Regularity Considerations in Instance–Based Locality Optimization

Claudia Leopold

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

Abstract. Instance–based locality optimization [6] is a semi–automatic program restructuring method that reduces the number of cache misses. The method imitates the human approach of considering several small program instances, optimizing the instances, and generalizing the structure of the solutions to the program under consideration. A program instance appears as a sequence of statement instances that are reordered for better locality. In [6], a local search algorithm was introduced that reorders the statement instances automatically.
This paper supplements [6] by introducing a second objective into the optimization algorithm for program instances: regularity. A sequence of statement instances is called regular if it can be written compactly using loops. We quantify the intuitive notion of regularity in an objective function, and incorporate this function into the local search algorithm. The functionality of the compound algorithm is demonstrated with examples, that also show trade–offs between locality and regularity.

1 Introduction

In memory hierarchies, (data) locality is a gradual property of programs that reflects the level of concentration of the accesses to the same memory block, during program execution. Locality optimization increases the degree of locality, and hence reduces the number of cache misses (or memory accesses in general). A major technique for locality optimization is program restructuring, i.e., reordering the statements of a given program.

In Ref. [6], a semi–automatic locality optimization method was introduced that is based on the optimization of program *instances* (PI). A PI is obtained from a program by fixing the input size and possibly more parameters. After unrolling loops and resolving recursions, the PI can be written as a sequence of statement instances (SIs) each of which is carried out once. An SI can be an elementary statement instance or a complex operation. In both cases, variables are instantiated, i.e., replaced by constant values.

The instance–based method follows a human approach to program optimization in which one first selects some small program instances, then optimizes the instances, and finally generalizes the structure of the optimized instances to the

program. Correspondingly, the instance–based method consists of three phases: an instantiation, an optimization and a generalization phase.

In the instantiation phase, the user fixes a sufficient number of parameters. The user must also choose the granularity of the SIs, i.e., decide if elementary or complex operations (such as function calls) should be considered as SIs. The parameters, particularly the input size, must be fixed at *small* values so that the SI sequence is of manageable length. When the parameters are known, the PI is automatically transformed into the required representation, and a data dependence graph is extracted.

Next, in the optimization phase, the SIs are reordered[1] such that the PI is improved with respect to locality. During the reordering, data dependencies are strictly obeyed. The reordering is controlled by a local search algorithm that is outlined in Sect. 2.1. The output of the optimization phase is a locally optimal SI ordering for the PI.

Finally, in the generalization phase, the user inspects the SI orderings produced for one or several PIs of the program. He or she has to recognize the structural differences that make the suggested orderings superior to the initial orderings, and generalizes them to the program. The user writes down the modified program and makes sure that data dependencies are not violated in the general case.

To facilitate the generalization phase, the output SI sequences of the optimization phase should preferably be structured, i.e., the indices attached to the names of the SIs should follow some regular patterns, as much as possible. This brings up a second objective in the optimization phase: regularity. Although [6] briefly mentions the implementation of a regularity–improving postprocessing phase, that implementation was preliminary and could only achieve few improvements.

This paper concentrates on the issue of regularity optimization, and supplements [6] in this respect. The paper makes two contributions: First, it derives a quite involved objective function that covers the intuitive notion of regularity, for a class of programs. Second, it incorporates the function into the local search algorithm of [6], such that the new algorithm optimizes PIs with respect to both locality and regularity. We report on experiments with nested loop programs and show trade–offs between locality and regularity. Extending the method of [6], our output is represented in a structured form, as a sequence of nested for–loops with constant bounds.

The paper is organized as follows. First, Sect. 2 recalls the formal statement of the optimization problem from Ref. [6], and states the regularity optimization problem. Section 3 defines the new objective function for regularity. Next, Sect. 4 explains how the function is integrated into the local search algorithm. In particular, the neighbourhood of the local search algorithm is extended by various loop transformations, and the two objective functions are combined. Sect. 5 demonstrates the functionality of the algorithm by examples. The paper finishes with an overview of related work in Sect. 6.

[1] [6] additionally reassigns data to memory blocks, but this is out of our scope.

2 The Problem

2.1 The Local Search Algorithm of Ref. [6]

According to Ref. [6], the input of the optimization phase consists of a set S of SIs, a data dependence graph *dag*, a set D of data instances (DIs), a function $SD : S \to D^*$, and memory hierarchy parameters C, L. The *dag* (a directed acyclic graph) contains precendence constraints between the SIs. A DI is a concrete location in memory, e.g. $A[5]$ is a DI while A or $A[i]$ are not. The function *SD* assigns to each SI s the sequence of DIs that are accessed by s. The parameters C and L are the capacity and line size of a hypothetical cache. They are chosen in relation to the size of the PI.

The output of the optimization algorithm is an SI sequence where each $s \in S$ appears once. It is called schedule. Ref. [6] optimizes the PIs with respect to an objective function OFL_{loc} that quantifies the intuitive notion of locality. The function is quite involved, and omitted here.

The optimization problem stated above is solved with a local search algorithm. Although we restrict ourselves to local search, an interesting consequence of this work is the provision of a framework that makes locality optimization accessible for various optimization techniques (including bio-inspired approaches such as genetic algorithms).

Local search is a well–known general heuristic solution approach to combinatorial optimization problems [1]. For each schedule *Sched* that agrees with the *dag*, a neighbourhood *NB(Sched)* is defined. In Ref. [6], *NB(Sched)* consists of all *dag*–compatible schedules that are obtained from *Sched* by moving a group of consecutive SIs as a whole from one position in *Sched* to another. Starting with a random schedule, the neighbours of the current schedule *Sched* are considered in some fixed order. If a neighbour with a higher objective function value is encountered, *Sched* is replaced by that neighbour immediately. The search stops when no improving neighbour exists.

2.2 The Regularity Optimization Problem

For regularity optimization, we need additional information about the SIs. In a structured representation, two or several SIs can be represented by a single loop statement only if they carry out the same operation. The operation may be modified by parameters that explicitly appear in the name of the statement. These parameters may but need not coincide with the indices of data that are accessed by the SIs.

Hence, our input additionally comprises functions *op*: $S \to \mathbb{N}$ and $V : S \to \mathbb{Z}^*$. Here, $op(s)$ characterizes the operation carried out by s, and $V(s) = (v_0(s), v_1(s), \ldots v_{r-1}(s))$ are the parameters that may modify the operation. If $op(s_1) = op(s_2)$, then s_1 and s_2 have the same number of parameters (written $r(s_1) = r(s_2)$).

The regularity optimization problem consists in reordering the SIs such that the final schedule can be written in a structured and intuitively simple form.

Currently, we restrict the set of permitted control structures to for-loops with constant bounds. We consider array accesses with index expressions that are affine functions of the loop variables. The final PI is represented by a sequence of possibly nested for-loops, using a PASCAL–like notation and synthetic loop variables.

3 An Objective Function for Regularity

The objective function is based on our intuitive notion of regularity, according to which the degree of regularity is higher,

- the more compact the schedule can be written with possibly consecutive and nested for–loops,
- the more common the step sizes in the loops are, and
- the simpler the index expressions are.

Thus, to measure the degree of regularity, we must identify loops (i.e., for–loops) in the schedule under consideration (denoted by *Sched*).

In *Sched*, a (non–nested) loop appears as a SI subsequence of the form

$$G = (s_{0,0}, s_{0,1}, \ldots, s_{0,w-1}, s_{1,0}, s_{1,1}, \ldots, s_{1,w-1}, \ \ldots \ , s_{l-1,0}, s_{l-1,1}, \ldots s_{l-1,w-1})$$

with $s_{i,j} \in S$, $l, w \in \mathbb{N}$, $l \geq 2$ and $w \geq 1$. $s_{i,j}$ and $s_{i,j+1}$, as well as $s_{i,w-1}$ and $s_{i+1,0}$ are consecutive in *Sched*. G can alternatively be written as

$$G = A_0 \circ A_1 \circ \cdots \circ A_{l-1} \quad \text{with} \quad A_i = (s_{i,0}, s_{i,1}, \ldots s_{i,w-1})$$

The A_i's are called aggregates. Let $V(s_{i,j}) = (v_{i,j,0}, v_{i,j,1}, \ldots, v_{i,j,r(s_{i,j})-1})$. G is denoted as *group* if it fulfills the following compatibility conditions:

1. $op(s_{i_1,j}) = op(s_{i_2,j})$, for all i_1, i_2, and
2. $v_{i,j,k} - v_{i-1,j,k} = v_{1,j,k} - v_{0,j,k} \underset{D}{=} D^G_{j,k}$ (or $D_{j,k}$, for short).

Each group can be written as a loop, where A_i corresponds to one iteration of the loop, l is the number of iterations, and w is the number of statements in the loop body. The correspondence between loops and groups is not one–to–one, since inner loops of the program text stand for several groups. G corresponds to the following loop:

FOR $i := 0$ TO $l - 1$ DO
  $op(s_{0,0})((f_{0,0}(i), f_{0,1}(i), \ldots f_{0,r(s_{0,0})-1}(i))$
  $op(s_{0,1})((f_{1,0}(i), f_{1,1}(i), \ldots f_{1,r(s_{0,1})-1}(i))$
  $\vdots$
  $op(s_{0,w-1})((f_{w-1,0}(i), f_{w-1,1}(i), \ldots f_{w-1,r(s_{0,w-1})-1}(i))$.

where $f_{j,k}(i) = v_{0,j,k} + i \cdot D_{j,k}$. We always represent loops in the above normalized form, where the step size is one and the loop starts from zero. Normalization

converts the requirement of common step sizes into the requirement of simple index expressions (manifested by common $D_{j,k}$ values).

We will see below that the groups within *Sched* can be easily identified. Unfortunately, groups may conflict, i.e., it may be impossible to realize them simultaneously in a structured formulation of the program. Fig. 1 shows three typical cases of conflicts that may occur. The conflict in Fig. 1c) arises since a proper nesting of the groups $s_0 \ldots s_{11}$ and $s_0 \ldots s_5$ is not possible, as $s_0 \ldots s_5$ and $s_6 \ldots s_{11}$ are structured differently.

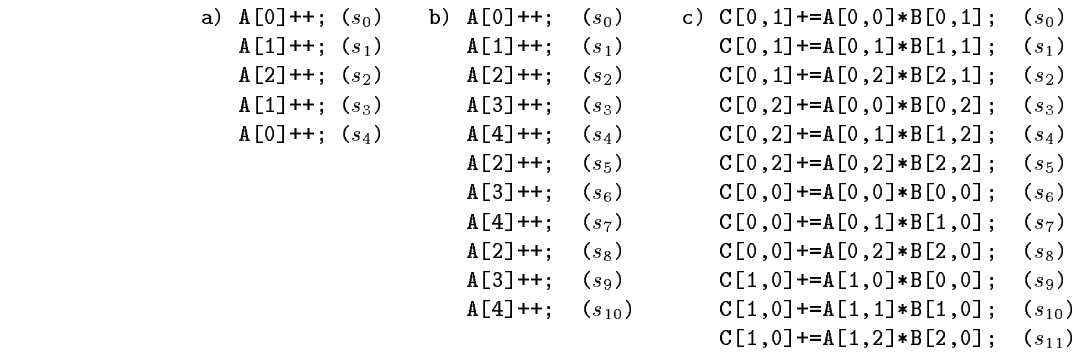

Fig. 1. Examples of conflicts

If there are no conflicts, groups can be nested. A nesting is permitted if all aggregates of the outer group have the same internal organization. Let $G = A_0 \circ \cdots \circ A_{l-1}$ be the outer group, and let $A_i = g_{i,0} \circ \cdots \circ g_{i,m_i}$ where $g_{i,j}$ stands for either a subgroup or a single SI. Then, A_{i_1} and A_{i_2} have the same internal organization iff

- $m_{i_1} = m_{i_2}$,
- $g_{i_1,d}$ and $g_{i_2,d}$ are either both groups or both SIs.
- If they are groups, $g_{i_1,d}$ agrees with $g_{i_2,d}$ in l, w, and $D_{j,k}^{g_{i_1,d}} = D_{j,k}^{g_{i_2,d}}$.
- If they are groups, the aggregates of $g_{i_1,d}$ and $g_{i_2,d}$ have the same internal organization.
- If they are SIs, $op(g_{i_1,d}) = op(g_{i_2,d})$.

The definition of the objective function for regularity (*OFR*) refers to a maximal set of conflict–free groups that can be identified in *Sched*. We define

$$OFR = \max_{\mathcal{G}} \sum_{G \in \mathcal{G}} Eval(G)$$

where $\mathcal{G}$ stands for any set of mutually conflict–free groups, and *Eval* is defined below.

Let G be a group characterized by l, w, and by the sequence $(D_{j,k})$ of essential $D_{j,k}$ values. A value $D_{j,k}$ is essential if the corresponding SI $s_{0,j}$ is either not

contained in a subgroup of G, or if $s_{0,j}$ recursively belongs to the first aggregates of all subgroups it is contained in. We define

$$Eval(G) = Dval^{1+2/l^2} \cdot l^{1+p_0} \quad .$$

Here, p_0 is a parameter (default 0.5), and *Dval* evaluates the $(D_{j,k})$ sequence according to our intuitive feeling of how a loop should preferably look like. We use the function

$$Dval(D_{j,k}) = W_{j,k} \cdot \begin{cases} 1 & D_{j,k} = 0 \\ 0.7 & D_{j,k} \neq 0 \text{ and} \\ & D_{j,k} = D_{j',k'} \text{ for some } (j',k') <_{\text{lex}} (j,k) \\ 1/(D_{j,k}+1) & D_{j,k} > 0 \\ 1/(|D_{j,k}|+2) & D_{j,k} < 0 \end{cases}$$

where

$$W_{j,k} = 0.9 + 0.1 \cdot \left(\sum_{d=0}^{j-1} r(s_{0,d}) + k + 1 \right)^{-1}$$

and where

$$Dval = \frac{1}{2} \cdot \left(\operatorname*{avg}_{j,k} Dval(D_{j,k}) + \max_{j,k} Dval(D_{j,k}) \right) \quad .$$

Both *Eval* and *Dval* were designed empirically. We started with some initial functions that roughly corresponded to our intuition. Then, we repeatedly considered two groups. If the relation of the values assigned by the current functions did not agree with that which we had intuitively expected to see, the functions were adjusted appropriately. Additionally, the functions were fine–tuned during their use in the local search algorithm.

For a given SI sequence and width w, it is easy to decide if the sequence forms a group of width w, by a simple test of the compatibility conditions. If we systematically consider all SI subsequences and widths, we find all groups contained in *Sched*. The selection of a conflict–free set of groups $\mathcal{G}$ with maximum $\sum_{G \in \mathcal{G}} Eval(G)$ value is, however, computationally expensive. Hence, we have implemented a greedy algorithm that finds a conflict–free subset $\mathcal{G}$ with a high value of $\sum_{G \in \mathcal{G}} Eval(G)$, but not necessarily with a maximum value. The algorithm is omitted for brevity.

4 Incorporating Regularity Optimization in the Local Search Algorithm

Since it is difficult to repair regularity deficiencies after the locality optimization process has finished, we combine locality and regularity optimization. Our

algorithm consists of two phases. In the first phase, emphasis is given to locality optimization, but regularity is considered as a secondary goal. The second phase optimizes the output schedule of the first phase with respect to regularity.

More detailed, the algorithm for the first phase differs from the algorithm of Ref. [6] in that the current schedule is no longer replaced by the first locality–improving neighbour encountered. Instead, several locality–improving neighbours are recorded. Whenever some number of candidates was found, the one that performs best is selected with respect to the combined objective function

$$p_1 \cdot OFL_{\mathrm{loc}} + (1 - p_1) \cdot OFR \quad \text{(for some parameter } p_1 \in [0 \ldots 1]).$$

In phase 2, the user can influence the trade–off between locality and regularity via a parameter $p_2 \geq 1$. The current schedule *Sched* is replaced by a neighbour *Sched'* if OFR(*Sched'*)$>$ OFR(*Sched*), and if additionally OFL_{cc}(*Sched'*) $\leq p_2 \cdot$ OFL_{cc}(*Sched*$_0$). Here, *Sched*$_0$ is the schedule after phase 1, and OFL_{cc} measures the number of simulated cache misses.

In experiments, the algorithm described so far was typically stuck with a local optimum. We analyzed the particular local optima, and could overcome the problem by extending the neighbourhood of the local search algorithm. In particular, the following transformations were added:

- *Move with Reversal:* This transformation corresponds to the neighbourhood of Ref. [6], except that the order of the moved SI subsequence is additionally reversed.
- *Loop Reversal, Permutation and Distribution*: These transformations are well–known compiler transformations [7].
- *Loop Extension:* This transformation simultaneously adds one or more new aggregates to all groups represented by a loop. It is based on the observation that the old aggregates $A_0 \ldots A_{l-1}$ uniquely imply how possible new aggregates (e.g. A_{-1} or A_l) must look like. In particular, they imply the *op* and V values of SIs that may form the new aggregate. If all the desired SIs are available somewhere in *Sched* (outside the loop under consideration), they are moved to their new positions, forming an extended loop.

The new neighbourhood consists of all SI sequences that are produced by the above transformations, and that additionally agree with the DAG.

After the second phase, the schedule is output in a structured representation. This representation is a by–product of the calculation of OFR, where the index expressions of the program statements are composed of the $v_{i,j,k}$ values of the first SI represented by the statement, together with the corresponding essential $D_{j,k}$ values of all surrounding loops.

As regularity optimization aims at supporting the generalization phase, regularity optimization should highlight the locality–relevant structure. Many optimized PIs have a tiled structure, where most reuse of cached data occurs within tiles. OFR is modified to prefer groups that are located within a tile, by scaling down $Eval(G)$ for groups G that cross a tile boundary. The experiments in Sect. 5 are based on a straightforward tile recognition scheme.

5 Examples

Figures 2–4 give the outputs of our algorithm for some example PIs. They show that the algorithm is able to produce structured outputs with a high degree of locality. They also show that the algorithm can make several reasonable suggestions that represent different trade–offs between locality and regularity. Currently, such trade–offs can not be found with other automatic locality optimization methods. Compiler transformations find one of the solutions, e.g. b) for Fig. 3, and c) for Fig. 4.

```
a) 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]
```

$OFL_{\mathrm{cc}} = 16$
(vs. 24 for input PI)

Fig. 2. optimized PI for 4×4 matrix transpose with $C = 8$ and $L = 2$.

```
a) FOR i1:=0 TO 1 DO
     FOR i0:=0 TO 1 DO
       C[0,i0]+=A[0,i1]*B[i1,i0]
   FOR i1:=0 TO 1 DO
     FOR i0:=0 TO 1 DO
       C[1,i0]+=A[1,1-i1]*B[1-i1,i0]
```

OFL_{cc}: 7

```
b) FOR i2:=0 TO 1 DO
     FOR i1:=0 TO 1 DO
       FOR i0:=0 TO 1 DO
         C[i2,i0]+=A[i2,i1]*B[i1,i0]
```

OFL_{cc}: 8

Fig. 3. optimized PI for 2×2 matrix multiplication with $C = 6$ and $L = 2$. a) $p_2 = 1.0$, b) $p_2 = 1.2$.

6 Related Work

The instance–based approach to locality optimization differs significantly from other approaches to locality optimization [4, 7, 8], as discussed in [6]. Since the instance–based method is new, the regularity optimization problem posed in this paper has not been investigated before. Some rather loosely related work exists in the areas of decompilers, software metrics, and genetic programming.

Decompilers [2] identify structures in a statement sequence. The structure already exists and the task is to make it explicit. The design of an objective function is not an issue.

Software metrics, particularly metrics for structural complexity [3], resemble our objective function in that they assess the structure of programs. However, software metrics do not consider the assignment of a meaningful value to an unstructured program, and are less sensitive to minor differences between programs.

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

OFL_{cc}: 13 (vs. 33 for input PI)

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

OFL_{cc}: 14

c)
```
FOR i2:=0 TO 2 DO
 FOR i1:=0 TO 2 DO
  FOR i0:=0 TO 2 DO
   C[i2,i0]+=A[i2,i1]*B[i1,i0]
```

OFL_{cc}: 15

Fig. 4. optimized PI for 3×3 matrix multiplication with $C = 9$ and $L = 3$. a) $p_2 = 1.0$, b) $p_2 = 1.1$, c) $p_2 = 1.2$

Genetic Programming also uses fitness measures that assess structural complexity. In Ref. [5], p. 91, structural complexity is defined as the number of elementary operations that appear in a structured representation of the program. This function is too rough, for our purposes, since we must compare PIs that are very similar to each other. The ability to react sensitively to minor differences is particularly important in local search, where it helps to escape from local optima. Furthermore, *OFR* is more detailed in that it reflects desirable properties such as the intuitive preference of upwards–counting over downwards–counting loops. In genetic programming, programs are in a structured form by construction, i.e., the recovery of the structure is not an issue.

References

1. Aarts, E.H.L. and Lenstra, J.K.: Local Search in Combinatorial Optimization. Wiley, 1997
2. Cifuentes, C.: Structuring Decompiled Graphs. Int. Conf. on Compiler Construction, 1996, LNCS 1060, 91–105
3. Fenton, N. E.: Software Metrics – A Rigorous Approach. Chapman & Hall, 1991, Chapter 10
4. LaMarca, A. and Ladner, R.E.: The Influence of Caches on the Performance of Sorting. Proc. ACM SIGPLAN Symp. on Discrete Algorithms, 1997, 370–379
5. Koza, J.R.: Genetic Programming II . MIT Press, 1994
6. Leopold, C.: Arranging Statements and Data of Program Instances for Locality. Future Generation Computer Systems (to appear)
7. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997
8. Wolf, M.E. and Lam, M.S.: A Data Locality Optimizing Algorithm. Proc. ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation, 1991, 30–44