* denotes the time during which this value is present in the cache. *Virtual cache lines* are composed of several cache values. In caches with low associativity, they can be represented via one of $\frac{m}{k}$ memory locations, where m is the memory size. They are mapped to physical cache lines. Figure 1 sums up these notions. For example, the DEC Alpha first level data cache has a size of 8192 bytes, partitioned into $k = 256$ lines of 32 byte length each. In scientific computing, common types are float and long, i.e., t is 4 resp. 8 bytes on the Alpha's C compiler.

Whenever a value requested in a calculation is present neither in registers nor in the cache, a *cache miss* occurs. *Compulsory misses* occur when filling up an empty cache. *Capacity misses* happen when the data does not fit fully into the cache. *Conflict misses* are due to the competition of memory locations for the same cache line; such misses are specific for direct mapped caches. We generalize this notion to conflicts between sets. In a *direct mapped* cache, the line that is replaced on a miss is determined by a modulo calculation out of the memory adress. *Full associativity* means free choice of cache location, allowing a

[1]In the following, the method is presented for direct mappings, modifications for higher set associativity are mentioned where necessary, otherwise just replace *line* with *set*.

```
for (i=0; i<SIZE; i++){
   for (j=0; j<SIZE; j++){
      for (k=0; k<SIZE; k++){
      c[i][j]+=a[i][k]*b[k][j];
      }
   }
}
```

```
for(i=1; i<SIZE; i++){
   x[k] = u[k]+r*(z[k]+r*y[k])
   +t*(u[k+6]+r*(u[k+5]+r*u[k+4])
   +t*(u[k+3]+r*(u[k+2]+r*u[k+1])))
}
```

Figure 2: Matrix Multiply and Livermore Kernel 7

choice among candidates for replacement (e.g. LRU, FIFO, see [HP96]). In a *set associative* cache, this replacement policy is applied to smaller units, so-called *sets*. As the design of fully associative caches is complex and expensive, mostly a low set associativity or a direct mapping is chosen in practice.

Cache thrashing occurs when a value is required that has just been evicted, in turn evicting a value tha will be required in the near future, etc. On a read miss, a value is loaded into one of the registers, at the same time loading the corresponding memory location and its surrounding values (depending on the line size and alignment) into the cache.

In a loop nest, the main cache pressure stems from accessing arrays depending on the loop indices. The *access pattern* is described by a vector which is given by a matrix J of multipliers and a constant vector $\vec{d}$ of displacements. Assume a loop nest of depth m with the corresponding index vector $\vec{i}$ and a n-dimensional array. For a triply nested loop and a twodimensional array $a_{i_1-i_2+1,2*i_3+2}$, $J = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$ and $\vec{d} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

The general form is $\begin{bmatrix} j_{1,1} & \cdots & j_{1,m} \\ \vdots & & \vdots \\ j_{n,1} & \cdots & j_{n,m} \end{bmatrix} \begin{bmatrix} i_1 \\ \vdots \\ i_m \end{bmatrix} + \begin{bmatrix} d_1 \\ \vdots \\ d_m \end{bmatrix}$. The array indices are thus generated by an *affine mapping* of the loop counter vector.

Consider two loop nests typical in scientific computing: matrix multiply and kernel 7 from the Livermore loops. In the first example, the access pattern is $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$ for c, $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$ for a and $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$ for b. The access patterns for u in the second example are trivially $[1][k] + [0], \ldots, [1][k] + [6]$, as there is only one loop. References differing only in the displacement are *uniformly generated.*

Temporal reuse of data in the cache occurs when the same data item is accessed several times. This was expressed formally by [WL91]: Temporal reuse is in direction $\vec{r_t}$, if $J * \vec{r_t} = \vec{0}$, spanning a vector space. This form of reuse is present for reference $c[i][j]$, where $J = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. The reuse vector space is spanned by $[0, 0, 1]$.

Spatial reuse means accesses to data in the same cache line. Spatial reuse occurs if accesses are made to data that lie in memory sequentially or at least with a distance of less than cache line size $\frac{l}{t}$. All but the innermost index must be identical in order to achieve self-spatial reuse. Let J' be J where all elements of the last row are set to 0, then the self-spatial reuse vectors fulfil $J' * \vec{r_s} = \vec{0}$. Self-temporal implies self spatial reuse.

For c in example 1, $J = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, $J' = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$.

Reuse has been defined wrt. unlimited cache size. Given limited cache size, *locality* denotes the property that no item is evicted that is still required. Reuse is a prerequisite for locality, however note that reuse does not imply locality except for unlimited cache size.

3 Starting Points

For the considerations in this paper, the restriction to perfectly nested loops without branches is made. Furthermore, only affine index functions are considered and indirect addressing is excluded, prerequisites fulfilled in many applications e.g. from scientific computing.

We restrict to array accesses without loop-carried dependencies for simplicity of presentation. Otherwise, data flow analysis for array references will have to be applied as proposed by Feautrier [Fea91]. It is assumed that array sizes are known to the compiler. The focus is on algorithms whose behaviour only depends on the input *size*, a subclass of the large class of *oblivious* algorithms (defined by Löwe and Zimmermann [ZL94]). Exclusively, reuse wrt. the innermost loop is considered, as the greatest effects can be achieved here. We concentrate on data caches. Instruction caches are not considered. Unless loops are unrolled extensively as e.g. in [DJ96], it is legitimate to leave instruction cache behaviour out of the focus.

As noted above, the scientific applications for which the optimizations apply are oblivious and perfectly nested.References in the matrix multiplication algorithm (figure 2) are not uniformly generated.There are three nested loops. In the Livermore kernel, there is only one loop, all references are uniformly generated. None of the running examples contain loop-carried dependencies.

4 Deriving the Memory Mapping

Conventionally, compilers map arrays to memory in the following way. Let there be an n-dimensional array, $size_k$ denoting the number of data items in dimension k. There are m nested loops denoted by $\vec{i}$. Row-major order is assumed. The memory mapping f is a function $f : \mathbb{N}^n \to \mathbb{N}$ of the loop indices:

$$f(i_1, \ldots, i_m) = i_1 * size_2 * \ldots * size_n + i_2 * size_2 * \ldots * size_n \ldots + i_n.$$

Obviously, such an arbitrary mapping of arrays often causes low reuse, e.g. for matrix b in example 1. A more adequate mapping must consider reuse and potentiality for conflicts:

$$\tilde{f}(x_1, \ldots, x_n) = x_1 * size_2 * \ldots * size_n + x_2 * size_2 * \ldots * size_n \ldots + x_n + \delta$$

where δ denotes a displacement of the array in memory[2].

Note that such a mapping is always correct, however may incur different *cost* due to cache miss penalty. If the parameters x_q and δ are chosen adequately, the following two goals are achieved:

1. Data should be combined into cache lines according to memory accesses. This can only be achieved by aligning data in memory (parameter x_q).
2. Data should be mapped to memory (parameter δ). in order to avoid conflicts.

Figure 3 depicts possible conflicts due to memory mapping in the running examples. Arrays a and b resp. vectors u and x aligned to the same memory address modulo cache size k (same colour) may cause *cache thrashing.* In section 5, this effect is shown by experiment.

An overview of the algorithm is given before the steps are applied to the running examples in the following. There are three main requirements for a single loop nest:

1. adequately represent temporal reuse
2. improve spatial reuse by a new memory mapping
3. determine a conflict minimal memory mapping

In the following subsections, the thus decomposed goals will be fulfilled by the steps of the optimization algorithm.

[2]As code is not modified, moving cache lines in order to avoid conflicts is impossible and far too fine-grained.

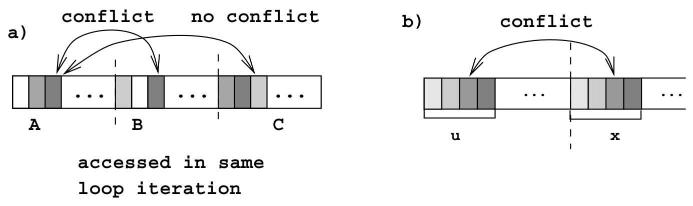

Figure 3: Conflicts due to Memory Mapping: a) Example 1, b) Example 2

4.1 Cache Value Life Times

Temporal reuse has to be respected. To this end, an item should be present in the cache as long as it is still accessed. As a feasible heuristics is of interest, the goal must be to derive a *pattern* dependent of loop indices rather than fully unrolling the loop nest. For this reason, a restriction to the innermost loop is made.

As the register allocation phase is assumed to be completed, physical registers have to be taken into account [Bri92]. Deriving cache value life times must take place *after* locality optimizations because they might influence the instruction execution sequence.

Figure 4 sketches the cases that can occur. Memory accesses are depicted by full dots, purely register operations by empty dots[3]. Number 1) shows the general case, where reuse is present. A remarkable fact is that often data are accessed only *once*, so that the cache value life time can be depicted as a single dot (number 2)). The third case is purely register computation, affecting registers, but not the cache (number 3)). Assume that load

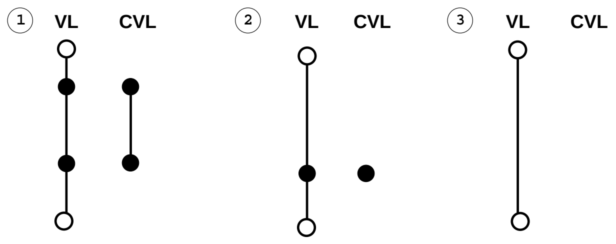

Figure 4: Relating Cache Value Life Times (CVL) to Variable Life Times (VL)

resp. store operations can be extracted on source code level. Given an array assignment, e.g. `a[i]=b[i]+k`, reference to array a on its left hand side is a store operation. The right hand side, which in source code might consist of many operations, usually contains loads from several array locations, here $b[i]$. Cache value life times are a representation of temporal reuse (figure 5). The life times of matrix elements in example 1 are mostly dot-shaped. In matrix multiplication (see section 2), self-temporal reuse wrt. the innermost loop can only be found in Matrix c. Extensive self-temporal reuse of u is present in the Livermore kernel. Cache value life times are chains of length 7. Elements of vectors x,y and z are not reused, their life ranges are dot-shaped.

Summing up, temporal reuse in the innermost loop has been captured by the notion of cache value life time. Now, this notion will be employed in order to derive an exact picture of a cache line.

[3]Note that reading a value from memory after definition of the corresponding variable is not reasonable, whereas it may happen that a value is stored before its last access via register.

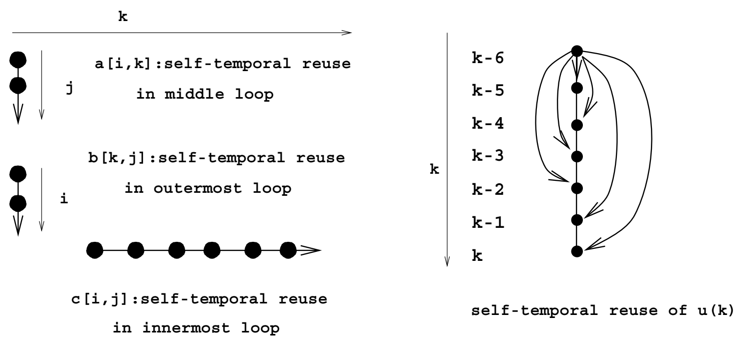
k
a[i,k]:self-temporal reuse
in middle loop
j
b[k,j]:self-temporal reuse
in outermost loop
i
c[i,j]:self-temporal reuse
in innermost loop
k
k-6
k-5
k-4
k-3
k-2
k-1
k
self-temporal reuse of u(k)

Figure 5: Cache Value Life Times for Running Examples - innermost loop unrolled

4.2 Virtual Cache Lines

The next goal is twofold: firstly *fix* the values which belong together into one cache line in order to reduce the degrees of freedom for the following stages and manage complexity of conflict detection. Secondly, enhance spatial locality. Note that the latter improvement is limited by $\frac{l}{t}$, however neglecting this option means giving away valuable chances for improvement.

The first step should fill up cache lines as good as possible. In order to deterministically partition the array into cache lines, the notion of reuse described above is employed.

Spatial reuse is exploited by a modified memory mapping function $\tilde{f} : \mathbb{N}^n \to \mathbb{N}$ if $\vec{r}$ determines the sequence that is placed in memory. A restriction is made to those accesses dependent on the innermost loop index. Reuse is now determined via J and $\tilde{J}$ as described in section 2.

The basic idea is to reflect those loop transformations of [WL91] aiming at the innermost loop by the memory mapping. For the moment, the approach is restricted to transposition and related techniques for simplicity. Transposition can be considered a "loop interchange" concerning just one set of uniformly generated references (especially references to each array of example 1 seperately). This excludes e.g. *blocking* techniques. I.e., only $\vec{i}$ has to be adapted to guarantee that the access pattern remains unchanged, preserving correctness: $\tilde{J} * \tilde{\vec{i}} + \vec{d} = J * \vec{i} + \vec{d}$. For a n-dimensional array and $\tilde{\vec{i}}$,

$$\tilde{f}(\tilde{i}_1, \ldots, \tilde{i}_m) = \tilde{i}_1 * size_2 * \ldots * size_n + \ldots + \tilde{i}_m$$

maps the array elements for improved reuse in the innermost loop.

Array a already has self-spatial reuse of factor $\frac{l}{t}$ in the innermost loop; accesses to c were already examined in section 2. They are independent of the innermost loop index and thus of no interest. For b, $J = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, $J' = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$.

I.e., interchanging loops for b wrt. the standard memory mapping f would create the same reuse behaviour as for a, traversing the rows of an array mapped row-major. Swapping loop indices k and j yields $\tilde{J}$ with $\tilde{J}' = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

$\vec{i}$ has to be adapted: $\tilde{\vec{i}} = \begin{bmatrix} i \\ k \\ j \end{bmatrix}$. The mapping prescribes to map array b columnwise in example 1. The indices k and j have to be swapped for accesses to b in the source code, yielding `c[i][j]+=a[i][k] * b[j][k]`. Note that this effect, also known as *matrix transposition*, cannot be achieved by loop transformations. On the right of figure 5, only the accesses to $\vec{u}$ in the body of Livermore loop 7 are shown. As there is temporal reuse, this implies spatial reuse.

The method presented so far is conservative as it disallows e.g. the merging of arrays[4]. at this point because this complicates the derivation of $\tilde{J}$ and $\tilde{\vec{i}}$.

Memory is virtually subdivided into cache lines, see section 2. Technically however, attention must be payed that any memory space delivered by routines such as `malloc` is aligned to element size only, whereas alignment to cache line boundaries is needed.

Until now, arrays have been considered separately. In the following, the interaction between arrays in memory will have to be considered.

4.3 Cache Line Allocation

The task is now to derive more information on conflicts, avoiding cache thrashing. The number of conflicts has to be derived the as cost for use in an objective function. Let k be an integer so that two memory locations map to the same cache line, i.e. their memory addresses are identical modulo k (the machine dependent number of cache lines in direct mapped cache). the goal is to determine an optimal *displacement* δ for each array in order to minimize conflicts with all other arrays.

By restricting to the innermost loop, a *cyclic* representation of cache values in loop iterations can be derived in analogy to Hendren et. al. [HGAM92]. It should be noted that the problem is easier for cache lines than for registers. For a practical application, cache lines cl are represented by the corresponding memory address modulo k, subscripted by the arrays they stem from. Then, the notion of conflict can be modelled easily. Let a and b be two arrays whose memory mapping has already been determined. In the same iteration i_m of the innermost loop, memory is mapped to the same cache location, if

$$cl_a(i_m) = cl_b(i_m) + c * k \quad \forall c \in I\!N$$

As an array is always represented by one cache line, δ can be chosen as displacement to the array starting address.

It might look like finding an adequate displacement is a very simple task. However, the interaction between all arrays has to be taken into account. Obviously, this approach is rather expensive as the conflict cost minimal combination of δ^{n-1} starting address combinations given n arrays has to be found. Experimental results will show (section 5) that randomization can be employed here.

5 Experimental Results

The DEC Alpha 21064 memory hierarchy is well documented. Measurements are made on this architecture, whose first level data cache is direct mapped. The specific analysis tool ATOM [SE94] allows specifying (e.g. cache) analyses in an elegant and flexible way. The tool allows analysis of C code on the DEC Alpha architecture by instrumenting binaries. It is used in order to get a realistic picture of cache behaviour. In the following, some preliminary measurements for a selection of scientific applications are presented. As already mentioned in section 2, there are $\delta = 256$ cache lines of $l = 32$ bytes each. The data type size t is 4 for a floating point value[5]. To give a clear picture, the absolute number of references resp. misses for the loop nest in question is included. Run times denote user times in *msec* of a mean of 100 runs. Weak optimization means optimization with respect to cache line combination, while memory is allocated more or less randomly. The worst case of strong optimization,

[4]Preliminary measurements not contained in the present paper have shown that merging arrays indeed has potentiality for some loop nests.

[5]In order to capture the miss rate of the loop nest in question only, the number of references as well as cache misses incurred by the rest of the program is subtracted from the total numbers. As the loop nests are at the end of the program code, the results do not differ significantly from those obtained by modifying the ATOM cache tool to measure the loop nest only.

problem size	optimization	references	misses	miss rate	time/ms
64*64	none	2097154	327232	0.156036	1235
64*64	weak	2097154	97718	0.046596	1219
64*64	thrashing	1972670	125164	0.063449	1273
55*55	none	1339998	33585	0.025062	775
55*55	weak	1339998	25399	0.018956	772
55*55	thrashing	1331002	72088	0.054161	779

Table 1: Evaluating Optimizations for Example 1

program	size	optimization	references	misses	miss rate	time/ms
LL 7	1000	none	17878	465	0.026010	15
LL 7	1000	thrashing	17000	3007	0.17688235	16
LL 18	1000*6	none	204816	9137	0.044611	356
LL 18	1000*6	thrashing	204416	29116	0.142157	366
LL 23	1000*7	none	119885	6086	0.050765	60
LL 23	1000*7	thrashing	119861	26859	0.224085	64
filter	1000*1000	none	9000002	230659	0.0256211	4583
filter	1000*1000	thrashing	8999994	230659	0.0256288	4587

Table 2: Conflict Miss Reduction: Livermore Kernels, Filter

mapping all arrays' starting addresses to the same cache line, is shown in the rows labelled "cache thrashing". The different number of references is due to alignment.

A comparison of different methods of optimizing matrix multiplication is shown in table 1. Mapping $\tilde{f}$ (i.e. the combination into cache lines) yields a transposition of matrix b, decreasing cache misses significantly. Aligning, in this case, reduces the miss rate by about two thirds. Provoking cache thrashing by aligning all arrays modulo $256 * \frac{32}{4} = 2048$ has little effect here. A matrix size of 55 was additionally examined, because the usual negative effects of matrix sizes of a power of 2 [PNDN97] have to be excluded. Surprisingly, cache thrashing effects are even more significant here. The size of the examples was chosen in order not to exceed the second level cache. Embedding the blocking algorithms of [WL91] into our framework is not too difficult and should achieve better results.

Table 2 shows results for the Livermore benchmark set and a filter loop nest typical for image processing. As already mentioned, the combination of values into cache lines has no effect on the memory mapping here. The dramatic increase in cache misses in the Livermore kernels after alignment modulo 2048 are due to cache thrashing.

The increase in cache misses incurred by strong cache alignment is significant. This indicates that a certain amount of "disorder" is desirable in order to avoid cache line conflicts. Cache thrashing effects are stronger when life times are longer, as is the case in the Livermore kernels. As there are many good and a few very bad choices of δ, the expectation value for conflicts is low. Thus, a randomized choice of displacements is very promising [MR95], which is confirmed by the good results for the weakly optimized case. Secondly, run time improvements fall a bit short behind expectations. This is due to the fact that cache misses have no effect on run time in pipelined processors with separate functional units unless the floating point pipeline is not completely filled. Otherwise, the actions overlap. The classes of problems for which the method is applicable with significant effect on run time will have to be determined.

6 Related Work

Classical approaches to cache optimization, aiming at the reduction of cache capacity misses by improving *locality*, can e.g. be found, as noted above, in [WL91] and in the work of McKinley et.al. [MCT96]. In contrast, reuse information is directly used to control the memory mapping here. By restricting to the innermost loop, compile-time complexity is significantly lower than theirs. Moreover, strictly speaking, the notion of *locality* applies to fully associative caches only. The more realistic case of low-associativity caches is not considered there. The combination of values into cache lines was examined in the context of cache analysis by Rawat [Raw93]. By not taking reuse information into consideration, the estimations made by Rawat's method are too coarse and often overestimate cache misses significantly. Panda, Nicolau et. al. [PNDN97] show a simple but striking approach to the problem of conflict misses in data caches which might provide an alternative approach to conflicts between arrays. Hashemi, Kaeli and Calder [HKC97] very recently applied *cache line coloring* for direct mapped instruction caches in order to obtain conflict-minimal mappings for procedures. Their optimizations are based trace-driven simulation and validated for the SPEC95 benchmark suite. In the approach presented here, by restricting to scientific applications more information can be obtained at compile time. Furthermore, data caches are challenging due to varying access patterns and huge amount of data.

7 Conclusions and Future Work

By exploiting the mapping properties of low-associativity caches, a structured approach to cache optimization is presented. Accounting for a composition of cache values to cache lines that respect temporal and spatial reuse a method for deriving conflict-minimal memory mappings is proposed. Using a framework from locality optimization enables to make some of the transformations available for memory mapping. More complicated mapping functions will have to be examined. The optimization presented here is complementary to classical loop transformations. Measurements document the flexiblity of the approach, which can be applied to a large class of scientific programs. Expectations wrt. negative effects of over-alignment due to cache thrashing have been confirmed.

However, there still are a lot of potentialities for further optimization. In the following, only the most important options are mentioned.

By now the source code is hand-optimized. The next step will be to embed the scheme into an experimental compiler. Interaction with other compiler phases, such as register allocation and instruction scheduling, has to be considered.

For the moment, each loop nest is analyzed separately. Arrays may be accessed in different ways in several loop nests that are part of a program. They may be replicated with respect to different access patterns. However, care must be taken not to exceed main memory size, which would occur when large arrays are considered. Alternatively, arrays would have to be remapped at run time, which is an expensive action. By choosing between alternative parameter settings, Eisenbiegler has specialized the cost directed configuration approach described by Moldenhauer [Mol97] for the purpose of compiler-supported data distribution for multiprocessors[Eis96]. Currently, a method to support the choice between remapping and replication is developed by us in analogy to the latter method. Data dependencies between array references have to be taken into account.

Finally, innermost loops are not specific to the area of scientific computing. In more general areas of application, data type sizes vary and structures of access are much less regular. It in an open question whether the method can be generalized.

Acknowledgement We wish to thank Uwe Assmann, Thilo Gaul, Gerhard Goos and Sylvain Lelait for productive discussions. Furthermore, we thank the authors of ATOM, Amitabh Srivastava and Alan Eustace for providing their tool.

References

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[DJ96] Jack W. Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable, optimizing compiler. In *Compiler Construction*, volume 1060 of *LNCS*, pages 59–73, April 1996.

[Eis96] Jörn Eisenbiegler. Datenverteilung als Konfigurationsproblem (data distribution as configuration problem). Technical Report 1996-22, Universität Karlsruhe (TH), Fakultät für Informatik, May 1996.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[HGAM92] L. Hendren, G. Gao, E. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *Proc. 4th Int. Conf. Compiler Construction*, volume 641 of *LNCS*, pages ?–? Springer-Verlag, 1992.

[HKC97] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *PLDI 1997*, pages 171–182, jun 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufman, 2nd edition, 1996.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[Mol97] Horst Moldenhauer. *Kostenbasierte Konfigurierung für Programme und SW-Architekturen (cost-based configuration of programs and software architectures).* PhD thesis, University of Karlsruhe (TH), June 1997.

[MR95] Rajeev Motwani and Praphakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[PNDN97] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and A. Nicolau. Improving cache performance through tiling and data alignment. In *IRREGULAR 1997*, pages 167–185. Springer LNCS 1253, 1997.

[Raw93] Jai Rawat. Static analysis of cache performance for real-time programming. Technical Report IASTATECS//TR93-19, Iowa state university, November 19 1993.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, jun 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[ZL94] Wolf Zimmermann and Welf Löwe. An approach to machine-independent parallel programming. In *VAPP: CONPAR 90–VAPP IV: Joint International Conference on Vector and Parallel Processing.* LNCS 457, Springer-Verlag, 1994.