Compiler-Directed Reordering of Data by Cyclic Graph Coloring

Daniela Genius[1] and Sylvain Lelait[2]

[1] Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Zirkel 2, 76128 Karlsruhe, Germany, E-mail: genius@ipd.info.uni-karlsruhe.de, Phone: (+49) 721 608-4763, Fax: (+49) 721 30 0 47

[2] Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria, E-mail: sylvain@complang.tuwien.ac.at, Phone:(+43) 1 588 01 185 11, Fax: (+43) 1 588 01 185 98

Abstract. We show that cyclic graph coloring techniques from register allocation are successfully applicable to caches. The presence of values in the cache can be modeled in analogy to register live ranges. By applying the meeting graph method, the compiler can determine an unrolling factor and guarantee the maximal number of colors, i.e. of cache lines required. Values of one color belong together, even if they stem from different data structures. For a class of scientific codes, the new data layout improves reuse and conflict behavior by a systematic merging.

1 Introduction

Scientific computing 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. 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.

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.

The focus is on applications with more complex access patterns, where often the same value is required in several iterations, and where a large number of arrays are accessed with the imminent danger of interfering with each other in the cache. For innermost loops, such interference and temporal reuse can be modeled together in a cyclic interval graph. By applying the meeting graph method, the compiler can compute an unrolling factor and determine the maximal number of colors, i.e. cache lines, required. Values with 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. For typical example codes from the above areas, cache behavior is considerably improved, yielding run time reductions. In addition, paging activity is significantly postponed in some cases.

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

2 Background

A *(physical) cache line* usually holds several data items, which we denote as *cache values*; this is reflected by partitioning memory into *memory lines* of the same size. Whenever a value requested in a calculation is present neither in registers nor in the cache, a *cache miss* occurs: the memory line surrounding the requested cache value is loaded into a cache line. For more details on cache organization we refer to [11].

There are two important reasons why cache misses cannot be overcome by hardware alone: *Capacity misses* are due to the fact that most of the values in a cache line remain unused. Memory lines stemming from different areas may evict still required data from a physical cache line, accounting for *conflict misses*. To overcome these problems, the temporal/spatial structure of accesses has to be changed by improving either of the following: Improving *spatial reuse* ensures that as many of the values in a cache line are used, while *temporal reuse* takes care that the same data item is accessed several times. *Cross interferences* occur when different arrays compete for a cache line.

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 are imposed. We consider only such array references that are generated by an *affine mapping* of the loop counter vector [20]. 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.

On the hardware side, the focus is on single processor, instruction-level parallel machines, typically workstations. For presentation we chose a Low Intermediate Representation (LIR) for RISC architectures, which means that arithmetic operations on registers can are clearly separated from load/store operations[1].

[1] The method is nevertheless applicable for any architecture; out-of-order execution will reduce precision.

3 Coloring the Cyclic Interval Graph

3.1 Extracting Cache Behavior from the Code

We denote by the *(cache) live range* the number of processor cycles in which a value is actually present in the cache [6]. In this paper, we restrict to outlining the basic idea and applying it to an example. The live range begins, in analogy to a register live range, when the memory location's content is first accessed (here: loaded into the cache). It ends when it is no longer required to be in the cache. There is a close relation between cache and register live ranges. For our class of programs, load and store instructions can be easily extracted from the LIR representation of the innermost loop.

Example 1 (Livermore Loop 7). Livermore kernel 7 of Figure 1 is rather well-behaved, except for conflict misses. Four arrays compete for the cache lines in only one loop. On the other hand, reuse is largely present. Access patterns are simple except for u, where seven subsequent elements are accessed in one loop. We will use this loop throughout the paper; it satisfies the code properties stated above.

```
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])))
```

Fig. 1. Livermore Loop 7

Figure 2.a shows the LIR of the innermost loop body[2]. Accesses to x,y and z last one cycle, the values are read and loaded into the registers. Immediately after this, they are no longer required to be in the cache. Values of vector u are heavily reused. *u(k+6)* is first used in iteration k, last in iteration $k+6$; thus live ranges last 7 cycles. The resulting cyclic interval graph is shown in Figure 2.b, where a "dot" denotes an access to the value.

3.2 A Togetherness Criterion

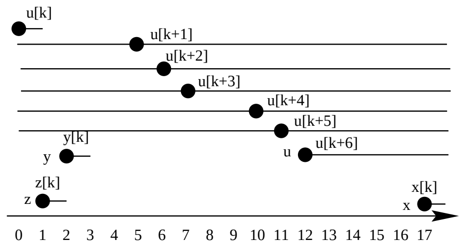
Cache lines usually hold more than one element. How can the compiler control which data is placed together in a physical cache line? We propose to achieve *togetherness* by chaining together live ranges for near-optimal cache line usage. Cyclic interval graph coloring can be used to indicate togetherness. Coarsely speaking, what belongs together in memory bears the same color. This allows us to formulate the following togetherness criterion:

Togetherness for m live ranges is a result of cyclic graph coloring: if they share the same color, they are grouped horizontally.

[2] For simplicity of presentation, multiplications with constants r and t are left out; all LIR instructions are assumed to require one cycle.

```
 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)

Fig. 2. Deriving cache value live ranges

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 shown in Figure 3.

3.3 Usual Cyclic Interval Graph Coloring

The small example of Figure 3.a, taken from [10], shows in Figure 3.c how live ranges in innermost loops can be represented by a *cyclic interval family*, which gives the *cyclic interval graph* of Figure 3.b. Cyclic interval graph q-coloring, also known as circular-arc graph q-coloring, is a polynomial problem [4]. But the algorithm is only practical for small values of q, therefore heuristics are used. One of the most efficient is the one introduced in [10]. The lower bound on the number of colors is the maximum number of live ranges overlapping a point in the interval family, often noted $MaxLive$ [12]. For instance in Figure 3.c, $MaxLive$ is equal to 2. Unfortunately this method can not handle intervals overlapping themselves. Furthermore this method does not guarantee a coloring with $MaxLive$ colors.

a)

```
Do i=1,N,1
   x1 = x3*10
   x2 = x4*20
   x3 = x1+5
   x4 = x2+x3
```

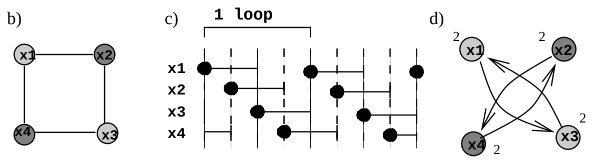


Fig. 3. Cyclic interval graph/meeting graph

3.4 Meeting Graph

On the other hand, with the *meeting graph* method [3], intervals which overlap themselves can be handled. The meeting graph heuristic was designed to obtain a family of intervals that can be colored with $MaxLive$ colors by unrolling the loop if necessary. It only works for families having $MaxLive$ intervals at each cycle, therefore some fictitious intervals may be added. Then the graph itself is built like in Figure 3.d. 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, u, one has to take the lcm of the weights, W_i, of disjoint circuits which can be found in the graph. In all cases finding a decomposition is a polynomial problem [3] and the coloring with $MaxLive$ colors is ensured. For this step of coloring, W_i colors are cyclically allocated to the intervals belonging to the u copies of each circuit.

In our context, the intervals depict the time that a value must stay in the cache. Minimizing the number of colors is equivalent to minimize the number of cache lines used. A decomposition of the meeting graph is computed, the lcm of the circuits weights gives the unrolling degree. Then the sequence of the intervals in the circuits gives the order of the arrays in the cache lines. In the following, we apply this method on the running example.

Example 2. Figure 4.b shows the meeting graph of Livermore Loop 7 which is derived from the cyclic interval family of Figure 2.b. 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. Hence 6 fictitious intervals have to be inserted per iteration, as shown in Figure 4.a. Figure 4.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 interval u from overlapping itself. The interval graph is unrolled 6 times and colored according to the result: 6 colors are required for the 6 live ranges issued of u and one additional color for those belonging to the other arrays.

4 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.

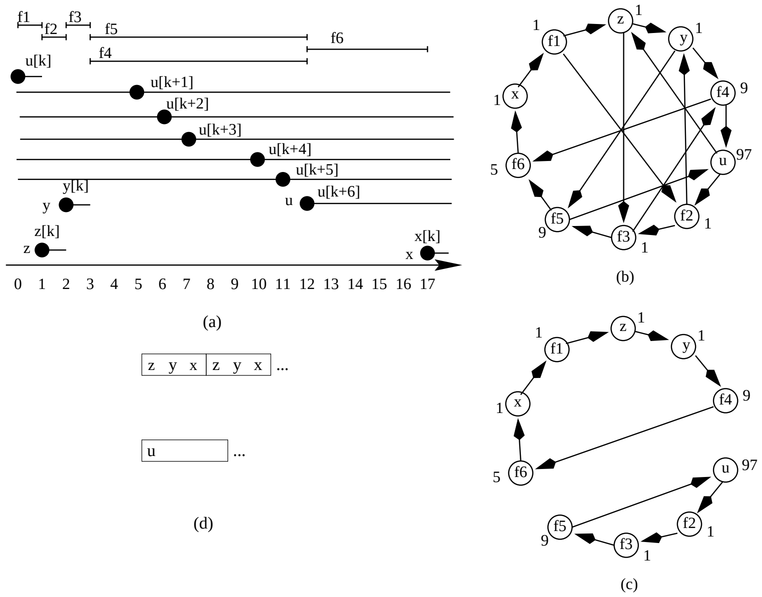


Fig. 4. Applying the meeting graph to Livermore loop 7

4.1 Standard Data Layout

Compilers usually 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. the 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 $dl : \mathbb{N}^n \to \mathbb{N}$:

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

4.2 Deriving a Coloring-directed Data Layout

In the general definition, a compiler-directed data layout can follow any (injective) function. Cyclic interval graph coloring now allows to intermix arrays systematically. We now present a simplified version of the algorithm which derives a data layout from the result of coloring. For full detail, see [5].

Assume that a loop nest was unrolled and then k colors $c_1, \ldots, c_k$ are used. Live ranges that have been identified as belonging to the same color c stem from p different arrays. These are combined in one *mergeset* $merge_c$.

In a first step, arrays are transposed so that the innermost loop variable i_n accesses dimension d_j of the array. 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.

Our method is not restricted to data items of equal size. Thus, in a second step, elements of size s are mixed. If there is only one array, no intermixing might be required, but a reordering of elements, that can also be handled by the formalism. For array a_i of a mergeset, let d_n be the offset in direction of the innermost loop index. The position i of the array in the mergeset determines which of the $p*s$ blocks is addressed, $d_n \bmod (p*s)$ yields the offset inside such a block. $merge : \mathbb{N} \to \mathbb{N}$ affects only the "innermost" dimension:

$$merge_{s,i}(d_n) = (p*s)*((d_n * p)\ div\, s) + (i-1) + (d_n \bmod (p*s))$$

Finally, we add starting address adjustments for the c_k different mergesets in order to avoid cross interference between the new compound arrays. Assuming that all mergesets were previously aligned to the same memory address modulo k, all arrays $a_{i_1}, \ldots, a_{i_p}$ of a mergeset m_c share the same adjustment, which is simply a constant added to the starting address of all arrays in m_c.

In total, we get for one array a_i belonging to mergeset m_c the following data layout function $dl_{s,i} : \mathbb{N}^n \to \mathbb{N}$:

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

Figure 5 summarizes the steps. Let us conclude by the new data layout for the running example.

In: Representation of cache live ranges, block size s
Out: Data layout $dl_{s,i}$

```
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
then put them into mergeset m_c
end if
for all mergesets m_{c_1}, ..., m_{c_k}
    for all arrays: n-dimensional array a_i ∈ m_C
        Transpose: π(d_1, ..., d_n)
        if |mergeset| > 1
        then merge_{s,j}(d_n)
        end if
    end for
end for
Adjust(m_1, ..., m_{k'})
```

Fig. 5. Deriving a New Data Layout

Example 3. After unrolling, there are no self-overlapping live ranges. The decomposition from Figure 4.c prescribes to merge z, y and x. The starting addresses of st_u and st_{zyx} are adjusted. No transposition is needed for one dimensional arrays. The resulting data layout is shown in Figure 4.d.

5 Experimental Results

We made experiments on Livermore Loop 7 and a set of scientific codes[3] using the realistic C codes transformed source-to-source for implementing the layout transformation; currently we are integrating our optimization into a larger compiler framework. As target architectures we use the DEC ALPHA AXP, a direct mapped cache architecture with separate data and instruction caches, and the INTEL I686. For the ALPHA, a specific analysis tool (ATOM [19]) is available, which allows realistic measurement of cache misses by instrumenting binaries[4]. Measurements are made in stand-alone mode on a 500 MHz 21164 with 8K of direct mapped first-level data cache (32 Byte cache lines) and on a 200 MHz Pentium Pro with the same cache and line size but 4-way set associativity.

In the following, *merge* stands for our new method, *standard* for the usual layout method, *separate* for the method of [6]. Figure 6 presents run time improvements for the running examples, and Table 1 results about cache misses.

Livermore kernel 7 is generally assumed to be rather well-behaved; up to now, nearly no method exists 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 stronger for the Alpha architecture.

We show also in Table 1 and Figure 6 sample miss rates and run times for 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. 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 well-established blocking technique (*block* [20]). Miss rates are significantly improved. Even if *separate* and *block* improve them even more, the former is much more expensive at run time due

[3] In the long version [5], the application of our method is shown for all examples.

[4] We do not present miss rate results for the Pentium Pro, because, unlike for the DEC AXP, we do not know any tool like ATOM for the PPro. Thus we can not measure cache misses without simulation, hence it would be unfair for comparison.

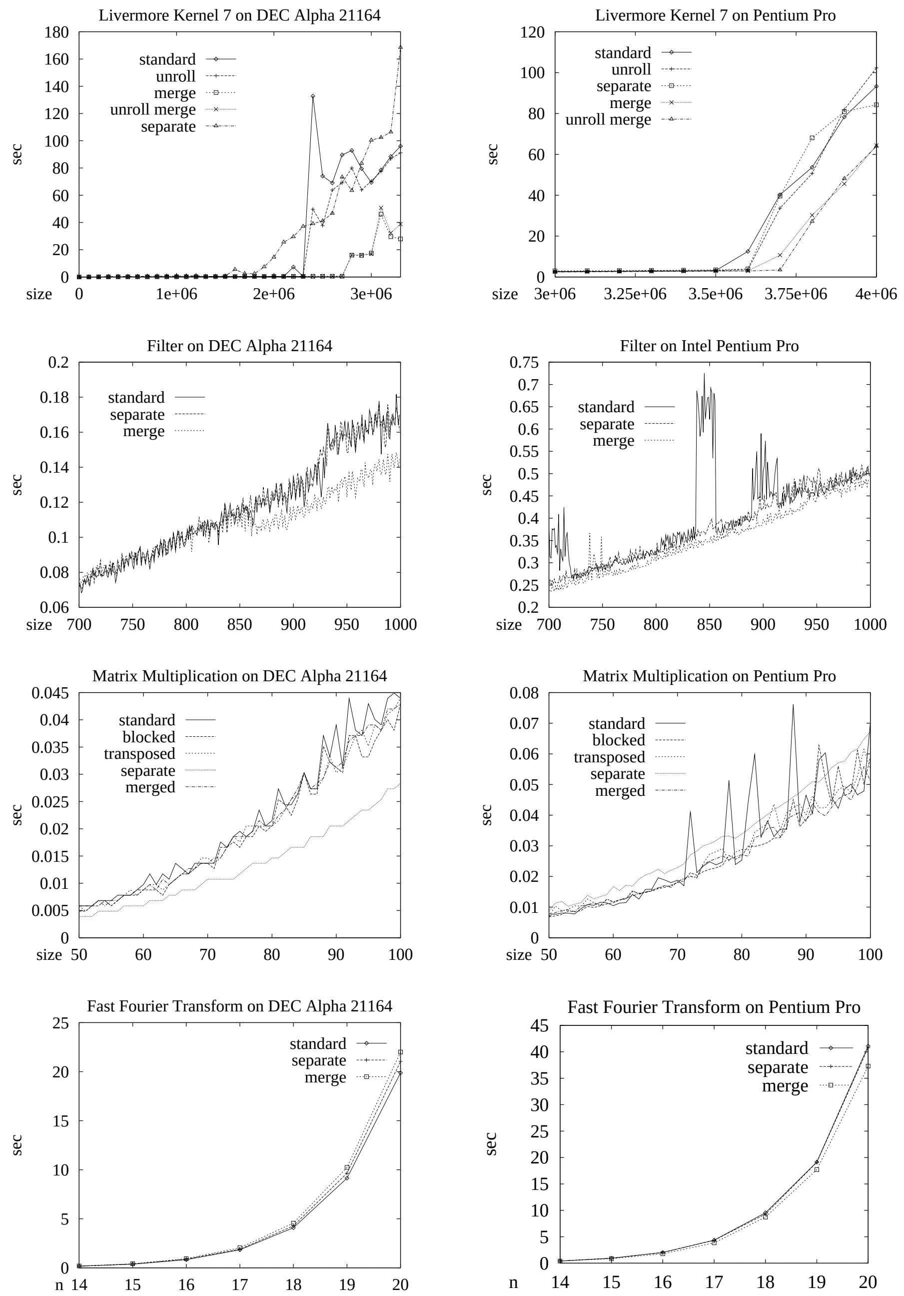


Fig. 6. Run times

Livermore loop 7				Filter			
size	miss rate %			size	miss rate %		
	none	separate	merge		none	separate	merge
100000	15.63	12.54	5.09	550	3.02	3.02	3.03
262144	20.3	17.8	5.83	1024	37.8	37.8	37.8

Matrix multiply						Fast Fourier Transform			
size	miss rate %					size	miss rate %		
	none	transpose	block	separate	merge		none	separate	merge
256	12.3	4.81	2.80	2.24	2.49	256	12.3	15.2	11.5
300	7.57	2.45	1.11	2.52	2.75	1024	12.6	15.5	12.0
301	6.33	2.26	1.12	2.53	2.76	4096	12.9	15.6	12.3

Table 1. Miss rates for benchmarks on DEC Alpha

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.

6 Related work

There are still few works on the impact of data layout on data cache misses, all of very recent origin. Mostly they are dealing with *padding* [2, 18], the insertion of gaps filled with unused data into data structures. Ghosh et al. [8] use cache miss equations to precisely select padding and tile size. Characteristic of padding is the danger of paging due to inflation of data structures.

The hyperplane transformations of Kandemir et.al. [13] target at spatial locality exclusively. They yield significant improvements at low additional run time cost, but are not applicable to complex access patterns. *Merging* is a simple but effective way to prevent the arrays from interfering with each other, usually performed by the programmer [11, 14]. The method we present enables the compiler to intermix arrays in a systematic way that goes beyond merging.

Array intermixing was done in [7] by borrowing a data distribution function from parallel computing and a posteriori-evaluation through a cost function, whereas the present approach allows more systematic intermixing.

Graph coloring wrt. caches was examined in the context of cache analysis by Rawat [17]. He contributes to 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 estimates made by Rawat overestimate cache misses significantly. Hashemi, Kaeli and Calder [9] 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 optimizations are based on trace-driven simulations. Restricting ourselves to scientific applications, we can obtain more information at compile time. [6] employed cyclic coloring for conflict prediction, not as togetherness criterion.

7 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 using this information. For a class of scientific applications, a memory layout with systematic merging is found, significantly reducing cache misses and improving run times.

Our approach is complementary to blocking [20, 15] which is promising because it will take us beyond innermost loops. As merging has proved to yield equally good results, a combination of both methods promises further improvement. We also plan to examine the interaction with compiler-controlled instruction scheduling in more detail. *Prefetching*, the preloading of values into the cache before it is demanded, can be easily integrated into our framework by extending live ranges into previous loop iterations. It remains to be shown if such a new, more exact prefetching scheme can do better than the one described in [16]. Innermost loops are not specific to the area of scientific computing. In application areas beyond scientific computing, data type sizes vary and accesses are less predictable. We are examining to what extent our approach can be applied in that context [1].

References

1. U. Assmann, D. Genius, P. Fritzson, H. Sips, R. Kurver, R. Wilhelm, H. Schepers, and T. Rindborg. Java and cosy technology for embedded systems: The joses project. In *Proc. EMMSEC'99*, Stockholm, June 1999. to appear.
2. David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94 – Integrated Solutions*, pages 270–282, Toronto, Ontario, 1994.

3. 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.
4. M.R. Garey, D.S. Johnson, G.L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Alg. Disc. Meth.*, 1(2):216–227, 1980.
5. D. Genius and S. Lelait. Improving data layout through coloring-directed array merging. Technical Report 1999-3, Universität Karlsruhe, Januar 1999.
6. Daniela Genius. Handling cross interferences by cyclic cache line coloring. In *Proceedings of the 1998 Parallel Architectures and Compilation Techniques Conference (PACT'98)*, pages 112–117, Paris, France, October 14-16 1998. IEEE.
7. Daniela Genius and Jörn Eisenbiegler. Reducing cache conflicts by a parametrized memory mapping. In *Proc. of the 4th Int. Conference of the Austrian Center for Parallel Computation (ACPC 99)*, Salzburg, Feb. 16-18 1999. to appear in LNCS.
8. S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.
9. Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *PLDI 1997*, pages 171–182, jun 1997. *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*
10. 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.
11. John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufman, 2nd edition, 1996.
12. Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. *SIGPLAN Notices*, 28(6):258–267, June 1993. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.
13. 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.
14. Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.
15. 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.
16. Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.
17. Jai Rawat. Static analysis of cache performance for real-time programming. Technical Report IASTATECS//TR93-19, Iowa state university, Nov 1993.
18. G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 Int. Conference on Supercomputing (ICS'98)*, pages 353–360, Melbourne,Australia, July 1998.
19. Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.
20. 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.*