Generating Structured Program Instances with a High Degree of Locality

Claudia Leopold
Fakultät für Mathematik und Informatik
Friedrich-Schiller-Universität Jena
07740 Jena, Germany
claudia@informatik.uni-jena.de

Abstract

Memory hierarchy-consciousness is an important requirement for the design of high-performance programs. We describe a tool that supports the programmer in restructuring performance-critical code sections. The tool works with small program instances, which are obtained by fixing program parameters such as loop bounds, and rewriting the program as an operation sequence. The tool automatically reorders the operations for better locality, and respects data dependencies. It outputs the optimized program instance in a structured form. The user finally recognizes the locality-relevant structure and generalizes it to the program.

The paper focuses on recent advances in the development of our method. In particular, we introduce a hierarchical clustering scheme that highlights operation subsequences with much data reuse. The scheme is applied to the generation of structured optimized program instances in which the locality-relevant structure is easy to recognize. Experimental results are included.

1. Introduction

Due to the large and growing gap between processor speeds and memory speeds, the design of high-performance programs requires to be conscious of the memory hierarchy. In particular, one should avoid frequent access to the main memory, and instead try to reuse cached data as often as possible. Memory hierarchy-consciousness is a requirement for sequential and even more for parallel computation. Compiler optimizations can help, but the scope of most compiler optimizations is limited to nested loops and to a few transformations under consideration. Hence, memory hierarchy-conscious program restructuring is often a task for the programmer or algorithm designer, a task that can be time-consuming and difficult.

This paper suggests a tool that helps the programmer. The tool automatizes one of the human activities in algorithm/program design: developing solution ideas for a given problem by looking at small-input-size instances of the problem and thinking about efficient solutions for these instances. We believe the automatization of human activities to be crucial for closing the still large gap between the kind of optimizations that can be applied automatically and those that have to be applied manually. The work described here is a step in this direction.

Our work deals specifically with locality optimization in memory hierarchies. The distribution of data and computation in shared- and distributed memory parallel computers raises similar questions ([1] vs. [2]). We restrict ourselves to the simple case of sequential memory hierarchies here, but think that our approach can be generalized to the more involved case of interprocessor communication.

Locality is a gradual property of programs that reflects the level of concentration of the accesses to the same memory block, during program execution. A higher level of concentration corresponds to a higher degree of locality, and in tendency causes fewer cache misses, page faults etc.

The major component of our tool is an algorithm for the optimization of program instances (PIs). A PI is derived from a program, or typically from a performance-critical section of code, by fixing the input size and possibly more parameters. We fix those parameters that are required to unroll loops and resolve recursions in a way that the PI becomes a sequence of elementary or complex operations. This operation sequence is the starting point for the optimization algorithm, which reorders the operations and reassigns the data to memory blocks with the objective of maximizing locality. The rearrangement process is detailed later. Although the final arrangement found refers to a particular program instance, it gives hints for a restructuring of the program in general. It is the user's task to accomplish the generalization step from the instance to a program. Tool

support for this step is provided in the form of a structured representation of the output operation sequence. We call our method Instance-Based Locality Optimization (IBLOpt).

This paper reflects the current state of the development of IBLOpt, with focus on recent advances. In particular, we introduce a hierarchical clustering scheme that highlights operation subsequences with much data reuse. We use the clustering scheme to establish a *locality-aware* structured representation.

The paper starts with an overview of related work in Sect. 2. Next, previously published work on IBLOpt [4, 5] is summarized in Sect. 3, to lay the groundwork for the rest of the paper. Sections 4 and 5 describe our recent advances, where Sect. 4 introduces the hierarchical clustering scheme, and Sect. 5 uses the scheme to improve the structured output. Section 6 gives experimental results for four example programs: a two-deep perfect loop nest, matrix transpose, matrix multiplication and mergesort. Section 7 finishes with conclusions.

2. Related work

There are two major areas of related work: compiler optimizations for locality, and memory hierarchy-conscious algorithm design.

Compiler optimizations for locality are an important research subject. A survey is, for instance, given in [6]. Most compiler optimizations use heuristics that reduce the number of cache misses via transformations such as loop permutation, loop distribution and loop tiling. The techniques for sequential memory hierarchies do in general resemble those suggested for symmetric multiprocessors and distributed-memory parallel computers (e.g. [2]). While earlier work has focused on code restructurings (e.g. [11]), more recent work also considers data reassignments, that is, storing arrays row-wise vs. column-wise, etc. (e.g. [1]). Most of the work is restricted to loop nests, where arrays are accessed via affine index expressions. It is a characteristic feature of the compiler optimization approach that program restructurings are based on a small set of transformations under consideration. If a program needs other techniques for improvement, the improvement will not be found. It is a prerequisite that compiler optimizations can be executed automatically and fast.

Memory hierarchy-conscious algorithm design has developed into an active research area during the last years. Its focus is on external memory algorithms (see [10] for a recent survey), and there is also some work on caches (e.g. [3]). The scope of this work is wider than the scope of our method, since locality optimization is combined with other tasks, particularly with the establishment of an algorithm's basic structure (choice of the operations to be carried out) and with the choreography of the data movement. The merit of our method is the provision of *automatic* support to the otherwise manual process of developing algorithmic ideas.

3. Instance–based locality optimization

3.1. Overview and tool

IBLOpt is an intermediate approach between algorithm design and compiler optimizations. It is neither restricted to a finite set of transformations nor to loop programs, and hence it has a broader applicability than current compiler optimizations. On the other hand, the scope of IBLOpt is narrower than the scope of algorithm design. IBLOpt has a higher time consumption than compiler optimizations, but can save human time during algorithm design.

The central concept of IBLOpt is that of a program instance (PI). A PI comprises a sequence of elementary or complex operations called statement instances (SIs). Each SI operates on a sequence of instantiated data (DIs). For data, instantiated means that the location is non-variable, e.g. $A[3]$ is a DI whereas A[i] is not. For simplicity, we assume all DIs to be the same size. Examples of SIs are "$A[5]$+=1" and "Merge sorted run at $A[1] \ldots A[4]$ with sorted run at $A[5] \ldots A[8]$". The SIs are taken as undivisible. We assume that there are no control dependencies between SIs, i.e., each SI is carried out exactly once, and branching is restricted to occurring within SIs. In addition to the SI sequence, a PI comprises an assignment of DIs to memory blocks. The assignment specifies which data are stored together in a memory block, it does neither specify the order within the block nor the identity of the block.

Our tool works in three phases: an instantiation, an optimization, and a generalization phase. In the instantiation phase, the user specifies the PI. Therefore, the user inputs the code section to be optimized, fixes the control flow-relevant parameters, and chooses the granularity of the SIs. Then, the system rewrites the PI as a sequence of SIs, by unrolling loops and resolving recursions. A data dependence graph [6] is derived automatically, too. In IBLOpt it is a directed acyclic graph, referred to as *dag*.

Next, in the optimization phase, the SIs are automatically reordered and the DIs are reassigned to memory blocks so that the PI is improved with respect to locality. The rearrangement is controlled by a local search algorithm that strictly obeys the data dependencies. To support the generalization phase, a second objective is regularity. Striving for regularity means that we favour operation sequences that can be written in a compact and intuitively appealing form using loops. The local search algorithm is detailed in Sect. 3.2. Its output is a locally optimal SI ordering for the PI, together with a structured representation thereof. In the structured representation, we allow for-loops with constant bounds.

Finally, in the generalization phase, the user inspects the SI ordering, to recognize the structural differences that make the suggested ordering superior to the initial ordering. After having found these differences, the user generalizes them to the program and writes down the improved program. The user can assess the quality of the new program by running the optimization algorithm on a different instance thereof.

Along with the PI, the user must also specify memory hierarchy parameters C, L. They denote the capacity and line size of a small hypothetical cache, which is assumed to be fully-associative and LRU. The parameters are used in the optimization process, and they are used to quantitatively assess the locality of our output PIs.

3.2. The optimization algorithm for program instances

The input of the optimization phase consists of a set S of SIs, a set D of DIs, a data dependence graph *dag*, and the memory hierarchy parameters C, L. The SIs are characterized by functions *SD*: $S \to D^*$, *op*: $S \to \mathbb{N}$ and V: $S \to \mathbb{Z}^*$. Here, *SD* assigns to s the sequence of DIs that are accessed by s. Further, *op*(s) characterizes the operation carried out by s, and $V(s)$ is a sequence of numbers that appear explicitly in the notation for s. That is, *op* and V do together encode the textual string that denotes the operation in the PI. A more detailed explanation can be found in [4, 5]. The function *SD* is used for locality optimization, while *op* and V are needed for producing a structured output.

The output of the optimization phase comprises a schedule, which is an SI sequence where each $s \in S$ appears once, and a data assignment function *DB*, where *DB*(d) is the number of the memory block to which $d \in D$ is assigned. The schedule is output in two forms: as an operation listing, and in a structured representation consisting of nested and consecutive for-loops with constant bounds. The optimization problem is constrained by the *dag*.

There are two objective functions: one for locality and one for regularity. The functions are combined in a way that is described later.

The locality function was introduced in [4], where two candidate functions were compared: OFL_{loc} and OFL_{cc}. The function OFL_{cc} measures the number of simulated cache misses, referring to the hypothetical parameters C, L. The function OFL_{loc} quantifies the intuitive notion of locality from Sect. 1. OFL_{loc} was established empirically and has quite a complex definition [4], omitted here. Briefly, OFL_{loc} is a sum where each summand assesses the access distance of a reuse tuple, which is a pair of successive accesses to the same memory block. The assessments follow the principle *the-closer-the-better*, which is in contrast to the *either-close-enough-or-not*–principle of OFL_{cc}. In [4], it was shown that OFL_{loc} outperforms OFL_{cc} with respect to the solution quality in local search, and that OFL_{loc} leads to more natural results. Hence, we use OFL_{loc}, and refer to OFL_{cc} chiefly for a clearer presentation of results. OFL_{loc} is to be maximized, while OFL_{cc} is to be minimized.

The objective function for regularity *OFR* was introduced in [5]. Regularity optimization aims at producing a schedule that can be written in a structured and intuitively simple form. Hence, the SIs should be arranged so that the sequence of values *op*(s) and $V(s)$ exhibits regular patterns. *OFR* is a complex function omitted for brevity [5]. Briefly, it is based on the identification of loop structures in the schedule under consideration. To each recognized loop, some gain is assigned, and the gains are summed up. The summation takes only part of the loops into account, namely a maximal conflict-free subset. A subset of loops is called conflict-free if all the loops can be simultaneously realized in a single structured representation. The assignment of gains follows an empirically established formula; a greedy heuristic is used to select the maximal conflict-free subset of loops. The evaluation of *OFR* is more costly (in terms of computing time) than the evaluation of OFL_{loc}.

In the present implementation, regularity optimization is restricted to the SI sequencing aspect. Hence, we will usually assume *DB* to be fixed as standard row-wise storage order.

We have implemented a local search algorithm to approximately solve the optimization problem. The algorithm starts from a random schedule and repeatedly considers some neighbourhood of the current schedule. If a neighbour with a higher objective function value is encountered, the current schedule is replaced by that neighbour (at least the new neighbour is a candidate for replacement, see below). The neighbourhood comprises moves of consecutive SI groups as well as loop transformations (permutation, reversal, distribution, and a novel transformation called loop extension [5]). All intermediate schedules must respect the *dag*.

Locality and regularity are typically conflicting goals that must be traded off. [5] combines them in a two-step algorithm. In Step 1, locality has priority. Regularity comes in by choosing the next schedule from k locality-improving neighbours according to a weighted sum of OFL_{loc} and *OFR*. The weight is a parameter to be specified by the user. In Step 2, a neighbour must improve *OFR* and must not degrade OFL_{loc} too much (controlled by another parameter).

Step 2 of the algorithm is costly since *OFR* must be evaluated for many neighbours. This is one shortcoming of the algorithm. Another shortcoming is the algorithm's tendency to produce distracting structured representations. A structured representation is called *distracting* if it highlights another than the locality-relevant structure, and is hence misleading. An example is given in Sect. 4. This paper ad-

dresses the shortcomings.

3.3. Previous results and relevance for this paper

Reference [4] gives experimental results for about 15 example programs, mostly taken from the compiler research literature. The experiments referred to both the case that *DB* was fixed and to the case that *DB* had to be found. Regularity optimization was not covered. The results show that the local search algorithm compares well with compiler optimizations. The OFL_{cc} values achieved for example PIs were always the same or better than those of compiler optimized reference PIs. Due to the choice of the example programs, the improvements were often small. In two cases, the restructuring found for the PI was generalized into a restructuring of the program with a performance gain of about 20–30% over the compiler-optimized version. This performance gain was measured for realistic input sizes on a real machine.

Reference [5] gives some examples of PIs that were optimized for both locality and regularity. Due to the shortcomings mentioned above, the experiments were restricted to PIs with a very small input size. The examples show trade-offs between locality and regularity.

In particular the experimental results of [4] are still relevant for the modified algorithm suggested in the present paper. The suggested modification makes it easier for the user to recognize the differences between the original and the optimized PI in the generalization phase, the modification does *not* change the program that is found by the user. That is, if the user inputs the same PI to the different variants of the algorithm suggested in [4], [5] and the present paper, he or she will normally find the same optimized program. It is, however, much harder for the user to find the program on the basis of the [4] output than it is now. Since the locality optimization potential of the algorithm has already been assessed in [4], we do not repeat the experiments here, and instead concentrate on the quality of the output representation.

4. Hierarchical clustering

Figure 1 shows an output schedule that distracts the user from the locality-relevant structure. It refers to the example of 4×4 matrix transpose. For clearness, only half of the program is printed. Assuming row-wise storage order and a cache line size of two, the following pairs of data are stored in the same cache line: $A[0,0]$–$A[0,1]$, $A[1,0]$–$A[1,1]$, $B[0,0]$–$B[0,1]$, $B[1,0]$–$B[1,1]$, and so on.

In the example, locality is supported best by a program in which the matrix is divided into 2×2 sub-matrices that are transposed one after another. Although the operation

```
B[0,0]  = A[0,0];
B[0,1]  = A[1,0];
B[1,0]  = A[0,1];
B[1,1]  = A[1,1];
B[1,2]  = A[2,1];
B[1,3]  = A[3,1];
B[0,2]  = A[2,0];
B[0,3]  = A[3,0];
        ⋮
```

or, in structured representation:

```
FOR  i:=0  TO  1  DO
  B[0,i]  = A[i,0];
FOR  i:=0  TO  3  DO
  B[1,i]  = A[i,1];
FOR  i:=0  TO  1  DO
  B[0,i+2]   = A[i+2,0];
        ⋮
```

Figure 1. A distracting output for matrix transpose.

listing on the top exhibits this structure, the second for-loop in the structured representation distracts from it. To be of more help, a structured representation should put the first and the next four operations, respectively, into a separate loop nest. In other words, the locality-relevant structure of the example reads that the first four operations belong together, and the next four operations belong together. The operations belong together since their accesses concentrate on a small number of memory blocks, so that there is much data reuse.

In this section, a hierarchical clustering scheme is introduced that makes the locality-relevant structure explicit. Based on this scheme, distracting loops can be eliminated. Moreover, the clustering scheme on its own is a help in the generalization phase, and supplements the structured representation.

The clustering scheme is based on a locality-optimized schedule, and highlights groups of successive SIs with a high degree of data reuse. The groups, called clusters, are further agglomerated into larger clusters with a still relatively high degree of data reuse. The agglomeration proceeds hierarchically, producing a tree.

Our task resembles hierarchical clustering as studied in pattern recognition [7,9]. Since only *successive* SIs are agglomerated, we are particularly dealing with contiguity-constrained hierarchical clustering [8]. There are important differences: In pattern recognition, a set of objects is given where each object is characterized by some feature vector. The features belong to the individual object, i.e., they do

not depend on other objects (not even for relational input data). In our case, data reuse is a property that only exists in the interplay of successive SIs. Additionally, our contiguity constraint is one-dimensional, whereas [8] handles two-dimensional constraints. Also, we agglomerate more than two objects/clusters per step if this is more natural. Hence, our clustering algorithm, although inspired by algorithms from pattern recognition, is quite different from these algorithms. Since our objective is stated vaguely (to find natural clusters that are helpful in the generalization phase), we assess our results empirically (in Sect. 6).

In the following, the clustering algorithm is outlined. Explanations are given below.

$\mathcal{C}$:= set of SIs;
for all $c \in C$ do
  Determine the reuse factor $f(c, succ(c))$
repeat
  f^* := $ct \cdot \max_{c \in C} \{f(c, succ(c))\}$
  while $\exists c : f(c, succ(c)) \geq f^*$ do
    new := *FormCluster*(c, f^*);
    $(\mathcal{C}, \{f(c, succ(c))\}_{c \in \mathcal{C}})$
    := *UpdateClustersAndReuseFactors*(*new*)
  end
until $|\mathcal{C}| = 1$

The reuse factor $f(c, c')$ characterizes the amount of data reuse between SI groups c and c'. Its definition considers all pairs of accesses to the same memory block with one access belonging to c and the other belonging to c'. The access in c must be the last access to that memory block in c, and the access in c' must be the first access to that memory block in c'. For each such pair, an access distance is determined as the sum of the number of accesses following the access in c, and the number of accesses preceding the access in c'. The access distance is assessed using the same *the-closer-the-better* function as in OFL_{loc} [4], and the assessments are summed up over all pairs. The resulting value is finally scaled with respect to the size of c and c'.

The scaling function is based on the intuition that the following combinations of c and c' should have the same reuse factor. (We describe each cluster by a list of memory block numbers. The list contains those memory blocks that are accessed within the cluster, in access order.)

- $c = (1),\ c' = (1)$,
- $c = (1, 2),\ c' = (2, 1)$,
- $c = (1, 2, 3),\ c' = (3, 2, 1)$,
- $c = (1, 1),\ c' = (1, 1)$, etc.

The combination $c = (1, 2, 3),\ c' = (3)$ should have a slightly lower reuse factor than the other combinations. The scaling function was empirically defined so that these intuitions are met.

The function *FormCluster*(c, f^*) forms a cluster *new* from two or more successive clusters around c (see below). *UpdateClustersAndReuseFactors* removes these clusters from $\mathcal{C}$ and replaces them by *new*. It also updates f(*new, succ(new)*) and f(*pred(new),new*), where *pred* and *succ* denote the predecessor and successor clusters in the schedule.

The function *FormCluster* starts combining c with its successor. Then it repeatedly extends the current cluster by a neighboured cluster, as long as an appropriate neighbour exists. Let us w.l.o.g. consider a neighbour c_0 that precedes the cluster, and let us denote the current cluster by $\bar{c}$. c_0 is added to $\bar{c}$ iff either $f(c_0, \bar{c}) \geq f^*$, or if there is a direct sub-cluster $\bar{\bar{c}}$ of $\bar{c}$ with $f(c_0, \bar{\bar{c}}) \geq f^*$.

The algorithm outputs a tree consisting of the clusters found in intermediate steps (returned by *FormCluster*). The parameter $ct \in [0 \ldots 1]$ (called cluster tolerance) is supplied by the user and controls the average cluster size. Clusters tend to be larger if *ct* is small.

5. The three–step optimization algorithm

The following three-step optimization algorithm avoids the shortcomings of the two-step algorithm from Sect. 3.2, and is used in the experiments of Sect. 6.

Step 1 of the algorithm corresponds to Step 1 of the algorithm from Sect. 3.2, that is, we reorder the schedule with locality as a primary and regularity as a secondary goal. In Step 2, the clustering is established as described in Sect. 4. The novel part is Step 3. It uses a modified function *OFR'* that assigns a gain to a loop only if the loop is non-distracting. A loop is non-distracting if for each cluster c and each loop iteration k, one of the following conditions holds: $c \subseteq k$, $k \subset c$, or $k \cap c = \varnothing$ (taking k and c as sets of SIs).

In addition, Step 3 restricts the neighbourhood of the local search algorithm to cluster-conform neighbours, i.e., to neighbours in which the SIs of each cluster are kept consecutive (the order may change). For the case that the neighbour is obtained by moving a group of consecutive SIs, it is easy to see that the neighbour is cluster-conform if and only if it can be produced by a reordering of the direct subclusters of some cluster. For the case of loop transformations, several clusters may have to be reordered in parallel. In any case, all cluster-conform neighbours can be generated in a systematic way. The number of cluster-conform neighbours tends to be much smaller than the original number of neighbours, at least if the cluster size is low. Since less neighbours need to be assessed in Step 3, the modification speeds up the algorithm. For the examples considered in this paper, the algorithm is speeded up to the extent that

Step 3 runs faster than Step 1.

As usual, Step 3 searches the neighbourhood in some deterministic, cyclic order. We replace the current schedule whenever a neighbour is found that fulfills the following three requirements:

- The neighbour improves *OFR*'.
- The OFL_{cc} value of the neighbour is at most $t3 \cdot OFL_{cc}^{1}$, where OFL_{cc}^{1} is the OFL_{cc} value of the schedule produced in Step 1, and *t3* is a user-supplied parameter (called tolerance for Step 3).
- Comparing the neighbour to the current schedule, the relation between the regularity gain and the OFL_{cc} loss (if any) is acceptable. The acceptance threshold is controlled by a user-supplied parameter *w3*. Expressed as a formula, we require

  $$OFR'_{\mathrm{new}} / OFR'_{\mathrm{old}} - 1 > w3 \cdot (OFL_{\mathrm{loc}}^{\mathrm{old}} / OFL_{\mathrm{loc}}^{\mathrm{new}} - 1)$$

  where *old* refers to the current schedule and *new* refers to the neighbour under consideration. *w3* is permitted to take any nonnegative real value.

Notwithstanding cluster-conformity, the neighbour may give rise to a different clustering if the new order permits the agglomeration of previously non-consecutive clusters. Larger clusters in tendency imply higher *OFR*' values. For a fair comparison between schedules, the clustering is *not* updated after a replacement of the current schedule. All we do is keeping the cluster coordinates up-to-date when the SIs have moved.

6. Experiments

This section gives experimental results for the following input PIs:

- 4×4 matrix transpose with $C = 8$ and $L = 2$ (Trapo),
- 4×4 matrix multiplication with $C = 12$ and $L = 2$ (Mult),
- Loop program in Fig. 2 with $N = 6$, $C = 4$ and $L = 2$ (Loop),
- Mergesort of $N = 32$ data with $C = 8$ and $L = 2$ (Merge).

The mergesort input program is iterative, i.e., it makes $\lceil \log_2 N \rceil$ passes over the data where the i-th pass merges runs of length 2^{i-1} into runs of length 2^i (as in base mergesort from [3]). The program alternatingly writes to two arrays. Each SI is a merge of two runs, with some likely internal access order supposed.

```
FOR i:=0 TO N DO
  FOR j:=0 TO N DO
    f(A[i],A[j]);
```

Figure 2. Loop program taken from [11]

.

Figure 3 summarizes our results and shows the influence of the parameters *t3* and *ct*. Some exemplary outputs are depicted in Fig. 4–7. Each entry in Figure 3 stands for five runs with a different random initialization. The entries give OFL_{cc}/NoOfLines for the best of the five runs, and a symbol that characterizes the general impression from all the five runs. For clearness, identical entries are written only once. In the cases marked by '+', the five outcomes were, in our intuitive judgement, about as good as the example outputs given in Fig. 4–7. In the cases marked by '(+)', the structuring was improvable but the clustering was helpful. The symbol '–' marks cases where the clustering/structuring was of little help in the generalization phase.

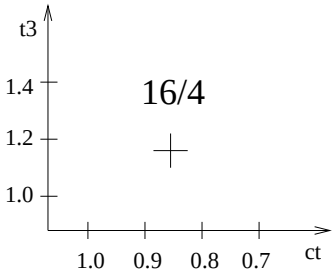
For brevity, the influence of the parameter *w3* is not illustrated in Fig. 3. We found that the influence of *w3* is minor, provided that *w3* is set to a small value such as 0.0 or 1.0. If *w3* is set to a large value such as 100.0, then Step 3 let the operation sequence unchanged. All experiments were carried out with *w3*$= 1.0$. Appropriate values were also used for the parameters introduced in [4, 5] .

The figures show that the clustering scheme does successfully highlight the locality-relevant structure and avoids distracting loops, provided that *ct* is set to an appropriate value. The trade-off between locality and regularity can be controlled by the parameter *t3*. For *t3*$= 1.0$, outputs with a very high degree of locality were produced. Slightly larger values of *t3* led to simpler solutions that may perform better in practice. The given *Mult* output with $OFL_{cc} = 40$ and the *Trapo* output correspond to compiler optimized programs. The *Loop* and *Merge* outputs generalize to programs that improve on compiler-optimized programs by about 20-30%, as shown in [4].

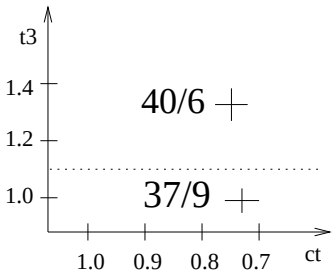
7 Conclusions

This paper has reported on recent progress in the development of the IBLOpt method. First, we have introduced a hierarchical clustering scheme that highlights the locality-relevant structure in an SI sequence. Second, we have applied the clustering scheme to produce a more meaningful structured representation of the final schedule, in which distracting loops are avoided. The new results are helpful in the generalization phase of our tool. The user finds the same optimized programs as with the [4, 5] version of IBLOpt, but the amount of human work needed for the generalization is reduced.

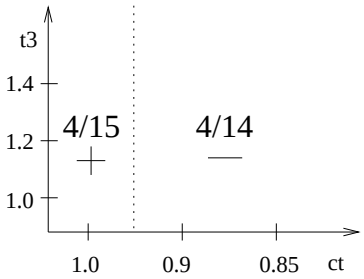
a) Trapo



b) Mult



c) Loop



d) Merge

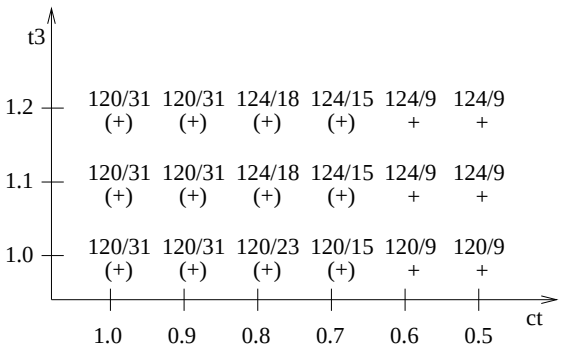


Figure 3. Results for various parameter settings.

```
f(A[0],A[4]); f(A[0],A[5]);
f(A[1],A[4]); f(A[1],A[5]);
f(A[4],A[0]); f(A[4],A[1]);
f(A[5],A[0]); f(A[5],A[1]);
f(A[0],A[2]); f(A[0],A[3]);
f(A[1],A[2]); f(A[1],A[3]);
f(A[2],A[0]); f(A[2],A[1]);
f(A[3],A[0]); f(A[3],A[1]);
f(A[0],A[0]); f(A[0],A[1]);
f(A[1],A[0]); f(A[1],A[1]);
f(A[2],A[2]); f(A[2],A[3]);
f(A[3],A[2]); f(A[3],A[3]);
f(A[4],A[4]); f(A[4],A[5]);
f(A[5],A[4]); f(A[5],A[5]);
f(A[4],A[2]); f(A[4],A[3]);
f(A[5],A[2]); f(A[5],A[3]);
f(A[2],A[4]); f(A[2],A[5]);
f(A[3],A[4]); f(A[3],A[5]);
```

```
FOR i2:=0 TO 1 DO
   FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
         f(A[i1],A[4+i0-2*i2]);
   FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
         f(A[4+i1-2*i2],A[i0]);
FOR i2:=0 TO 2 DO
   FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
         f(A[i1+2*i2],A[i0+2*i2]);
FOR i2:=0 TO 1 DO
   FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
         f(A[4+i1-2*i2],A[2+i0+2*i2]);
```

OFL_{cc}= 4 NoOfLines = 15

Figure 4. Example output for Loop: operation listing with clustering, and structured representation.

```
FOR  i2:=0  TO  1  DO
  FOR  i1:=0  TO  3  DO
    FOR  i0:=0  TO  1  DO
      A[i1,i0+2*i2]=B[i0+2*i2,i1]
```

Figure 5. Example output for Trapo (structured representation only).

```
FOR i3:=0 TO 1 DO
  FOR i2:=0 TO 3 DO
    FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
        C[i2,i0+2*i3]+=
        A[i2,i1+2*i3]
        *B[i1+2*i3,i0+2*i3]
  FOR i2:=0 TO 3 DO
    FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
        C[3-i2,i0+2*i3]+=
        A[3-i2,2+i1-2*i3]
        *B[2+i1-2*i3,i0+2*i3]
```

OFL_{cc}: 37

```
FOR i4:=0 TO 1 DO
  FOR i3:=0 TO 1 DO
    FOR i2:=0 TO 3 DO
      FOR i1:=0 TO 1 DO
        FOR i0:=0 TO 1 DO
          C[i2,i0+2*i3]+=
          A[i2,i1+2*i4]
          *B[i1+2*i4,i0+2*i3]
```

OFL_{cc}: 40

Figure 6. Example outputs for Mult.

```
FOR i3:=0 TO 1 DO
  FOR i2:=0 TO 1 DO
    FOR i1:=0 TO 1 DO
      FOR i0:=0 TO 1 DO
        Merge two runs of length 1,
        first run starts at
        2*i0+4*i1+8*i2+16*i3
      END;
      Merge two runs of length 2,
      first run starts at
      4*i1+8*i2+16*i3
    END;
    Merge two runs of length 4,
    first run starts at 8*i2+16*i3
  END;
  Merge two runs of length 8,
  first run starts at 16*i3..16*i3+7
END;
Merge two runs of length 16, namely
0..15 and 16..31
```

OFL_{cc}: 120

Figure 7. Example output for Merge.

References

[1] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th Int. Conf. on Supercomputing*, pages 269–276. ACM Press, July 1997.

[2] M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 236–247. IEEE Computer Society Press, 1997.

[3] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms*, pages 370–379, Jan. 1997.

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

[5] C. Leopold. Regularity considerations in instance-based locality optimization. In *Workshop Proceedings Int. Parallel Processing Symp. and Symp. on Parallel and Distributed Processing, LNCS 1586*, pages 230–238, 1999.

[6] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[7] F. D. Murtagh. *Multidimensional Clustering Algorithms*. Physica-Verlag, Vienna, 1985.

[8] F. D. Murtagh. Contiguity-constrained hierarchical clustering. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 19:143–152, 1995.

[9] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313–1325, Aug. 1995.

[10] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. DIMACS Series on Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999.

[11] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991.