A Closer Look at Loop-Carried Code Placement

Peter Faber, Martin Griebl, Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
email: {faber,griebl,lengauer}@fmi.uni-passau.de

Abstract. Traditional code optimization techniques treat loops as non-predictable structures and do not consider expressions containing array accesses for optimization. We show that, in the typical setting, in which loop bounds and array subscripts can be expressed as affine functions, the polyhedron model can be used to implement code placement techniques that exploit equalities of expressions that hold between arbitrary loop iterations. We give a detailed description of our method and a comparison with classical optimization techniques. We also present preliminary results that show the effectiveness of our method on a simple example.

1 Introduction

Traditional code optimization techniques treat loops as non-predictable structures. In order to exploit the equality of expressions across loop iterations in code placement or code motion, loops may be partially unrolled [Ste96,KRS99]. However, a loop may have to be unrolled infinitely often to remove *all* redundancies, which leads to unbounded data structures. The polyhedron model [Fea92,Len93] offers a way of analyzing memory accesses in loops by considering symbolically expressible sets of loop iterations. These symbolic descriptions are usually restricted to affine expressions in the index variables of surrounding loops and constants. Thus, in general, it is not possible to analyze a complete program. But code fragments can be analyzed in the polyhedron model if they

- contain only statements without side effects, i.e., only assignments and calls to `PURE` functions and subroutines as defined in `HPF` and `Fortran 95`, and
- contain only accesses to scalars or arrays indexed by affine expressions.

A widely used tool for manipulating the necessary set representations is the Omega library, which can handle general sets that are expressed by Presburger formulae [KMP+96].[1]

Although the polyhedron model was originally developed to capture loop restructuring methods for automatic parallelization [Len93], its general representation of loop iterations as sets also provides a means for other code transformations based on the analysis of array accesses in loop nests: Slowik developed loop restructuring methods for optimizing cache usage [Slo99]. Recently, Wonnacott extended constant propagation and dead code elimination to arrays in loop nests [Won00].

In a short version of this paper, we sketched a method to extend a further class of optimization strategies to arrays – namely code placement optimizations [FGL01]. In this work, we give a more detailed description of our method. Our aim is to improve code that has been written by a programmer without attention to possible recomputations of the same value in different loop iterations. Although the method is, in principle, also applicable to `WHILE` loops, we only consider `DO` loops here.

The remainder of this paper is organized as follows: Section 2 gives an overview of the principal method of loop-carried code placement, which is applied to an example that we present along with initial performance measurements in Section 3; we close with general remarks and aspects for future work in Section 4.

[1] Presburger formulae are first-order logic formulae on equality or inequality predicates comparing affine expressions.

2 Loop-Carried Code Placement

Informally, the central question of code placement is where to put the computation of an expression in order to ensure that this computation is performed only if it is needed, and at the same time avoid recomputations of the same value. Relevant data flow information is on the one hand which computations have to be performed before an expression can be computed, and on the other hand which expressions share the same value. These questions are tackled in most state-of-the-art approaches by purely syntactic analysis. There are also approaches for the analysis of scalar references that introduce semantic properties, such as the analysis of Steffen, Knoop and Rüthing [SKR90], which is based on Herbrand equivalence and exploits equalities that result from assigning terms to variables. We restrict ourselves to the purely syntactic – but loop-carried – equivalence of expressions, i.e., we use affine expressions in loop indices to infer the equality of two terms.

As a cost model, we employ the number of function calls and arithmetic operations executed, while we view reads and writes to variables as zero-cost operations. We then minimize the amount of computations performed during the complete execution of the loop nest by replacing computations of already computed values by array lookups.

Our analysis is restricted to the intra-procedural case. We view a subroutine as an expression that is combined using a binary prefix operator `;(_,_)` from statements – which, again, are composed of an operator symbol and, possibly, a list of argument expressions. Similarly, we view an assignment as an ordinary operator `:=(`*expression*`,`*variable*`)`. Thus, each point in the program text – including assignments and subroutine calls – can be interpreted as an expression. As already mentioned, the restrictions on our input will in most cases not hold for a complete program, so that we consider only code fragments. For the sake of simplicity, we assume that all variables that are written in the code fragment under consideration are *live* – i.e., may be read – after execution of that code fragment.

2.1 Basic Structures

The structures used in our method are generalizations of the polyhedron model.

A *textual access* is a reference r to a memory cell in the program. Note that we assume a reference to an array element, e.g., `A(i+1,i+j-1)` to be atomic, i.e., we do not resolve the affine indices of an array reference into composed expressions.

For our instancewise approach, we need to distinguish between different run-time instances, caused by surrounding loops. These instances are identified by the *index vectors*, i.e., the values of the surrounding loop indices. The set of all possible index vectors for a statement S is called the *index space* $\mathrm{IS}(S) \subseteq \mathbb{Z}^n$. Thus, an *access instance* $\langle v, r\rangle$ of a textual access r in a statement S is defined by the index vector $v \in \mathrm{IS}(S)$ and the textual access r. It is the atomic item of the dependence tests employed here. The access instances of a program are grouped into disjoint sets $\mathbf{W}$ of writes and $\mathbf{R}$ of reads. We take the term access to mean an access instance.[2]

We work on expressions, which are compositions of several textual accesses, including operators and function calls. Thus, we generalize the concept of single textual accesses to *occurrences* o of expressions, i.e., o is the textual appearance of an expression in the program. We define O as the set of all occurrences. As for textual accesses, we refine the definition of an occurrence o in statement S to an *occurrence instance* $\langle v, o\rangle$, $v \in \mathrm{IS}(S)$. OI is the set of all occurrence instances.

Usually, the occurrences of a term are represented as elements of $\mathbb{N}^*$. We abandon this convention in order to integrate occurrences smoothly into the polyhedron model. We represent an occurrence by a natural number that is appended as least significant element to the index vector. For the determination of suitable numbers, we use a *syntax tree*, in which control structures govern their bodies as well as the arguments that control the execution, while function and subroutine calls govern their arguments [ASU86]. Traversing this tree following an appropriate pattern, we can assign an increasing number as occurrence to every node so that the execution order of the

[2] A dummy argument of a subroutine that may be written as well as read is, for this purpose, replaced by a read access followed by a write access.

sequential program is defined by the lexicographic order of the occurrence instances. A bijective mapping Γ associates the occurrence with the node in the syntax tree. We write the index vector of an occurrence instance α as $\mathrm{Idx}(\alpha)$, and its occurrence as $\mathrm{Occ}(\alpha)$.

Next, we define the dependences between occurrence instances. The analysis used in the polyhedron model already supplies the suitable granularity, since it is based on access instances. If the dependence analysis cannot yield exact results, we use, as appropriate, the optimistic ($\delta_{\subset}$) or pessimistic ($\delta_{\supset}$) approximation instead (i.e, $\delta_{\subset} \subseteq \delta \subseteq \delta_{\supset}$). δ^f denotes flow dependences (first write, then read), δ^a anti dependences (first read then write), δ^o output dependences (both writes), and δ^i denotes input dependences (both access instances are reads). For the sake of an exact presentation, we will always use $\delta_{\subset}$ and $\delta_{\supset}$ in the remainder of the paper. However, to keep the presentation clear, we also use the shorthand $I := \delta^i_{\subset}$.

It remains to include dependences arising from combining accesses or occurrences to composed occurrences. The idea is that the application of a function or operator (aside from `;`) depends on its input arguments, and the output arguments depend on the function application, creating new flow dependences. Formally: Given m occurrence instances $\alpha_1, \ldots, \alpha_m$ with $\Gamma(\mathrm{Occ}(\alpha_i)) = e_i$, and α_0 with $\Gamma(\mathrm{Occ}(\alpha_0)) = f(e_1, \ldots, e_l, \ldots, e_m)$, where $e_1, \ldots, e_l$ are only read, while $e_{l+1}, \ldots, e_m$ are only written, and $\mathrm{Idx}(\alpha_0) = \cdots = \mathrm{Idx}(\alpha_m)$, we define:

$$(\alpha_1, \alpha_0) \in \delta^f_{\subset}, \ldots, (\alpha_l, \alpha_0) \in \delta^f_{\subset} \tag{1}$$

$$(\alpha_0, \alpha_{l+1}) \in \delta^f_{\subset}, \ldots, (\alpha_0, \alpha_m) \in \delta^f_{\subset} \tag{2}$$

Let F denote the textually carried dependences defined by (1) and (2). Using this definition, we create a graph that mimics the structure of the syntax tree, and enables us to argue about different loop iterations. A vertex in this *occurrence instance graph* (OIG) $G = (OI, E)$ is an occurrence instance α. In addition to the flow dependences that are extended by F, the anti and output dependences that are discovered by the access-based dependence test are included in the edge relation E:

$$E := \delta^f_{\supset} \cup \delta^a_{\supset} \cup \delta^o_{\supset}$$

Thus, the edges of the OIG are the "normal" dependences between occurrence instances: any order on OI that is compatible with E defines a valid execution of the program.

Example 1.

Figure 1 shows the OIG of the following code fragment:

```
DO j=1,n
 DO i=1,n+1
! Occurrences:
!  [5]     [4] [1]   [3] [2]
   D(i,j)   =  A(i) **  B(j)
 END DO
END DO
```

There are five occurrences to be considered: the two reads, the exponentiation, the assignment operation and the write. The occurrences are given by the bracketed numbers above the corresponding program text. Each of these occurrences is executed for each element in $\{(i,j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq n+1\}$.

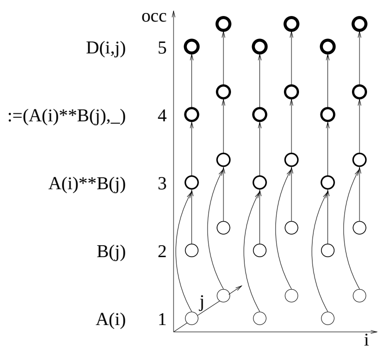

Fig. 1. OIG of Example 1 ($\mathtt{n} = 2$)

As can be guessed from Figure 1, the resulting graph G is constructed just like our syntax tree; the only difference is that each vertex represents an occurrence instance rather than an occurrence. Note that this is a conceptually clean but impractical structure since, in general, it is unbounded in size. The approach taken in the polyhedron model is to use set representations to combine different instances of a statement to a single representation.

In order to define equality on expressions, we define an equivalence relation $\equiv$ based on the input dependences δ^i. For occurrence instances α, β that represent array accesses, we define

$$\alpha \equiv \beta \Leftrightarrow (\alpha, \beta) \in \mathrm{clos}(I)$$

If we need to use approximated input dependences, we use $\delta^i_\subset$ to build the equivalence classes, because the optimistic approximation assumes *no* dependence and, thus, no equivalence in the case that the presence of a dependence could not be ascertained. Note that we are only interested in *direct* input dependences, i.e., in input dependences which are executed without any intermediate write to the same memory cell. Formally:

$$\alpha \, I \, \beta \Longrightarrow \neg(\exists \gamma : \gamma \in OI : \alpha \, \delta^a_\supset \, \gamma \wedge \gamma \, \delta^f_\supset \, \beta)$$

Two occurrence instances α_0, β_0 representing composed terms are equivalent wrt. $\equiv$ iff, for every input argument α_0, there is an equivalent input argument of β_0.
More formally: For $\alpha_0, \beta_0 \in OI$, with $\Gamma(\mathrm{Occ}(\alpha_0)) = f(e_1, \ldots, e_m)$, $\Gamma(\mathrm{Occ}(\beta_0)) = f(e'_1, \ldots, e'_m)$, $(\forall i : i \in \{1, \ldots, m\} : \; \Gamma(\mathrm{Occ}(\alpha_i)) = e_i, \Gamma(\mathrm{Occ}(\beta_i)) = e'_i)$, where f is a function or operator that is free of side effects, we define

$$\alpha_0 \equiv \beta_0 \Leftrightarrow (\forall i : i \in \{1, \ldots, m\} : \; (\alpha_i, \beta_i \notin \mathbf{W} \Rightarrow \alpha_i \equiv \beta_i))$$

Note that write references cannot be input dependent on each other. This definition avoids cycles in the condensed OIG defined in Section 2.2, which identifies occurrences that are equivalent wrt. relation $\equiv$.

2.2 Code Placement by Affine Scheduling

2.2.1 The Principle of Placement by Scheduling

Our aim is to perform code placement based on affine scheduling methods. The basic procedure here is as follows:

1. Construct the OIG (with control structures like loops being represented by sets of vertices).
2. Schedule equivalence classes of vertices in the OIG, and construct a placement function completing the schedule.
3. Generate the new syntax tree with loops governing assignment statements from the scheduled equivalence classes.

Here, Step 1 is easily accomplished by a scan across the variable accesses. The determination of equivalence classes in Step 2 is based on the input dependences computed by a dependence test such as Feautrier's [Fea91]. The subsequent calculation of a schedule in Step 2 is based on flow dependences resulting from the same dependence test. Step 3 generates the resulting code; we will not go into the deeper details of this last step: the different occurrences have to be rewritten according to $\equiv$, and some optimization of temporary variables should be performed.

Now that we have introduced the OIG (OI, E) and the additional equivalence relation $\equiv$ on its vertices, we are ready to formulate the central steps of our method.

The equivalence relation $\equiv$ divides the set OI of occurrence instances into equivalence classes $OI/\equiv$. The elements aggregated in a single equivalence class are guaranteed to represent the same value. Two expressions are considered equal if:

- they are affine expressions in the induction variables and structure parameters and provably represent the same arithmetic value, or
- their syntactic representation is equal (up to the aforementioned equality on affine subexpressions), and no write access changes the represented value in any execution path of the original program between these expressions.

Note that computing $\equiv$ at least requires the computation of transitive closures; these are not generally computable, which may lead to approximations [KPRS94]. We will later schedule the equivalence classes $OI/\equiv$ instead of the occurrence instances. However, we still have to consider the dependences between elements of different equivalence classes to obtain legal schedules. Therefore, the dependences that define the edges of the OIG also define edges of the *condensed OIG* $(OI/\equiv, E_{\equiv})$:

$$(\forall(\alpha,\beta) : \alpha,\beta \in OI : \ (\alpha,\beta) \in E \Rightarrow ([\alpha],[\beta]) \in E_{\equiv})$$

Note that – as in the OIG – all discovered dependences appear as edges in the condensed OIG (COIG).

We can now proceed and construct schedules and placements for the equivalence classes of occurrence instances rather than for statement instances.

Before we go into further details, let us compare classical code placement with code placement based on affine mappings.

2.2.2 Comparison of Placement Techniques

As mentioned earlier, classical code optimization is still largely based on purely syntactic analysis. Similarly, data flow analysis in the polyhedron model – which we use in our method – is a syntactic analysis that can additionally compute and compare the values of affine expressions in induction variables and constant parameters. Control flow information that can be expressed in terms of inequalities of those expressions can, therefore, be described exactly in this model. In order to identify loops and their induction variables, a loop recognition phase may have to be executed first [ASU86]. On the other hand, if the control flow or data flow is based on non-affine expressions in the previously recognized induction variables, we have to adopt conservative strategies – as is done in classical code optimization for all control flow decisions.

The principle of code motion – or code placement, respectively – is to move the computation of an expressions as far as possible out of a specific context, so that it has to be executed only once and further references to the computed value can be implemented by a single lookup. In the classical code optimization framework, the order in which computations are performed is given by their placement in the control flow graph. We adopt the common practice in the polyhedron model, and determine this order by a schedule function, i.e., a function which takes occurrence instances and returns an integer vector with the lexicographic order on integer vectors defining the order of execution.

The affine scheduling problem is to find a piecewise affine function θ such that

$$\alpha\, \delta_{\sqsupset}\, \beta \Rightarrow \theta(\alpha) < \theta(\beta) \qquad \text{(correctness)}$$
$$(\forall\eta : \eta \text{ piecewise affine, correct} : \ (\forall\alpha : \alpha \in OI : \ \eta(\alpha) \geq \theta(\alpha))) \qquad \text{(optimality)}$$

for arbitrary occurrence instances α, β and a dependence relation $\delta = \delta^a \cup \delta^o \cup \delta^f$. With the generalizations discussed above, we can treat occurrence instances in just the same way as the usual statement instances. Optimality cannot be guaranteed in general. But, if we restrict ourselves to affine dependence relations and to schedules that can be expressed as the pointwise minimum of affine functions, there is an optimal schedule, which the algorithm by Feautrier is able to compute [Fea92].

To stress the correspondence to classical code placement: the correctness criterion is equivalent to the availability condition of classical code placement. Many schedules meet this restriction – e.g., ordering the execution of the equivalence classes in the order of their respective lexicographic minima. The optimality criterion places computations as early as possible. This early placement is not needed in order to establish a more general context, yet it does have the effect of placing the use of a value as close to its definition as possible, reducing the scope of the variable needed to store that value.

Schedules derived this way do not guarantee that a value computed is indeed needed in every program execution. However, restricting the input of the scheduler to those points that are sources of a dependence does have this effect [Won00].

2.2.3 Placement by Affine Scheduling

Now, we can legally place the code by constructing a schedule for each vertex in the COIG $(OI/\equiv, E_\equiv)$. We may do so by choosing occurrence instances from the equivalence classes as representatives – e.g., the lexicographic minimum – and schedule these representatives. However, if an occurrence instance β is represented by α, i.e., $(\alpha, \beta) \in I$, dependences with source β still have to be considered for scheduling; therefore we have to add a corresponding subset of $E \circ I$ to the new dependence relation $E_\equiv$. The schedule then determines the execution time of a given computation. In this step, independent computations may be scheduled at the same time step. Therefore, computations that may be executed simultaneously are assigned different processing units by a a so-called allocation function. Schedule and allocation together form a space-time mapping that maps each index vector to a point in space and time and that, thus, defines when and where a computation should be executed.[3] Since this space-time mapping is a bijection, neither schedule nor allocation have any influence on the cost of the target program (they do not contribute to shrinking the index space of a given occurrence). We are free to choose any legal schedule, completed to a space-time mapping.

As noted earlier, we have to switch to a set representation of $OI/\equiv$ in an implementation, since the COIG may be infinite. These sets are then transformed by a space-time mapping (mathematically, this represents a change of basis in $\mathbb{Z}^n$). The transformed sets are then enumerated by scanning techniques such as the one employed by Omega [KPR94] or the method of Quilleré and Rajopadhye [QR00]. This is the usual transformation applied in the polyhedron model, although we apply this transformation to sets of occurrence instances instead of statement instances.

In principle, one could compute a schedule for the instances of each occurrence separately; however, this is unnecessarily complex. For each vertex $[\alpha] \in OI/\equiv$ that has a unique successor $[\beta]$ given by $E_\equiv$, there is no need to store the value computed in memory and, since, in this case, we can deduce the mapping for $[\alpha]$ from the mapping of $[\beta]$, there is no need to compute an individual space-time mapping for a set of these occurrence instances. However, the dependences with sources $[\alpha]$ still have to be considered when scheduling $[\beta]$ in lieu of the complete set $\{[\alpha], [\beta]\}$.

To be precise, a vertex $[\alpha]$ of the COIG has to be instantiated by statement instances if:

- The occurrence represented by $[\alpha]$ is a write, and $[\alpha]$ is source or target of a dependence (other than those of F), or
- $[\alpha]$ is an operator and:
  - $[\alpha]$ is the source of flow dependences to several other vertices, i.e., $(\exists [\beta], [\gamma] : [\beta] \neq [\gamma] \in OI/\equiv : ([\alpha], [\beta]) \in E_\equiv \wedge ([\alpha], [\gamma]) \in E_\equiv)$, or
  - $[\alpha]$ is the source of a flow dependence to an equivalence class that represents a write access, i.e., $(\exists [\beta] : [\beta] \in OI/\equiv : ([\alpha], [\beta]) \in E_\equiv \wedge \beta \in \mathbf{W})$.[4]

If $[\alpha]$ is a write and neither source nor target of a dependence, its schedule may be taken from the occurrence that represents the operator doing the write.
If $[\alpha]$ is an operator, there are two cases:
In the first case, an assignment statement has to be created, because the value computed is used for the computation of different values. Therefore, in order to execute as few computations as possible, an assignment to a possibly new variable has to be performed, so that further references to that value can be replaced by a simple lookup.
In the second case, a statement has to be instantiated, because the value computed may be needed outside of the code fragment under consideration.

Note that the number of dependences for which a given occurrence o is the source may actually vary depending on the point in the index space. So, it is unclear which set of occurrence instances to schedule: all instances of o together, or only separate subsets. It may be beneficial to consider the interaction of this approach with index set splitting as investigated, e.g., by Griebl, Feautrier,

[3] Note that we use the terms space-time mapping and allocation without implying that any loop iterations have to be performed in parallel; the preocessing units may well be implemented as a sequential loop.

[4] Note that the definition of $\equiv$ on occurrence instances guarantees that all occurrences of an equivalence class represent the same operation.

and Lengauer [GFL00]. Yet, we leave this issue for future work and produce the partitioning with the finest grain.

We illustrate these partitionings of occurrence instance sets with an example.

Example 2.

Figure 2 shows part of the COIG for the following code fragment:

```
DO j=1,n
 DO i=1,n+1
! Statement S1
! Occurrences:
!  [7]     [6] [1]  [3] [2]  [5] [4]
   D(i,j)  =  A(i) **  B(j)  +  C(i,i)
! Statement S2
! Occurrences:
!  [14]   [13] [8] [10] [9] [12] [11]
   E(i,j)  =  A(i) **  B(j)  +  C(i,j)
 END DO
END DO
```

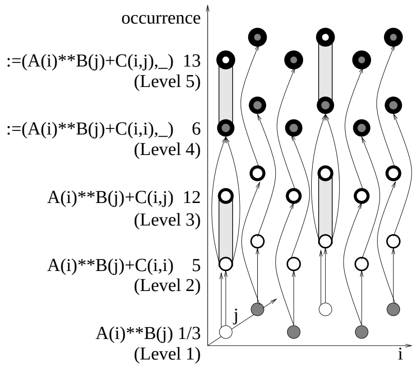

Fig. 2. COIG for Example 2, $\mathtt{n} = 2$

Again, bracketed numbers above the program text denote the respective occurrences assigned to the corresponding points in the code. The occurrence dimension in the diagram is divided into levels. The instances of occurrences 3 and 10 are *both* represented by the points on the lowest level (they belong to the same equivalence classes). The second level represents the instances of occurrence 5, the third level the instances of occurrence 12. Levels 4 and 5 represent the occurrences 6 and 13, respectively. The fact that for $\mathtt{i} = \mathtt{j}$ the same term is assigned at both levels is reflected by the shaded areas connecting the corresponding points of the diagram; the occurrences with the smaller numbers are chosen as representatives. The occurrence instances that are passed to the scheduler are shaded grey:

- **Level 4/5:** Every occurrence instance represents an assignment; however, the diagonal points of Levels 4 and 5 are equivalent, therefore only the representatives (on Level 4) need to be passed to the scheduler.
- **Level 3:** All points have a unique successor on Levels 4/5.
- **Level 2:** The diagonal elements are equivalent to points of Level 3; these points have a unique successor on Level 4 (representing Levels 4 and 5). The other points have unique successors on Level 4 (here, representing only Level 4).
- **Level 1:** The diagonal elements of Level 1 have a unique successor: both targets of dependences from those points belong to the same equivalence class. Therefore, these elements of Level 1 do not have to be scheduled. The others do have several successors and must be scheduled.

After placing equivalence classes according to a schedule, code generation can be performed as usual in the polyhedron model. In contrast to traditional optimization techniques, this results in the generation of new loop nests. In our approach, the generated code results in a statement instance for each equivalence class scheduled; i.e., a statement is created for each occurrence that is scheduled as a representative for its equivalence class. If an occurrence is scheduled because it is the source of flow dependences to several targets, a new array variable has to be introduced (formally, changing Γ) that may conveniently be indexed by the surrounding loop nest – as is done in single assignment conversion [Fea91,Coh99]. Since this introduction of new arrays may increase the memory requirements of the program, array elements may be reused following the method of Lefebvre and Feautrier [LF98] – however, this may introduce complicated array indices.

The targets that are flow dependent on these occurrences must then be replaced by lookups to the introduced arrays, with the index function given directly by the flow dependence relation between the representative of the source and the representative of the target. Note that different code may have to be generated for the same representative depending on which occurrences belong to the same equivalence class at this point in the index space.

3 Practice of Loop-Carried Code Placement

Let us now apply our method to the code fragment of Example 2. The occurrence instances OI and flow dependences of F are:

$$
\begin{aligned}
OI &= \{(j,i,o) \mid 1 \le j \le n, 1 \le i \le n+1, 1 \le o \le 14\}\\
F &= \{((j',i',o'),(j,i,o)) \mid 1 \le j' = j \le n \wedge 1 \le i' = i \le n+1 \wedge\\
&\quad ((o' \in \{1,2\} \wedge o = 3) \vee (o' \in \{3,4\} \wedge o = 5) \vee (o' \in \{5,6\} \wedge o = o'+1) \vee\\
&\quad (o' \in \{8,9\} \wedge o = 10) \vee (o' \in \{10,11\} \wedge o = 12) \vee (o' \in \{12,13\} \wedge o = o'+1))\}
\end{aligned}
$$

Since there are no other flow dependences, $E = F$. We used our tool *LooPo* to obtain the input dependences on access instances. Here, it is important that not only the last read before another read is the source for a dependence, but every read that is not "killed" by a write. We can deduce the input dependences on occurrence instances in a bottom-up traversal of the syntax tree, applying our definition to the dependences found by the access based dependence test:

$$
\begin{aligned}
I = &\{((j,i,3),(j,i,10)) \mid 1 \le j \le n \wedge 1 \le i \le n+1\} \cup\\
&\{((j,j,5),(j,j,12)) \mid 1 \le j \le n\} \cup \{((j,j,6),(j,j,13)) \mid 1 \le j \le n\}
\end{aligned}
$$

Using Omega, we then construct the equivalence classes by computing the minima of OI with respect to I; if an occurrence instance β is represented by α, i.e., $(\alpha,\beta) \in I$, we add the corresponding subset of $E \circ I$ to the new dependence relation $E_{\equiv}$. As described in Example 2, only part of the COIG needs to be scheduled. So, we now select the occurrence instances that are the source of flow dependences to several distinct occurrence instances and those that are the source of a flow dependence to a write. If we now leave out an occurrence instance β in favor of α with $(\beta,\alpha) \in E$, we have to add the appropriate subset of $E \circ E$, just as before. The resulting sets R of occurrence instances and ρ of (flow) dependences are:

$$
\begin{aligned}
R &= \{(j,i,3),(j,i,13) \mid 1 \le j \le n \wedge 1 \le i \le n+1 \wedge i \ne j\} \cup\\
&\quad \{(j,i,6) \mid 1 \le j \le n \wedge 1 \le i \le n+1\}\\
\rho &= \{((j,i,3),(j,i,6)),((j,i,3),(j,i,13)) \mid 1 \le j \le n \wedge 1 \le i \le n+1 \wedge i \ne j\}
\end{aligned}
$$

Feautrier's allocation and scheduling methods [Fea94,Fea92] then yield the following results:

occurrence	index points	schedule	allocation
3	$1 \le j \le n \wedge 1 \le i \le n+1 \wedge i \ne j$	0	(j,i)
6	$1 \le j \le n \wedge 1 \le i \le n+1 \wedge i \ne j$	1	(j,i)
6	$1 \le j \le n \wedge 1 \le i = j \le n$	0	(j,i)
13	$1 \le j \le n \wedge 1 \le i \le n+1 \wedge i \ne j$	1	(j,i)

Using Omega's code generator for enumerating the corresponding polyhedra and introducing the appropriate help variables, we obtain the following target code:

```
DO t2=1,n
   DO t3=1,t2-1
      TMP1(t3,t2)=A(t3)**B(t2)
   END DO
   TMP2(t2)=A(t2)**B(t2)+C(t2,t2)
   D(t2,t2)=TMP2(t2)
   E(t2,t2)=TMP2(t2)
   DO t3=t2+1,n+1
      TMP1(t3,t2)=A(t3)**B(t2)
   END DO
```

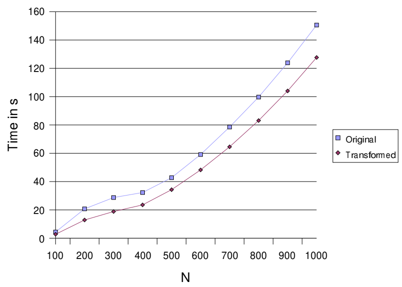

Fig. 3. Timings on a 1 GHz Pentium III

```
END DO
DO t2=1,n
   DO t3=1,t2-1
      D(t3,t2)=TMP1(t3,t2)+C(t3,t3)
      E(t3,t2)=TMP1(t3,t2)+C(t3,t2)
   END DO
   DO t3=t2+1,n+1
      D(t3,t2)=TMP1(t3,t2)+C(t3,t3)
      E(t3,t2)=TMP1(t3,t2)+C(t3,t2)
   END DO
END DO
```

Figure 3 shows the timing results on a 1 GHz Pentium III running Linux. The times are accumulated over 500 runs of the code fragment. The transformed code performs 12% to 37% better than the original one.

4 Remarks

Our method is aimed at improving source code given by a human programmer for better performance, where most work is done in the computation on array elements – in scientific applications, these are usually expensive floating-point operations.

We are currently in the process of incorporating an implementation of our method into our code restructurer *LooPo*. We plan to perform more substantial performance measurements when the implementation is available.

We have only used syntactic equivalence, augmented by equivalence on expressions that could be identified as affine for a certain vector space. Especially for the improvement of code written by a human, further exploitation of associativity and commutativity is important. Also, the possible impact of different schedulers and allocators on the transformed program remains to be investigated.

We made computations the cost-determining factor, which, again, is a common choice. However, for modern machine architectures, a more realistic cost model should also consider different costs for memory access, e.g., between data retrieval from cache and main memory. Especially in the context of parallel programs, where local computations may be cheaper than communication with other processors, this is a crucial point. However, we leave it for future work.
Additionally, the overhead introduced by the new arrays and possibly complicated index functions still has to be reduced; ideally, the cost model should be made sensitive to computations in index functions and the time spent in memory lookups.

Acknowledgements. This work is supported by the DFG through project *LooPo/HPF*.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Coh99] A. Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages.* PhD thesis, PRiSM, Université de Versailles, 1999.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem. *Int. J. Parallel Programming*, 21(5/6), 1992. Two-part paper.

[Fea94] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[FGL01] P. Faber, M. Griebl, and C. Lengauer. Loop-carried code placement. In *Euro-Par 2001*, LNCS, 2001. To appear.

[GFL00] M. Griebl, P. A. Feautrier, and C. Lengauer. Index set splitting. *Int. J. Parallel Programming*, 28(6):607–631, 2000.

[KMP+96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library*, 1996.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Department of Computer Science, Univ. of Maryland, 1994.

[KPRS94] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. Technical Report UMIACS-TR-95-48, University of Maryland, College Park, MD 20742, apr 1994.

[KRS99] J. Knoop, O. Rüthing, and B. Steffen. Expansion-based removal of semantic partial redundancies. In S. Jähnichen, editor, *Compiler Construction*, LNCS 1575, pages 91–106. Springer-Verlag, 1999.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[LF98] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *PC*, 24:649–671, 1998.

[QR00] F. Quilleré and S. Rajopadhye. Code generation for automatic paralelization in the polyhedral model. *Int. J. Parallel Programming*, 28(5):469–498, 2000.

[SKR90] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In N. D. Jones, editor, *3rd European Symposium on Programming (ESOP'90)*, LNCS 432, pages 389–405. Springer-Verlag, 1990.

[Slo99] A. Slowik. *Volume Driven Selection of Loop and Data Transformations for Cache-Coherent Parallel Processors*. PhD thesis, Universität-GH Paderborn, 1999.

[Ste96] B. Steffen. Property-oriented expansion. In R. Cousot and D. A. Schmidt, editors, *Static Analysis, 3rd International Symposium (SAS '96)*, LNCS 1145, pages 22–40. Springer-Verlag, 1996.

[Won00] D. Wonnacott. Extending scalar optimizations for arrays. In M. Gupta and S. M. J. Moreira, editors, *13th Workshop on Languages and Compilers for Parallel Computing (LCPC 2000)*, LNCS. Springer-Verlag, 2000. To appear.