Replicated Placements in the Polyhedron Model

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. Loop-carried code placement (LCCP) is able to remove redundant computations that cannot be recognized by traditional code motion techniques. However, this comes possibly at the price of increased communication time. We aim at minimizing the communication time by using replicated storage.

1 Introduction

The basic idea of the polyhedron model is to create a mathematical model of the computation to be done. This representation can be transformed in order to carry out code optimizations or automatic parallelization. In earlier work, we developed an optimization technique called *loop-carried code placement* (LCCP), which is able to remove redundant computations that are executed in different iterations of a loop. However, this improvement may come at the price of increased communication time. This is because intermediate results have to be stored in arrays which may introduce communications – actually a problem that also occurs in other programs (where intermediate results are stored).

Communication time may depend on various factors. An effective placement method should take into account as many specific traits of the underlying system as possible. Such specific traits may include certain communication patterns for which efficient communication code can be generated and should have the ability to use replicated data storage in order to decrease communication time.

Our present aim is to model as many of these traits as flexibly as possible with regard to some underlying cost model. The idea is then to "plug in" a cost function that evaluates the current placements and choose the placements that incur the least cost. Section 2 gives a short example of the use and limits of LCCP. In Section 3, we discuss the basic model used in our method. Section 4 describes the method of computing a placement in detail.

2 Motivation

Primarily, we are concerned with improving the results of an LCCP transformation, although the use of replicated placements is not limited to this scenario. In Example 1, a code fragment is given that is very well suited for applying LCCP.

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 303–308, 2003.

Example 1. Let us consider the following code fragments:

```
!HPF$ INDEPENDENT
     DO j=2,N-2
!HPF$ INDEPENDENT
        DO k=1,N
          L(k,j)= (y(j)  *2)*k&
                 -(y(j+2)*2)/k&
                 +(y(j+1)*2)+k&
                 +(y(j-1)*2)
        END DO
     END DO
```

```
!HPF$ INDEPENDENT
     DO j=1,N
       tmp(j)=y(j)*2
     END DO
!HPF$ INDEPENDENT
     DO j=2,N-2
!HPF$ INDEPENDENT
        DO k=1,N
          L(k,j)= tmp(j)  *k &
                 -tmp(j+2)/k &
                 +tmp(j+1)+k &
                 +tmp(j-1)
        END DO
     END DO
```

The (synthetic) code fragment on the left contains four computations of `y(j)*2`. The same value is computed repeatedly for different iterations of both the `k`-loop and the `j`-loop. LCCP can transform this code fragment into the one on the right (the actual transformation depends on the scheduler used). In the transformed code, `y(j)*2` is calculated once and then assigned to a temporary.

Let us suppose that `L` is `(BLOCK,BLOCK)`-distributed. Depending on the distribution of `y`, different distributions of `tmp` may be beneficial: most probably, an alignment with `y` is a good approach; if `y` is replicated, it is probably more useful to align with `L`.

3 The Model

In this section, we discuss our model in closer detail. We give a representation for replicated storage in our model, and finally show how to adapt dependence information to this representation.

3.1 Base Language

Our method depends on the presence of an initial data placement. If we restrict ourselves to `HPF` programs, the user has already defined some data placement. The result of our method is a set of further placements that can be used in an `HPF` (or `HPF`-style) program for intermediate computations whose results are stored in arrays. Therefore, our method can be viewed as a compilation phase within an `HPF` compiler, or possibly as a preprocessor.

3.2 Occurrence Instances

We consider occurrence instances with dependence relations between them as described in our earlier work [FGL01b] (and in a technically expanded version

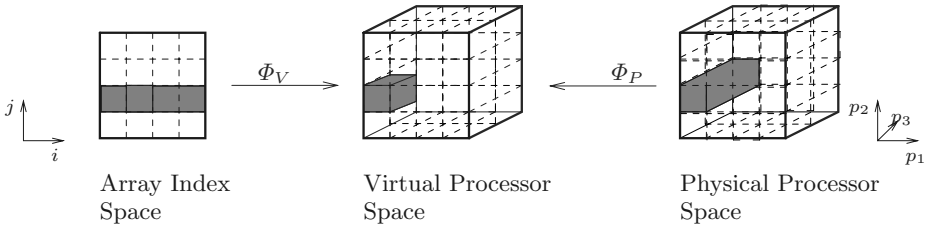

Fig. 1. Mappings representing a distribution: Φ_V maps array elements to a virtual processor, Φ_P maps physical processors to a virtual processor.

[FGL01a]), which is based on the polyhedron model [Lam74]. Thus, every computation executed can be viewed as a function application $f(e_1, \ldots, e_q, \ldots, e_r)$, where expressions $e_1, \ldots, e_l$ are only read, while $e_{q+1}, \ldots, e_r$ are only written, and each function application is assigned a unique number – its occurrence. This means that our method works on the expression level (and not – as usual – on the coarser level of statements). In the polyhedron model, placement functions defining where to execute a given occurrence are affine functions in the indices of loops surrounding the occurrence. We view these n-dimensional affine functions as linear functions in an $(n{+}1)$-dimensional space using homogenous coordinates.

We employ a dependence analysis as the one of Feautrier [Fea91] to produce a dependence graph on occurrence instances which we call an *occurrence instance (dependence) graph* (OIG).

Actually, the usual input for our method will not be an OIG, but its reduced form, a *condensed occurrence instance graph* (COIG) as defined in our earlier work [FGL01b]. In a COIG, equivalent occurrence instances are reduced to a single representative (a single occurrence instance). However, for the work presented here, this distinction is unimportant.

3.3 How to Model Replication

Our aim is to use replication (of data and computations) to reduce communication costs. The replicated mapping of a point $\alpha \in \mathbb{Z}^{n+1}$ to a subset of $\mathbb{Z}^m$ can be described as follows: we introduce a template T (corresponding to $\mathbb{Z}^m$) and a pair of affine functions $\Phi = (\Phi_V, \Phi_P)$. α is then mapped to T via Φ_V, while all the points of $\mathbb{Z}^m$ to which α should ultimately be mapped are themselves mapped to that same template element by Φ_P (see Figure 1).

The set of processors on which an occurrence instance α is to be stored is:

$$\Phi(\alpha) = \Phi_P^{-1} \circ \Phi_V(\alpha)$$

A consistency condition is that the image of Φ_V be a subset of the image of Φ_P. For the placements given as input, we can safely assume this; the placements we compute in Section 4 satisfy this property by construction.

This approach corresponds to the description of replicated data in `HPF`, as sketched in Figure 1. An array, represented by the square with dimensions i and j on the left, is to be placed on a three-dimensional processor grid on the right.

3.4 Dependences in the Presence of Replication

In order to estimate the costs for communication induced by the target program, we view the dependences with respect to space coordinates – i.e., after application of the placement relation $\Phi = (\Phi_V, \Phi_P)$. Since neither of the two functions defining the placement relation is necessarily invertible, it is not immediately clear how to represent dependences in the target program.

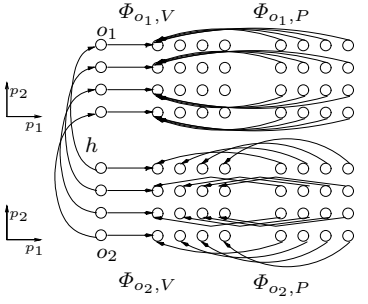

Fig. 2. Dependence from instances of occurrence o_1 to instances of occurrence o_2.

Figure 2 shows an example of a dependence given by an h-transformation h as introduced by Feautrier [Fea91]. In the upper part of the figure, four instances of an occurrence o_1 are mapped to a virtual processor grid by $\Phi_{o_1,V}$; physical processors are mapped to the same grid by $\Phi_{o_1,P}$. The h-transformation maps a target occurrence instance α to its source occurrence instance $h(\alpha)$. The instances of occurrence o_1 are stored in a replicated fashion (all processors with the same p_1 coordinate own a copy of an occurrence instance).[1] Just as the mappings $(\Phi_{o_1,V}, \Phi_{o_1,P})$ define a placement relation for the instances of o_1, the mappings $(\Phi_{o_2,V}, \Phi_{o_2,P})$ define a placement for the instances of o_2, as depicted in the lower part of Figure 2. The instances of o_2 are only allocated on the first column of the processor space by their placement.

This means that all instances of occurrence o_2, each of which depends on a different instance of o_1, may "choose" the source processor from which to load the data needed for computation.

With the names taken from Figure 2, the new dependence relation is $h' = \Phi^{-1}_{o_1,P} \circ \Phi_{o_1,V} \circ h \circ \Phi^{-1}_{o_2,V} \circ \Phi_{o_2,P}$. We observe that there are two sets involved in the computation of h':

- $\Phi^{-1}_{o_1,P} \circ \Phi_{o_1,V}(\alpha)$: the possible sources of α (copies of the same value).
- $\Phi^{-1}_{o_2,V} \circ \Phi_{o_2,P}(\beta)$: the set of occurrence instances to be executed on the processor with space coordinates β.

If $\Phi^{-1}_{o_1,P} \circ \Phi_{o_1,V}(\alpha)$ is a set, we may choose any point within that set for the representation of our dependence. Therefore, we can account for $\Phi^{-1}_{o_1,P}$ by using a generalized inverse that yields a single point.

However, $\Phi^{-1}_{o_2,V} \circ \Phi_{o_2,P}$ has to be represented as a set, e.g., by using two mappings to represent the relation h', just as with placement relations.

4 The Placement Method

With this description of replicated placements and resulting dependence relations, we can compute further placements. In order to allow almost any cost

[1] Note that occurrence instances may represent either, computations or data.

model to be "plugged in", we choose a naïve approach that basically considers all possible placements and selects the cheapest solution (names as in Figure 2):

1. For all sets of occurrence instances: propagate a placement candidate from source o_1 to target o_2 ($\Phi_{o_1} \circ h$) and from target to source ($\Phi_{o_2} \circ h^{-1}$).
2. For each combination of candidate placements for the occurrence instances:
   a) Compute the dependence information according to the current placements. Candidate placements Φ_1, Φ_2 can be combined into a new placement Φ_3 placing all occurrence instances to both sets by asserting $\Phi_{3,P}^{-1} \circ \Phi_{3,V}(\alpha) \supseteq \Phi_1(\alpha) \cup \Phi_2(\alpha)$ for all α.
   b) Compute a cost for the given dependence information, select the combination of placements that incurs the lowest cost.

As noted elsewhere [FGL01b], it is not necessary to compute a placement for *all* occurrence instances of a program. Which sets do need placements can be deduced from the dependence information.

Placements are ultimately propagated from data. Therefore, we have to consider the transitive closure of the dependence relation in Step 1. Then we compute placement relations that can be expressed by two affine mappings. If an occurrence o_2 depends on an occurrence o_1 with h-transformation h, the placements Φ_{o_1}, Φ_{o_2} should satisfy

$$\Phi_{o_2,P}^{-1} \circ \Phi_{o_2,V}(\alpha) \subseteq \Phi_{o_1,P}^{-1} \circ \Phi_{o_1,V}(h(\alpha)) \tag{1}$$

If placement relations are propagated from source to target, this can be guaranteed by computing the Φ_{o_2} as: $\Phi_{o_2,P} := \Phi_{o_1,P}, \Phi_{o_2,V} := \Phi_{o_1,V} \circ h$.

If the placement is propagated from target to source, the computation is a bit more complex since we have to consider dependences to several different tragets. However, we can compute a placement relation that ensures condition (1) by using replication: the smallest subspace for which replication has to occur to satisfy condition (1) can be computed as the solution of a linear equality system.

5 Preliminary Results

Let us reconsider Example 1. We use a block distribution for both dimensions of `L`, and a replicated distribution for `y` along the first dimension of `L` (with `y(j)` aligned to `L(*,j)`). We compiled the code using the ADAPTOR compiler and measured the runtime on 16 nodes of an SCI-connected PC-cluster. We applied a cost model that only distinguishes between communication patterns (local data [cost 1], shifts [cost 10], array-triplets [cost 100], general accesses [cost 1000]). According to that cost model, we had to choose replicated storage for `tmp` in our LCCP-optimized code. This resulted in about 10% improvement wrt. the case had we chosen a distribution that can be expressed by a function such as an alignment with `L` (which already performed better than the original code).

6 Related Work

The dHPF compiler uses a similar approach to take advantage of replication for computation partitioning [MCAB+02]. It can generate code for a union `ON HOME` specifications, which can therefore be propagated from a dependence target to its sources. This statement-driven approach leaves the control structure constant.

In contrast, our method is based on a fine-grained polyhedron model, i.e., we represent calculations of expressions inside loops as integer polyhedra and dependences as relations between these polyhedra. The control structure is later regenerated from the geometric description by loop scanning [QR00].

During the last decade, several placement algorithms based on the polyhedron model have been proposed, such as the placement algorithm by Feautrier [Fea94]. However, these methods do not consider replication.

7 Future Work

We are currently implementing our method as part of our code restructurer LooPo. We will conduct experiments on a substantial set of benchmarks when a stable implementation is available. In addition, we are working on an accurate cost model for our main `HPF` compiler, ADAPTOR. Finally, an important optimization is the partitioning of the iteration space into a form for which efficient code can be generated; loop transformations in the polyhedron model typically ignore the cost of enumerating loops, which is not always precise enough.

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

References

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

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

[FGL01a] P. Faber, M. Griebl, and C. Lengauer. A closer look at loop-carried code replacement. In *Proc. GI/ITG PARS'01*, PARS-Mitteilungen Nr.18, pages 109–118. Gesellschaft für Informatik e.V., November 2001.

[FGL01b] P. Faber, M. Griebl, and C. Lengauer. Loop-carried code placement. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Euro-Par 2001: Parallel Processing*, LNCS 2150, pages 230–234. Springer, 2001.

[Lam74] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.

[MCAB+02] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarría-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the Rice dHPF compiler. *Concurrency and Computation: Practice and Experience*, 14(8–9):741–767, July/August 2002.

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