Improving Data Layout through Coloring-Directed Array Merging

Daniela Genius
Institut für Programmstrukturen und Datenorganisation,
Fakultät für Informatik Universität Karlsruhe
Zirkel 2, 76128 Karlsruhe, Germany
E-mail: genius@ipd.info.uni-karlsruhe.de,
WWW: http://i44www.info.uni-karlsruhe.de/~genius
Tel.: (+49) 721 608-4763
Fax: (+49) 721 30 0 47

Sylvain Lelait
Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
E-mail: sylvain@complang.tuwien.ac.at
WWW: http://www-rocq.inria.fr/~lelait
Tel.:(+43) 1 588 01 185 11
Fax: (+43) 1 588 01 185 98

January 29, 1998

Abstract

Scientific computing and image processing applications access large amounts of data in regular patterns. In order to relieve the memory bottleneck, caching tries to keep recently referenced data available in fast storage. This is increasingly important as the gap between processor and memory hierarchy speed has widened in recent years.

There are two main difficulties that cannot be dealt with by hardware alone. Firstly, a cache line usually holds several values; often only one of them is actually used. Secondly, conflicting accesses to one cache line cause data to be evicted which is still required. In the extreme case, data is replaced on every access, a situation we denote as cache thrashing. To overcome these problems, the temporal/spatial structure of accesses has to be changed.

Compile-time cache optimizations exploit regular access patterns. Loop transformations as e.g. tiling are well-established. For caches with limited associativity, it is often crucial to additionally adjust the placement of data in memory. We show that compiler techniques for register allocation, namely graph coloring, support a systematic data placement.

For innermost loops, conflicts and temporal reuse can be modeled together in a cyclic interval graph. If reuse stretches over several loop iterations, live ranges may overlap themselves, prohibiting usual cyclic coloring. By applying the meeting graph method, the compiler can determine an unrolling factor and determine the maximal number of colors, i.e. of cache lines required.

Values of the same color are mapped to memory together. Since these values may stem from different data structures, our technique offers a natural way of dealing with conflicts between different arrays.

We implement this scheme through modifying the standard memory mapping. At run time, the new compile time mapping function is used as index function. The tradeoff between the additional cost for more complex indexing and reduced miss penalty is reflected by a cost function.

For typical example codes from the above areas, reuse and conflict behavior are considerably improved, yielding moderate run time reductions. In addition, on a more coarse level, paging activity is often significantly postponed.

1 Introduction

Scientific programs cause large numbers of cache misses due to competition for a cache line as well as bad cache line utilization. Although to some extent the hardware can be improved, e.g. by increasing cache line size or associativity, good cache utilization for high-performance codes often remains a task for hand-optimization. There are though a number of approaches to having the compiler improve cache behavior. For instance, Temam and McKinley have shown that for typical loop nests, conflict misses account for up to 50% of all cache misses [MT96]. For caches with limited associativity, it is possible to influence cache behavior via the memory layout. But alternative data layouts have rarely been considered, although their importance has been fully recognized by now [RT98].

An analysis of memory accesses in innermost loops informs us which portions of an array must be present in the cache. Interference and temporal reuse can then both modeled by a cyclic interval graph. In order to determine potential conflicts and to find out which data should be loaded into the same cache line, we apply some heuristics from register allocation. As array live ranges in the cache are similar to variable lifetimes handled in loops by cyclic graph coloring methods, we take advantage of this formalization. The obtained coloring gives us information for the occupation of the cache lines by the arrays in order to reduce the cache misses. Then this allow us to deduce a new data layout for these arrays in memory. In this framework, cache misses are reduced in two ways: a better utilization of the cache lines reduces capacity problems, while also many conflicts are avoided. Our experimental results show improvements concerning cache miss rates and also run times improvements for a set of benchmarks typical of scientific code.

Section 3 contains basic terminology and prerequisites, while Section 2 sums up recent related work. We show in Section 4 how cache value live ranges are derived and describe in Section 5 how the meeting graph is applied. Section 6 shows the application of the result to actual memory layout. Results of measurements can be found in Section 7. We conclude by outlining further work.

2 Related work

Graph coloring wrt. caches was examined in the context of cache analysis by Rawat [Raw93]. He contributes the notion of *togetherness* of values in a cache line. Coloring as a heuristic is more appropriate for optimization. By not taking reuse information into account, the estimations made by Rawat overestimate cache misses significantly. Hashemi, Kaeli and Calder [HKC97] applied a coloring technique for direct mapped instruction caches in order to obtain conflict-minimal mappings for procedures. For this area coloring applies more easily. Their opti-

mizations are based on trace-driven simulations. Restricting to scientific applications, we can obtain more information at compile time.

There are still few works on the impact of data layout on cache misses, all of very recent origin. They are mostly dealing with *padding*, the insertion of useless data into data structures. Rivera and Tseng [RT98] classify padding more precisely and provide comprehensive experimental results. Panda et.al. [PNDN97] propose to combine padding with a tiling scheme. Their run time improvements are very small due to operation overhead. Ghosh et al. [GMM97] use cache miss equations to select padding and tile size however they do not provide much experimental data. The gaps filled with useless data that are characteristic for padding are no longer required. Kandemir et.al. [KCR+98] target exclusively at spatial locality.

Merging is a simple but effective way to prevent the arrays from interfering with each other, usually performed by the programmer [HP96, LW94]. The method we present enables the compiler to intermix arrays in a systematic way that goes beyond merging. We have already employed the meeting graph in previous work [Gen98], where we proposed a general way of deriving data layouts from array indexes. In this approach, coloring was performed *after* building cache lines, it was not used as a togetherness criterion.

3 Basic notions

3.1 Cache Terminology

For the basics of cache architecture we refer to [HP96]. Whenever a value requested in a calculation is present neither in registers nor in the cache, a *cache miss* occurs: A value is loaded into one of the registers. At the same time the corresponding memory location and its surrounding values are loaded into a (physical) cache line. *Compulsory misses* occur when filling up an empty cache. When the size of a portion of data that is required to be in the cache exceeds cache size, *capacity misses* are caused. *Conflict misses* are due to the competition of memory locations for the same cache line; they do not occur in fully associative caches. *Self interference* denotes conflicts that are caused by accesses to the same array. *Cross interference* occurs when different arrays compete for a cache line. A comprehensive analysis can be found in [TFJ94].

By *cache thrashing* we denote the situation when on every iteration of the innermost loop, references cause data to be evicted from the cache. This notion corresponds to *severe conflicts* in [RT98].*Temporal reuse* of data in the cache occurs when the same data item is accessed several times. *Spatial reuse* means accesses to data in the same cache line. Temporal reuse is a special case of spatial reuse.

3.2 Prerequisites

Our primary application area is scientific computing, where nested loops with regular accesses to large portions of memory have to be handled. The goal is to extract as much information as possible at compile time, so the following restrictions have to be imposed. We consider such array references that are generated by an *affine mapping* of the loop counter vector [WL91]. One loop nest without branches is considered at a time. We focus on the improvement of innermost loops as the greatest effects can be achieved here. We concentrate on first-level data caches with limited associativity. Unless loops are unrolled extensively, it is legitimate to leave the instruction cache behavior out of the focus.

We present our method for a Low Intermediate Representation (LIR) for a RISC architecture, which means that arithmetic operations are performed only on registers. The method is applicable for any architecture without out-of-order execution. Furthermore we denote memory addresses by their corresponding array locations: `load(a[i])` stands for a load from a memory address calculated from i.

The running examples satisfy the code properties stated above. For presenting our method we only show the interesting loop nests, whereas we use the entire programs for measurement.

Example 1 (Benchmarks)
Consider some main loops from scientific benchmarks. In the Livermore kernel of Figure 1.a, there is only one loop; references to all four arrays are uniformly generated, values of vector `u` are heavily reused. Matrix multiply, shown in Figure 1.b, contains accesses to three arrays. In a preliminary step, matrix `b` is transposed for improving spatial reuse. A typical application taken from image processing is *filtering*, presented in Figure 1.c, which exhibits a bad cache miss behavior because values from former/future iterations of the outer loop are reused. Images are usually rather big, so these values will be evicted before they are reused. Fast Fourier Transform, shown in Figure 1.d, also suffers from the problem that from the $\log(N-1)$th iteration on a value will be evicted before it is reused, as it is pointed out in [PNDN97]. The additional problem here is that problem sizes are always a power of two, which is pathological when dealing with caches.

4 Extracting Cache Behavior from the Code

A *physical cache line* is represented in memory as a *virtual cache line* consisting of *cache values* of data item size. The number of processor cycles in which a value is actually present in the cache is called the *cache value live range* [Gen98]. From the RISC-LIR representation of the innermost loop body, load and store instructions can be easily extracted.

a) LL7

```
for(k=1; k<SIZE; k++){
   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])))
}
```

b) Matrix Multiply

```
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[j][k];
    }
  }
}
```

c) Filter

```
for ( k=1 ; k<SIZE ; k++ ){
  for ( j=1 ; j<SIZE ; j++ ){
      b[i][j]=0.25*
         (a[i-1][j]+a[i][j-1]
         +a[i+1][j]+a[i][j+1])
  }
}
```

d) FFT

```
le=2^n; windex=1; wptrind=0;
for(l=0;l<n;l++){
   le=le/2;
   for(j=0;j<le;j++){
      wpr=wreal[wptrind];
      wpi=wimag[wptrind];
      for(i=j;i<2^n;i+=2*le){
         tmpr=sigreal[i];
         sigreal[i]+=sigreal[i+le];
         tmpi=sigimag[i];
         sigimag[i]+=sigimag[i+le];
         tr=tmpr-sigreal[i+le];
         ti=tmpi-sigimag[i+le];
         sigreal[i+le]=tr*wpr-ti*wpi;
         sigimag[i+le]=tr*wpi-ti*wpr;
      }
     wptrind+=windex;
   }
   windex=windex*2;
}
```

Figure 1: Benchmarks

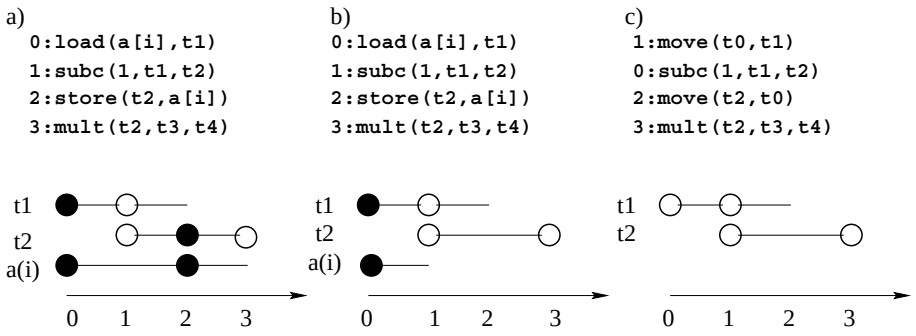


Figure 2: Deriving cache value live ranges

Assume two short sequences of LIR code and consider the content of *a(i)*. Figure 2 relates cache value live ranges (*a(i)*) to register live ranges (*t3,t4*): while the latter refer to registers, the former refer to cache — and thus memory — locations when dealing with limited associativity. Memory accesses are depicted by full dots, register operations by empty dots[1]. Figure 2.a shows that *a(i)* is reused, although it is kept in different registers and its content is changed. If data are accessed only *once*, we consider them as surviving one cycle. This is the case if only loads are considered like in Figure 2.b. When the value is loaded from a register *t0* rather than a memory location *a(i)*, register-only computation does not affect the cache as shown in Figure 2.c.

Example 2
Values of vector u in the Livermore kernel LL7 of Figure 5.a are heavily reused. *u(k+6)* is first used in iteration k, last used in iteration *k+6*; live ranges last 7 cycles. Accesses to x,y and z last one cycle.

Example 3
In the matrix example of Figure 6.a, values of a and b live one cycle, while values of c stretch over all iterations. To achieve a form suitable for our representation, scalar expansion is applied. The loop body now has the form *c(i,j,k)=c(i,j,k-1)+(a(i,k)*b(k,j))*, so that lifetimes for `c` stretch over four cycles.

Example 4
In the filter example shown in Figure 7.a, elements are accessed twice in the innermost loop (i.e. *a(i,j-1)* and *a(i,j+1)*). Live ranges survive three iterations, e.g. from iteration *j-1* to iteration *j+1*. Reuse in direction of i is unlikely because after the array has been fully traversed by the innermost loop in direction of j at "level" i, values of "level" *i-1* will have been evicted.

[1]Note that this notions can be easily transferred to non-RISC architectures: any memory-affecting instruction is considered a load or store.

Example 5
Fast Fourier Transform (FFT, Figure 8.a) suffers similarly from the problem that from the $n-1$th iteration on a value will be evicted before it is reused, as was pointed out by [PNDN97]. The additional problem with FFT is that problem sizes are always a power of two, a pathological situation causing a lot of cache conflicts. In order to overcome the problem caused by the outer loop which makes every array location alive most of the time, we unroll it completely and we execute in fact n times the loop nest composed of the two other loops.

4.1 A Togetherness Criterion

Cache lines usually hold more than one element. In which way can we control which data is placed in a physical cache line? That is, how is *togetherness* achieved?

Intra-array regrouping, *vertical grouping*, is done by displacing cache line sized pieces of data in memory via an injective function that redistributes cache line sized pieces of data in order to reduce interferences [Gen98]. The main disadvantages are that arrays have to be mapped wrt. spatial reuse in a preliminary phase. As arrays stay separate, cross interference has to be dealt with separately.

On the other hand, it is possible to have a cyclic interval graph handling also togetherness. By chaining together live ranges, *horizontal grouping*, we determine which elements should be together in a cache line. In order to reduce the degrees of freedom we always assume that the order in memory is the same as the sequence of lifetimes.

Figure 3 shows the dilemma for two arrays; the upper part shows vertical, the lower part horizontal grouping. In the following, choosing the second approach allows us to formulate the following togetherness criterion:

Claim 1
Togetherness is given for m lifetimes as a result of cyclic graph coloring: if they share the same color, they are grouped horizontally.

5 Coloring the Cyclic Interval Graph

We have now achieved live ranges of the form required by cyclic interval graph coloring algorithms. A *variable or register live range* originally denotes the period during which a certain value is present in a register during the program execution. Finding a conflict layout for possibly infinitely many live ranges onto a limited number of resources is also the aim of register allocation techniques like [Cha82] (Figure 4.b): given k registers, an undirected graph is checked for k-colorability, where nodes and edges stand for live ranges and interferences, respectively. The small example of Figure 4.a, taken from [HGAM92], shows in Figure 4.c how live

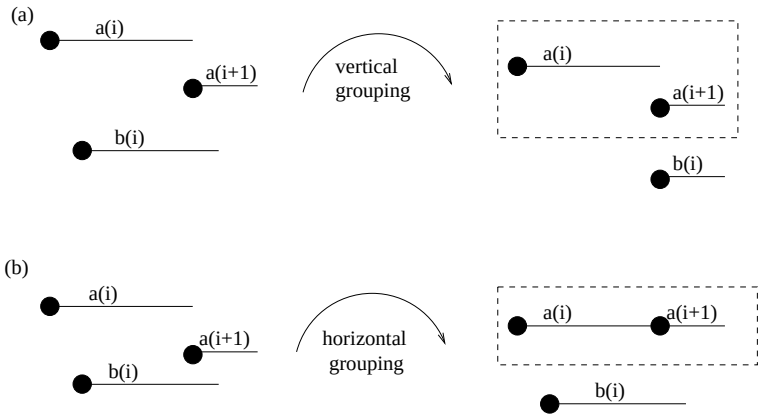


Figure 3: The grouping dilemma

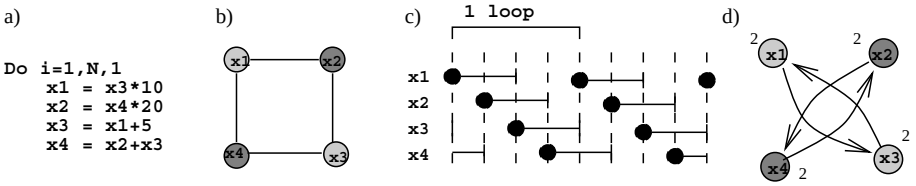


Figure 4: Cyclic interval graph/meeting graph

ranges in innermost loops can be represented by a *cyclic interval family*, which gives the *cyclic interval graph* of Figure 4.b. On the other hand, the intention of the *meeting graph* method [ELM95] is to handle loop unrolling and register allocation simultaneously for live ranges lasting more than one iteration of the loop.

5.1 Usual Cyclic Interval Graph Coloring

Cyclic interval graph coloring, also known as circular-arc graph coloring, is a polynomial problem [GJMP80]. But the q-coloring algorithm is only practical for small values of q, therefore heuristics are used. One of the most efficient is the one introduced in [HGAM92]. They try to minimize the number of colors used. The lower bound on the number of colors is the maximum number of live ranges overlapping a point in the interval family, often noted $MaxLive$. For instance in Figure 4.c, $MaxLive$ is equal to 2. Unfortunately they can not handle intervals overlapping themselves. In this case a step of unrolling is necessary to have a set of intervals not overlapping themselves. Furthermore this method does not guarantee a coloring with $MaxLive$ colors, even if it is usually rather efficient.

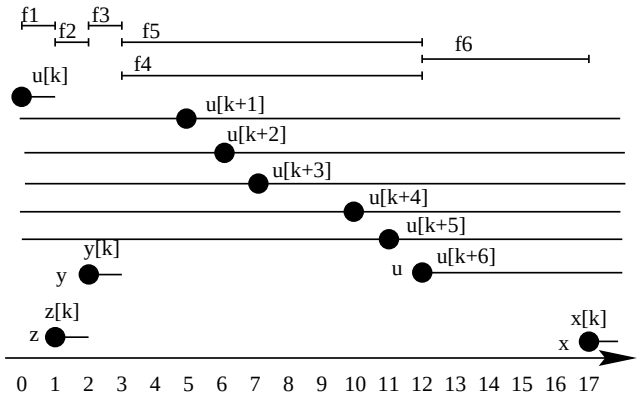
5.2 Meeting Graph

With this method, intervals overlapping themselves can be handled. The meeting graph heuristic was designed to minimize the unrolling degree necessary to obtain a family of intervals that can be colored with $MaxLive$ colors. It only works for families of constant width, therefore some fictitious intervals may be added. Then the graph itself is built like in Figure 4.c. We have one node for each interval, and an arc between a node i and a node j if the interval corresponding to i ends at the point where the interval corresponding to j begins. A weight w corresponding to the length of the interval is added to each node. Similarly each circuit has a weight equals to the sum of the weights of its nodes divided by the number of cycles of the loop. To find a suitable unrolling degree one has to take the lcm of the weights of disjoint circuits which can be found in the graph, hence the weights of its connected components if it is not decomposed. Several heuristics have been proposed to find a decomposition which minimizes the lcm [ELM95]. In all cases finding a decomposition is a polynomial problem and the coloring with $MaxLive$ colors is ensured. The optimal unrolling degree is the least lcm computed over all the possible decompositions, finding it is an exponential problem.

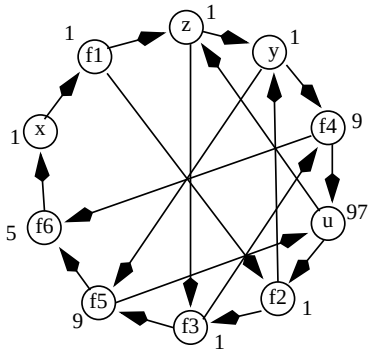
In our context, the intervals depict the time that a value must stay in the cache. The meeting graph is used to find an unrolling degree, and a coloring of the cyclic interval graph. A decomposition is computed, the lcm of the circuits weights gives the unrolling degree. An unrolling degree equal to 1 means that no unrolling is necessary. Then the sequence of the intervals in the circuits gives the order of the arrays in the cache lines. Furthermore minimizing the number of colors is equivalent to minimize the number of cache lines used. In the following, we apply this method on the running examples.

Example 6 (Example 2 continued)
Figure 5.b shows the meeting graph of Livermore Loop 7 which can be derived easily from the cyclic interval family of Figure 5.a. Each live range is represented by a node labeled with the number of cycles it is required to be in the cache. As $MaxLive = 7$ during only the last cycle, we must add fictitious intervals, 1 during cycles 0-2 and 12-16, and 2 during cycles 3-11. As no interval begins or ends during cycles 4-10 and 13-15, the unit intervals were merged to build $f4$,$f5$ and $f6$. This does not change the results, but allows to have less nodes in the graph. Thus, per iteration, six fictitious intervals have to be inserted. Figure 5.c shows one possible decomposition, yielding a circuit of weight 1 and one of weight 6. An unrolling factor of $lcm(1,6) = 6$ is required to prevent u from overlapping itself. The interval graph is unrolled six times and colored according to the result: six colors are required for the live ranges of u, one additional color for those belonging to the other arrays.

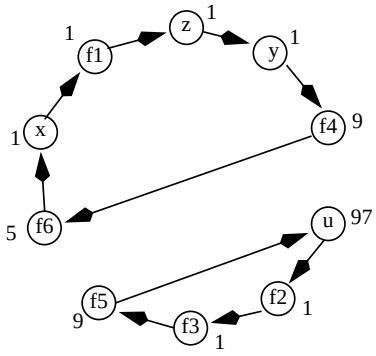
```
 0:load(u[k],t1)
 1:load(z[k],t2)
 2:load(y[k],t3)
 3:add(t2,t3,t4)
 4:add(t1,t4,t5)
 5:load(u[k+1],t6)
 6:load(u[k+2],t7)
 7:load(u[k+3],t8)
 8:add(t7,t8,t9)
 9:add(t5,t9,t10)
10:load(u[k+4],t11)
11:load(u[k+5],t12)
12:load(u[k+6],t13)
13:add(t12,t13,t14)
14:add(t11,t14,t15)
15:add(t15,t10,t16)
16:add(t16,t4,t17)
17:store(t17,x[k])
```



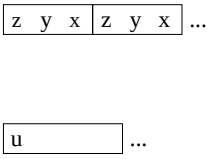
(a)



(b)



(c)



(d)

Figure 5: Coloring Livermore loop 7

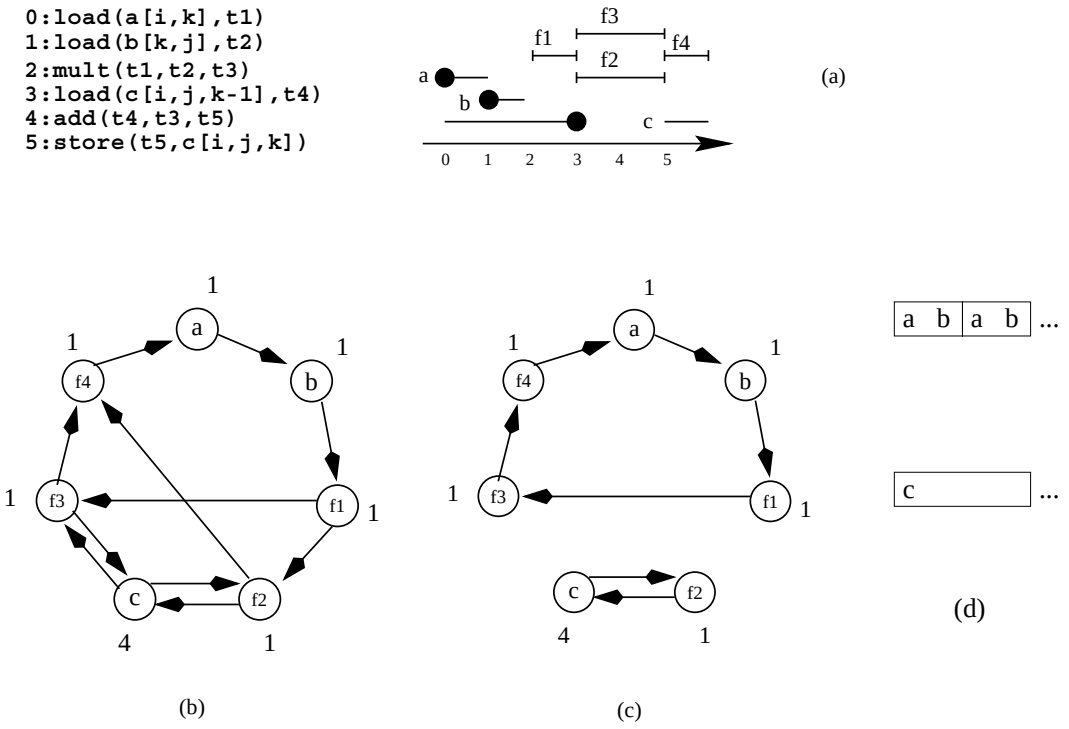


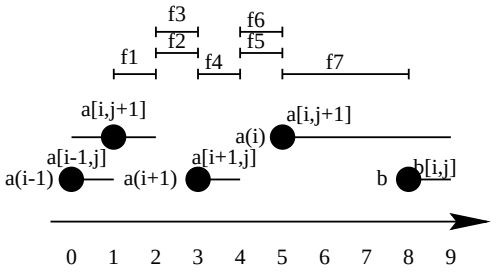
Figure 6: Coloring matrix multiply

Example 7 (Example 3 continued)
Consider again matrix multiply of Figure 6.a. Fictitious intervals $f1$ to $f4$ have to be inserted in order to derive the meeting graph. Figure 6.b shows the corresponding meeting graph. It has two connected components, each with a weight equal to 1: $\frac{1+1+3}{5}$ and $\frac{5}{5}$ respectively as shown in Figure 6.c. The cyclic interval graph can thus be colored with two colors without unrolling.

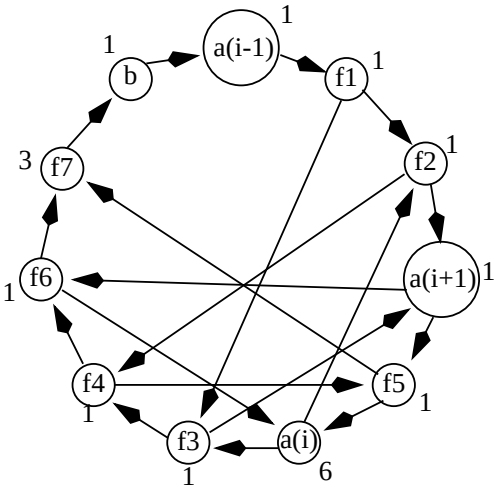
Example 8 (Example 4 continued)
The meeting graph for the filter example is shown in Figure 7.b. Because of the more complex access pattern, seven fictitious intervals are required. It can be decomposed into two circuits as shown in Figure 7.c, each with a weight of 1 ($\frac{9}{9}$). Thus we don't need to unroll the cyclic interval graph in order to color it with two colors.

Example 9 (Example 5 continued)
For each loop nest of the FFT example, we get the interval representation of Figure 8.a. Figure 8.b presents the corresponding meeting graph. It can be decomposed in 2 circuits as shown in Figure 8.c.

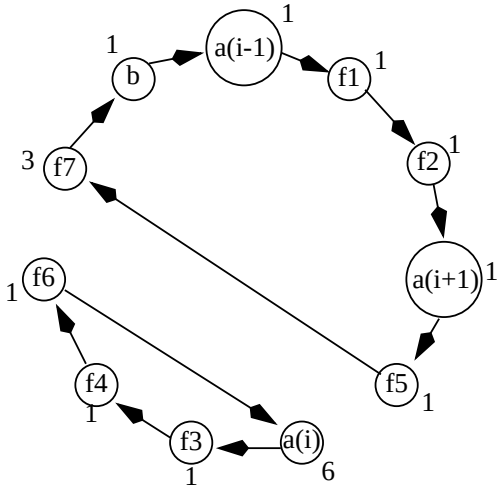
```
0: load (a[i-1,j],t1)
1: load (a[i,j-1],t2)
2: add (t1,t2,t3)
3: load (a[i+1,j],t4)
4: add (t3,t4,t5)
5: load (a[i,j+1],t6)
6: add (t5,t6,t7)
7: mulc (t7,0.25,t8)
8: store (t8,b[i,j])
```



(a)



(b)



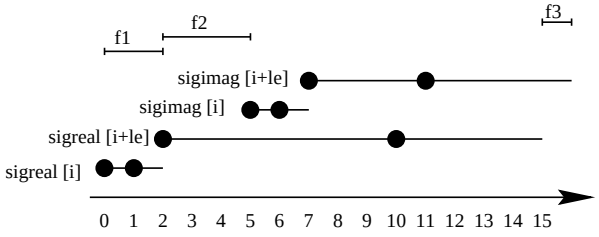
(c)

a(i-1)	a(i+1)	b
a(i)		
a(i-1)	a(i+1)	b
a(i)		

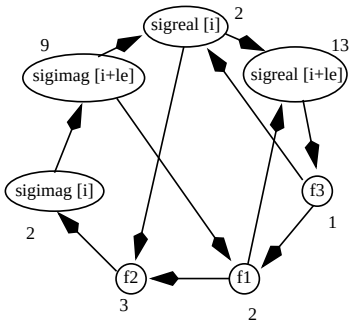
(d)

Figure 7: Coloring Filter

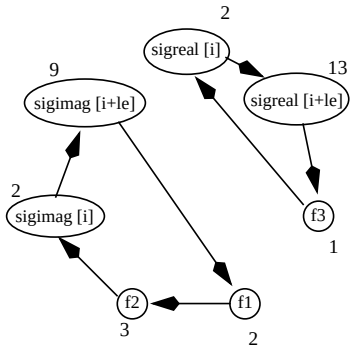
```
 0:load(sigreal[i],t1)
 1:load(sigreal[i],t2)
 2:load(sigreal[i+le],t3)
 3:add(t2,t3,t4)
 4:store(t4,sigreal[i])
 5:load(sigimag[i],t5)
 6:load(sigimag[i],t6)
 7:load(sigimag[i+le],t7)
 8:add(t6,t7,t8)
 9:store(t8,sigimag[i])
10:load(sigreal[i+le],t9)
11:load(sigimag[i+le],t10)
12:fr(t9,t10,t12)
13:fi(t10,t9,t13)
14:store(sigreal[i+le])
15:store(sigimag[i+le])
```



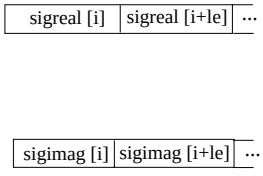
(a)



(b)



(c)



(d)

Figure 8: Coloring FFT

6 Determining a new Data Layout

To avoid a table that maps each element separately to its new location, which incurs high run time cost for indexing and additional cache misses, the new location should be calculated from the position within the original array [Gen98].

6.1 Standard Data Layout

C compilers allocate contiguous memory space more or less arbitrarily from an arbitrary starting address *start*. Let a_i be an n-dimensional array. The d_i denote the offsets in direction of the array dimensions, $size_j$ the boundary of d_j (e.g. column length). C maps arrays row major, FORTRAN column major. In the following, we consider a *linearized* C layout. A standard *data layout dl* for one array is a function $dl : \mathbb{N}^n \to \mathbb{N}$:

$$dl(d_1, \ldots, d_n) = d_1 * size_2 * \ldots * size_n + d_2 * size_3 * \ldots * size_n + \ldots + d_n + start.$$

6.2 Deriving a Coloring-directed Data Layout

In the general definition, a compiler-directed data layout can follow any injective function. Horizontal grouping now results in systematically intermixing arrays. Assume that p arrays are accessed in a loop nest which was unrolled and colored with $k' \leq k$ colors $c_1, \ldots, c_{k'}$. Live ranges that have been identified as belonging to the same color c_i stem from $p_i \leq p$ different arrays. These arrays have to be combined in a *mergeset* m_i. In the following, we describe the transformations for one array a_i from one such mergeset.

While cyclic interval graph coloring handles only the innermost loop, the resulting layout affects all dimensions of an array that are accessed through the innermost loop index i_n. In a first step, arrays are transposed so that the innermost loop variable i_n accesses dimension d_j of the array[2]. Let $\pi : \mathbb{N}^n \to \mathbb{N}^n$ be a *permutation* such that $\pi(d_1, \ldots, d_j, \ldots, d_n) = (d_1, \ldots, d_n, d_j)$. If nothing is to be done, π is the identity permutation.

In a second step, data items of size s are mixed. They stem either from one or from several different arrays. The number s of consecutive items from one array ranges from 1 over row size to arrays size, representing elementwise, piecewise, rowwise merging and no merging at all, respectively. $s = 1$ is the usual case. For the jth array a_j of a mergeset, let d_n be the offset in direction of the innermost loop index. The position j of the array in the mergeset determines which of the $p_i * s$ blocks is addressed, $d_n \bmod (p_i * s)$ yields the offset inside such a block. $merge : \mathbb{N} \to \mathbb{N}$ affects only the "innermost" dimension:

$$merge_{s,j}(d_n) = (p_i * s) * ((d_n * p_i) \; div \, s) + (j - 1) + (d_n \bmod (p_i * s))$$

[2]Note that in contrast to loop transformations [WL91], d_j is modified instead of the loop index vector $(i_1, \ldots, i_n)^T$.

In a third step, we add starting address adjustments for the $c_{k'}$ different mergesets. To avoid cache thrashing, which is now a greater risk because arrays are accessed side-by-side, it is simply required that they do not start at memory addresses that are equal modulo the total number of cache lines k. Assuming that all mergesets were previously aligned to the same memory address modulo k, all arrays $a_{j_1}, \ldots, a_{j_{p_i}}$ of a mergeset m_j share the same adjustment, which is simply a constant added to the starting address of all arrays in m_j:

$$\begin{aligned} adjust \quad & (m_1, \ldots, m_{k'}) = \forall j, l : (a_{j_l} \in m_j : adjust(a_{j_l})) \\ & \text{where } \forall\, m_j, m_{j'}\ start(m_j) \bmod k \neq start(m_{j'}) \bmod k \end{aligned}$$

In total, we get for one array a_{i_j} belonging to mergeset m_j the following data layout function $dl_{s,j} : \mathbb{N}^n \to \mathbb{N}$:

$$dl_{s,j}(d_1, \ldots, d_n) = merge_{s,j} \circ dl \circ \pi(d_1, \ldots, d_n) + adjust(a_j).$$

Proposition 1
Injectivity is preserved.

This holds simply because coloring assigns only one color to one mergeset; a value belongs to one mergeset only. Merging does not endanger injectivity as *merge* maps items from different arrays onto different locations: values subsequent in a live range are also subsequent in memory (parameter j).

Theorem 1
For every innermost loop, if its cyclic interval graph can be colored with $k' \leq k$ colors, then it is guaranteed that there are no conflicts stemming from this loop.

We will just give a sketch of the proof. The cyclic coloring heuristic guarantees that the cyclic interval graph can be colored with at most k colors. Conflicting data items bear different colors, thus they belong to different mergesets. These are mapped according to $dl_{s,j}$. From Proposition 1 follows that this mapping is injective.

Figure 6.2 summarizes the steps of our method. Let us consider the new data layouts for the running examples.

Example 10 (Example 6 continued)
There is no danger of self-interference when Livermore kernel 7 is unrolled 6 times, just the starting addresses of st_u,st_x,st_y and st_z are adjusted in order to avoid cache thrashing. The decomposition from Figure 5.c prescribes to merge z, y and x as described in Figure 5.d. One can observe that cache line boundaries do not matter in this approach: if a cache line can hold 8 items, just imagine 3 subsequent lines holding 8 sequences of z,y,x.

Example 11 (Example 7 continued)
In the matrix example, mergeset m_1 contains *a(i,k)* and *b(k,j)*, thus arrays a and b are *merged*; b is also transposed as shown in Figure 6.d.

In: representation of cache value live ranges, item size s
Out: data layout $dl_{s,j}$

determine the array live ranges
build the meeting graph and find an unrolling degree
perform cyclic interval graph coloring: k' colors
if two live ranges have the same color $c_i \in (c_i, \ldots, c_{k'})$
then put them into mergeset m_i
end if
for all mergesets $m_1, \ldots, m_{k'}$
  for all arrays: n-dimensional array $a_j \in m_j$
    transpose: $\pi(d_1, \ldots, d_n)$
    if $|mergeset| > 1$
    then $merge_{s,j}(d_n)$
    end if
  end for
end for
$adjust(m_1, \ldots, m_{k'})$

Figure 9: Deriving a New Data Layout

Example 12 (Example 8 continued)
Here, m_1 contains *a(i-1,j)*,*a(i+1,j)* and *b(i,j)*, m_2 contains $a(i, j)$. Rows *a(i-1,j)* and *a(i+1,j)* are thus interleaved with row *b(i,j)*. This is depicted in Figure 7.d. Here, the parameter s is of row size. We can note that in such cases, graphs similar to Figure 7.c indicate the need for *tiling* which we plan to integrate into our method.

Example 13 (Example 9 continued)
It is recommended to have *sigreal(i+le)* follow *sigreal(i)*; the same is true for *sigimag* as shown in Figure 8.d. This is defined for a fixed size of *le*. Ideally, thus, for every iteration of the loop over *le*, memory would have to be reordered. This is not realistic: for small *le* that do not exceed the cache size, there is no self-interference. For the others, the cost of reordering has to be weighed up against the cost of conflict misses. For problems of this structure, reordering is worthwhile because the array is one dimensional, whereas on every step a cache miss occurs.

7 Experimental Results

In the following, we show experimental results for the four running examples. We used the realistic C codes transformed source-to-source for implementing the layout transformation; currently we are integrating our optimization into a larger compiler framework. The DEC Alpha memory hierarchy is typical for a direct mapped separate data and instruction cache architecture. Measurements are made in standalone mode on a 200 MHz 21064 with 8K of direct mapped firstlevel data cache (32 Byte cache lines), on a 500 MHz 21164 with the same firstlevel cache architecture, on a 200 MHz Pentium Pro with the same cache and line size but 4-way set associativity, and on a 266MHz Pentium (i586) with also 8KB size, but 2-way set associativity.

We show sample miss rates and run times for Livermore kernel 7, a cross filter kernel, matrix multiply and Fast Fourier Transform (FFT) on Alpha and Pentium Pro, respectively. They were chosen because of the ways in which their memory access patterns differ from each other.

In the following, *standard* and *thrash* denote standard memory allocation and cache thrashing, respectively, while *GE98* stands for the data layout of [Gen98], *merge* for the new layout. All miss rates were measured with the -O4 option. Concerning the run times results for LL7 and FFT, run times were taken from 100 repetitions for accuracy.

The graph in Figure 10 presents miss rate improvements for Livermore kernel 7. As we measure only first-level cache misses, the sharp increase that is characteristic when data exceeds second level cache capacity does not occur in the results. The two new layout methods perform generally better. Note that, as cache miss rates oscillate heavily, lines between the points may not be drawn; we

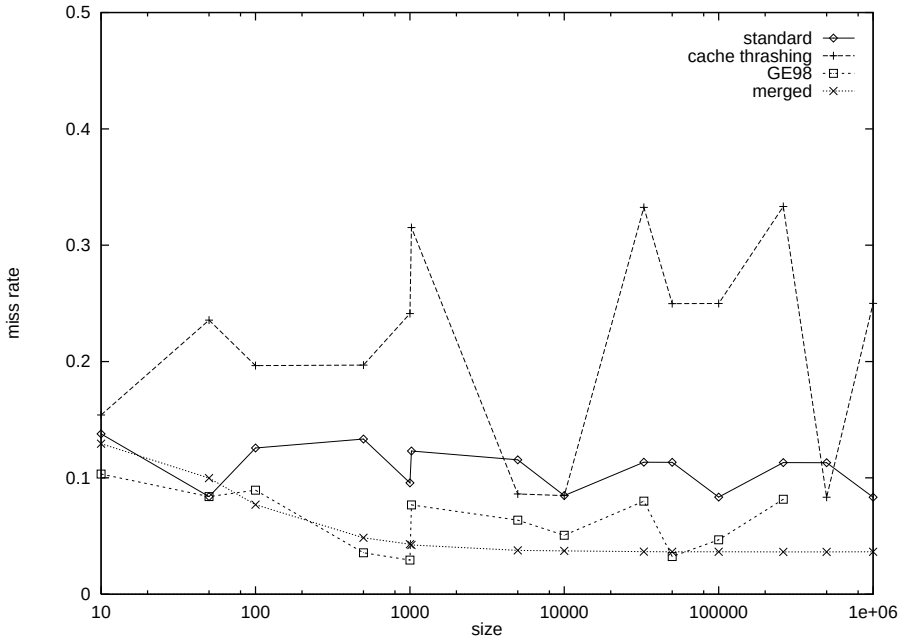


Figure 10: Miss rates for Livermore kernel 7

do so only to improve visibility. *merge* outperforms all other methods, only *GE98* is better twice, and it decreases regularly unlike other methods. We also checked sizes of a power of two as pathological cases, to which *merge* is quite insensitive; a similar effect was observed in [PNDN97].

For matrix multiply, in addition to the naive algorithm we compare a transposition for improving spatial reuse (*transpose*) and a well-established blocking technique (*block* [WL91]). Miss rates are significantly improved. Even if *separate* and *block* improve it even more, the former is much more expensive at run time due to integer division and modulo calculations, and the latter is a somewhat unfair competition, as it is no data, but a loop transformation.

The results for *filter* shown in table 1 indicate that miss rates are very similar for all methods.

For *fft*, the miss rate results are very good in comparison with the standard method and particularly with *GE98*. Concerning cache misses *merge* wins on both architectures due to the avoidance of cross interference: remember that in FFT, four arrays of a size of a power of two are accessed, making it extremely prone to cross interference; in this example, reuse is mainly present in the outer loop.

Figures 11 to 18 present run time improvements for the running examples.

Compared to the miss rates presented in Figure 10 and Table 1, it becomes obvious that run times do not improve proportionally to miss rates, in contrast to what is often assumed. The explanation is as follows: for multiple functional

Matrix multiply					
size	miss rate %				
	none	transpose	block	separate	merge
256	12.3	4.81	2.80	2.24	2.49
300	7.57	2.45	1.11	2.52	2.75
301	6.33	2.26	1.12	2.53	2.76

Filter			
size	miss rate %		
	none	separate	merge
550	3.02	3.02	3.03
1024	37.8	37.8	37.8

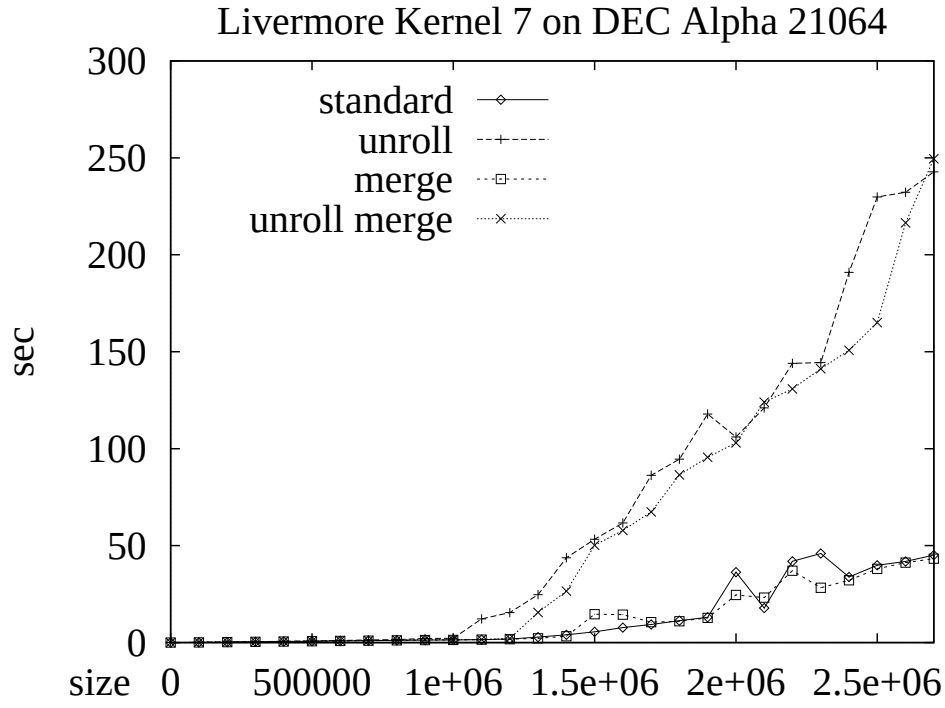
Fast Fourier Transform			
size	miss rate %		
	none	separate	merge
256	12.3	15.2	11.5
1024	12.6	15.5	12.0
4096	12.9	15.6	12.3

Table 1: Miss rates for Matrix multiply, cross filter and FFT on DEC Alpha

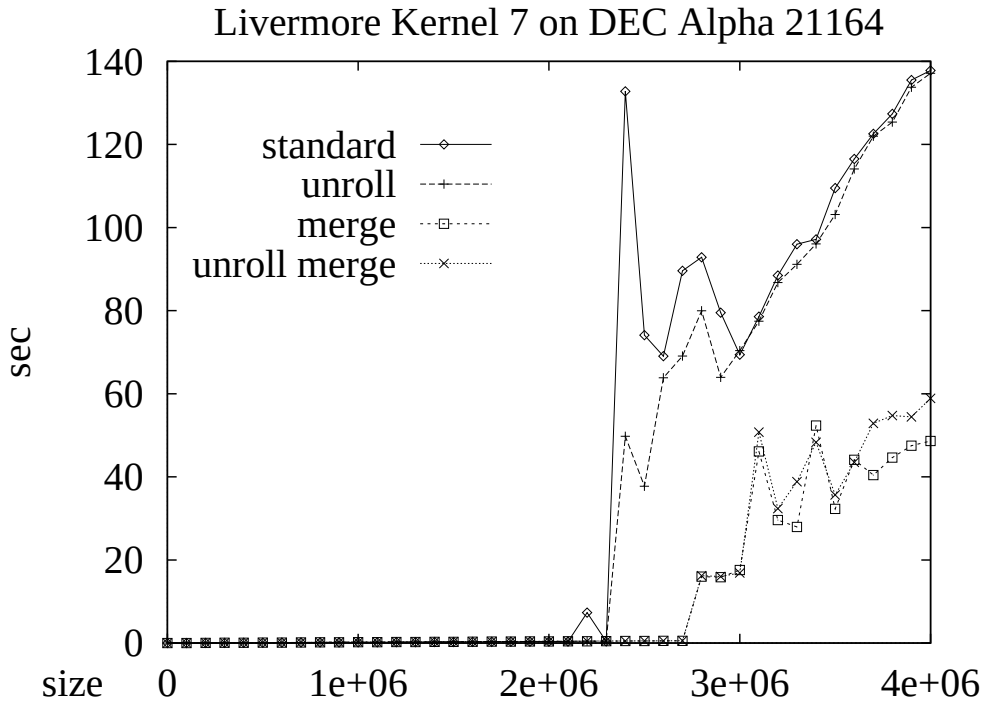
units, cache misses affect run time only if the floating point unit is not completely filled and suffers a stall; otherwise, the actions overlap. This becomes particularly obvious for the Livermore kernel (LL7) in Figure 10.

Livermore kernel 7 is generally assumed to be rather well-behaved; up to now, nearly no methods exist to improve it further. We also checked sizes of powers of two – known as pathological cases for caches; *merge* is quite insensitive to those. For run time measurements, we included the standard layout with loop unrolling as well as a version where x, y and z were merged without unrolling. It is known that unrolling generally may have positive effects on the run time; the larger part of the improvement however is due to merging. Furthermore, paging is postponed for *merge* because the number of different memory pages is reduced; this effect is much stronger on the Alpha architectures.

The filter calculates image data from its four neighbors; array accesses are cross-shaped. Reuse occurs both within the inner and the outer loop. Miss rates are very similar for all methods. The run times for *merge* are up to 10% faster than all others on Alpha and suffer much less from jumps on Pentium. For matrix multiply, we compare additionally a transposition for improving spatial reuse (*transpose*) —merging implies a transposition of one matrix— and a

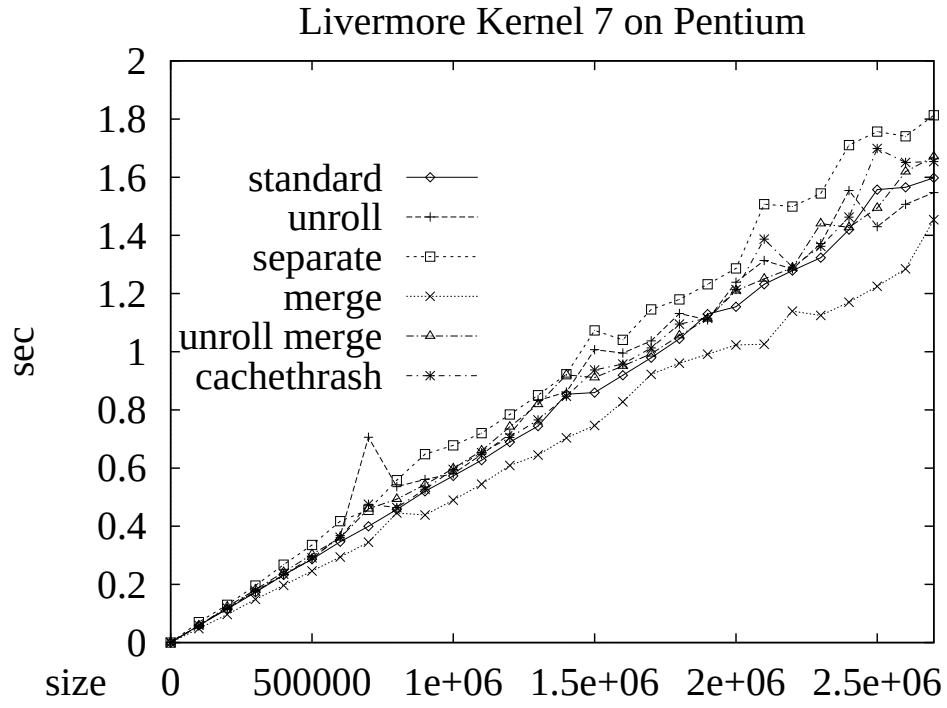


(a)

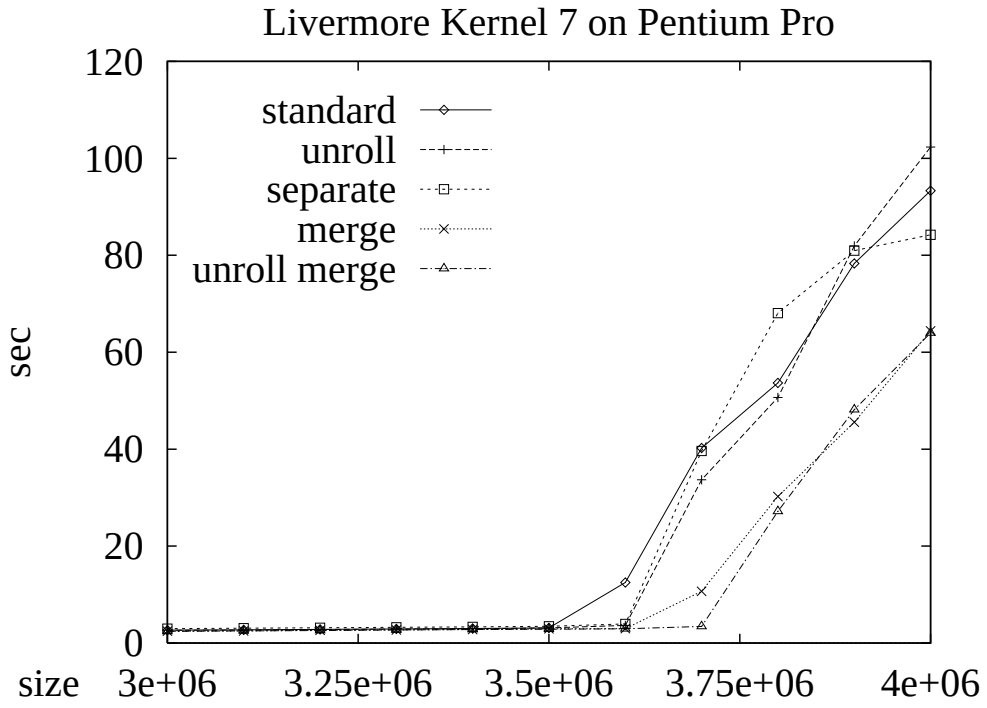


(b)

Figure 11: Run times for Livermore kernel 7 on Alpha

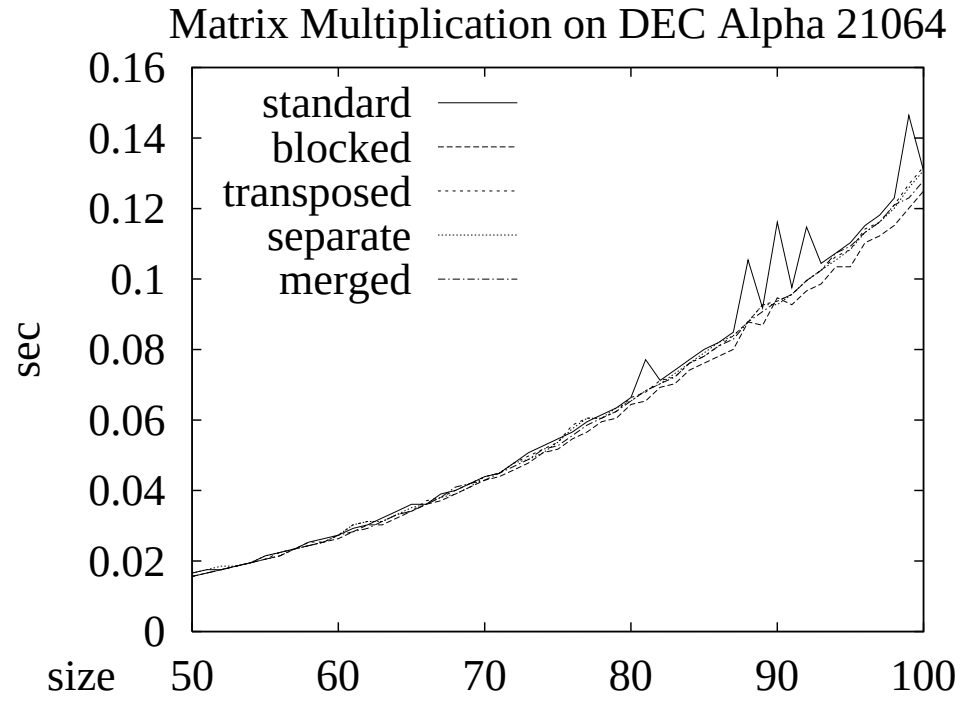


(a)

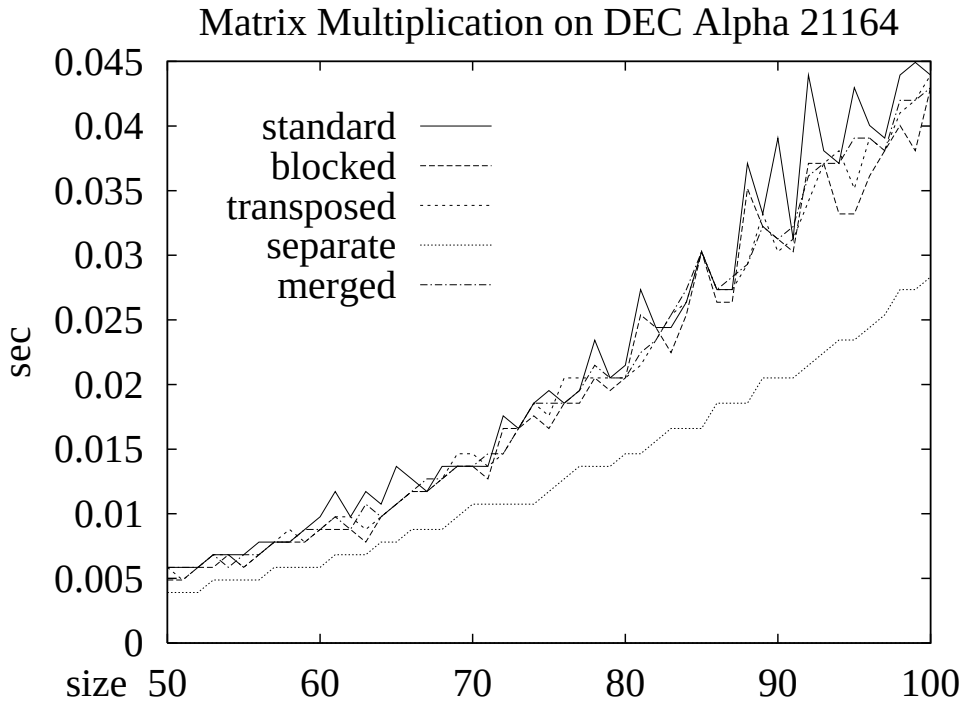


(b)

Figure 12: Run times for Livermore kernel 7 on Pentium

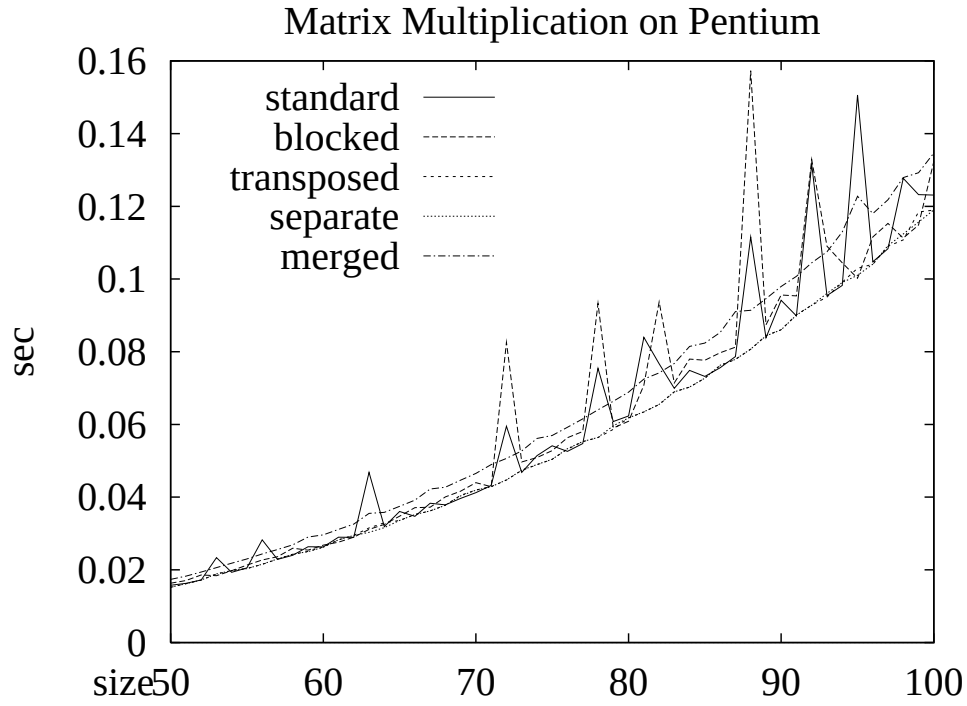


(a)



(b)

Figure 13: Run times for matrix multiply on Alpha



(a)

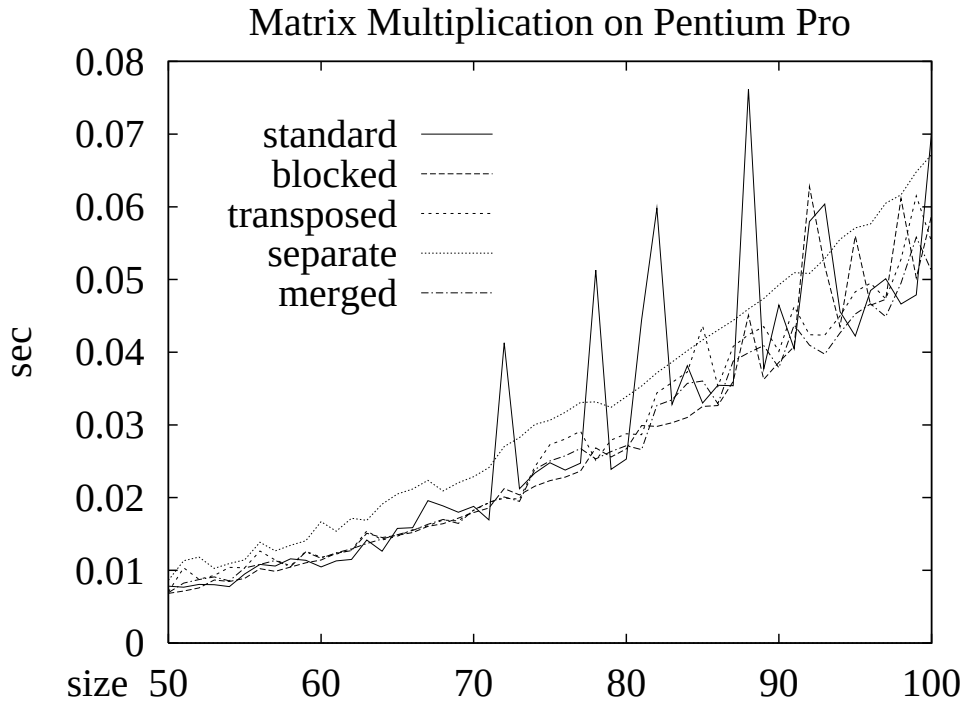
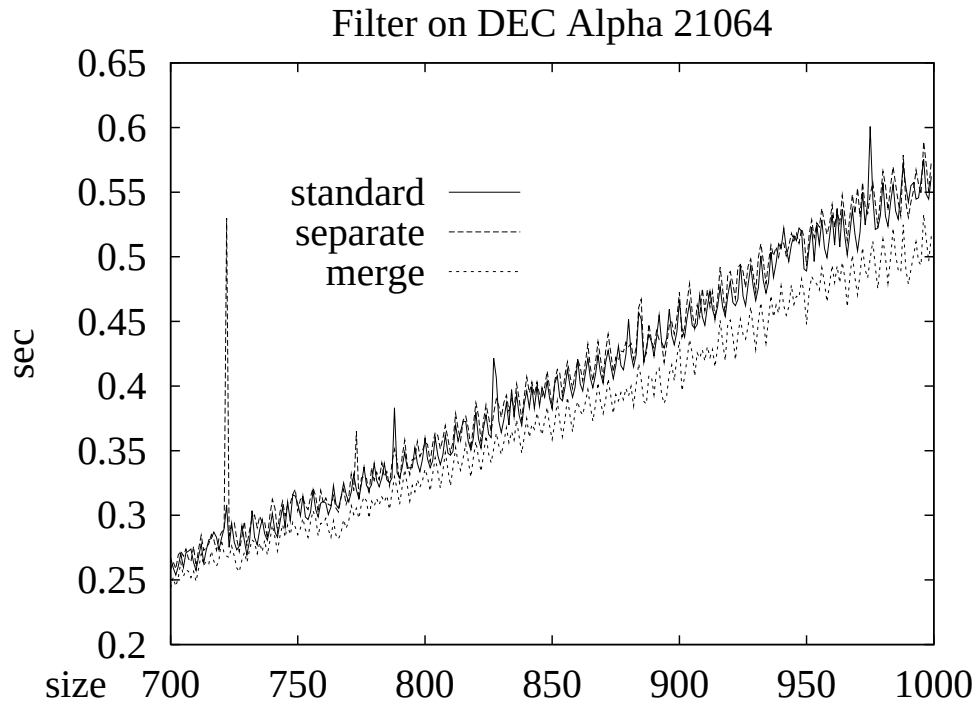
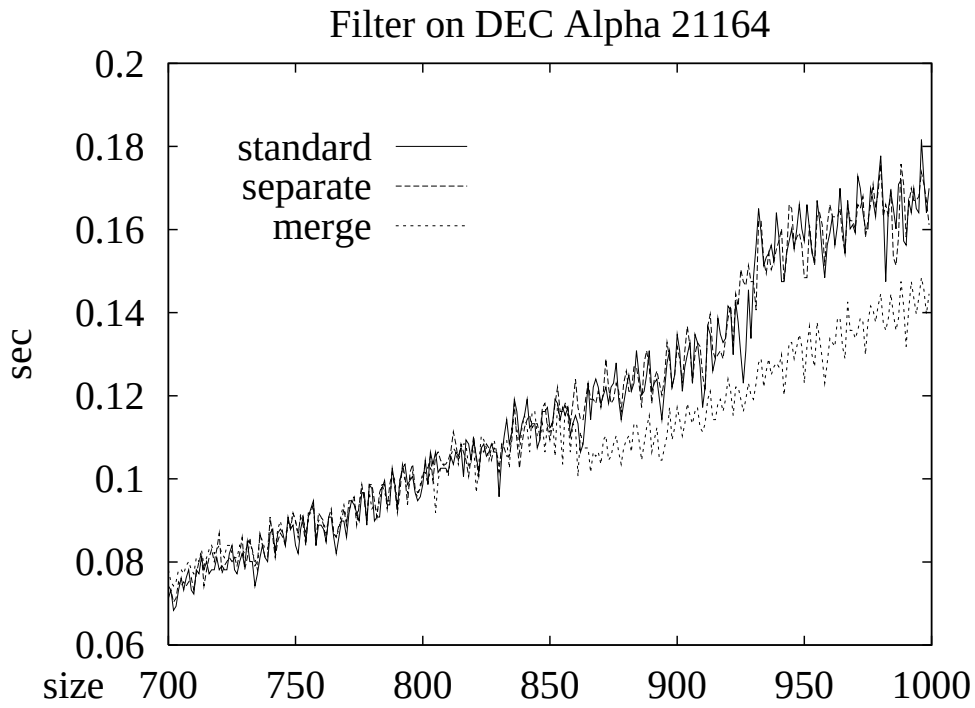


Figure 14: Run times for matrix multiply on Pentium

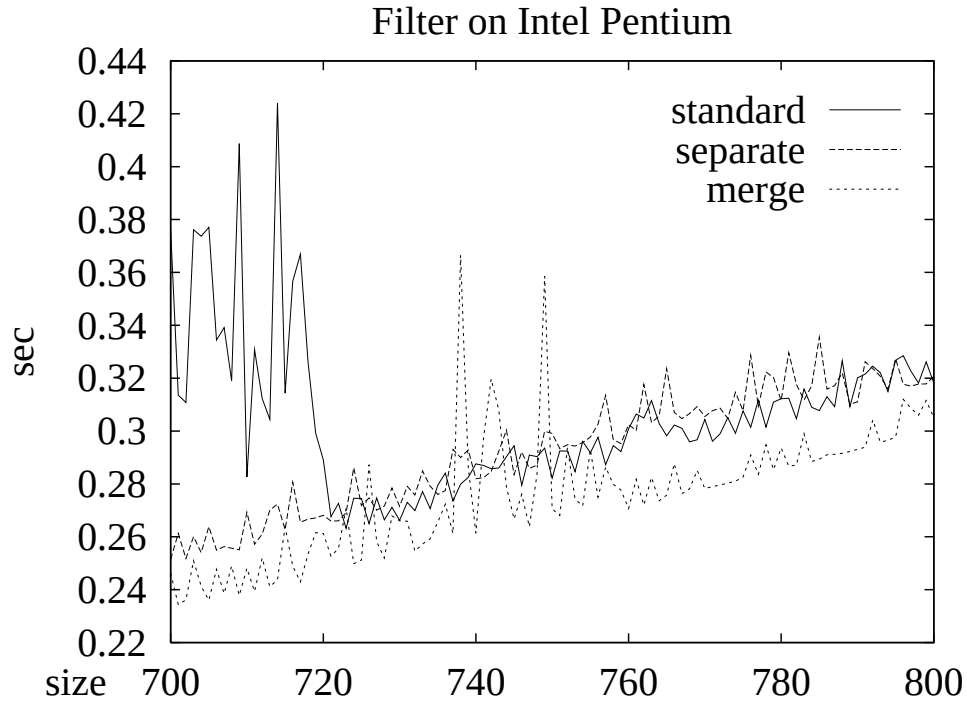


(a)

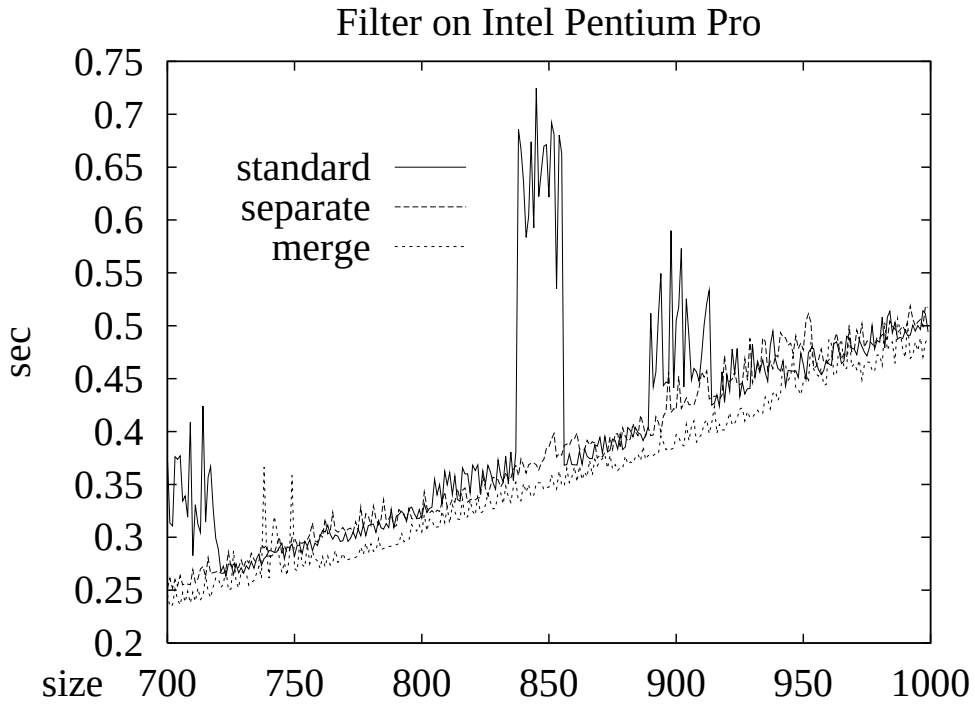


(b)

Figure 15: Run times for filter on Alpha

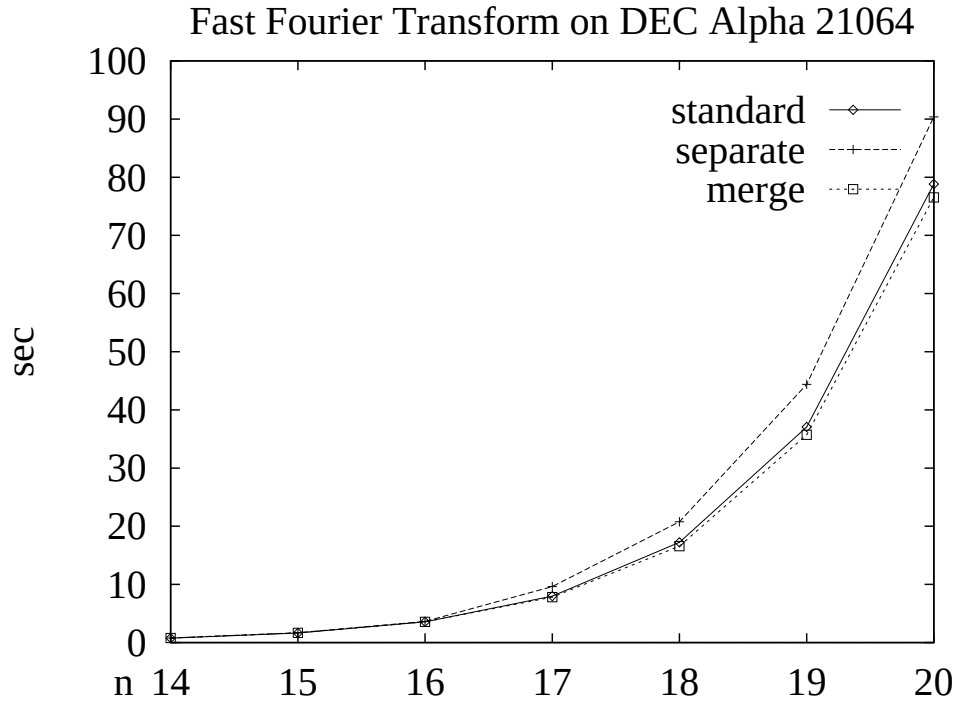


(a)

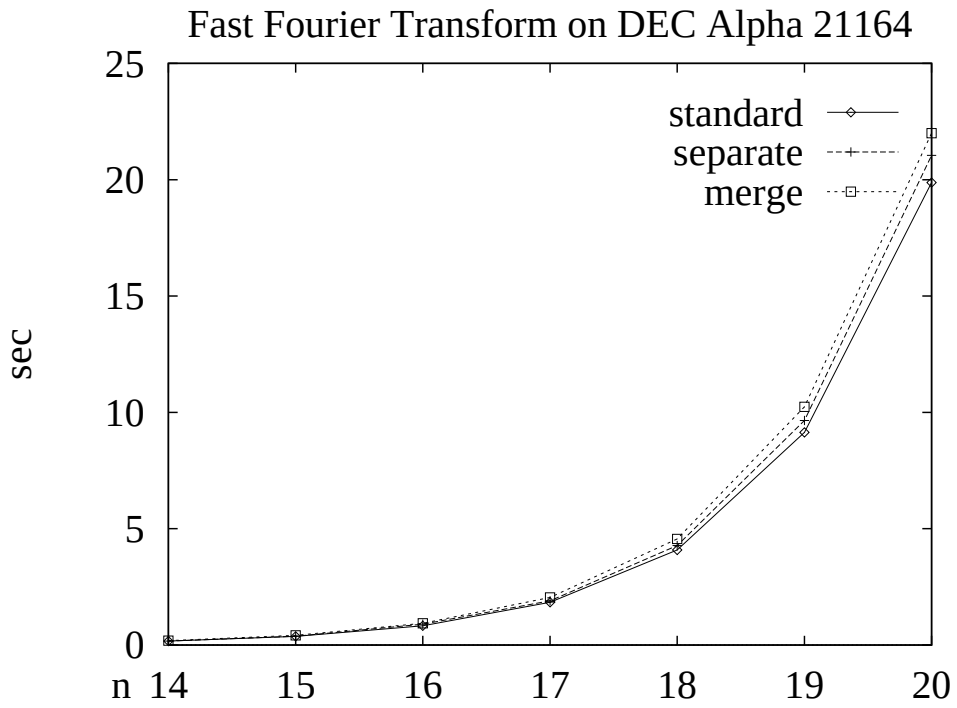


(b)

Figure 16: Run times for filter on Pentium

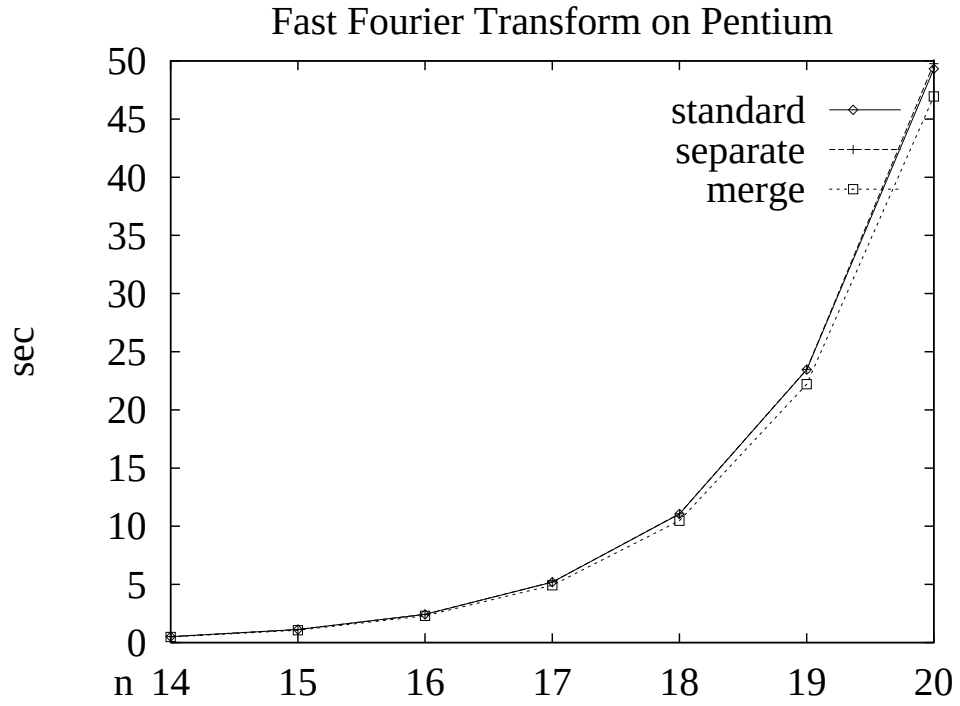


(a)

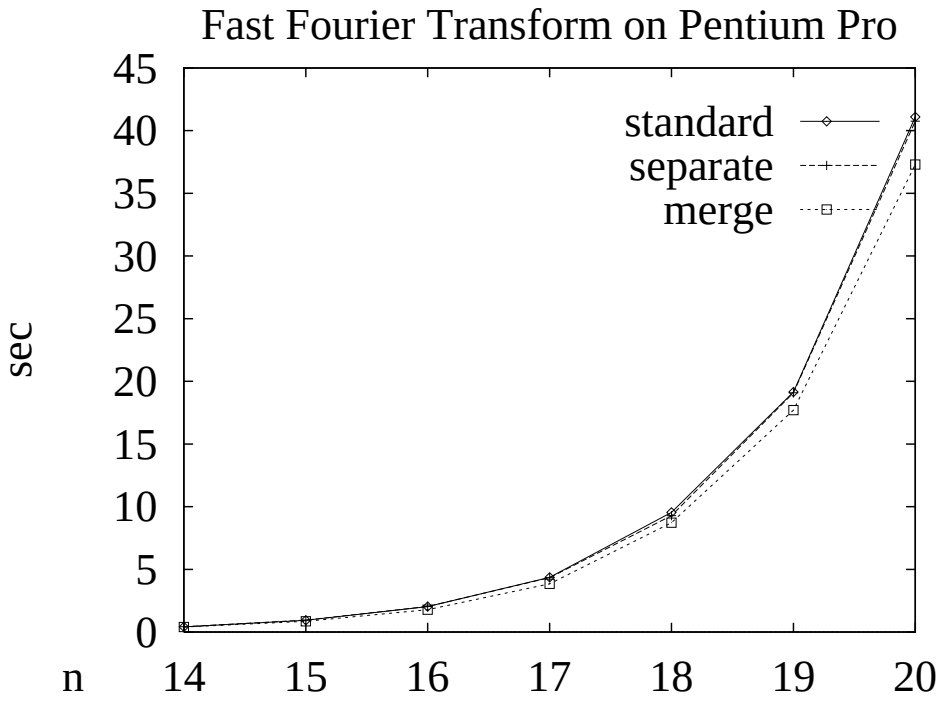


(b)

Figure 17: Run times for FFT on Alpha



(a)



(b)

Figure 18: Run times for FFT on Pentium

well-established blocking technique (*block* [WL91]). Miss rates are significantly improved. Even if *separate* and *block* improve them even more, the former is much more expensive at run time due to integer division and modulo calculations, and the latter is a somewhat unfair competition, as it is no data, but a loop transformation. About run times, the performance curve for matrix multiply with row major layout suffers from jumps. The performance of *merge* is comparable to that of blocking (*block*), but more even. This applies for both architectures. *separate* suffers heavily from the more expensive indexing functions on Pentium, while on Alpha, it beats all other methods due to a drastic reduction of cache conflicts. For FFT run times, *merge* performs comparably to the standard method on Alpha, while it performs up to 10% better on Pentium Pro. Concerning cache misses *merge* wins on both architectures due to the avoidance of cross interference: in FFT, four arrays of a size of a power of two are accessed, making it extremely prone to cross interference. Reuse is mainly present in the outer loop.

For codes with extensive reuse in the innermost loop, i.e. LL7, the new layout yields the most significant speedup. FFT and filter fall behind because a lot of reuse is due to the outer loop. Several other Livermore kernels and many image processing and compression algorithms can be classified along with LL7, making a more thorough examination clearly promising.

8 Conclusions and future work

We have introduced a method that improves cache line usage with the help of the meeting graph and a new notion of togetherness. Cyclic graph coloring is used to determine the distribution of the arrays in the cache lines. Thus a new data layout is then computed thanks to this information. For a class of scientific applications, we gain exact control over the memory layout and significantly reduce conflict misses. The experimental results show improvements in the cache miss rates, especially for the Livermore kernel 7, as well as run times improvements for all benchmarks.

Let us outline some directions of further research. *Prefetching*, the preloading of data from memory into cache ahead of time [MLG92], can be easily integrated into our framework, and our approach is complementary to tiling [LRW91, MCT96]. Furthermore our approach will have to be extended beyond innermost loops. Innermost loops are not specific to the area of scientific computing. In more general application areas, data type sizes vary and accesses are less regular. Still, a clever data layout helps to improve performance as shown in [GTZ98].

We plan also to examine the interaction with compiler-controlled instruction scheduling in more detail. In the presence of branches, a specific data flow analysis is required.

Acknowledgments This work was initiated by discussions with Uwe Assmann; we furthermore thank the authors of ATOM providing their tool. S. Lelait is funded by the Austrian Science Fund (FWF). D. Genius is on a grant from Graduiertenkolleg "Beherrschbarkeit Komplexer Systeme" of the German Science Foundation (DFG).

References

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105. ACM, ACM, 1982. Available as SIGPLAN Notices 17(6) June 1982.

[ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: A new model for loop cyclic register allocation. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings PACT'95*, pages 264–267, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[Gen98] Daniela Genius. Handling cross interferences by cyclic cache line coloring. In *1998 Parallel Architectures and Compilation Techniques Conference (PACT'98)*, Paris, France, October 14-16 1998. to appear, IEEE.

[GJMP80] M.R. Garey, D.S. Johnson, G.L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chord s. *SIAM J. Alg. Disc. Meth.*, 1(2):216–227, June 1980.

[GMM97] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 317–324, New York, July7–11 1997. ACM Press.

[GTZ98] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using sandwich types. In *Proceedings of the 1998 Types in Compilation workshop*, Kyoto, Japan, march 1998. Springer LNCS.

[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 176–191. 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.

[KCR+98] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. Enhancing spatial locality via data layout optimzations. In *Proceedings of EuroPar'98*, pages 422–434, Heidelberg, September 1998. Springer LNCS.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 8–11, 1991.

[LW94] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.

[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.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[MT96] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[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, Nov 1993.

[RT98] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, june 1998.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271, New York, NY, USA, May 1994. ACM Press.

[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.*