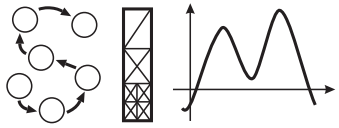
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)



Cache Performance Tuning of Numerically Intensive Codes

Markus Kowarschik, Christian Weiß

Lehrstuhlbericht 02-2

Cache Performance Tuning of Numerically Intensive Codes*

Markus Kowarschik[1] and Christian Weiß[2]

[1] Lehrstuhl für Systemsimulation (Informatik 10)
Institut für Informatik
Friedrich–Alexander–Universität Erlangen–Nürnberg, Germany
Markus.Kowarschik@cs.fau.de

[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Fakultät für Informatik
Technische Universität München, Germany
Christian.Weiss@cs.tum.edu

Abstract. Today's computer architectures employ fast cache memories in order to hide both the low main memory bandwidth and the latency of main memory accesses which is slow in contrast to the floating–point performance of the CPUs. Efficient program execution can only be achieved, if the codes respect the hierarchical memory design. This is particularly true for numerically intensive codes which are characterized by small computational kernels which are based on nested loop structures and contain a high potential of data locality. This paper introduces the design of cache memories, illustrates techniques to enhance the cache utilization of numerical codes, and provides various examples of current research focusing on cache–aware numerical methods.

1 Introduction

In order to mitigate the impact of the growing gap between CPU speed and main memory performance, today's computer architectures implement hierarchical memory structures. Usually, there is a small and expensive high speed memory sitting on top of the hierarchy which is usually integrated within the processor chip to provide data with low latency and high bandwidth; i.e., the CPU registers. Moving further away from the CPU, the layers of memory successively become larger and slower. The memory components which are located between the processor core and main memory are called *cache memories* or *caches*. They are intended to contain copies of main memory blocks to speed up accesses to frequently needed data [27, 28]. The next lower level of the memory hierarchy is the main memory which is large but also comparatively slow. While external memory such as hard disk drives or remote memory components in a distributed computing environment represent the lower end of any common hierarchical memory design, this paper focuses on optimization techniques for enhancing cache performance.

The levels of the memory hierarchy usually subset one another so that data residing within a smaller memory are also stored within the larger memories. A typical memory hierarchy is shown in Figure 1.

Efficient program execution can only be expected if the codes respect the underlying hierarchical memory design. Unfortunately, today's compilers cannot introduce highly sophisticated cache–based transformations and, consequently, much of this optimization effort is left to the programmer [24, 38].

This is particularly true for numerically intensive codes, which our paper concentrates on. Such codes occur in almost all science and engineering disciplines; e.g., computational fluid dynamics, computational physics, and mechanical engineering. They are characterized both by a large portion of floating–point (FP) operations as well as by the fact that most of their execution time is spent in small computational kernels based on loop nests. Thus, instruction cache misses have no significant impact on execution performance. However, the underlying data sets are typically by far too large to be kept in a higher level of the memory hierarchy; i.e., in cache.

* This research is being supported in part by the *Deutsche Forschungsgemeinschaft* (German Science Foundation), projects Ru 422/7–1,2,3.

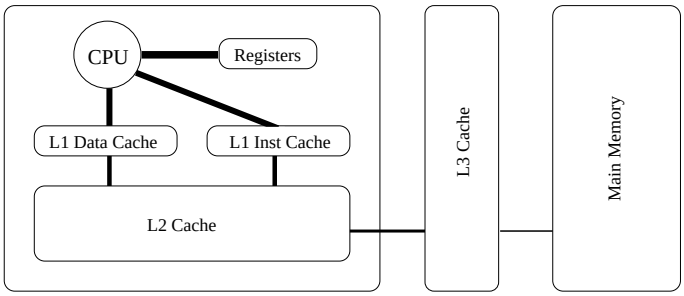


Fig. 1. A typical memory hierarchy containing two on–chip L1 caches, one on–chip L2 cache, and a third level of off–chip cache. The thickness of the interconnections illustrates the bandwidths between the memory hierarchy levels.

Due to data access latencies and memory bandwidth issues, the number of arithmetic operations alone is no longer an adequate means of describing the computational complexity of numerical computations. Efficient codes in scientific computing must necessarily combine both computationally optimal algorithms and memory hierarchy optimizations. Multigrid methods [57], for example, are among the most efficient algorithms for the solution of large systems of linear equations. The performance of such codes on cache–based computer systems, however, is only acceptable if memory hierarchy optimizations are applied [59].

Our paper is structured as follows. In Section 2, we will introduce some elementary cache characteristics, including a brief discussion of cache performance analysis tools. Section 3 contains a general description of important cache optimization techniques. In Section 4, we will provide examples which illustrate how these techniques have been used to tune a variety of implementations of numerical algorithms; e.g., codes from numerical linear algebra. Furthermore, we will outline how design aspects of memory subsystems can even guide the development of new numerical algorithms. Section 5 concludes the paper.

2 Characteristics and Performance of Cache Memories

2.1 Cache Organization

Typically, a memory hierarchy contains a rather small number of registers on the chip which are accessible without delay. Furthermore, a small cache — usually called *level one (L1) cache* — is placed on the chip to ensure low latency and high bandwidth. The L1 cache is often split into two separate parts; one only keeps data, the other instructions. The latency of on–chip caches is commonly one or two cycles. The chip designers, however, already face the problem that large on–chip caches of new microprocessors running at high clock rates cannot deliver data within one cycle since the signal delays are too long. Therefore, the size of on–chip L1 caches is limited to 64 Kbyte or even less for many chip designs. However, larger cache sizes with accordingly higher access latencies start to appear.

The L1 caches are usually backed up by a *level two (L2) cache*. A few years ago, architectures typically implemented the L2 cache on the motherboard, using SRAM chip technology. Currently, there is a tendency that the L2 cache is located on–chip as well; e.g., in the case of Intel's Pentium 4. Off–chip caches are much bigger, but also provide data with lower bandwidth and higher access latency. On–chip L2 caches are usually smaller than 512 Kbyte and deliver data with a latency of approximately 5 to 10 cycles. If the L2 caches are implemented on–chip, an off–chip *level three (L3) cache* may be added to the hierarchy. Off–chip cache sizes vary from one Mbyte to 16 Mbyte. They provide data with a latency of about 10 to 20 CPU cycles[1].

[1] See also the chapter on *Algorithms for Hardware Caches and TLB* in this volume.

2.2 Locality of References

Because of their limited size, caches can only hold copies of recently used data or code. Typically, when new data are loaded into the cache, other data have to be replaced. Caches improve performance only if cache blocks which have already been loaded are reused before being replaced by others. The reason why caches can substantially reduce program execution time is the *principle of locality of references* [28] which states that recently used data are very likely to be reused in the near future. Locality can be subdivided into *temporal locality* and *spatial locality*. A sequence of references exhibits temporal locality if recently accessed data are likely to be accessed again in the near future. A sequence of references exposes spatial locality if data located close together in address space tend to be referenced close together in time.

2.3 Block Placement

Data within the cache are stored in *cache lines*. A cache line holds the contents of a contiguous block of main memory. If data requested by the processor are found in a cache line it is called a *cache hit*. Otherwise, a *cache miss* occurs. The contents of the *memory block* containing the requested word are then fetched from a lower memory layer and copied into a cache line. For this purpose, another data item must typically be replaced. Therefore, in order to guarantee low access latency, the question into which cache line the data should be loaded and how to retrieve them henceforth must be handled efficiently.

In respect of hardware complexity, the cheapest approach to implement block placement is *direct–mapping*: the contents of a memory block can be placed into exactly one cache line:

cache line address = (block address) MOD (number of cache lines)

Direct–mapped caches have been among the most popular cache architectures in the past and are still very common for off–chip caches. However, computer architects have recently focused on increasing the *set–associativity* of on–chip caches. An a–way set–associative cache is characterized by a higher hardware complexity, but usually implies higher hit rates.

The cache lines of an a–way set–associative cache are grouped into sets of size a. The contents of any memory block can be placed into any cache line of the corresponding set. The set in which the contents will be placed is determined as follows:

set address = (block address) MOD (number of sets)

Finally, a cache is called *fully associative* if the contents of a memory block can be placed into any cache line. Usually, fully associative caches are only implemented as small special–purpose caches; e.g., *translation lookaside buffers (TLBs)* [28]. Direct–mapped and fully associative caches can be seen as special cases of a–way set–associative caches; a direct–mapped cache is a 1–way set–associative cache, whereas a fully associative cache is C–way set–associative, provided that C is the number of cache lines.

2.4 Block Replacement

In a fully associative cache and in a k–way set–associative cache, a memory block can be placed into several alternative cache lines. The question into which cache line a memory block is copied and which block thus has to be replaced is decided by a *(block) replacement strategy*. The most commonly used strategies for today's microprocessor caches are *random* and *least–recently used (LRU)*. The random replacement strategy chooses a random cache line to be replaced. The LRU strategy replaces the block which has not been accessed for the longest time interval. According to the principle of locality, it is more likely that a data item which has been accessed recently will be accessed again in the near future. Less common strategies are *least–frequently used (LFU)* and *first in, first out (FIFO)*. The former replaces the memory block in the cache line which has least frequently been used, whereas the latter replaces the data which have been residing in cache for the longest time.

Eventually, the *optimal* replacement strategy replaces the memory block which will not be accessed for the longest time. It is impossible to implement this strategy in a real cache, since it requires information about future cache references. Thus, the strategy is only of theoretical value; for any possible sequence of references, a fully associative cache with optimal replacement strategy will produce the minimal number of cache misses among all types of caches of the same size [54].

2.5 Measuring and Simulating Cache Behavior

In general, *profiling tools* are used in order to determine if a code runs efficiently, to identify performance bottlenecks, and to guide code optimization [24]. One fundamental concept of any memory hierarchy, however, is to hide the existence of caches. This generally complicates data locality optimizations; a speedup in execution time only indicates an enhancement of locality behavior, but it is no evidence.

To allow performance profiling regardless of this fact, many microprocessor manufacturers add dedicated registers to their CPUs in order to count certain events. These special–purpose registers are called *hardware performance counters*. The information which can be gathered by the hardware performance counters varies from platform to platform. Typical quantities which can be measured include cache misses and cache hits for various cache levels, pipeline stalls, processor cycles, instruction issues, and branch mispredictions. Some prominent examples of profiling tools based on hardware performance counters are the *Performance Counter Library (PCL)* [8], the *Performance Application Programming Interface (PAPI)* [11], and the *Digital Continuous Profiling Infrastructure (DCPI)* (Alpha–based Compaq Tru64 UNIX only) [5].

Another approach is based on *code instrumentation*. Profiling tools such as *GNU prof* [20] and *ATOM* [19] insert calls to a monitoring library into the program to gather information for small code regions. The library routines may either include complex programs themselves (e.g., simulators) or only modify counters. Instrumentation is used, for example, to determine the fraction of the CPU time spent in a certain subroutine. Since the cache is not visible to the instrumented code the information concerning the memory behavior is limited to address traces and timing information.

Eventually, cache performance information can be obtained by *cache simulation* [29, 53] or by *machine simulation* [12, 48]. Simulation is typically very time–consuming compared to regular program execution. Thus, the cache models and the machine models often need to be simplified in order to reduce simulation time. Consequently, the results are often not precise enough to be useful.

3 General Techniques for Improving Cache Efficiency

3.1 Data Access Optimizations

Data access optimizations are code transformations which change the order in which iterations in a loop nest are executed. The goal of these transformations is mainly to improve temporal locality. Moreover, they can also expose parallelism and make loop iterations vectorizable, for example. Note that the data access optimizations we present in this paper maintain all data dependencies and do not change the results of the numerical computations[2].

Usually, it is difficult to decide which combination of transformations must be applied in order to achieve a maximum performance gain. Compilers typically use heuristics to determine whether a transformation will be effective or not. Loop transformation theory and algorithms found in the literature focus on transformations for perfectly nested loops [63]. However, loop nests in scientific codes are not perfectly nested in general. Hence, initial enabling transformations like loop skewing, loop unrolling, and loop peeling are required. Descriptions of these transformations can be found in the compiler literature [3, 6, 41, 66].

In the following, a set of loop transformations will be described which focus on improving data locality for one level of the memory hierarchy; typically a cache. As we have already mentioned in Section 1, instruction cache misses have no severe impact on the performance of numerically intensive codes since these programs typically execute small computational kernels over and over again. Nevertheless, some of the transformations we present in this section can be used to improve instruction locality as well.

[2] However, these transformations may trigger an aggressively optimizing compiler to reorder FP operations. Due to the properties of finite precision arithmetic, this may cause slightly different numerical results.

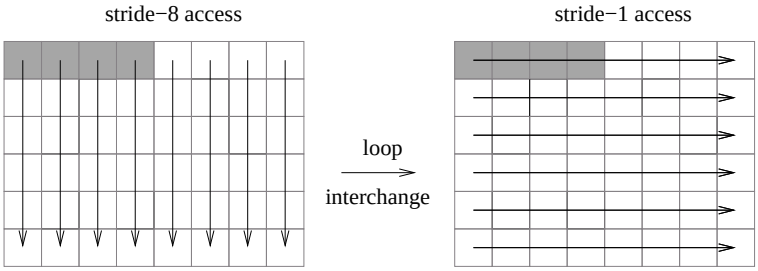


Fig. 2. Access patterns for interchanged loop nests.

Loop Interchange. This transformation reverses the order of two adjacent loops in a loop nest [1, 2, 65]. Generally speaking, loop interchange can be applied if the order of the loop execution is unimportant. Loop interchange can be generalized to *loop permutation* by allowing more than two loops to be moved at once and by not requiring them to be adjacent.

Loop interchange can improve locality by reducing the stride of an array–based computation. The stride is the distance of array elements in memory accessed within consecutive loop iterations. Upon a memory reference, several words of an array are loaded into a cache line. Accesses with large stride only use one word per cache line with arrays being larger than the cache. The other words which are loaded into the cache line are evicted before they can be reused.

Loop interchange can also be used to enable and improve vectorization and parallelism, and to improve register reuse. The different targets may be conflicting. For example, increasing parallelism requires loops with no dependences to be moved outward, whereas vectorization requires them to be moved inward.

Algorithm 3.1 Loop interchange

```
double sum;
double a[n, n];

1: // Original loop nest:
2: for j = 1 to n do
3:    for i = 1 to n do
4:       sum+ = a[i, j];
5:    end for
6: end for

1: // Interchanged loop nest:
2: for i = 1 to n do
3:    for j = 1 to n do
4:       sum+ = a[i, j];
5:    end for
6: end for
```

The effect of loop interchange is illustrated in Figure 2. We assume that the $(6, 8)$ array is stored in memory in *row major order*; i.e., two array elements are stored adjacent in memory if their second indices are consecutive numbers. The code corresponding to the left part of Figure 2, however, accesses the array elements in a column–wise manner. Consequently, the preloaded data in the cache line marked with grey color will not be reused if the array is too large to fit entirely in cache. However, after interchanging the loop nest as demonstrated in Algorithm 3.1, the array is no longer accessed using stride eight, but stride one. Consequently, all words in the cache line are now used by successive loop iterations. This is illustrated by the right part of Figure 2.

Loop Fusion. Loop fusion [16] is a transformation which takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop. The loop fusion — sometimes also called *loop jamming* — is the inverse transformation of *loop distribution* or *loop fission* which breaks a single loop into multiple loops with the same iteration space. Loop fusion is legal as long as no flow, anti, or output dependences in the fused loop exist for which instructions from the first loop depend on instructions from the second loop [3].

Fusing two loops results in a single loop which contains more instructions in its body and therefore offers increased instruction level parallelism. Furthermore, only one loop is executed, thus reducing the total loop overhead by approximately a factor of two.

Algorithm 3.2 Loop fusion

```
1: // Original code:
2: for i = 1 to n do
3:    b[i] = a[i] + 1.0;
4: end for
5: for i = 1 to n do
6:    c[i] = b[i] * 4.0;
7: end for
```

```
1: // After loop fusion:
2: for i = 1 to n do
3:    b[i] = a[i] + 1.0;
4:    c[i] = b[i] * 4.0;
5: end for
```

Loop fusion also improves data locality. Assume that two consecutive loops perform global sweeps through an array as in the code shown in Algorithm 3.2, and that the data of the array are too large to fit completely in cache. The data which are loaded into the cache by the first loop will not completely remain in cache, and the second loop will have to reload the same data from main memory. If, however, the two loops are combined with loop fusion only one global sweep through the array will be performed. Consequently, fewer cache misses will occur.

Loop Blocking. Loop blocking (also called *loop tiling*) is a loop transformation which increases the depth of a loop nest with depth n by adding additional loops to the loop nest. The depth of the resulting loop nest will be anything from $n+1$ to $2n$. Loop blocking is primarily used to improve data locality [3, 23, 52, 62, 64].

The need for loop blocking is illustrated in Algorithm 3.3. Assume that the code reads an array a with a stride one, whereas the access to array b is of stride n. Interchanging the loops will not help in this case since it would cause the array a to be accessed with stride n instead.

Tiling a single loop replaces it by a pair of loops. The inner loop of the new loop nest traverses a *block* of the original iteration space with the same increment as the original loop. The outer loop traverses the original iteration space with an increment equal to the size of the block which is traversed by the inner loop. Thus, the outer loop feeds blocks of the whole iteration space to the inner loop which then executes them step by step. The change in the iteration space traversal of the blocked loop in Algorithm 3.3 is shown in Figure 3.

A very prominent example for the effect of the loop blocking transformation on data locality is the matrix multiplication algorithm [9, 35, 37, 61]. In particular, the case of sparse matrices is considered in [42].

Data Prefetching. The loop transformations discussed so far aim at reducing the capacity misses which occur in the course of a computation. Misses which are introduced by first–time accesses are not addressed by these optimizations. Prefetching [58] allows the microprocessor to issue a data request before the

Algorithm 3.3 Loop blocking for matrix transposition

```
1: // Original code:
2: for i = 1 to n do
3:    for j = 1 to n do
4:       a[i, j] = b[j, i];
5:    end for
6: end for
```

```
1: // Loop blocked code:
2: for ii = 1 to n by B do
3:    for jj = 1 to n by B do
4:       for i = ii to min(ii + B − 1, n) do
5:          for j = jj to min(jj + B − 1, n) do
6:             a[i, j] = b[j, i];
7:          end for
8:       end for
9:    end for
10: end for
```

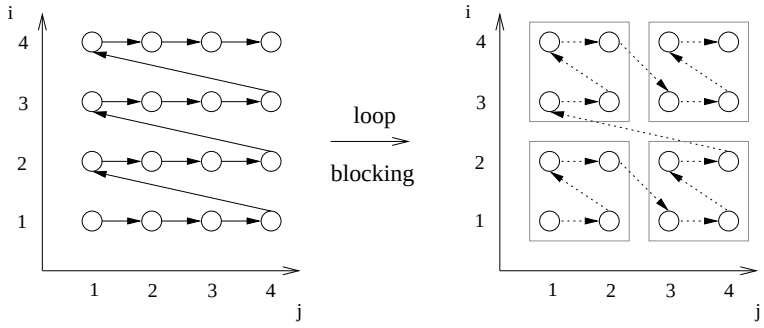


Fig. 3. Iteration space traversal for original and blocked code.

computation actually requires the data. If the data are requested early enough the penalty of cold misses as well as capacity misses not covered by loop transformations can be hidden.

Many modern microprocessors implement a *prefetch* instruction which is issued as a regular instruction. The *prefetch* instruction is similar to a load, with the exception that the data are not forwarded to the CPU after it has been cached. The *prefetch* instruction is often handled as a hint for the processor to load a certain data item, but the actual execution of the prefetch is not guaranteed by the CPU.

Prefetch instructions can be inserted into the code manually by the programmer or automatically by a compiler [34, 40, 43]. In both cases, prefetching involves overhead. The prefetch instructions themselves have to be executed; i.e., pipeline slots will be filled with prefetch instructions instead of other instructions ready to be executed. Furthermore, the memory addresses of the prefetched data must be calculated and will be calculated again when the load operation is executed which actually fetches the data from the memory hierarchy into the CPU.

Besides software prefetching, hardware schemes have been proposed and implemented which add prefetching capability to a system without the need of prefetch instructions. One of the simplest hardware–based prefetching schemes is *sequential prefetching* [51]; whenever a cache line l is accessed the cache line $l+1$ and possibly some subsequent cache lines are prefetched. More sophisticated prefetch schemes have been invented [14, 31, 33], but most microprocessors still only implement stride–one stream detection or even no prefetching at all.

In general, prefetching will only be successful if the data stream is predicted correctly either by the hardware or by the compiler and if there is enough space left in cache to keep the prefetched data together with memory references that are still active. If the prefetched data replace data which are still needed this will increase bus utilization, the overall miss rates, as well as memory latencies [13].

3.2 Data Layout Optimizations

Data access transformations have proven to be able to improve the data locality of applications by reordering the computation, as we have shown in the previous section. However, for many applications, loop transformations alone may not be sufficient for achieving reasonable data locality. Especially for computations with a high degree of conflict misses [46], loop transformations are not effective in improving performance[3].

Data layout transformations modify how data structures and variables are arranged in memory. These optimizations aim at avoiding effects like cache conflict misses and false sharing [28]. They are further intended to improve the spatial locality of a code.

Data layout optimizations include changing base addresses of variables, modifying array sizes, transposing array dimensions, and merging arrays. These techniques are usually applied at compile time although some optimizations can also be applied at runtime.

Array Padding. If two arrays are accessed alternatingly as in Algorithm 3.4 and the data structures happen to be mapped to the same cache lines, a high number of conflict misses is introduced.

In the example, reading the first element of array a will load a cache line containing this array element and possibly subsequent array elements for further use. Provided that the first array element of array b is mapped to the same cache line as the first element of array a, a read of the former element will trigger the cache to replace the elements of array a which have just been loaded. The following access to the next element of array a will no longer be satisfied by the cache, thus force the cache to reload the data and in turn to replace the data of array b. Hence, the array b elements must be reloaded, and so on. Although both arrays are referenced sequentially with stride one, no reuse of data which have been preloaded into the cache will occur since the data are evicted immediately by elements of the other array, after they have been loaded. This phenomenon is called *cross interference* [37] of array references.

A similar problem — called *self interference* — can occur if several rows of a multidimensional array are mapped to the same set of cache lines and the rows are accessed in an alternating fashion.

[3] For a classification of cache misses we again refer to the chapter on *Algorithms for Hardware Caches and TLB*.

Algorithm 3.4 Applying inter array padding.

```
1: double a[1024];
2: #ifdef APPLY_PADDING
3: double pad[x];
4: #endif
5: double b[1024];
6: for i = 0 to 1023 do
7:    sum+ = a[i] * b[i];
8: end for
```

For both cases of interference, array padding [10, 56] provides a means to reduce the number of conflict misses. *Inter array padding* inserts unused variables (pads) between two arrays in order to avoid cross interference. Introducing pads modifies the offset of the second array such that both arrays are then mapped to different parts of the cache.

Intra array padding, on the other hand, inserts unused array elements between rows of a multidimensional array by increasing the leading dimension of the array; i.e., the dimension running fastest in memory is increased by a small number of extra elements. Which dimension runs fastest in memory depends on the programming language. For example, in Fortran77 the leftmost dimension is the leading dimension, whereas in C/C++ the rightmost dimension runs fastest.

The sizes of the pads depend on the mapping scheme of the cache, the cache size, the cache line size, its set–associativity, and the data access pattern of the code. Typical padding sizes are multiples of the cache line size, but different sizes may be used as well. Array padding is usually applied at compile time. Intra array padding can, in principle, be introduced at runtime. However, knowledge of the cache architecture is indispensable, and information about the access pattern of the program will improve the quality of the selected padding size [45–47]. The disadvantage of array padding is that extra memory is required for pads.

Array Merging. This layout optimization technique can be used to improve the spatial locality between elements of different arrays or other data structures. Furthermore, array merging can reduce the number of cross interference misses for scenarios with large arrays and alternating access patterns, as we have introduced in the previous paragraph. The array merging technique is also known as *group–and–transpose* [32].

Algorithm 3.5 Applying array merging.

```
1: // Original data structure:
2: double a[1024];
3: double b[1024];

1: // array merging using multidimensional arrays:
2: double ab[1024][2];

1: // array merging using structures:
2: struct{
3:     double a;
4:     double b;
5: } ab[1024];
```

Array merging is best applied if elements of different arrays are located far apart in memory but usually accessed together. Transforming the data structures as shown in Algorithm 3.5 will change the data layout such that the elements become contiguous in memory.

Array Transpose. This technique permutes the dimensions within multidimensional arrays and eventually reorders the array as shown in Algorithm 3.6 [15]. This transformation has a similar effect as loop interchange, see Section 3.1.

Algorithm 3.6 Applying array transpose.

```
1: // Original data structure:
2: double a[N][M];

1: // Data structure after transposing:
2: double a[M][N];
```

Data Copying. In Section 3.1, loop blocking has been introduced as a technique to reduce the number of capacity misses. Research has shown [21, 62] that blocked codes suffer from a high degree of conflict misses introduced by self interference. This effect is demonstrated by means of Figure 4. The figure shows a part (block) of a big array which is to be reused by a blocked algorithm. Suppose that a direct–mapped cache is used, and that the two words marked with x are mapped to the same cache location. Due to the regularity of the cache mapping, the shaded words in the upper part of the block will be mapped to the same cache lines as the shaded words in the lower part of the block. Consequently, if the block is accessed repeatedly, the data in the upper left corner will replace the data in the lower right corner and vice versa, thus reducing the reusable part of the block.

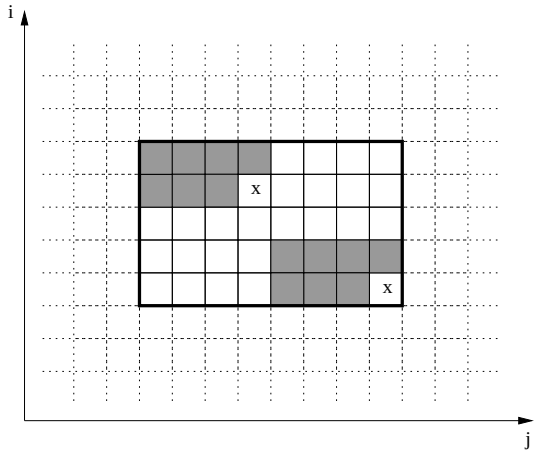


Fig. 4. Self interference in blocked code.

Lam et al. have proposed a *data copying* technique to guarantee high cache utilization for blocked algorithms [62]. The approach copies the non–contiguous data from a block into a contiguous area of memory. Hence, each word of the block will be mapped to its own cache location, effectively avoiding self interference within the block.

The technique, however, involves a copy operation which increases the total cost of the algorithm. In many cases the additional cost will outweigh the benefits from copying the data. Therefore, Temam et al. have introduced a compile time strategy to determine when to copy data [55]. This technique is based on an analysis of cache conflicts.

4 Examples of Current Research Efforts

4.1 Overview

Presently, a lot of research is focusing on data locality optimizations for various kinds of numerically intensive codes which arise in scientific computing applications. These efforts comprise the investigation of code transformations which are either based on or related to the data layout and data access optimization techniques we have presented in Section 3. See for example [7, 17, 59] for the optimization of stencil–based iterative codes on regular meshes; e.g., Jacobi's method and multigrid.

The introduction of high level code transformations by hand often turns out to be a tedious and cumbersome task. Several research groups have therefore been developing new compiler technologies in order to automatically apply sophisticated code transformations, which cannot be introduced by native compilers the vendors deliver with their machines, see [44] for example.

Other research addresses the problem of tailoring code transformations for specific platforms. Such transformations usually depend on the specification of various tuning parameters; e.g., block dimensions and array paddings. Many researchers argue that, due to the complexity of current computing platforms, abstract machine models can never cover all essential aspects in full detail, but only guide code optimization. Consequently, a suitable or even nearly optimal set of tuning parameters cannot be derived from a simplified machine model and determined analytically at compile time. Therefore, they propose approaches which are based on empirically searching the parameter spaces at compile time in order to find appropriate sets of tuning parameters. This argument for example motivated the development of the matrix multiplication package *PhiPAC* [9] and the *FFTW* software infrastructure [22] for efficiently computing Fast Fourier Transforms. It also motivates the numerical linear algebra project *ATLAS* [61], see Section 4.2 for further details.

More radical research efforts concentrate on the development of new numerical algorithms which per se contain a higher potential of data locality. This approach has an impact on both cache utilization as well as parallel scalability and parallel performance in distributed computing environments.

Of course, this survey neither can nor is intended to be complete at all. In the following, we will thus briefly present a few examples and explain how they are related to the previous description of research directions.

4.2 Automatically Tuned Linear Algebra Software (ATLAS)

The ATLAS project is based on a paradigm in building general–purpose libraries for high performance scientific computing which the developers call *AEOS (Automated Empirical Optimization of Software)*. The basic idea is to integrate code transformations and their performance evaluation into the process of building the libraries.

Currently, the ATLAS project focuses on the *BLAS* package as well as on parts of *LAPACK*[4]. The transformations address the efficient utilization of both the memory hierarchy and the FP units of the underlying processor. Therefore, the techniques which are applied while building the library cover cache optimization techniques like loop blocking as well as other optimizations techniques like loop unrolling. The latter primarily aims at reducing loop overhead and increasing instruction level parallelism [3]. See [61] for further details on ATLAS.

4.3 Efficient Iterative Solvers for Large Linear Systems

Many scientific problems involve the numerical solution of partial differential equations (PDEs). The discretization of PDEs often leads to large systems of linear equations which are typically solved using iterative numerical schemes; e.g., SOR, the conjugate gradient method as well as other Krylov subspace methods, and multigrid. See [26] for a discussion of iterative methods for large linear systems.

First considerations of data locality optimizations for iterative methods have been published by Douglas [17]. They include data locality optimizations comparable to the *loop fusion* technique described in

[4] *BLAS:* Basic Linear Algebra Subprograms, *LAPACK:* Linear Algebra PACKage.

Section 3.1. More recent work by Douglas, Hu, and others [18, 30] focuses on optimizations for multigrid methods on unstructured grids. The focus of data locality optimizations for unstructured grids is slightly different to the optimization for structured grids, since the data structures involved with unstructured grids are more complicated due to the extensive use of pointers. Thus, minimizing the latency involved with accessing data through chains of pointers becomes dominant. Hu has proposed grid partitionings and corresponding blocking strategies to improve cache behavior [30].

Research in a similar area has been reported by Keyes et al. [25]. This work focuses on the solution of large problems in computational fluid dynamics. Keyes et al. manually apply data locality optimizations to implementations of complex iterative methods based on domain decomposition approaches on unstructured grids. They achieve significant performance improvements with array merging, loop blocking, and node reordering techniques. They illustrate the efficiency of these techniques by presenting detailed miss rates for caches and TLB. Similarly, the *FEAST* project by S. Turek et al. has also been concentrating on applications in computational fluid dynamics; cache–aware iterative solvers for patch–based mesh structures have been developed [4].

The research project *DiME (Data–local Iterative Methods)* focuses on the development of cache–aware multigrid implementations [36, 59, 60]. Both data layout optimizations as well as data access optimizations have been investigated in order to enhance the cache performance of multigrid codes on structured grids. These efforts address both constant coefficient as well as variable coefficient codes in two and three dimensions. Current research activities focus on the investigation of a new non–standard multigrid approach; the combination of *patch–adaptivity* and *adaptive processing techniques*. This approach is based on the idea to split the computational domain into subdomains (patches) whose sizes are tailored for the cache size of the underlying architecture [39]. The computational process then adaptively focuses on those patches where the errors in the numerical approximations are still large and need to be further reduced [49].

Quinlan et al. [7] have introduced the term *temporal blocking* for stencil–based operations. Temporal blocking is a loop transformation comparable to loop blocking. They have demonstrated the application of temporal blocking to Jacobi's method on a regular mesh.

Finally, recent work by Sellappa et al. [50] is based on the temporal blocking idea as well. They present a two–dimensional blocking approach for the method of Gauss–Seidel based on either a lexicographic or a red–black ordering of the unknowns. This approach is comparable to the blocking techniques presented in [59, 60].

5 Conclusions

From the programmer's point of view, the most convenient approach towards the generation of highly efficient code would be a compiler performing fully automated optimizations including cache performance enhancements. Unfortunately, current compiler technology is not able to significantly optimize the data locality behavior of complex numerically intensive codes, particularly iterative methods from numerical linear algebra which contain a high potential of temporal locality.

As a consequence, sophisticated cache optimizations of numerical codes are mainly applied by hand. While this is the most promising approach in terms of execution speedup, it is often tedious and error–prone. Extended strategies such as AEOS integrate tuning and performance evaluation into the code generation process in order to automate large parts of the optimization process. However, this approach is still far away from a fully automated optimization, and further efforts are necessary to bridge the remaining gap.

Future architecture trends show the number of transistors on chip increasing beyond one billion. Computer architects have announced that most of the transistors will be used for larger on–chip caches and on–chip memory. Most of the forecast systems will be equipped with memory structures similar to the memory hierarchies currently in use.

While those future caches will be bigger and smarter, the data structures presently used in real–world scientific codes already exceed the maximum capacity of forecast cache memories by several orders of magnitude. Today's applications in scientific computing typically require several Megabytes up to hundreds of Gigabytes of memory.

Consequently, due to the similar structure of present and future memory architectures, data locality optimizations for numerically intensive codes will further on be essential for all computer architectures which employ the memory hierarchy concept.

References

1. R. Allen and K. Kennedy. Automatic Loop Interchange. In *Proceedings of the SIGPLAN Symposium on Compiler Constructs*, volume 19 of *SIGPLAN Notices*, pages 233–246, Montreal, Canada, June 1984.
2. R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
3. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, San Francisco, California, USA, 2001.
4. M. Altieri, C. Becker, and S. Turek. On the Realistic Performance of Linear Algebra Components in Iterative Solvers. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, *High Performance Scientific and Engineering Computing, Proceedings of the 1998 International FORTWIHR Conference on HPSEC*, volume 8 of *Lecture Notes in Computational Science and Engineering*, pages 3–12. Springer, March 1998.
5. J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, October 1997.
6. D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High–Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
7. F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object–Oriented Scientific Frameworks on Cache–Based Architectures. In *Proceedings of the International Conference Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, Nevada, USA, October 1998.
8. R. Berrendorf and B. Mohr. PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. Technical report, Research Centre Juelich GmbH, Juelich, Germany, September 2000. http://www.fz-juelich.de/zam/PCL/.
9. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High–Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, July 1997.
10. W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Managment. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, Arizona, USA, December 1989.
11. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
12. D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin–Madision, Madison, Wisconsin, USA, June 1997.
13. D.C. Burger, J.R. Goodman, and A. Kägi. The Declining Effectiveness of Dynamic Caching for General–Purpose Microprocessors. Technical Report CS TR-95-1261, Computer Sciences Department, University of Wisconsin–Madison, Madison, Wisconsin, USA, 1995.
14. T.-F. Chen and J.-L. Baer. Effective Hardware Based Data Prefetching for High–Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.
15. M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared–memory Machines. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, California, USA, June 1995.
16. A. Darte. On the Complexity of Loop Fusion. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 149–157, Newport Beach, California, USA, October 1999.
17. C.C. Douglas. Caching in With Multigrid Algorithms: Problems in Two Dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.
18. C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, February 2000.
19. A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 303–314, January 1995.

20. J. Fenlason and R. Stallman. *GNU gprof*. Free Software Foundation, Inc., Boston, Massachusetts, USA, 1998.
21. J. Ferrante, V. Sarkar, and W. Trash. On Estimating and Enhancing Cache Effectiveness. In U. Banerjee, editor, *Fourth International Workshop on Languages and Compilers for Parallel Computing*. Springer, August 1991.
22. M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'98)*, volume 3, pages 1381–1384, May 1998.
23. D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.
24. S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.
25. W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith. High Performance Parallel Implicit CFD. *Parallel Computing*, 27(4):337–362, March 2001.
26. W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*, volume 95 of *Applied Mathematical Sciences*. Springer, 1993.
27. J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.
28. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, second edition, 1996.
29. M.D. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transaction on Computers*, 38(12):1612–1630, December 1989.
30. J. Hu. *Cache–Based Multigrid on Unstructured Grids in Two and Three Dimensions*. PhD thesis, Department of Mathematics, University of Kentucky, Lexington, Kentucky, USA, 2000.
31. Y. Jegou and O. Temam. Speculative Prefetching. In *Proceedings of the International Conference on Supercomputing*, pages 57–66, Tokyo, Japan, July 1993.
32. T. Jeremiassen and S. Eggers. Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformation. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, California, USA, July 1995.
33. N. Jouppi. Improving Direct–Mapped Cache Performance by the Addition of a Small Fully–Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 388–397, Barcelona, Spain, June 1990.
34. A.C. Klaiber and H.M. Lewy. Architecture for Software–controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–63, Toronto, Canada, May 1991.
35. I. Kodukula, N. Ahmed, and K. Pingali. Data–Centric Multi-Level Blocking. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, Nevada, USA, June 1997.
36. M. Kowarschik, C. Weiß, and U. Rüde. Data Layout Optimizations for Variable Coefficient Multigrid. In *Proc. of the 2002 International Conference on Computational Science (ICCS 2002), Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 642–651, Amsterdam, The Netherlands, April 2002. Springer.
37. M.S. Lam, E.E. Rothberg, and M.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 (ASPLOS IV)*, pages 63–74, Palo Alto, California, USA, April 1991.
38. D. Loshin. *Efficient Memory Programming*. McGraw–Hill, 1998.
39. H. Lötzbeyer and U. Rüde. Patch–Adaptive Multilevel Iteration. *BIT*, 37(3):739–758, 1997.
40. T.C. Mowry. *Tolerating Latency Through Software–Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, March 1994.
41. S.S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, California, USA, 1997.
42. J.J. Navarro, E. Garcia-Diego, J.-L. Larriba-Pey, and T. Juan. Block Algorithms for Sparse Matrix Computations on High Performance Workstations. In *Proceedings of the International Conference on Supercomputing*, pages 301–308, Philadelphia, Pennsylvania, USA, October 1996.
43. A.K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, USA, May 1989.
44. D. Quinlan. ROSE: Compiler Support for Object–Oriented Frameworks. In *Proceedings of the 8th Workshop on Parallel Compilers (CPC2000)*, Aussois, France, jan 2000. Also in special issue of Parallel Processing Letters, Vol. 10, Nos. 2 & 3 (2000), pages 215–226.
45. G. Rivera. *Compiler Optimizations for Avoiding Cache Conflict Misses*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, Maryland, USA, 2001.
46. G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998.

47. G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the ACM/IEEE SC00 Conference*, Dallas, TX, USA, November 2000.
48. M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems & Applications*, 4(3):34–43, 1995.
49. U. Rüde. Fully Adaptive Multigrid Methods. *SIAM J. Numer. Analysis*, 30(1):230–248, February 1993.
50. S. Sellappa and S. Chatterjee. Cache–Efficient Multigrid Algorithms. In *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001), Part I*, volume 2073 of *Lecture Notes in Computer Science*, pages 107–116, San Francisco, California, USA, May 2001. Springer.
51. A.J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
52. Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, Georgia, USA, May 1999.
53. R.A. Sugumar and S.G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 24–35, Santa Clara, California, USA, May 1993.
54. O. Temam. Investigating Optimal Local Memory Performance. In *Proceedings ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Diego, California, USA, October 1998.
55. O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile–Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proceedings of the ACM/IEEE Supercomputing 1993 Conference*, Portland, Oregon, USA, November 1993.
56. J. Torrellas, M. Lam, and J. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2, pages 266–270, Pennsylvania, USA, August 1990.
57. U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.
58. S.P. Vanderwiel and D.J. Lilja. Data Prefetching Mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.
59. C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids*. PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, December 2001.
60. C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the ACM/IEEE Supercomputing 1999 Conference*, Portland, Oregon, USA, November 1999.
61. R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the International Conference on Supercomputing*, Orlando, Florida, USA, November 1998.
62. M.E. Wolf and M.S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 33–44, Toronto, Canada, June 1991.
63. M.E. Wolfe. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, California, USA, August 1992.
64. M.J. Wolfe. More Iteration Space Tiling. In *Proceedings of the International Conference on Supercomputing*, pages 655–664, Reno, Nevada, USA, November 1989.
65. M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, USA, 1989.
66. M.R. Wolfe. *High–Perfomance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, California, USA, 1996.