Index Set Splitting*

Martin Griebl
FMI, Universität Passau
D-94030 Passau, Germany
griebl@fmi.uni-passau.de

Paul Feautrier
PRiSM, Université de Versailles
45 avenue des États-Unis
F-78035 Versailles, France
Paul.Feautrier@prism.uvsq.fr

Christian Lengauer
FMI, Universität Passau
D-94030 Passau, Germany
lengauer@fmi.uni-passau.de

19th October 1999

Abstract

There are many algorithms for the space-time mapping of nested loops. Some of them even make the optimal choices within their framework. We propose a preprocessing phase for algorithms in the polytope model, which extends the model and yields space-time mappings whose schedule is, in some cases, orders of magnitude faster. These are cases in which the dependence graph has small irregularities. The basic idea is to split the iteration domain of the loop nests into parts with a regular dependence structure and apply the existing space-time mapping algorithms to these parts individually.

This work is based on a seminal idea in the more limited context of loop parallelization at the code level. We elevate the idea to the model level (our model is the polytope model), which increases its applicability by providing a clearer and wider range of choices at an acceptable analysis cost.

Index set splitting is one facet in the effort to extend the power of the polytope model and to enable the generation of competitive target code.

1 Introduction

Space-time mapping methods for the automatic parallelization of loop nests relate every instance of every statement to a virtual point in time (*schedule*) and a virtual processor (*allocation*) [Fea96, Len93]. Linear algebra and linear programming are typically employed in the search for schedules and allocations. The information this search is based on is the set of dependences between different instances of the statements.

*A short version has been presented at the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99) [GFL99].

For the case of uniform dependences, Darte, Khachiyan and Robert [DKR91] proved that there are methods which yield a schedule with (asymptotically) optimal latency. However, for the case of affine, non-uniform dependences, this optimum is sometimes missed by orders of magnitude. The use of different schedules for different iterations of the same statement frequently improves this situation. Thus, the idea of this paper is, to partition the index sets of all statements independently of the problem size into a fixed number of parts and compute individual schedules for each part.

In the same way, it turns out that a piecewise defined placement function might improve the quality of an allocation, but this has never been pointed out in the literature.

2 Definitions

We are given a dependence graph whose nodes are statements and whose edges are dependences. Each node S is associated with a subset $I(S)$ of $\mathbb{N}^{p_S}$, where p_S is the nesting depth of S. An edge e from S to T is associated with a dependence d_e, a relation from $I(S)$ to $I(T)$.

The elements of $I(S)$ are called *iteration vectors*. Each iteration vector is associated with an execution of statement S, which we also call an *operation*. The execution of S for iteration vector x is denoted $\langle S, x\rangle$. In more abstract contexts, operations are denoted by letters like $u, v, \ldots$. The fact that v *depends* on u is written $u\,\delta\,v$. The set of operations, i.e., the disjoint union of all $I(S)$ is named Ω.

In the following, all iteration domains and dependence relations are parametric polytopes. In other words, they are defined by systems of affine constraints with parameters. To simplify the presentation, only one parameter, named n, is taken into account, meaning that the size of the iteration domain increases with n. We assume n to be unbounded.

A schedule is a function θ from the set of operations to the set of integers which satisfies the following causality condition:

$$\forall u, v \in \Omega : u\,\delta\,v \Rightarrow \theta(u) + 1 \leq \theta(v) \tag{1}$$

As a matter of fact, one can omit the integrity condition on schedules since, if θ satisfies the causality condition, then so does $\theta'(u) = \lfloor \theta(u) \rfloor$ [Qui87].

From a causal schedule, one can deduce a parallel program whose figure of merit is its latency:

$$L = \max_{u\in\Omega} \theta(u) - \min_{u\in\Omega} \theta(u)$$

The latency can be interpreted either as the running time on a parallel computer with "enough" processors, or as the minimum number of synchronization points. Whatever the interpretation, it is clear that the latency must be minimized.

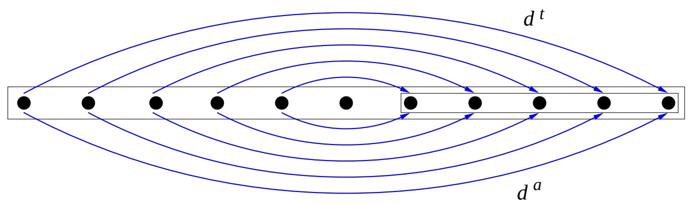

Figure 1: Simple example showing the necessity of splitting

3 Statement of the Problem

It is not generally possible to find an arbitrary schedule with minimum latency. Usually, one restricts the search to a subset of all possible functions, the functions which are affine in the loop counters. Conceptually, one builds an affine template for every schedule function (i.e., an affine function with unknown coefficients) and writes inequality (1) for all possible values of u and v. The unknowns are the coefficients of the scheduling functions, and it is easy to see that the resulting constraints are affine in these unknowns. This must be done for all values of n, yielding an infinite set of constraints. Fortunately, thanks to special properties of affine functions, this set can be shown to be equivalent to a finite set of affine constraints, which can be solved by the usual linear programming methods.

For some problems, the resulting linear program is found to be infeasible. In this case, one resorts to multi-dimensional schedules. One can find a maximal subset of the dependences which still gives a feasible program. The resulting function is the first component of the multi-dimensional schedule. Then one applies the same algorithm to the unsatisfied dependences, obtaining the next component of the schedule, and so on until all dependences are satisfied.

One may wonder whether the schedule found in this way bears any relation to the minimum latency schedule. It has been proved [DV96] that, when all dependences are uniform, the two schedules are equivalent in the asymptotic sense. However, there are well known counter examples showing that this is not true for arbitrary affine dependences.

Example 1 Consider:

```
do i = 0, 2*n
  a(i) = a(2*n-i)
end do
```

The index set and its dependence graph are given in Figure 1. The best affine schedule is:

$$\theta_1(i) = i/2 \tag{2}$$

while the minimum latency schedule is:

$$\begin{aligned} \theta_2(i) &= 0 \ \ \textbf{if}\ \ 0 \le i \le n && (3)\\ &= 1 \ \ \textbf{if}\ \ n < i \le 2n && (4) \end{aligned}$$

θ_2 can be found by splitting the iteration domain into two subdomains, $I_1 = [0, n]$ and $I_2 = [n+1, 2n]$, and postulating two separate scheduling functions in I_1 and I_2. The details of the resolution method are not affected by the splitting; the number of unknowns, however, is doubled. This splitting can also be interpreted as a code transformation yielding the program:

```
do i = 0, n
   a(i) = a(2*n-i)
end do
do i = n+1, 2*n
   a(i) = a(2*n-i)
end do
```

followed by the application of any convenient scheduling algorithm.

Our aim in this paper is to derive an algorithm for deciding when splitting is useful, and for finding these splits.

4 Related Work

Our notion of index set splitting seems very similar to tiling [AI91]: both techniques partition the index sets. Tiling is still a very active research area [FGRT98]. However, the goal of tiling has been either to increase granularity (e.g., [ARY98]), or to block for cache optimization (e.g., [BKA97]), or simply to map virtual processors to real processors. In all these cases, the idea is to enumerate the given index set in a higher-dimensional space: one set of dimensions for the tiles and another set of dimensions for the points inside a tile; all tiles are treated equally. In contrast, index set splitting does not change the number of dimensions but benefits from an individual treatment of the various partitions.

Like index set splitting, the scheduling method by Feautrier [Fea92a, Fea92b] can also result in piecewise affine functions. However, the schedules found are minima of a finite set of affine functions, and most piecewise affine schedules cannot be cast in this form. Example 1 is a case in point.

The idea of index set splitting goes back to Wolfe [Wol89], and further to Allen/Kennedy [AK87] and Banerjee [Ban79]. Our method expands on these seminal efforts by incorporating them into the polytope model.

The work most closely related is by Jemni and Mahjoub [MJ95, MJ96] and deals with partitioning the index set at points where the type of a dependence changes, e.g., from true to anti. Since their method is not based on a model, they can separate the index domain only along planes parallel to the coordinate directions.

5 Analysis

Let us first explain why schedule (2) is suboptimal. An arbitrary affine function can be written as follows:

$$\theta(S, x) = \tau_S.x + \sigma_S$$

and obviously has the property that

$$\theta(S, x + v) - \theta(S, x) = \tau_S.v$$

depends on v but not on x. Suppose that there is a dependence from (S, x) to $(S, x + v)$ for some x and v. Then:

$$\theta(S, x + v) - \theta(S, x) = \tau_S.v \geq 1$$

Iterating this result k times, we obtain:

$$\theta(S, x + kv) - \theta(S, x) \geq k \tag{5}$$

The concept of latency can be extended to individual statements S:

$$L_S = \max_{x \in I(S)} \theta(S, x) - \min_{x \in I(S)} \theta(S, x)$$

and it is clear that the latency of a statement is a lower bound on the latency of the program. From (5) we deduce:

$$L_S \geq \max\{k \mid x + kv \in I(S)\} - \min\{k \mid x + kv \in I(S)\}$$

Since $I(S)$ depends linearly on a size parameter n, we may expect that L_S also increases linearly with n.

In Example 1, we have a one-dimensional dependence vector, $v = (2)$, from instance $n-1$ to $n+1$. Since v can be iterated n times within the index space, we have a latency of at least n for the execution of statement S, which is exactly the latency of schedule (2).

Suppose now that $I(S)$ has been partitioned into two subdomains, I_1 and I_2, and that $x \in I_1$ and $x + v \in I_2$. The above reasoning no longer applies, since the schedule θ is not necessarily the same affine function in I_1 and I_2. Hence, the above estimate of the latency of S no longer holds, and we may hope for a better schedule.

If we split the index set of the target statement of a dependence d into I_1, which contains the image of d (and therefore has to satisfy the schedule constraints for d), and I_2, which contains the rest of the index space (and therefore need not satisfy constraints which are due to d), we get a constant schedule for I_2 (if d is the only dependence). If there are no dependences inside I_1, i.e., the domain of d is contained in I_2, we also get a constant schedule for I_1.

For our example, the dependence analysis gives us the information that only the instances $n+1, \cdots, 2n$ are in the range of the existing dependences. Since iterations $0, \cdots, n$ are not images of any dependence, they need not satisfy any scheduling condition and, thus, should be given an individual schedule template. This splitting enables us to find schedule (3), which has a constant latency.

The same reasoning applies to placement problems. Here the goal is to find a placement function $\pi(S, x) = \lambda_S.x + \mu_S$ which gives the number of the processor that executes operation $\langle S, x\rangle$. The goal is that two operations which access the same memory location are executed on the same processor. If $\langle S, x\rangle$ and $\langle S, x + v\rangle$ access the same memory cell, we must have

$$\pi(S, c) = \pi(S, x + v) \Rightarrow \lambda_S.v = 0$$

and all operations $\langle X, x + kv\rangle$ will use the same processor as $\langle S, x\rangle$, whether there is a reason or not. On the other hand, putting $\langle S, x\rangle$ and $\langle S, x + v\rangle$ in different subsets of the iteration space invalidates this reasoning and may result in better parallelism. In the case of Example 1, the existence of a dependence vector $v = (2)$ entails $\lambda_S = 0$, all operations are on the same processor, and there is no parallelism. After splitting, the only condition is that operations $\langle S, x\rangle$ and $\langle S, 2n - x\rangle$ are on the same processor, which can be arbitrary. The degree of parallelism is n.

6 A First, Naïve Splitting Algorithm

As we have seen in Section 5, suboptimal schedules result from the fact that some parts of a statement's index set are in the image of a dependence, and some are not. If the distinguished statement is the target of several dependences, we should apparently subdivide its iteration space, each part corresponding to a selection of the incoming dependences. Furthermore, if there is a dependence d_1 from statement R to statement S and a dependence d_2 from S to T, we should use the composite dependence $d_2 \circ d_1$ for splitting $I(T)$. Thus, a naïve approach for index set splitting tries to compute all possible paths in the dependence graph and split every statement according to all incoming paths.

Even if some drawbacks are obvious, let us analyze this method in more detail. It will illustrate the limits of what we can expect from the final algorithm to be presented in Section 7.

6.1 Merging multiple incoming paths

First, we realize that a single statement can be reached via several paths. Thus, independently from the chosen splitting algorithm, we will have to merge the splits of multiple incoming dependences. In principle, we split the index set according to the first incoming dependence, then split every part according to the next incoming dependence, and so on.

It is important to see that the order of treating the incoming dependences is irrelevant: to merge splits means to compute a conjunction of the image (or its complement which we call the non-image) of an incoming dependence and the parts already split, and conjunction is associative and commutative.

The complexity of merging the splits of k arbitrary incoming (composite) dependences is 2^k, i.e., in general, we must expect an algorithm with at least exponential time complex-

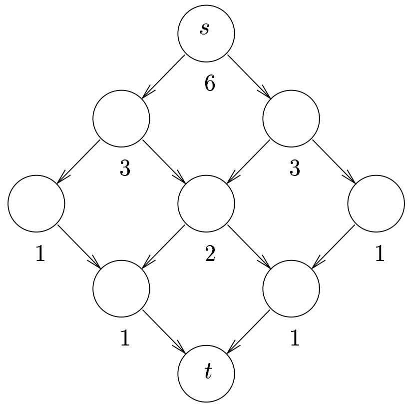

Figure 2: A dag with an exponential number of paths

ity. Therefore, our main concern must be to reduce the number k of different incoming (composite) dependences as much as possible, in order to get an efficient algorithm.

Note that, if the dependence graph is a tree, the complexity is only linear in the length of the paths, i.e., the depth of the tree: various incoming paths can only differ in length (for two different paths reaching a tree node, one must be a postfix of the other). Therefore, the image of the longer path is a subset of the image of the shorter path, i.e., we never need to split the non-images, which avoids the exponential growth.

6.2 Numbers of paths

We have just seen that in a tree the number of different paths to a given node is linear in the depth of the tree, i.e., logarithmic in the number of nodes. How many paths are there in a directed acyclic graph (DAG)?

There can be exponentially many paths between two statements in a DAG. For an illustration, consider Figure 2. For every node, we annotate the number of possible paths to node t. As we can see, the numbers correspond to Pascal's triangle, i.e., the numbers are the binomial coefficients $\binom{l}{k}$, where l is the length of the path from the considered node to t, and k is the horizontal position in the graph. In other words, the DAG in Figure 2 has $O(l^2)$ nodes but $O(2^l)$ paths.

Hence, if we consider all different paths from s to t, we end up with exponentially many incoming splits in t, which must then be merged with the exponential method as explained in Section 6.1. All in all, this results in a doubly exponential algorithm, which we consider impractical.

Obviously, the naïve method is not effective if there are loops in the statement dependence graph since, within strongly connected components, the number of different possible paths is unbounded. As simplest example, take a single statement with one self loop: every different number of loop traversals results in a different path. Note that, in this simple situation of only one self loop, the number of splits would increase linearly instead of exponentially (like in the case of the tree above), but it is still unbounded.

6.3 Preparing an effective algorithm

As we have just seen, directly using paths as descriptions of composite dependences results, in general, in a doubly exponential algorithm. Hence, our splitting algorithm must abstract from precise paths in order to be efficient.

For this purpose, we treat composite dependences systematically by associating a finite automaton with the dependence graph. The states of this automaton are the statements and the transitions are the dependences. There are well known algorithms (e.g., by Kleene [FB94]) for associating any two states S and T with a *regular expression* representing all paths from S to T. The letters in this regular expression are dependence *names*, and the operators are the dot (concatenation), the vertical bar (set union), and the Kleene star. The composite dependence from S to T is obtained by replacing in this expression each dependence name by the dependence itself, the dot by relation composition, the vertical bar by relation union, and the Kleene star by transitive closure. In suitable cases, one can compute the composite dependence in closed form by using, for instance, the Omega calculator [PW92]. However, since the transitive closure of an affine relation is not always affine, the above computation does not always succeed. Our proposal is to ignore a composite dependence when a closed form cannot be computed.

However, if we base our splitting algorithm on the path descriptions given by the Kleene algorithm, we find many practical examples which cannot but should be split. This is because the use of the operators $|$ and * entails a loss of precision. In fact, we lose all information on the *delay* associated with the composite dependence.

Example 2 Consider the following program:

```
do i = 0, m
   do j = 0, 2*n
      a(i,j) = a(i,2*n-j) + a(i-1,j+2)
   end do
end do
```

The iteration dependence graph is depicted in Figure 3. The range of the combined dependence $d = d_{\text{upwards}} \mid d_{\text{downwards}}$ is $[0,m] \times [0,2n] \setminus [0,0] \times [0,n]$, as is the range of d^+. The desired split into $[0,m] \times [0,n]$ and $[0,m] \times [n+1,2n]$, which leads to a linear execution time (see Figure 3), is not found if we base the algorithm on Kleene's path descriptions only.

The difficulty in this example is that the different dependences have individual properties, which are merged by the union operation, making the distinctions invisible. However, treating all dependences separately is too costly, in general, as shown in Section 6.2.

Our heuristic solution is to consider (in addition to all combined paths) every single dependence once, and split the index set of its target statement into its range and the rest. This guarantees that the properties of different dependences are considered at least once,

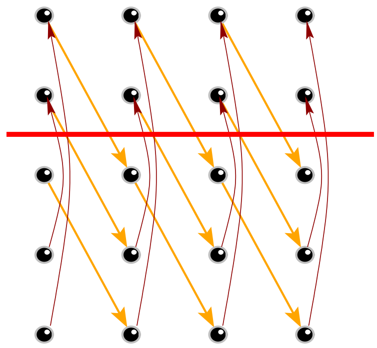

Figure 3: Splitting due to the initial phase

and that complexity is increased by only a linear factor (D dependences instead of 2^D paths). In other words, by splitting the target index set $I(S)$ of every single dependence d into the range of d and the rest, we already have a conservative approximation of the range of all *paths* to S whose last dependence is d.

Our idea is then to propagate these ranges of d along every path to all other statements, where we now accept the loss of information by using the path descriptions from Kleene's algorithm, in order to keep the computational effort reasonably small.

Practical experiments show that this mixture of working with separate dependences on the one hand and combined path descriptions on the other exhibits a good balance between power and execution complexity of our algorithm.

7 The Proposed Splitting Algorithm

With the ideas in the previous section, we can now formulate a first version of the effective splitting algorithm:

1. For all dependences d, compute a polytope R_d (as small as possible) containing the range of d, and split the index set $I(T)$ of the target statement of d into R_d and $I(T) \setminus R_d$.

2. Compute a description for the set of all paths in the statement dependence graph, using Kleene's algorithm.

3. For every pair of statements (T, S) and for every dependence d with target statement T and every path p from T to S do: interpret path description p as a composition of relations which maps points of the index space $I(T)$ to points of the index space $I(S)$, and compute the image $p(R_d)$ of this composed relation when applied to the polytope R_d computed in Step 1. This will divide the index set $I(S)$ into a part which is in the image of R_d under p, and the rest. Usually, this step can be computed

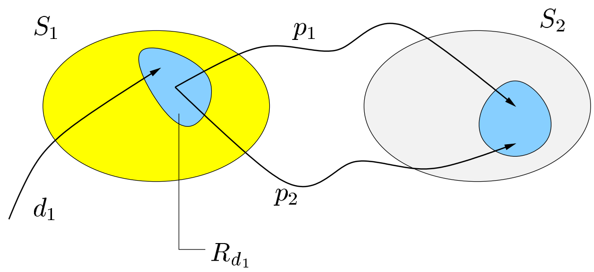

Figure 4: An illustration of the splitting algorithm

with the Omega calculator [PW92]. However, if p contains a cycle whose transitive closure cannot be computed precisely by Omega, then delete the cycle form p before the propagation.

4. For any statement S, combine all splits obtained in this way as in Section 6.1.

The algorithm is illustrated in Figure 4. In Step 1, the index set of S_1 is split into the image R_{d_1} of d_1 and the rest. Step 2 computes all paths between every pair of statements. Assume that there are two paths from S_1 to S_2, denoted with p_1 and p_2. Step 3 computes the image of R_{d_1} w.r.t. $p_1|p_2$ within the index space of S_2, where the possible images are all in the dark subset.

Basically, this algorithm computes, for every statement, the approximate ranges of all (transitively) incoming dependences. It is obvious that, for each such range, we might obtain a different possible schedule and, thus, want a separate template.

However, in practice, there are some additional considerations:

- It is easy to see that splitting an index set due to a uniform dependence is worthless, since the range of a uniform dependence is (approximately) the complete target index space. Therefore, we optimize the algorithm by applying Step 1 only to non-uniform dependences. Note that, in Step 3, we still need to consider uniform dependences, as we shall see in Example 4.

- Due to the condensed description of the set of all paths and the overestimation of the reflexive transitive closure by Omega, it often happens that we lose precision and, thus, do not find precise splits and sometimes even not all splits (if two approximations yield the same set).

  Note that these overestimates by Omega are due partly to the theoretical impossibility of computing the reflexive transitive closure as a finite union of polyhedra and partly to technical limitations inside Omega.

- Sometimes it is useful to unroll a cycle a fixed number of times, in order to achieve the optimal split (cf. Example 6). If, for a cycle c, we can find this fixed number k by the methods described in Appendix A, we extend the computation of $p(R_d)$ in

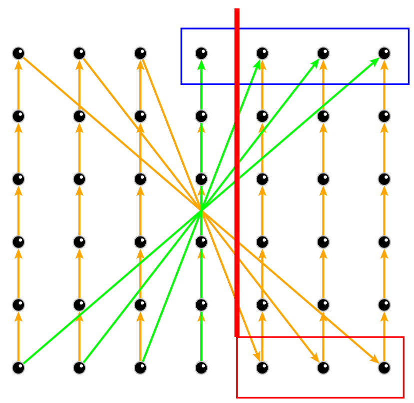

Figure 5: Splitting a two-dimensional index set by propagation

Step 3 of our splitting algorithm: if the propagation path p contains c, we rewrite p as $p_1.c^*.p_2$, and we compute $p_1.c^i.p_2(R_d)$ for $0 \le i \le k$ instead of the approximation $p_1.c^*.p_2(R_d)$.

8 Examples

Let us reconsider our initial example and modify it in several ways in order to get a feeling for the performance of our method.

Example 3 Let us apply our algorithm to Example 1. The iteration domain is the set $[0, 2n]$ and the range R of the two dependences is $[n+1, 2n]$. The set of all paths in the statement dependence graph is given by $(d_t \mid d_a)^*$. Propagating R along d^+ ($^+$ meaning at least once) is not possible since the domain of d^+ does not intersect R. Thus, the algorithm already terminates after the initial step and splits the index set into $[0, n]$ and $[n+1, 2n]$, as expected.

Example 4 Extending the program of Example 1 to a two-dimensional example, let us consider:

```
do i = 0, 2*n
   do j = 0, m
      a(i,j) = a(2*n-i,j+m) + a(i,j-1)
   end do
end do
```

For the uniform dependence, no split is derived. The non-uniform true dependence has range $[n+1, 2n] \times [0, 0]$. Propagating along the combined self-dependence leads to the desired split into $[0, n] \times [0, m]$ and $[n+1, 2n] \times [0, m]$ (see Figure 5). The non-uniform anti-dependence has the range $[n, 2n] \times [m, m]$, which is not increased by propagation. So, finally, we end up with three subdomains: $[0, n] \times [0, m] \setminus \{(n, m)\}$, the singleton $\{(n, m)\}$, and $[n+1, 2n] \times [0, m]$.

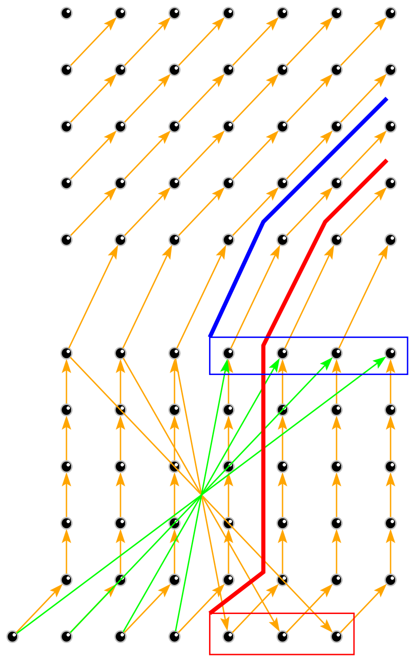

Figure 6: Splitting for imperfect loop nests

Note that, in the previous example, treating the singleton individually does not improve the schedule and, hence, should be avoided in practical implementations, if possible. However, in general, it is impossible to decide locally, i.e., at one given statement, whether a split is useful or not, since global information of the dependence graph is necessary. In other words, we would have to run the scheduler in order to detect whether we should split the input of the scheduler! Thus, in our current implementation, we accept possibly useless splits.

Example 5 In order to get a better feeling for the behaviour of our algorithm in the case of multiple statements and imperfect loop nests, let us consider:

```
      do i = 0, 2*n
S1:      a(i,0) = a(2*n-i,m)
         do j = 1, m
S2:         a(i,j) = a(i,j-1)
         end do
      end do
      do i = 0, 2*n
         do j = 1, k
S3:         a(i,j+m) = a(i-1,j+m-1)
         end do
      end do
```

In this example, we find two uniform true dependences and four non-uniform dependences (three true and one anti), i.e., we obtain four initial splits. After propagating each of them along the combined path to each of the three statements, we have to merge the four initial splits with the 12 propagated splits. After simplification, we get the desired splits as indicated in Figure 6.

Example 6 In the case of a self-dependence d, a constant number of propagations may give us an interesting schedule. Let I be the iteration space of the dependence. Successive propagation yields subsets $I \setminus d(I), d(I) \setminus d^2(I), \ldots, d^r(I) \setminus d^{r+1}(I)$ and, in the interesting case, there exists a constant k such that $d^k = \emptyset$. It can be proved that k is bounded by the cardinality of I, but this bound depends on the size parameters of the program and cannot be evaluated at compile time. The problem can be solved in the special case that d is defined by a square matrix A in one of the following senses:

$$\begin{array}{rl} & \langle x, y \rangle \in d \Rightarrow x = Ay \\ \text{or} & \langle x, y \rangle \in d \Rightarrow y = Ax \end{array}$$

One can prove (see the appendix) that, if $A^k = I$, then $d^k(I(S)) = \emptyset$. k can be determined by computing the characteristic polynomial of A and testing its divisibility by the cyclotomic polynomials of degree less than the dimension n of the iteration space of S, i.e., the number of loops surrounding S. For likely values of n, the number of tests is very small.

Consider the following program:

```
do i = 1,n
   do j = 1,n
      do k = 1,n
         a(i,j,k) = a(j,k,i)
      end do
   end do
end do
```

Classical scheduling yields $\theta(i, j, k) = \binom{i}{j}$, or $O(n)$ parallelism. There are two dependences:

$$\begin{array}{rl} & d_1 = \{\langle i, j, k \rangle \to \langle j, k, i \rangle \mid i < j\} \\ \text{and} & d_2 = \{\langle i, i, k \rangle \to \langle i, k, i \rangle \mid i < k\} \end{array}$$

For brevity, we have omitted the bound constraints $1 \leq i \leq n$ and the like, which stay invariant in the splitting process.

d_1 is generated by the matrix: $A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ whose characteristic polynomial is $P(x) = x^3 - 1 = (x - 1)(x^2 + x + 1)$. This is the product of the cyclotomic polynomial of order 1 and 3, and suggests the computation of A^3 which is found to be the unit matrix. The corresponding subsets are:

$$\begin{aligned} & d_1(I(S)) = \{i,j,k \mid k < i\} \\ \text{and} \quad & d_1^2(I(S)) = \{i,j,k \mid k < i, i < j\} \\ \text{and} \quad & d_1^3(I(S)) = \emptyset \end{aligned}$$

The domain of d_2 is disjoint from the domain of d_1, hence we are justified in handling both dependences independently. We leave it to the reader to check that $d_2^2(I(S))$ is empty. All in all, the latency is 2 and the degree of parallelism is $O(n^3)$.

Example 7 Consider now:

```
do i = 1,n
   do j = 1,n
      a(i,j) = a(2*i+j, i+j)
   end do
end do
```

The dependences are associated to the matrix $A = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 1 \end{smallmatrix}\right)$, whose characteristic polynomial is $P(x) = x^2 - 3x + 1$. P is not divisible by either $x-1$ or $x+1$ or x^2+x+1, and hence there is no k such that $A^k = I$. As a consequence, there is no hope of improving the schedule by repeated propagation.

9 Selective Splitting

The method described so far treats the program as a whole and searches for all potentially useful splits for improving the schedule. However, especially for large programs, it is desirable to apply index set splitting only when and where it is necessary. Furthermore, we are often only interested in increasing parallelism by orders of magnitude, but not by constant factors.

Our method can easily be adapted to these situations. The basic idea is that we first schedule the program without splitting and, if the result is not satisfactory, try to improve the solution by index set splitting:

1. Compute a schedule according to one of the usual methods [Fea92a, Fea92b, DV95, DV97] without any splitting.
2. Analyze the schedule, with a view of selecting an interesting candidate for splitting. For this purpose, take the statement S with the highest dimensionality of the schedule. If there are several candidates, choose one which is minimal w.r.t. the order imposed by the acyclic condensation of the statement dependence graph.
3. As we have seen in Section 5, significantly improving a schedule for S means opening a cycle going through S. Edges of such cycles belong to the strongly connected component (SCC) of S in the dependence graph. Thus, we must split the statements in the SCC via the original algorithm in Section 7:
   (a) For all non-uniform dependences d of the SCC, compute the initial split as in Step 1 of the original algorithm.

(b) For every pair of statements (T, R) of the SCC, propagate the initial splits of T to R as in Step 3 of the original algorithm. If the composite dependence d from R to R is non-uniform and generated by a matrix A of order k (see Example 6), continue propagating the split $k-1$ times via d.

4. Schedule the new dependence graph.

5. When the dimensionality of the schedule is satisfactory or when all SCC's of the dependence graph have been considered, then stop; else start again with Step 2.

Note that this algorithm – like all approaches to index set splitting presented so far – attempts to increase parallelism. This is not useful for statements which have more parallel instances than there are real processors available. Taking resource constraints into account, we have an additional selection criterion for Step 2:

(a) The basic idea is to find the statement which contributes the most to the parallel execution time of the program, and which holds some hope of improvement. To this end, we need to know the "characteristic size" N of the problem and the number P of processors. The important factor is, in fact, the least exponent α such that $N^\alpha \gg P$.

(b) Let S be a statement with an n_S-dimensional iteration space and a p_S-dimensional schedule. If $N^{n_S - p_S} \gg P$, or $n_S - p_S \geq \alpha$, all processors are occupied, the running time is N^{n_S}/P, and it cannot be improved by modifying the schedule.

(c) On the other hand, if $n_S - p_S < \alpha$, then the running time is N^{p_S} and can be improved by reducing p_S.

(d) Hence the rule is to select the statement S with the largest p_S among those satisfying $n_S - p_S < \alpha$.

Remark: Note that our iterative splitting approach yields fewer splits than the original algorithm: dependences which are not part of a strongly connected component are never considered. This limits the complexity of the target code, but the detected parallelism is still of the same order of magnitude as if we used the original algorithm.

10 Conclusions

We have proposed a method to split the iteration domain of loop nests into parts in order to obtain better space-time mappings. It can be used to improve any space-time mapping algorithm which is based on templates (templates are typically used, e.g., for dealing with different statements in the loop body).

10.1 Trading off quality for time

Our method can be adapted simply for trading off quality of parallelism for analysis time:

- One can search for more splits and, thus, even satisfy a variety of optimality criteria, but at the cost of a doubly exponential search.
- One can turn off the propagation phase in order to save compilation time at the price of a loss of some useful splits (e.g., in Example 4).
- On the other hand, one can try to improve the solution by propagating a split through cycles more than once, as indicated in Example 6.
- Alternatively, one can also first schedule a program without splitting and afterwards try to split only those statements which are responsible for a possibly bad quality of the schedule, as shown in Section 9.

10.2 Experiments

In our experiments, we decided to use the presented algorithm, which seems to be a good compromise between analysis time and quality of the resulting splits. More experiments on practical examples with our prototype will have to confirm this observation.

For this purpose, we are currently implementing the described methods in the prototype parallelizer LooPo [GL97]. This implementation takes as input the results of the existing dependence analysis modules and is itself very close to the algorithm in Section 7, with the following technical modifications:

- In Step 1, we do not actually execute the splits but just store them per statement in a list, in order not to increase the number of statements in this phase.
- In Step 3, we only compute the images; the computation of the rest (i.e., the non-images) does not commence before the end of Step 4 and, again, the derived splits are just stored in lists in order to limit complexity.
- In Step 4, we first merge all stored splits and then compute the rest, i.e., the non-images – which may lead to further splitting, due to technical limitations, if the non-images cannot be described by a single polytope but only by a union of polytopes. Finally, every split, i.e., description of a subset of the original index space gets a copy of the body statements and, thus, can be viewed by all subsequent parallelization phases as if it were a separate statement with surrounding loops in the input program. Note that our model-based approach saves us from computing a loop nest which really enumerates all these split subsets; this is advantageous because the construction of such a loop nest would be very costly.

10.3 Putting the method into context

Finally, let us recapitulate what we set out to do and what we have achieved.

The principal idea of index set splitting is that every dependence d causes irregularities in the target statement T of d: some instances of T are in the image of d – and, thus, have to satisfy some constraints, e.g., for scheduling – and some are not in the image of d.

The same is true for paths of dependences instead of single dependences. Hence, the basic task is to compute every dependence path p and split the target statement of p into the points in the image of p and the rest.

Since the number of paths is unbounded (in the absence of cycles bounded but still exponential) or, to be more precise, dependent on the problem size, we must find a different representation of the set of all paths which is problem-size independent.

Kleene's path descriptions satisfy this condition and can be computed in cubic time. However, with the use of these path descriptions we leave the classical polytope model: the price for the cubic description of an unbounded number of paths is a loss of precision, as illustrated, e.g., by Example 2.

Our approach deviates in one point notably from typical methods based on the polytope model: we make extensive and central use of heuristics.

Our heuristic solution for the trade-off between precision and efficient computation is to use dependences for two separate tasks:

- For every single dependence, we compute the split, i.e., a partitioning of the iteration dependence graph.
- We propagate every initial split along the path descriptions of the statement dependence graph.

The first task requires no heuristics – it can be computed precisely with standard methods of the polytope model [Fea91]. The second task is solved with methods outside the classical polytope model, and incurs a possible loss in precision, for the following reason: in effect, we compute precisely the images of all paths p_1 of length 1 and, since the image of some path $p_2.p_1$ is a subset of the image of p_1, we accept a possible approximation of its image, because we already have an upper estimate: the image of p_1.

Another case in which we apply heuristics is selective splitting (Section 9). In this scenario, we are not interested in optimal solutions (within the model). By concentrating on the order of magnitude by which the running time is reduced, we achieve a significant reduction in the cost of the compile-time analysis.

Our trick is to apply the powerful but costly polytope model only to subproblems for which it is cost-effective. Following the same idea, one can first apply simpler but fast scheduling techniques and re-schedule with more elaborate methods only those strongly connected components of the statement dependence graph which have the highest latency. Other approaches like iterative array data flow analysis [CG99] have a similar potential.

In this sense, the polytope model can be viewed as a powerful alternative to be applied to subproblems for which simple heuristics do not yield satisfactory solutions. We believe that this is a rôle which it can play effectively in practical parallelizing compilers.

Acknowledgements

Financial support was gratefully received from the German Research Foundation (DFG) under project *RecuR* and from the French-German exchange programme *PROCOPE* through DAAD and APAPE.

Thanks to Mohamed Jemni for discussing his method with us in Passau, which started off this work. We are grateful to Jean-François Collard and Bernhard Lehner and to the participants of Dagstuhl Seminar 99161 [AELS99] for fruitful discussions. Thanks to Gottlieb Leha and Niels Schwartz for mathematical advice; the second author acknowledges the help of Vital Chauve with the material in Example 6 and the appendix.

A Appendix

Let us consider the case of a dependence d from a statement S to itself. We will pretend that d is the sole dependence with target S: in this way, we will obtain a lower bound on the latency of S.

If we split iteratively according to d, we generate a succession of sets $I(S) \setminus d(I(S))$, $d(S) \setminus d^2(I(S))$, ... , $d^r(I(S)) \setminus d^{r+1}(I(S))$. The question is: is there a k for which this process terminates because $d^k(I(S)) = \emptyset$?

The answer is yes, and k is bounded by the cardinality of $I(S)$. The proof is left to the reader. (Hint: remember that if y depends on x then $x \ll y$, where $\ll$ is lexicographic order.)

This result is not useful for a compile time analysis, since the size of $I(S)$ is likely to be unknown at compile time. Are there cases where k can be computed a priori?

We have been able to answer the question only in the case where d is a function or the inverse image of a function. In this case, there exists a matrix A such that either $\langle x, y\rangle \in d \Rightarrow x = Ay$ or $\langle x, y\rangle \in d \Rightarrow y = Ax$. A is a square matrix of size $n \times n$, where n is the dimension of $I(S)$, i.e., the number of loops surrounding S. This constellation is quite frequent. For instance, if we have decided to consider only dataflow (or value-based) dependences, then the dependence relation will always be the inverse of a function [Fea91]. We are ignoring the possibility that there is a constant term in the expression of the dependence, since symbolic and numeric constants can be subsumed by the use of homogeneous coordinates.

One can prove that, if there exists an integer k such that $A^k = I$, then $d^k(I(S)) = \emptyset$. The reason is that, if $d^k(I(S))$ is not empty, then there exists an operation x such that $x \ll A^k x = x$, a contradiction.

Every square matrix A can be put into Jordan normal form:

$$A = U^{-1}DU$$

where D either is diagonal or has some non-zero elements in the first subdiagonal. The diagonal elements of D are the eigenvalues of A, i.e., the roots of the characteristic polynomial $P_A(x) \equiv \det(A - xI)$.

If $A^k = I$, then $D^k = I$. One can prove that this implies that D is diagonal and that the eigenvalues of A are mth roots of unity, where $m \mid k$ (m divides k). The primitive roots of unity are roots of the cyclotomic polynomial:

$$C_m(x) \equiv \prod_{\gcd(j,m)=1} (x - e^{2\pi ij/m}).$$

The polynomial $C_m(x)$ has integral coefficients, is irreducible over $Z[x]$, and has degree $\phi(x)$ where ϕ is Euler's function:

$$\phi(m) = m \prod_{p|m} (1 - 1/p), \ p \text{ is a prime number.}$$

Since C_m is irreducible, either the gcd of P_A and C_m is 1, and the mth roots of unity are not eigenvalues of A, or C_m divides P_A. In the latter case, the degree $\phi(m)$ of C_m is at most n. Hence, a first version of an algorithm for finding k, if it exists, is:

- Compute P_A and set $P = P_A$.
- For $m = 1$ to L do:
  - While $C_m \mid P$ do: set $P = P/C_m$.
  - If $P = 1$, stop; k is the least common multiple of all m for which the division has been successful.
- Stop; there is no k such that $A^k = I$.

If this algorithm terminates successfully, one must test whether the Jordan normal form of A is diagonal, which can be done most simply by direct computation of A^k in time $O(n^3 \log k)$. This test can be avoided if all eigenvalues of A are simple, i.e., if the While loop in the algorithm above iterates at most once.

It remains to state how to select L. The correct value is:

$$L = \max\{m \mid \phi(m) \leq n\}.$$

However, we have to prove that this L exists, and this is difficult since the behaviour of ϕ is chaotic. Let us consider instead the obvious minorant:

$$\phi'(m) = m \prod_{p \leq m} (1 - 1/p)$$

One can prove that ϕ' is non-decreasing and that:

$$\phi'(x) \approx e^{-\gamma} \frac{x}{\log x},$$

where γ is Euler's constant (see Theorem 428 of Hardy/Wright [HW90]). Thus, we can define L' by:

$$L' = \max\{m \mid \phi'(m) \leq n\},$$

and either use this value in the above algorithm or compute a tighter value of L:

$$L = \max\{m \mid m \leq L', \phi(m) \leq n\}.$$

Recall that n is the number of loops surrounding S and is never very large. For instance, for $n = 3$, there are exactly five candidate divisors. This number goes up to 20 for the unlikely value $n = 10$. Hence, the cost of this test is likely to be very small.

References

[AELS99] Damal K. Arvind, Kemal Ebcioglu, Christian Lengauer, and Robert S. Schreiber, editors. Instruction-level parallelism and parallelizing compilation. Technical Report 237, Schloß Dagstuhl, 1999.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP'91)*, pages 39–50. ACM Press, 1991.

[AK87] John R. Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[ARY98] Ruman Andonov, Sanjay Rajopadhye, and Nicola Yanev. Optimal Orthogonal Tiling. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98: Parallel Processing*, LNCS 1470, pages 480–490. Springer-Verlag, 1998.

[Ban79] Utpal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report 79-989.

[BKA97] Rajeev Barua, David Kranz, and Anant Agarwal. Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors. In David Sehr, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239, pages 350–368. Springer-Verlag, 1997.

[CG99] Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Jean Ferrante, editor, *Languages and Compilers for Parallel Computing (LCPC'99)*, LNCS. Springer-Verlag, 1999. To appear.

[DKR91] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, December 1991.

[DV95] Alain Darte and Frédéric Vivien. Revisiting the decomposition of Karp, Miller, and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.

[DV96] Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96: Parallel Processing, Vol. I*, LNCS 1123, pages 379–388. Springer-Verlag, 1996.

[DV97] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. J. Parallel Programming*, 25(6):447–496, December 1997.

[FB94] Robert W. Floyd and Richard Beigel. *The Language of Machines – An Introduction to Computability and Formal Languages*, chapter 4.4. Computer Science Press, 1994.

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

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, 1992.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

[FGRT98] Jeanne Ferrante, Wolfgang Giloi, Sanjay Rajopadhye, and Lothar Thiele, editors. Tiling for optimal resource utilization. Technical Report 221, Schloß Dagstuhl, August 1998.

[GFL99] Martin Griebl, Paul A. Feautrier, and Christian Lengauer. On index set splitting. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 274–282. IEEE Computer Society Press, October 1999.

[GL97] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo—Announcement. In David Sehr, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239, pages 603–604. Springer-Verlag, 1997.

[HW90] Godfrey H. Hardy and Edward M. Wright. *An Introduction to the Theory of Numbers*. Oxford Science Publications. Oxford University Press, 5th edition, 1990.

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

[MJ95] Zaher Mahjoub and Mohamed Jemni. Restructuring and parallelizing a static conditional loop. *Parallel Computing*, 21(2):339–347, February 1995.

[MJ96] Zaher Mahjoub and Mohamed Jemni. On the parallelization of single dynamic conditional loops. *Simulation Practice and Theory*, 4:141–154, 1996.

[PW92] William Pugh and Dave Wonnacott. Eliminating false data dependences using the Omega test. *ACM SIGPLAN Notices*, 27(7):140–151, July 1992. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'92)*.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In Françoise F. Soulié, Yves Robert, and Maurice Tchuenté, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.