Loop-Carried Code Placement

Peter Faber, Martin Griebl, and Christian Lengauer

Fakultät für Mathematik und Informatik. Universität Passau
D–94030 Passau, Germany
{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 the polyhedron model can be used to implement code placement techniques that exploit equalities of expressions that hold between loop iterations. We also present preliminary results for 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 [9,5]. However, a loop may have to be unrolled infinitely often to remove *all* redundancies, which would lead to data structures that are unbounded in size. The polyhedron model [3,7] offers a way to analyze memory accesses in loops by considering symbolically expressible sets of loop iterations. These descriptions are usually restricted to affine expressions in the index variables of surrounding loops and constants. Thus, in general, it is only possible to analyze code fragments. The representation of loop iterations in the polyhedron model provides a means for transformations based on the analysis of array accesses in loop nests: Wonnacott extended constant propagation and dead code elimination to arrays [10].

In this work, we use the polyhedron model to extend a further class of optimization techniques to arrays – namely code placement optimizations. Our aim is to improve code 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.

2 Loop-Carried Code Placement

The central question of code placement is where to put the computation of an expression in order to ensure this computation to be performed only once and only if needed. Relevant data flow information is gathered in most approaches by syntactic analysis. There are also approaches for the analysis of scalars that introduce semantic properties, such as the one employed by Steffen, Knoop and Rüthing [5], which exploits equalities resulting from assigning terms to variables.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 230–235, 2001.

We restrict ourselves to a syntactic – but loop-carried – equivalence of expressions: we use affine expressions in loop indices to infer equality. This enables us to prove the equality of terms that are not syntactically identical, e.g., `C(i,i)` and `C(i,j)` in Ex. 1, which represent the same value for `i` = `j`.

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

Example 1.

```
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
```

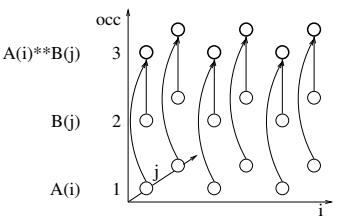

Fig. 1. OIG for `A(i) ** B(j)` of Ex. 1 (`n` = 2)

In the program text above, the term `A(i)**B(j)` occurs twice and represents the same value in both locations, so it suffices to compute this term once. However, if `i` = `j`, the complete terms on the right hand side (RHS) of the assignments are identical; in this case, it suffices to compute the RHS once. We remove the redundant computations that occur in the second statement (the computation of `A(i)**B(j)` for all `i`, `j`, and the computation of `A(i)**B(j)+C(i,j)`, `i` = `j`), replacing them by memory lookups.

2.1 Basic Structures

We use generalizations of the usual structures in the polyhedron model that are described in detail, e.g., by Cohen [1] and by Lengauer [7].

A subroutine is a term combined of statements, which are terms composed of an operator and, possibly, a list of arguments. Each term is identified with a unique number, its *occurrence*. In contrast to the conventional polyhedron model, which is based on read and write *accesses*, we base our method on occurrences, enabling references to any point in the program text. In Ex. 1, the occurrences are placed in brackets on the comment lines above the corresponding code. Note that the order of occurrences corresponds to the order of target code a compiler would generate. Analogously to the conventional polyhedron model, we consider different *occurrence instances*. These are – in a nest of n loops – represented by a vector $\alpha \in \mathbb{Z}^{n+1}$ with the first n components given by the *index vector* defining the loop iteration and the last component given by the occurrence. The set of occurrence instances is denoted OI. Fig. 1 shows the instances of occurrences

1–3 from Ex. 1, for $\mathtt{n} = 2$. The index space spans the $\mathtt{i} \times \mathtt{j}$-plane of the diagram, while occurrences are assigned to the vertical `occ`-axis.

The usual dependence analysis in the polyhedron model – such as Feautrier's [2] – captures dependences between occurrences that represent accesses. However, in order to model execution on the level of occurrences, we have to include dependences arising from the combination of occurrences. For flow dependences, the application of a function or operator depends on its input arguments, and the output arguments depend on the function application. Anti and output dependences do not occur between compound occurrences. We combine all these dependences in a relation E.

With E as edge relation, we create a graph (OI, E) that mimics the structure of the terms of the loop body and enables us to argue about different loop iterations. The edges in this *occurrence instance graph* (OIG) are the "normal" dependences between occurrence instances: any order on OI that is compatible with E defines a valid execution of the program.

Fig. 2 shows the part of the OIG that contains the compound occurrence instances of Ex. 1. The solid arrows represent the dependences in E. The approach taken in the polyhedron model to represent such structures that are unbounded in size is to use set representations to combine different instances to a single set. E.g., the Omega tool [4] provides this functionality.

2.2 Code Placement by Affine Scheduling

In order to determine whether a value is computed twice, we have to introduce a notion of equality. Our equality relation $\equiv$ is based on the input dependence relation δ^i. We are only interested in dependences between reads that are executed without any conflicting write in between. Fig. 2 shows the OIG of the compound terms of Ex. 1 with dashed arrows indicating equivalences as defined above (the arrows indicate the ordered, non-reflexive part of $\equiv$).

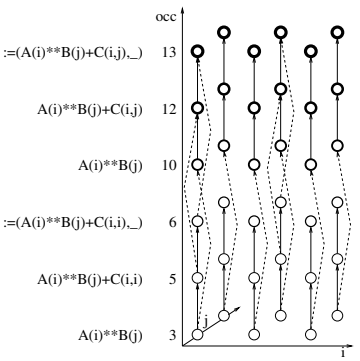

Fig. 2. OIG for the compound terms of Ex. 1 ($\mathtt{n} = 2$)

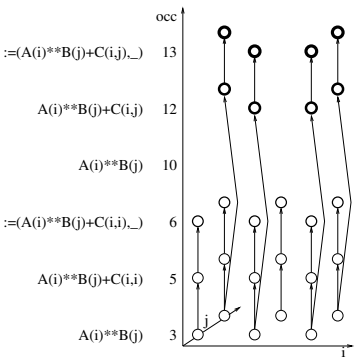

Fig. 3. COIG representation for the compound terms of Ex. 1 ($\mathtt{n} = 2$)

The relation $\equiv$ divides the OI into equivalence classes $OI/\equiv$. The occurrence instances in a single equivalence class are guaranteed to represent the same value. So, in order to compute a value only once, we have to generate a single computation for each element in $OI/\equiv$. Still, these computations must be executed in an order compatible with E. We assert this by computing a schedule for the equivalence classes. I.e., we construct the condensation of the OIG and compute a schedule for the vertices of this *condensed occurrence instance graph* (COIG).

A representation of the COIG that can be directly passed to a scheduler can be obtained by choosing a single representative of an equivalence class – we choose the lexicographic minimum – and adjusting dependence information appropriately. Fig. 3 shows this representation for Ex. 1 ($\mathtt{n} = 2$).

We may now compute a schedule for the vertices $OI/\equiv$ using, e.g., Feautrier's scheduler [3] – just as one would compute a schedule for the *statement instances* in conventional loop parallelization – and obtain a transformation that is applied to the index space of $OI/\equiv$. The resulting index space description defines a legal execution order and places lookups as early as possible. From this description, we can generate code using scanning techniques as the one employed by Omega [4] or the one of Quilleré and Rajopadhye [8]; this code is then rewritten to use newly introduced arrays for storing the value of scheduled occurrence instances.

3 Results and Final Remarks

We applied our method to Ex. 1. We transformed the loop nest according to Feautrier's scheduler, scanned the resulting polyhedron with the Omega code generator, and introduced auxiliary variables. This resulted in the following 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
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
```

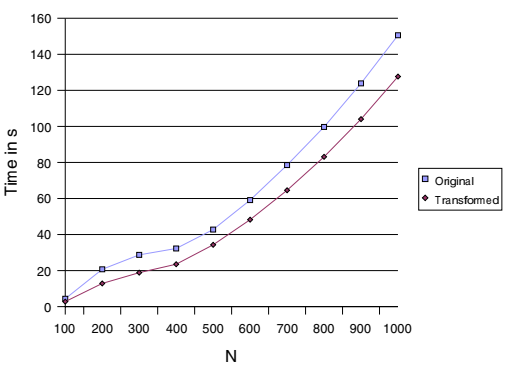

Fig. 4. Timings on a 1 GHz Pentium III

Fig. 4 shows the timing results on a 1 GHz Pentium III Xeon 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.

Our method is aimed at speeding up source code written by a human, where most time is spent in the computation on array elements (in scientific codes, these are usually expensive floating-point operations).

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

An optimization that has to be performed when applying this method to real codes is minimizing array sizes following, e.g., Feautrier and Lefebvre [6].

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 on the transformed program remains to be investigated.

We chose computations as the cost-determining factor. A more realistic cost model should consider different costs for memory accesses, e.g., cache vs. main memory. This is a crucial point that remains 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 DAAD through project PROCOPE and by the DFG through project *LooPo/HPF*.

References

1. A. Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, PRiSM, Université de Versailles, 1999. 231
2. P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991. 232
3. P. Feautrier. Some efficient solutions to the affine scheduling problem. *Int. J. Parallel Programming*, 21(5/6), 1992. Two-part paper. 230, 233
4. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994. 232, 233
5. 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. 230
6. V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *PC*, 24:649–671, 1998. 234
7. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993. 230, 231
8. F. Quilleré and S. Rajopadhye. Code generation for automatic paralelization in the polyhedral model. *Int. J. Parallel Programming*, 28(5):469–498, 2000. 233
9. 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. 230

10. 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. 230