Using the *iblOpt* Tool for Locality Optimization of Stencil Codes

Claudia Leopold

Institut für Informatik
Friedrich-Schiller-Universität Jena
Germany
claudia@minet.uni-jena.de

Abstract. In previous work, we have suggested instance-based locality optimization (*iblOpt*) as a semi-automatic approach to restructure performance-critical code sections for better cache usage. This paper reports on recent progress in the development of an *iblOpt* tool and discusses practical aspects of tool usage. We state general guidelines for setting up a series of *iblOpt* experiments and illustrate these guidelines with the example of Gauß-Seidel kernels.

1 Introduction

On all current computers, cache misses have a high impact on program performance. Optimizing compilers successfully reduce the cache misses for many applications. Nevertheless, other applications exist for which the compiler-optimized programs are suboptimal. Examples include stencil codes such as the Jacobi and Gauß-Seidel iterative solvers [13, 16].

If compiler transformations are not sufficient for a given program, application-specific transformations can be devised manually, as it was done for stencil codes [3, 13, 15, 16]. Application-specific transformations often find their way into compilers and libraries later, but the necessity to develop such transformations for new applications will likely persist in the future.

Manual design of transformations is time-consuming and difficult. It is a creative process with little automatic support as yet. Instance-based locality optimization (*iblOpt*) [4, 7] provides a type of support that may or may not be helpful in a particular case.

We assume to be given a short, performance-critical program section. As its central idea, *iblOpt* does not optimize this program section directly, but instead considers one or several small instances thereof, which are obtained by setting loop bounds and other variable values to small constants. The *iblOpt* approach is based on the assumption that these program instances (PIs) have a similar pattern of cache usage like the original program (for a smaller cache), and thus the original program profits from the same restructurings as the PIs. Vice versa, if the PIs can not be profitably restructured, *iblOpt* concludes that the original program can not be restructured either.

We consider PIs because they have some favorable properties: 1) due to their lower complexity, they are more amenable to elaborate optimization procedures than the original code, 2) cache misses can be determined quickly through simulation (if cache parameters are scaled down in relation to the problem size), and 3) cache behavior can be visualized in detail.

The *iblOpt* approach has a wider scope than current compiler transformations insofar as it considers general restructurings as opposed to a fixed set of transformations. By general restructuring, we mean any re-scheduling of the statements in the PI (that respects data dependencies) in combination with any re-grouping of the data into memory blocks. A compiler, in contrast, refers to some pool of transformations (loop permutation, loop tiling etc.), and combines these transformations in order to improve a given program. The pool of transformations is large, but limited, and thus *iblOpt* can find additional restructurings. On the backside, programs must be relatively regular to profit from *iblOpt* because the program is modeled by PIs.

We have implemented the approach in a tool called *iblOpt*. The current prototype applies to programs that operate on arrays. Locality is the only goal of optimization. We consider a two-level memory in a uniprocessor machine, which consists of cache and main memory. In line with common notation, locality denotes the degree of concentration of the accesses to the same data (temporal locality) or to data from the same memory block (spatial locality). Cache misses are distinguished into

– *cold misses*, which correspond to the first access to a memory block and can not be avoided,
– *capacity misses*, which are due to limited cache capacity, and
– *conflict misses*, which are due to limited cache associativity.

The tool increases locality and thus reduces capacity misses. Conflict misses are out of scope and should be reduced in a postprocessing phase through tile size selection, array padding, and/or copying [14].

This paper starts with an outline of *iblOpt* in Sect. 2, which summarizes both previously published work and recent progress. In Sect. 3, we state general guidelines for *iblOpt* usage, including rules for defining PIs of a given program. Furthermore, Sect. 3 discusses a possible integration with other tools. Section 4 uses stencil codes as an example to demonstrate *iblOpt* usage. Most of the restructurings found by *iblOpt* have already been developed manually in the past. We state that such transformations can be found more easily with *iblOpt*. Furthermore, *iblOpt* provides an estimate of locality potential, and thus an *iblOpt* user is less likely to overlook relevant transformations. The paper finishes with a review of related work and conclusions.

2 The *iblOpt* Tool

The central concept of *iblOpt* is a program instance (PI). A PI is derived from a program by setting loop bounds and other control flow-relevant variable values to small constants, and then unrolling all loops completely. Thus, a PI is a sequence of statement instances (SIs), which may be elementary assignments or complex function calls. In either case, SIs are considered as units for scheduling, and each SI is characterized by the sequence of data elements that it accesses.

The *iblOpt* approach is based on the assumption that a program with a relatively regular structure is likely to profit from the same restructurings as a small instance of the same program. Thus, instead of a given program, *iblOpt* considers one or several small instances thereof, and optimizes these PIs by re-scheduling the SIs and re-grouping the data into memory blocks. The optimized PIs provide hints for the optimization of the original program.

To state it in more detail, instance-based locality optimization proceeds in three phases:

- a *specification phase*, in which the user chooses one or several PIs,
- an *optimization phase*, in which the PIs are optimized automatically, and
- a *generalization phase*, in which the user generalizes the restructurings found for the PIs to the original program.

We have recently implemented a prototype *iblOpt* tool, which supports the three phases as follows:
Specification phase: First, the user inputs the program section of interest in a restricted C-like syntax. Then, a parser analyzes the code and asks for loop bounds and other variable values that it needs to construct a PI. The user must also specify cache parameters that are scaled down in relation to the PI size, and the user must specify the optimization mode in which *iblOpt* should run (as explained in Sect. 3). Finally, the tool completely unrolls all loops, and so generates a PI.
Optimization phase: An optimization algorithm automatically restructures the PI through

- *code transformations*, which are re-schedulings of the SIs that respect data dependencies, and/or
- *data transformations*, which are re-groupings of the data into memory blocks.

The objective is optimization with respect to locality and, optionally, with respect to regularity. Regularity denotes the simplicity of a PI's structure and will be explained later.

An objective function that measures *locality* has been introduced in [4]. We have recently modified this function for use in the important special case of pure code transformations. The modified function eliminates the need for a threshold parameter and slightly improves solution quality. For data transformations, we still use the function from [4]. The new function measures the locality of a PI by

$$\sum_{reuse\ pairs} f(distance(reuse\ pair)) \ ,$$

where a reuse pair is a pair of successive accesses to the same memory block. Unlike in the function of [4], we define distance as the number of intervening accesses to *distinct* memory blocks. The measure thus corresponds to the well-known concept of reuse distance [1, 18].

For cache capacity C and line size L, function f is defined by

$$f(\mathit{distance}) = \begin{cases} 1 - 1/(C/L + 2 - \mathit{distance}) & \mathit{distance} < C/L \\ 1/\sqrt{\mathit{distance} - C/L + 4} & \mathit{distance} \geq C/L \end{cases} .$$

Thus, f is a smooth function with $f(C/L) = 0.5$, and $\lim\limits_{\text{distance} \to \infty} f(\text{distance}) = 0$. In our context, a smooth function performs better than the crisp measure of cache misses, as has been shown in [4].

With respect to SI order, a PI has a high degree of *regularity* if the PI can be written as a short sequence of nested loops. A corresponding objective function has been given in [7]. With respect to the grouping of data into memory blocks, a PI has a high degree of regularity if the data of each memory block correspond to a simple-shaped portion of a program array (such as a rectangular subblock).

At present, *iblOpt* supports regularity optimization for the data grouping in the case that memory blocks have size two. Consider a program array A and a memory block b that contains array elements $d = A[i_0, i_1, \ldots, i_r]$ and $d' = A[i'_0, i'_1, \ldots, i'_r]$. Then, we define the regularity degree of b as $\sum\limits_{k=0\ldots r} 1/(|i_k - i'_k| + 1)$, and we define the regularity degree of a PI as the sum of the regularity degrees of all memory blocks. This objective function did not receive much tuning, and thus regularity optimization is less effective for data than it is for SIs.

The optimization algorithm is a *local search* that repeatedly moves a group of successive SIs to another position in the schedule, applies loop transformations, and exchanges data between memory blocks, until the objective functions for locality and regularity can not be further improved [5]. In recent work, we have somewhat speeded up the local search algorithm. In particular, the locality objective function is now updated incrementally, and the search order has been adapted to this incremental update.

Generalization phase: After optimization, the tool reports the numbers of simulated cache misses for the original PI and for the optimized PI. These numbers suggest whether locality optimization is feasible, and how much gain can be expected. If there is locality potential, the user will probably want to restructure the code. This restructuring must be performed manually. Tool support is provided insofar as the tool outputs the optimized PIs. In particular, it presents the PIs as an SI sequence and in a structured representation. Furthermore, a visualization component graphically depicts array accesses on the screen and so illustrates the execution order of the optimized PIs. This output helps the user to recognize the transformations that improve the PIs.

In previous work, we have carried out a few experiments with a preliminary version of *iblOpt*. For instance, we have improved some two-deep loop nest by a factor of about two by exploiting non-uniform reuse [6].

3 Guidelines for *iblOpt* Usage

The optimization of a program with *iblOpt* typically requires consideration of several PIs, and of several classes of restructurings (e.g. pure code transformations). In this section, we state guidelines on how to set up a series of *iblOpt* experiments, and on how to choose the parameters that define a PI. In the course of this section, we also explain different invocation modes of *iblOpt*, which can be selected in the specification phase.

As mentioned in Sect. 2, *iblOpt* can be used to estimate locality potential, or to support manual program restructuring after potential has been discovered. These two use cases are supported by different invocation modes. Technically, the modes differ in whether regularity optimization is included in the local search algorithm or not. At present, restructuring mode is slower than potential-estimation mode by a factor of order 1000. While the exact value is implementation-dependent, the fact that there is a difference is inherently due to a higher problem complexity. In potential-estimation mode, the current *iblOpt* prototype processes a PI of 500 SIs within a few seconds (on a 1 GHz AMD Athlon PC, for the 1D code from Sect. 4).

As a possible usage scenario of potential-estimation mode, we envision the integration of *iblOpt* into a performance analysis and optimization tool in the spirit of FINESSE [12]. Such a tool would invoke potential-estimation mode of *iblOpt* automatically after having found a performance-critical code section with a high number of cache misses. In potential-estimation mode, *iblOpt* is easy to use. We think that the corresponding calls can be automated in the near future, based on the guidelines for parameter selection given below or a refined version thereof. A tool that incorporates *iblOpt* should have access to feedback from previous program runs to select PI parameters. When the tool detects locality potential, it would report the potential to the user and offer to run *iblOpt* in restructuring mode. Invoking *iblOpt* on a regular basis through integration into a tool would help to discover weaknesses in compiler-optimized programs, and thus contribute to a steady improvement of the compiler techniques.

In the following, we assume that *iblOpt* is used as a standalone facility, and that we are given a program section to be restructured if profitable. We organize *iblOpt* experiments into three phases, to minimize the use of the time-consuming restructuring mode:

1) Determine locality potential, thereby run *iblOpt* in potential-estimation mode,
2) Decide whether restructuring is worthwhile, and
3) Devise a restructuring, thereby run *iblOpt* in restructuring mode.

Orthogonal to potential-estimation and restructuring modes, *iblOpt* can be invoked in code-transformation, data-transformation, and mixed code/data-transformation modes. In phase 1, we investigate four types of locality potential in separate groups of experiments:

a) Potential of temporal locality,

b) Potential of spatial locality that can be used with standard, (e.g. row-major) storage order,
c) Potential of spatial locality that can be used with another common storage order, and
d) Potential of spatial locality that can be used with an unconventional storage order.

Experiments for potential of type d) are run with mixed mode, whereas the other experiments are run with code-transformation mode (as explained below). Mixed mode has a larger search space, and thus the optimization takes longer than in cases a)–c), and the results tend to be farther away from the global optimum. The division of experiments into four groups gives us a more detailed picture of locality potential. After phase 1, we know which types of locality are profitable, and can restrict the experiments in phases 2 and 3 to these types. So we save both computer time for running less experiments with possibly smaller PIs, and human time for making restructuring easier. In general, it is easiest to restructure a program for a)-potential, and hardest to restructure a program for d)-potential, in the sense that it is easier or harder for the user to recognize the corresponding transformations.

Experiments for a)-potential use $L = 1$. Experiments for c)-potential are run with a special invocation mode of *iblOpt* that iterates through all combinations of common orders by which the program arrays can be initialized. At present, we consider row-major, column-major, and rectangular-subblock orders for two-dimensional arrays; analogous orders for multi-dimensional arrays; and block/cyclic orders for one-dimensional arrays. After each initialization, *iblOpt* optimizes the given PI by code transformations, and finally outputs the best overall result.

In each of the four groups of experiments in phase 1, we consider one or several PIs. Additionally, we run the optimization algorithm repeatedly for each PI, to improve the chance that local search finds a global optimum. For each PI, the first run starts from the original SI order, and the others start from random SI orders. The exact number of experiments depends on the context in which *iblOpt* is run, since any particular application calls for a different tradeoff between optimization expense (number of runs etc.) and locality gain (likelihood that all relevant transformations are found).

In the following, we list rules for choosing parameters that define a PI and corresponding cache. The rules reflect our experiences in using the tool. In the rules, N stands for parameters that influence an array size, and T stands for parameters that influence the iteration count of a loop. A PI may have several parameters of types N and T; the rules are meant to apply to all of them. As before, C denotes cache capacity and L denotes cache line size:

– **Minimum rule:** Start experiments with the smallest values permitted by the rules and observe optimization time. Later, increase parameters as appropriate. Small instances can be optimized faster, and it is more likely that local search finds a global optimum.

- **L = 2 rule**: For potential of types b)–d), use $L = 2$, since two is the smallest value that captures spatial locality. Additional experiments with $L = 3$ may increase confidence in the final result.
- **L – C rule**: $L \,|\, C$ and $C > L$.
- **L – N rule**: L should evenly divide array dimensions to avoid a special treatment of borders that complicates an optimized PI's structure.
- **C – SI rule**: The cache should be large enough to simultaneously hold all data that are accessed by two SIs, even if the data belong to different memory blocks. This rule captures reuse between successive SIs. The requirement is pessimistic insofar as the data sets of successive SIs typically overlap, and optimistic insofar as reuse may involve more than two SIs. The smoothness in the locality objective function favors reuse pairs that are close even if the distance is slightly larger than C. Therefore *iblOpt* can warn about unfortunate C values. Nevertheless, it is useful to run experiments with a larger C value, as well, if time permits.
- **N – C rule**: The total number of data must be larger than C, at least by factor 2.
- **T rule:** Parameters of type T should be at least 2, better 4, so that patterns can be recognized.
- **Feedback rule:** The relative size of parameters should correspond to the real case. In particular, the cache should hold an array or a row of an array if and only if the same holds for typical program runs.

After phase 1, we know whether the given program has locality potential, and whether the use of this potential requires data transformations. In phase 2, we only consider the promising locality types, and carry out some further experiments to decide whether it is worthwhile to tackle the restructuring process. In particular, phase 2 compares the *iblOpt* output to the corresponding instance of a compiler-optimized program (if this has not yet been done before), and estimates scalability of the potential. For the latter, we simply compare the locality potential of PIs with different parameter sizes. The result of phase 2 is a decision on whether to proceed with phase 3.

Phase 3 aims at finding a restructuring that exploits the potential discovered before. We start with an inspection of the phase 1 and 2 outputs and observe for which parameter constellations the gain is highest. Then, we formulate one or several PIs with these properties. The PIs may be different from the PIs of phase 1 since the minimum rule is even more pressing now. We make sure that the PIs have a significant locality potential before invoking restructuring mode.

In restructuring mode, *iblOpt* is preferably run over night so that multiple runs of the local search algorithm can be accomplished. Then, we pick the most interesting optimized PIs in terms of cache misses and number of lines (in a structured representation) and inspect these PIs with the visualization component of *iblOpt*. Hopefully, patterns can be recognized that are responsible for the reduction in cache misses. Possibly, additional PIs have to be considered. Phase 3 requires creative work and is therefore not amenable to automatization.

Finally, the user restructures the program according to the patterns found for the PIs, and makes sure that data dependencies are respected. Successful locality optimization is no guarantee for speedup since other performance factors such as the number of mispredicted branches may have an opposite impact. Hence, the gain must be examined experimentally, as it is common practice for manual restructurings.

4 Optimization of Stencil Codes

In this section, we apply the guidelines to 1D, 2D, and 3D Gauß-Seidel kernels. As before, C denotes cache capacity and L denotes cache line size.

Optimization of 1D Gauß-Seidel Code with Time Loop

We consider the following code:

```
for (t=0; t<T; t++)
  for (i=1; i<N-1; i++)
    A[i] += A[i-1] + A[i+1];
```

Phase 1: For potential of type a), the C–SI rule implies $C \geq 6$, the N–C rule implies $N \geq 2C$, and the T rule implies $N \geq 6$ and $T \geq 4$. The feedback rule is ignored, since this code does not usually need cache optimization in practice. We start with $N = 12, T = 4, C = 6$. Since optimization runs fast, larger parameters are considered thereafter.

For potential of type b), the C–SI rule implies $C \geq 8$ since each SI accesses two memory blocks, that is, four data elements. For potential of types c), d), the C–SI rule implies $C \geq 12$ since each SI accesses up to three memory blocks, that is, six data elements. For better comparability among b)–d), we set $C = 12$ in most PIs. Table 1 summarizes the PIs and optimization results.

Table 1. Cache misses of 1D Gauß-Seidel PIs. N, T are program parameters, C denotes cache capacity, and L denotes cache line size. The numbers are simulated cache misses for the input program, the compiler-optimized program, and *iblOpt*-optimized PIs, respectively. The *iblOpt* results correspond to the best value that is obtained with 10 initializations per PI.

N	T	C	L	Input	Compiler	iblOpt a) or b)	iblOpt c)	iblOpt d)
12	4	6	1	48	18	18	–	–
12	8	6	1	96	40	36	–	–
24	4	9	1	96	24	24	–	–
16	4	8	2	32	8	8	–	–
24	5	12	2	60	12	12	12	12
24	12	12	2	144	27	24	24	35

Phase 2: We refer to the following compiler-optimized program after skewing and tiling [17]:

```
for (ii=1; ii<N-1+T; ii+=s)
  for (t=0; t<T; t++)
    for (i=max(ii,t+1); i<min(ii+s,N-1+t); i++)
      A[i-t] += A[i-t-1] + A[i-t+1];
```

The values in Tab. 1 are optima for tile sizes $s = 1 \ldots N$. Thus, these values are somewhat optimistic (compilers select a particular size, which may be sub-optimal). The locality potential seems to scale since improvements are achieved for large values of T.

Phase 3: We choose the PIs such that T is large: $N = 12$, $T = 8$, $C = 6$, $L = 1$ and $N = 16$, $T = 8$, $C = 8$, $L = 2$.

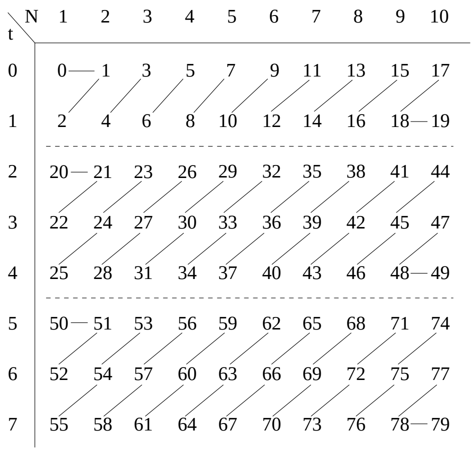


Fig. 1. Optimized PI for 1D Gauß-Seidel code with $N = 12$, $T = 8$, $C = 6$, $L = 1$ after running *iblOpt* in restructuring mode.

The first instance yields a clearly structured optimized PI that is depicted in Fig. 1. The second PI yields a less clear output, so we repeat the experiment with another T (not further discussed here). In Fig. 1, numbers are to be read as that the first SI (number 0) of the optimized PI updates $A[1]$ for $t = 0$, the next SI (number 1) updates $A[2]$ for $t = 0$, and so on. It can be observed that the optimized PI is blocked along the t-axis. Within each horizontal stripe, the execution order resembles that of the compiler-optimized instance (given in Fig. 2), except that tile width is one. The compiler-optimized instance induces 40 cache misses, and the *iblOpt* instance induces 36 cache misses. The *iblOpt* order corresponds to the following optimized program:

```
for (tt=0; tt<T; tt+=s)
  for (i=tt+1; i<N-2+min(tt+s,T); i++)
    for (t=tt; t<min(tt+s,T); t++)
      if (i>=t+1 && i<N-1+t)
```

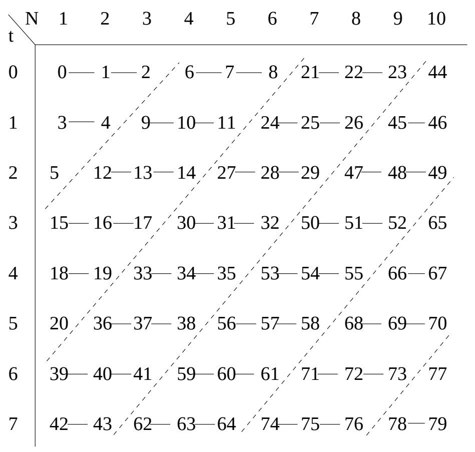


Fig. 2. Compiler-optimized PI for 1D Gauß-Seidel code with $N = 12$, $T = 8$, $C = 6$, $L = 1$.

```
A[i-t]+=A[i-t-1]+A[i-t+1];
```

To understand the difference, we look at where cache misses arise. With *iblOpt* order, they arise at horizontal stripe boundaries, and with compiler order, they arise at tile boundaries. The fact that there are no cache misses within the horizontal stripes depends on tiles being thin. The *iblOpt* order reuses data of neighbored tiles within a stripe and thus, in a sense, overlaps tiles.

For analysis, we assume an LRU cache replacement policy and cache line size 1. Then, the optimal strip width is $C-3$ for the *iblOpt* scheme, and the optimal tile width is C–3 for the compiler scheme, as well. Since capacity misses occur along stripe and tile boundaries, respectively, the *iblOpt* scheme takes about $2 \cdot N \cdot \lfloor T/(C-3) \rfloor$ capacity misses, and the compiler scheme takes about $2 \cdot T \cdot \lfloor N/(C-3) \rfloor$ capacity misses. For $N \gg T$, the first value is typically smaller than the second, especially if $\lfloor T/(C-3) \rfloor$ equals 1 or 2. A similar analysis applies to $L > 1$.

We also compared the schemes experimentally on a Linux PC with 1.4 GHz AMD Athlon(tm) XP 1600+ processor. For a 256 kByte L2 cache, we used $N = 200\ 000$, $T = 40\ 000$, and array element type double. After compilation with gcc -O4, we measured running times 327 sec for the input program, 234 sec for the compiler-optimized program, and 211 sec for the *iblOpt*-optimized program (for optimal tile sizes as determined experimentally).

Optimization of 2D Gauß-Seidel Code without Time Loop

We consider the following code:

```
   for (i=1; i<N-1; i++)
```

```
    for (j=1; j<N-1; j++)
      A[i][j] += A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1];
```

Phase 1: For potential of type a), the C–SI rule implies $C \geq 10$, and the N–C rule implies $N^2 \geq 2C$. It has been observed previously [13] that cache optimization of 2D codes is not an issue unless $N > C/2$. Therefore we assume $N > C/2$, even though many programs do not fulfill this assumption. We start with $N = 6$, $C = 10$ and, since optimization runs fast, then consider $N = 10$, $C = 10$ and $N = 14$, $C = 10$.

For potential of type b), the C–SI rule requires $C \geq 16$, so we consider $N = 8$, $C = 16$, $L = 2$. For comparability with c), d), we proceed with $N = 12$, $C = 20$, $L = 2$, and $N = 16$, $C = 20$, $L = 2$. In all cases, *iblOpt* reduces cache misses by a factor of about 1.5.

Phase 2: We refer to the following compiler-optimized program after tiling, and set s to the optimum from $s = 1 \ldots N$:

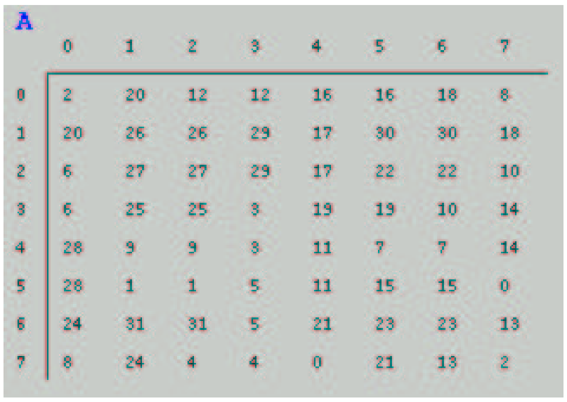
```
for (jj=1; jj<N-1; jj+=s)
  for (i=1; i<N-1; i++)
    for (j=jj; j<min(jj+s, N-1); j++)
      A[i][j] += A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1];
```

We make the following observations:

– In a)–c), the compiler-optimized PIs take the same number of cache misses as the *iblOpt*-optimized PIs, or somewhat less. This outcome suggests that tiling is optimal. The differences are due to *iblOpt* running a local search.
– Cache misses are minimized for $s = 3$ if $C = 10$, for $s = 4$ if $C = 16$, and for $s = 5$ if $C = 20$. Taking into account that a tile's working set comprises neighbors to be read in addition to the data that are updated, these values for s correspond to non-square tiles. Compiler heuristics, in contrast, favor square tiles in absence of conflict misses. Non-square tiles reflect a difference from the traditional tiling scheme that loads a block of data into cache, works with these data, loads the next block, and so on. Non-square tiles exploit overlap between neighbored tiles, similar as the 1D scheme above. Note that *iblOpt* optimization was not required to recognize this modification, but the approach of considering PIs was helpful as it allowed us to quickly determine cache misses for all tile sizes.
– In d), the *iblOpt*- optimized PIs take somewhat less cache misses than their compiler-optimized counterparts (86 vs. 95 for $N = 12$, $C = 20$, $L = 2$ and 170 vs. 180 for $N = 16$, $C = 20$, $L = 2$). We conclude that there is potential beyond tiling, which can be exploited through data transformations.

Phase 3: Since combined code/data transformations are hard to recognize, we first investigate whether pure data transformations are sufficient to realize the reduction in cache misses. We start from the tiled program and get about the same results as in d): 85 for $N = 12$, $C = 20$, $L = 2$, and 166 for $N = 16$, $C = 20$, $L = 2$. Hence, pure data transformations are sufficient.

Since regularity optimization for the data grouping is not yet mature in our present prototype, we have to look at several optimized PIs until we recognize the pattern that is responsible for the reduction in cache misses. It is a mixed column-row layout that stores data row-wise in the interior of a tile and column-wise at vertical tile boundaries. Figure 3 depicts an optimized PI from which this layout can be recognized. We have analyzed the layout in [8], where we show that it slightly reduces cache misses as compared to both column-major and row-major layouts.



A	0	1	2	3	4	5	6	7
0	2	20	12	12	16	16	18	8
1	20	26	26	29	17	30	30	18
2	6	27	27	29	17	22	22	10
3	6	25	25	3	19	19	10	14
4	28	9	9	3	11	7	7	14
5	28	1	1	5	11	15	15	0
6	24	31	31	5	21	23	23	13
7	8	24	4	4	0	21	13	2

Fig. 3. Layout for tiled program with $N = 8$, $s = 3$, $C = 16$, $L = 2$.

Optimization of 3D Gauß-Seidel Code without Time Loop

We consider the following code:

```
for (i=1; i<N-1; i++)
  for (j=1; j<N-1; j++)
    for (k=1; k<N-1; k++)
      A[i][j][k] += A[i-1][j][k] + A[i+1][j][k] + A[i][j-1][k]
                  + A[i][j+1][k] + A[i][j][k-1] + A[i][j][k+1];
```

Phase 1: For potential of type a), the C–SI rule implies $C \geq 14$, the N–C rule implies $N^3 \geq 2C$, the T rule implies $N \geq 6$, and the feedback rule implies $N < C < N^2$ [13]. For potential of types c), d), the C–SI rule implies $C \geq 28$. This leads us to the PIs in Tab. 2.
Phase 2: The values in Tab. 2 refer to the following compiler-optimized program after tiling, with sj, sk set to the optima from $[1 \ldots N]$:

```
for (jj=1; jj<N-1; jj+=sj)
  for (kk=1; kk<N-1; kk+=sk)
```

Table 2. Cache misses of 3D Gauß-Seidel PIs. Notation as in Tab. 1.

N	C	L	Input	Compiler	iblOpt a) or b)	iblOpt c)	iblOpt d)
6	15	1	352	260	239	–	–
6	20	1	256	224	226	–	–
7	14	1	675	545	491	–	–
8	30	1	792	648	660	–	–
6	28	2	168	168	151	150	102
8	28	2	720	513	470	462	400

```
for (i=1; i<N-1; i++)
  for (j=jj; j<min(jj+sj, N-1); j++)
    for (k=kk; k<min(jj+sk, N-1); k++)
      A[i][j][k] += A[i-1][j][k] + ...
```

We make the following observations:

– There is some potential of type a), but it does not seem to scale. For completeness, we investigate this potential in phase 3, but a typical *iblOpt* user would ignore it.
– There is some potential of type b).
– There seems to be some potential of type c), but actually this is not the case: In b), the local search algorithm is run with 10 initializations per PI, and in c) it is run with 80 initializations per PI (10 for each of the 8 combinations of storage orders). For the given code, all storage orders are equivalent, and so the higher number of initializations gives c) an advantage. If we repeat b) with 80 initializations, we get the same results as in c).
– There is some potential of type d).

Phase 3: Potential of type a): Optimization of $N = 6, C = 15$ in restructuring mode yields a tiled PI with 2×2 tiles. It differs from the compiler scheme only in the tile-internal access order, and this order is the same for all tiles. A closer look at this order reveals that it eliminates long reuse distances within the tile. This is profitable for very small caches, but makes no difference if the cache holds the complete working set of a tile. Thus the modification does not scale.
Potential of type b): For $N = 6$, $C = 28$, $L = 2$, restructuring mode yields a PI that is tiled along the i- and j-axes. Along k-axis, the "tiles" are L-shaped instead of rectangular such that memory blocks are never cut by tile boundaries. This observation is actually counterintuitive since a memory block that is cut incurs one capacity miss, but a pair of memory blocks that are not cut incurs two capacity misses [9]. A closer look at the sequence of memory accesses reveals that the advantage of non-cut blocks can be attributed to neighbored blocks that are read but not updated. For large caches, only few blocks are removed from cache in-between their read and update (the blocks at the corner of a tile), and thus this potential does not scale either.
Potential of type d): Again, we start from the tiled program, and find that

pure data transformations are sufficient. The layout for $N = 8$, $C = 28$, $L = 2$ and optimal tile size $sj = 2$, $sk = 3$ leads to the following observations:

– The boundary and interior elements of A, respectively, are stored in different sets of memory blocks.
– Elements $A[i][j][1]$ and $A[i][j][2]$ as well as elements $A[i][j][5]$ and $A[i][j][6]$ are often stored in the same memory block, but elements $A[i][j][3]$ and $A[i][j][4]$ (which are updated in different tiles) are not.
– Element $A[i][j][3]$ is often stored in the same memory block as $A[i-1][j][3]$, $A[i][j-1][3]$, $A[i-1][j+1][3]$, or $A[i-1][j-1][3]$ (similarly for $A[i][j][4]$).
– In general, most memory blocks contain data that are updated in the same tile.

These observations lead us to the conjecture that a mixed column-row layout reduces cache misses. In this layout, memory blocks contain elements $A[i][j][k]$, $A[i][j][k+1], \ldots$ in the interior of a tile and at tile boundaries in j-direction, and memory blocks contain elements $A[i][j][k]$, $A[i][j+1][k], \ldots$ at tile boundaries in k-direction.

5 Related Work

The *iblOpt* restructurings resemble application-specific transformations that have been devised previously. In particular, Leiserson et al. [3] use blocking along the t-loop, and Sellappa [15] and Rivera and Tseng [13] use overlapping tiles in a slightly different context.

In [8, 9], we have analyzed tiling transformations for stencil codes. These papers do not refer to *iblOpt*, but instead discuss the restructurings from an application-oriented point of view. The main results are lower bound proofs, which show that tiling gets close to optimum in terms of capacity cache misses.

Compiler optimizations for locality have received much attention in previous research. Traditionally, code transformations such as loop permutation, distribution, skewing, and tiling have been considered [17]. Especially in recent years, code transformations have increasingly been combined with data transformations [2]. Moreover, limitations to unimodular transformations and perfect loop nests have been overcome [10]. In addition to the formalism-based approaches, a number of heuristics have been suggested [11].

6 Conclusions

This paper has explained how the *iblOpt* tool supports manual program restructuring for locality. We have briefly described our current prototype and outlined some recent improvements. In the main part of the paper, we have stated general guidelines for *iblOpt* usage, and applied these guidelines to the optimization of stencil codes. Future work should investigate other codes and thereby refine the guidelines, as well as further improve the tool with respect to optimization time and output quality.

References

[1] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 205–217, 1995.

[2] M. T. Kandemir. A compiler technique for improving whole-program locality. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 179–192, 2001.

[3] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences*, 54(2):332–344, Apr. 1997.

[4] C. Leopold. Arranging statements and data of program instances for locality. *Future Generation Computer Systems*, 14:293–311, 1998.

[5] C. Leopold. Generating structured program instances with a high degree of locality. In *Proc. Euromicro Workshop on Parallel and Distributed Processing*, pages 267–274. IEEE Press, 2000.

[6] C. Leopold. Exploiting non-uniform reuse for cache optimization: A case study. In *Proc. ACM Symp. on Applied Computing*, pages 560–564, 2001.

[7] C. Leopold. Structuring statement sequences in instance-based locality optimization. *Future Generation Computer Systems*, 17:425–440, 2001.

[8] C. Leopold. On optimal locality of linear relaxation. In *IASTED Int. Conf. on Applied Informatics, Proc. Parallel and Distributed Computing and Networks*, pages 201–206, 2002.

[9] C. Leopold. Tight bounds on capacity misses for 3D stencil codes, 2002. To appear in Proc. Int. Conf. on Computational Science.

[10] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proc. ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP)*, pages 103–112, 2001.

[11] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[12] N. Mukherjee, G. D. Riley, and J. R. Gurd. FINESSE: A prototype feedback-guided performance enhancement system. In *Proc. Euromicro Workshop on Parallel and Distributed Processing*, pages 101–109. IEEE Press, 2000.

[13] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Proc. SC'2000*. Available at http://www.supercomp.org.

[14] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th Int. Conf. on Compiler Construction*, pages 168–182. Springer LNCS 1575, 1999.

[15] S. Sellappa. Cache-efficient multigrid algorithms. Master's thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 2000.

[16] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proc. of the Supercomputing Conf.* Available in ACM Digital Library, 1999.

[17] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 30–44, 1991.

[18] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In same proceedings.