Uniform Techniques for Loop Optimization†

William Pugh
Dept. of Computer Science and Institute for Advanced Computer Science
Univ. of Maryland, College Park
pugh@cs.umd.edu

Abstract

Many different kinds of loop transformations have been described, such as loop interchange, loop skewing and loop fusion. Each transformation requires its own particular set of dependence analysis tests and parallelizing a section of code may require performing a series of transformations. The only way to decide if there is a way of parallelizing a section of code is to try all possible sequences of transformations, which presents a difficult search problem.

We present a uniform method of performing loop optimizations. Rather than optimizing a program by performing a murky search through a series of transformations, our method considers a very powerful class of program transformations that includes any transformation that also can be obtained by any sequence of standard loop transformations. This optimization technique uniformly encompasses the effects of parallelization, loop fusion, loop splitting, loop interchange, loop skewing and statement reordering, as well as transformations not previously described. Thus, we only need to perform one program transformation.

We show that standard techniques for representing dependencies (dependence directions and dependence distances) are insufficient for this type of optimization and describe a more powerful technique, dependence relations, that have sufficient descriptive power.

1. Introduction

Many different kinds of loop transformations have been described [AK87, Poly88, Wol89, Wol90], such as loop interchange, loop skewing and loop fusion. However, there is no automatic way to decide which sequence of transformations need to be applied to parallelize a section of code. We describe a new automatic code transformation technique for parallelizing code. If any combination of standard loop transformation can parallelize the code, our techniques will parallelize the code.

For example, consider the following code fragment:

```
for i := k+1 to n do
   B[i, i] := B[i-1, k+1]*B[k, k]
   for j := k+1 to n do
      B[i, j] := B[i, j]+B[i-1, j]+B[i, j-1]+B[i, k]*B[k, j]
```

Our methods can automatically transform this code fragment into the following parallel code fragment, a task that is difficult or impossible for other automatic methods.

```
for t := 2*k+2 to n+k+1 do
   B[t-i-k, t-i-k] := B[t-2-k, k+1]*B[k, k]
   forall j := k+1 to t-k-1 do
      B[t-j, j] := B[t-j, j]+B[t-j-1, j]+B[t-j, j-1]+B[t-j, k]*B[k, j]
for t := n+k+2 to 2*n do
   forall j := t-n to n do
      B[t-j, j] := B[t-j, j]+B[t-j-1, j]+B[t-j, j-1]+B[t-j, k]*B[k, j]
```

Our approach consists of finding a schedule $T_p[x]$ for each statement s_p describing the moment at which each iteration x of s_p will be executed. For example, the code above is produced using the schedules $T_1[i] = i+k+1/2$ and $T_2[i, j] = i+j$. A schedule is feasible if it respects the orderings imposed by the dependencies in the program. Different schedules are isomorphic to the effects of different combinations of loop and code transformations. We optimize a program by evaluating the code produced by different feasible schedules. Finding a feasible schedule requires a more sophisticated method of analyzing dependencies than is normally used. The techniques described here are not as fast as simple optimization techniques and are intended to be used in practice only when simpler techniques are insufficient. The techniques described here, when specialized for "typical cases," may prove to be as fast as simpler techniques.

The idea of equating code-rearrangement with finding schedules for an iteration space has been examined by several authors [Lam74, Rib90, Wol90, Ban90]. Previous approaches have been severely limited in their ability to deal with imperfectly nested loops and complicated or transitive dependencies. In this paper, we propose methods for handling dependencies and imperfectly nested loops that move this approach substantially forward to reaching its full potential as an optimization method that can subsume all other loop optimizations.

Most previous work on program transformations use data dependence directions and data dependence distances to summary the dependencies between two array references. For our purposes, these techniques are too crude. We use integer programming techniques to represent and evaluate dependencies exactly. This approach provides more information about the types of transformations allowable and also allows us to properly handle transitive dependencies.

Integer programming is a NP-Complete problem and solving such problems may require exponential time. However, we have found that problems arising in practice can be efficiently manipulated, simplified and solved. We make use of the Omega test [Pug91], an integer programming simplification and decision algorithm designed for use in dependence analysis. Most of the problems encountered can be solved in 1-3 milliseconds (on a Decstation 3100, a 12 MIPS workstation).

† This research is supported by NSF grant CCR-8908900.

operation	Description	Definition
$F \circ G$	The composition of F with G	$\langle x,z \rangle \in F \circ G \Leftrightarrow \exists y$ s.t. $\langle x,y \rangle \in G \wedge \langle y,z \rangle \in F$
F^{-1}	The inverse of F	$\langle x,y \rangle \in F^{-1} \Leftrightarrow \langle y,x \rangle \in F$
$F \cup G$	The union of F and G	$\langle x,y \rangle \in F \cup G \Leftrightarrow \langle x,y \rangle \in F \vee \langle x,y \rangle \in G$
$F \cap G$	The intersection of F and G	$\langle x,y \rangle \in F \cap G \Leftrightarrow \langle x,y \rangle \in F \wedge \langle x,y \rangle \in G$
I	Identity function	$\langle x,x \rangle \in I$
F^*	The transitive closure of F	least fixed point of $F^* = I \cup F \circ F^*$
$F \setminus S$	The relation F restricted to domain S	$\langle x,y \rangle \in F \setminus S \Leftrightarrow \langle x,y \rangle \in F \wedge x \in S$
F / S	The relation F restricted to range S	$\langle x,y \rangle \in F / S \Leftrightarrow \langle x,y \rangle \in F \wedge y \in S$
$F \dot{+} G$	The pointwise sum of F and G	$\langle x,y+z \rangle \in F \dot{+} G \Leftrightarrow \langle x,y \rangle \in F \wedge \langle x,z \rangle \in G$
$F \dot{-} G$	The pointwise difference of F and G	$\langle x,y-z \rangle \in F \dot{-} G \Leftrightarrow \langle x,y \rangle \in F \wedge \langle x,z \rangle \in G$
$S \cup T$	The union of S and T	$x \in S \cup T \Leftrightarrow x \in S \vee x \in T$
$S \cap T$	The intersection of S and T	$x \in S \cap T \Leftrightarrow x \in S \wedge x \in T$
domain(F)	The domain of F	$x \in$ *domain*(F) $\Leftrightarrow \exists y$ s.t. $\langle x,y \rangle \in F$
range(F)	The range of F	$y \in$ *range*(F) $\Leftrightarrow \exists x$ s.t. $\langle x,y \rangle \in F$
$\exists x \in S$	Is S empty?	Returns "False" if $S = \varnothing$ otherwise some tuple x in S

Table 1 – Operations on tuple sets and relations, where F and G are tuple relations and S and T are sets of tuples:

To achieve the best results from the methods described in this paper, we must apply techniques such as constant propagation, forward substitution, induction variable substitution, scalar expansion and variable copying [ZC91] to programs before we attempt to optimize them.

This paper is broken into the following sections. Section 2 describes a class of operations on sets and relations of integers tuples (the implementations of these operations are described in Appendix 1). Section 3 describes how to obtain a set of scheduling dependencies for a code fragment. Section 4 discusses how to find feasible schedules for a set of statements. Section 5 show how a feasible schedule is used to rearrange code. Section 6 discusses the relationship between different schedules and optimizations. Section 7 shows that our techniques can find an optimization equivalent to any combination of a series of standard loop optimizations. Section 8 discusses related work and Section 9 summarizes the paper. Sections 4, 5 and 6 show the parallelization of two examples that could not be automatically parallelized by standard techniques.

Appendix 1 describes the implementation of the operations described in Section 3. Appendix 2 gives an overview of the Omega test, which is used for manipulating the integer programming problems that arise in the methods described here. The methods presented in the main body of the paper only discuss highly parallel schedules (schedules that take at most $O(mn)$ time, where m is the number of statements and n is the maximum number of iterations of any loop). Appendix 4 briefly discusses nested time, a method that allows our techniques to be used even in circumstances where no highly parallel schedule exists. Appendix 4 shows the parallelization of an iterative smoothing program, and Appendix 5 shows the parallelization of a program that does QR decomposition via Givens Rotations.

2. Integer Tuple Relations and Sets

In this section, we talk about techniques for the uniform manipulation of integer tuple relations. An integer k–tuple is simply a point in Z^k. For example, [3, 4, –3] is a point in Z^3. In manipulating tuple relations, we also will discuss of integer tuples sets.

A tuple relation is a mapping from tuples to tuples. A single tuple may be mapped to zero, one or more tuples. A relation can be thought of as a set of pairs, each pair describing an input tuple and the output tuple it is mapped to. All the relations we consider map from k–tuples to k'–tuples for some fixed k and k' (e.g., we do not consider relations take as input both 1-tuples and 2-tuples).

We use integer programming techniques to represent and manipulate relations and sets. A relation is defined by a collection of linear equalities and inequalities involving the elements of the input and output tuples, symbolic constants and wildcards. Such a relation contains exact those mappings of integer tuples to integer tuples that satisfy the constraints. A set is defined by a collection of linear equalities and inequalities involving the elements of the input and output tuples, symbolic constants and wildcards, and such a set contains exact those integer tuples that satisfy the constraints. An example of a relation from 2-tuples $[s_1, s_2]$ to 2-tuples $[t_1, t_2]$ is $\{ t_1 = s_1+1; t_2 = s_1+2s_2 \}$ and an example of a set of 2-tuples $[s_1, s_2]$ is $\{ 1 \le s_1 \le s_2 \le 10 \}$.

In the representation of a set, a system of linear equalities and inequalities can refer to the components of tuples in the set, (denoted as $s_1, s_2, ..., s_k$) symbolic constants (denoted by the use of typewriter font) and wildcards (any other variables). For example, the set of 2-tuples represented by $\{ s_1 + s_2 = 2\alpha \}$ is the set of all tuples $[x, y]$ such that $x+y$ is even, and the set of 2-tuples represented by $\{ s_1 + s_2 \le n \}$ is the set of all tuples $[x, y]$ such that $x+y$ is at most n.

The representation of a relation is similar, except that the linear equalities and inequalities for a relation can refer to the components of the input (i.e., argument) tuple (denoted as s_1, s_2, ...,s_k) and also the components of the output (i.e., result) tuple (denoted as $t_1, t_2, ..., t_{k'}$), as well as referring to wildcards and symbolic constants.

Note that we can accommodate integer division and remainder operations. For example, if m is a known constant, $x = y$ div m can be represented by $\{ 0 \le y - m\,x < m \}$ and $x = y$ mod m can be represented by $\{ x = y - m\,\alpha; 0 \le x < m \}$

If sets and relations are represented this way, we can efficiently compute the results of all of the operations shown in Table 1, save transitive closure. We can only find an exact, closed form for the transitive closure of a relation in certain (commonly occurring) special cases. In other situations, we can use an approximation such as $F^* = I \cup F \cup F \circ F \cup F \circ F \circ F \cup F \circ F \circ F \circ F$. Note that this is not a conservative approximation of F^*, but for the situations in which we need to compute F^*, we do not need a conservative approximation (also, we don't often need to compute transitive dependencies).

To allow us to discuss sets and relations more concisely, we denote relations using the form

$$F\,[e_1, e_2, ..., e_k] = [f_1, f_2, ..., f_{k'}]\ \{P\}$$

to denote the relation that would be more fully represented by

$\{ s_1 = e_1; s_2 = e_2; ...; s_k = e_k; t_1 = f_1; t_2 = f_2; ...; t_k = f_k; P \}$.

For example, the relation G denoted by

$\{ s_1+1 = s_2; t_1 = s_1; 1 \le t_1 < t_2 \le n \}$

can be more concisely described as

$G\,[i, i+1] = [i, j]\ \{ 1 \le i < j \le n \}$.

In Appendix 1, we describe how to combine integer programming problems to obtain the functions in Table 1. In Appendix 2, we describe how integer programming methods are used to manipulate and simplify the resulting relations.

3. Data dependence relations

We now wish to consider how to analyze the data dependencies within a program. For our analysis, we assume that programs consist solely of looping constructs and conditional assignments[1]. We also assume all loops are normalized so as to have a unary step value. Let $s_1, s_2, \ldots, s_k$ be the assignment statements within the code being analyzed. Statement s_p is contained within d_p nested loops and we use $s_p[i_1, i_2, \ldots, i_{d_p}]$ to refer to the iteration of s_p when the outermost loop index is equal to i_1, the next outermost is equal to i_2, ... and the innermost loop index is equal to i_{d_p}.

We now wish to compute the set of dependence relations in the program. For example, we might determine that $s_2[i]$ must occur before $s_3[i+1, j]$ for all i and j. We denote this as: $s_2[i-1]\ \delta\ s_3[i, j]$. This dependence could arise because:

- iteration $[i]$ of s_2 updates a value that should be read by iteration $[i+1, j]$ of s_3,
- because iteration $[i]$ of s_2 must read a value before it is overwritten by iteration $[i+1, j]$ of s_3 or
- because iteration $[i]$ of s_2 writes a value that must be overwritten by iteration $[i+1, j]$ of s_3.

Range constraints are associated with a dependence relation, as in $s_2[i]\ \delta\ s_3[i+1, j]$ $(1 \le i < j \le n)$. We implement dependence relations using tuple relations. Thus, the previous dependence relation is described by $F[i] = [i+1, j]$ $(1 \le i < j \le n)$.

We differ from the standard notation for dependencies in two respects. First, we annotate the dependencies with more information about the exact instances that are constrained. The dependence we denote $s_3[i, j]\ \delta\ s_3[i, j+1]$ would be described by others [AK87, Ban88, Wol89, Wol90] as $s_3\ \delta_{(0,1)}\ s_3$ (giving the dependence distance) or as $s_3\ \delta_{(=,<)}\ s_3$ (giving the dependence direction). For our purposes, dependence distances and directions are inadequate for dependencies such as $s_2[i]\ \delta\ s_3[j, i]$ $(i \le j)$ and $s_1[i, i]\ \delta\ s_4[i, j]$ $(1 \le i \le j \le n)$ (which conventional notation would denote $s_2\ \delta_{(\le)}\ s_3$ and $s_1\ \delta_{(=,\le)}\ s_4$).

Second, we do not annotate our dependencies with as much information about the kind of dependence. Standard notation uses either δ^f or just δ is used for a dependence from a write to a read (a flow dependence), δ^a or $\bar{\delta}$ for a dependence from a read to a write (an anti-dependence) and δ^o for a dependence from a write to a write (an output dependence) and δ^* for an arbitrary dependence. Since the work described here does not distinguish between flow dependencies, anti-dependences and output dependencies, we simply use δ for any dependence.

Reduction dependencies

When compiling for parallel execution, we often allow the compiler to make the assumption that addition and multiplication in machine arithmetic is associative. Although this is a false assumption, it often does not produce significant differences in the results of the program and can produce significant speed-ups. Assume $s_3[i, j]$ adds b[j] to a[i] and the j loop counts upward. Convention dependence analysis would determine that there is a data dependence from $s_3[i, j]$ to $s_3[i, j+1]$ and require that $s_3[i, j]$ happen before $s_3[i, j+1]$. If we assume machine addition is associative, we can perform these operations in either order (assuming this is consistent with all other dependencies), but not simultaneously. Operations that can be treated as reductions include addition, multiplication, and, or, min, and max. When handle such dependencies by treating them as reductiondependencies [Wol90].

In summary, if $s_p[i_1, i_2, \ldots, i_n]$ and $s_q[j_1, j_2, \ldots, j_m]$ perform compatible updates (i.e., order-independent updates) of the same location, we denote this as $s_p[i_1, i_2, \ldots, i_n]\ \delta^R\ s_q[j_1, j_2, \ldots, j_m]$.

Broadcast "dependencies"

When executing a program on a SIMD machine, we may wish to avoid having all processors simultaneously requiring read access to the same value. On a shared-memory SIMD machine, this could result in memory-conflicts. In a distributed memory architecture[2], this probably would require a broadcast of the required information. This information may be represented by a broadcast "dependence" [Wol90]. If $s_3[i, j]$ and $s_3[i, j']$ read the same information, we denote this as $s_3[i, j]\ \delta^B\ s_3[i, j']$.

Computing dependence relations

We create dependencies by noting when two statements read and/or write the same locations in memory. We do this by building dependencies that tell which locations of which arrays are read or written by the execution of a statement.

Let a_1^R be a relation that gives the locations of an array a read by statement s_1 and a_2^W be a relation that gives the locations of a written by statement s_2. These relations should include restrictions on the bounds of the loop variables and the portion of the guard (if any) expressible as linear equalities and inequalities on the loop variables and symbolic constants. The relation $(a_1^R)^{-1}[x]$ gives the iterations of s_1 that read $a[x]$ and $((a_1^R)^{-1} \circ a_2^W)[i_1, i_2, \ldots, i_{d_2}]$ gives the iteration(s) of s_1 in which the same location is read as is written in iteration $[i_1, i_2, \ldots, i_{d_2}]$ of s_2. To produce the dependence relation from s_1 to s_2, the relation is then restricted such that $\langle x, y \rangle$ is in the relation only if iteration x of s_1 would occur before iteration y of s_1 under normal semantics (this is ignored for broadcast and reduction dependencies).

As an example, we consider the first dependence of Example 1 below:

$s_2[i, j]$ updates B[i, j]	$B_2^W[i, j] = [i, j]$ $(k < i, j \le n)$
$s_1[i]$ reads B[i–1, k+1]	$B_1^R[i] = [i-1, k+1]$ $(k < i \le n)$ and $(B_1^R)^{-1}[i, k+1] = [i+1]$ $(k \le i < n)$
$\therefore$	$((B_1^R)^{-1} \circ B_2^W)[i, k+1] = [i+1]$ $(k < i < n)$ and $s_2[i, k+1]\ \delta\ s_1[i+1]$ $(k < i < n)$

Example 1, part 1

Consider the following code fragment (this example is based on one given on page 115 of [Wol89], but additional dependencies have been introduced to make it more difficult by requiring loop skewing to parallelize this loop):

```
    for i := k+1 to n do
s1    B[i, i] := B[i–1, k+1] * B[k, k]
      for j := k+1 to n do
s2      B[i, j] := B[i, j] + B[i–1, j] + B[i, j–1] + B[i, k]*B[k, j]
```

Using the techniques describe in this section, we can easily determine the dependence relations shown in Table 2, which are shown graphically in Figure 1. The from and to columns

[1] Conditional assignment is an assignment which is executed only if a boolean guard is true. Standard assignment statements are just conditional assignments with a boolean guard of *true*.

[2] There any many other factors concerning properties an algorithm must have in order to be able to be executed efficiently on a SIMD distributed-memory machine, some of which are discussed in [Wolfe90].

#	*Direct dependencies:* *dependence relation*	*range*	from	to	*dependence direction*
1	$s_2[i,k+1]\ \delta\ s_1[i+1]$	$k<i<n$	s_2 U B[i, j]	s_1 R B[i−1, k+1]	$s_2\ \delta_{(<)}\ s_1$
2	$s_2[i,j]\ \delta\ s_2[i+1,j]$	$k<i<n\ \&\ k<j\le n$	s_2 U B[i, j]	s_2 R B[i−1, j]	$s_2\ \delta_{(<,=)}\ s_2$
3	$s_2[i,j]\ \delta\ s_2[i,j+1]$	$k<i\le n\ \&\ k<j<n$	s_2 U B[i, j]	s_2 R B[i, j−1]	$s_2\ \delta_{(=,<)}\ s_2$
4	$s_1[i]\ \delta\ s_2[i,i]$	$k<i\le n$	s_1 U B[i, i]	s_2 U B[i, j]	$s_1\ \delta_{(=)}\ s_2$
5	$s_1[i]\ \delta\ s_2[i+1,i]$	$k<i<n$	s_1 U B[i, i]	s_2 R B[i−1, j]	$s_1\ \delta_{(<)}\ s_2$
6	$s_1[i]\ \delta\ s_2[i,i+1]$	$k<i<n$	s_1 U B[i, i]	s_2 R B[i, j−1]	$s_1\ \delta_{(=)}\ s_2$

Table 2 – Dependencies for Example 1

#	*Direct dependencies*	*range*	from	to
1	$s_1[i,j]\ \delta\ s_2[i+2,j+2]$	$3\le i\le n-2\ \&\ 3\le j\le m-3$	s_1 U c[i, j]	s_2 R c[i−2, j−2]
2	$s_2[i,j]\ \delta\ s_1[i,j+1]$	$3\le i\le n\ \&\ 3\le j\le m-2$	s_2 U b[i, j]	s_1 R b[i, j−1]
3	$s_2[i,j]\ \delta\ s_1[i+1,j-1]$	$3\le i\le n-1\ \&\ 4\le j\le m-1$	s_2 U b[i, j]	s_1 R b[i−1, j+1]

Table 3 – Dependencies for Example 2

show the array accesses that produced the dependence. These columns show the statement involved in the dependence, whether the access performed by the statement is an update (U) or a read (R), and the array location accessed.

Previous dependence analysis techniques would calculate the dependence direction vectors shown in the rightmost column (or dependence distance vectors that would convey no more information). Neither standard technique would distinguish between dependence 4 and 6 unless the i and j loops were interchanged (and the dependencies updated). Normally, this is acceptable since the difference between these two dependencies is significant only when the i loop is contained within the j loop. But since we want to consider optimization as a single transformation, not a search through a tree of transformations, we need the additional information our dependence relations provide.

Example 2, part 1

Consider the following code fragment:

```
      for i := 3 to n do
          for j := 3 to m-1 do
s1:           c[i, j] := f(b[i, j-1], b[i-1, j+1])
s2:           b[i, j] := g(c[i-2, j-2]);
```

The dependencies for this code segment are shown in Table 3.

4. Scheduling a set of statements

A feasible schedule for a set of m statements is a set of functions $T_1, T_2, \ldots, T_m$ such that if $s_p[x]\ \delta\ s_q[y]$, then $T_p(x) < T_q(y)$ and if $s_p[x]\ \delta^R\ s_q[y]$ or $s_p[x]\ \delta^B\ s_q[y]$ then $T_p(x) \neq T_q(y)$ We limit our attention to integer affine schedules: the time at which $s_p[x]$ is executed must be given by an integer affine function of x.

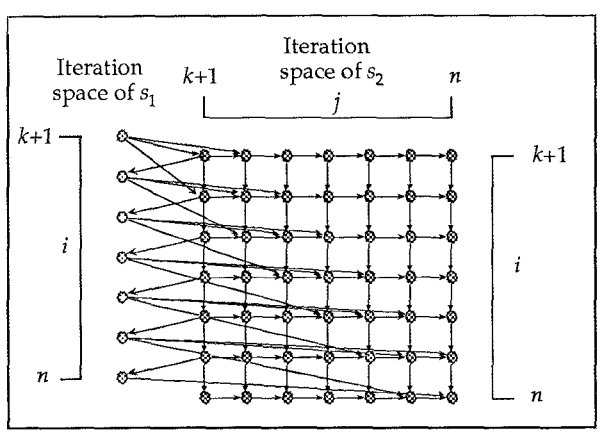

Figure 1 – Graphical illustration of dependencies for Example 1

Initially, we consider only *highly parallel* integer affine schedules. A *highly parallel* schedule is one such the total time to execute the program is $O(nm)$, where n is the maximum number of iterations of any loop and m is the number of statements. For situations in which we cannot use highly parallel schedules, we need to use nested time, a technique described in Appendix 3.

Given a schedule for each statement, it is simple to check that the schedules satisfies all the dependencies (i.e., is a feasible schedule). Unfortunately, the number of possible schedules is unlimited. Even if we restrict ourselves to reasonable schedules (with coefficients with an absolute value of at most 3 or 4), the number of possible schedules is exponential in the number of statements and the number of loops.

There are several methods we can use to limit the number of candidate schedules to a mere handful. We therefore describe methods for finding restrictions on the schedules of individual statements and restrictions between the schedules of different statements. Schedules that pass these restrictions are nominated as candidates, and are then tested to see if they are feasible.

It is perfectly acceptable to nominate some infeasible schedules. We can therefore use any partial application of the techniques described below so long as the number of remaining candidate schedules is reduced to a reasonable number.

We initially find a candidate schedule for a single statement. To do this, we must be able to characterize all of the restrictions on the schedule of that statement. In order to do that, we need to consider both the direct and transitive dependencies of a statement. In Example 1 above, statement s1 is involved in several transitive dependencies. including: $s_2[i,k+1]\ \delta\ s_1[i+1]\ \delta\ s_2[i+1, i+1]$. Note that reduction and broadcast dependencies are not transitive.

Consider a graph with a vertex for each statement and an edge from vertex s_p to vertex s_q iff there is a dependence from s_p to s_q. We can schedule each strongly connected component of the graph separately[3]. Within a SCC (strongly connected component), we consider all the self-dependences of a single statement (including transitive dependencies). If possible, we choose a feedback vertex to schedule (a vertex that, when removed, eliminates all cycles in the SCC).

Transitive self-dependencies

Transitive self-dependencies can be simple, or they can be complicated, involving a dependence chain of many statements. We describe below a method of computing the complete set of transitive self-dependencies for a statement. Restricting the number of candidate schedules to a reasonable number may require only a full application of the methods below.

The complete list of self-dependencies for a statement can be generated by considering the set of statements as a multi-graph, with each statement corresponding to a vertex and each dependence between statements corresponding to an edge. An edge from vertex p to vertex q is labeled with a dependence F_{pq}, that describes, given an iteration of s_p, the iterations of s_q that must happen later. Multiple edges from vertex p to vertex q indicate that dependencies from p to q are the union of the dependencies

[3] We may wish to schedule larger sections of the graph at a time by, allowing us to consider additional transformation that might reduce overhead in the final code.

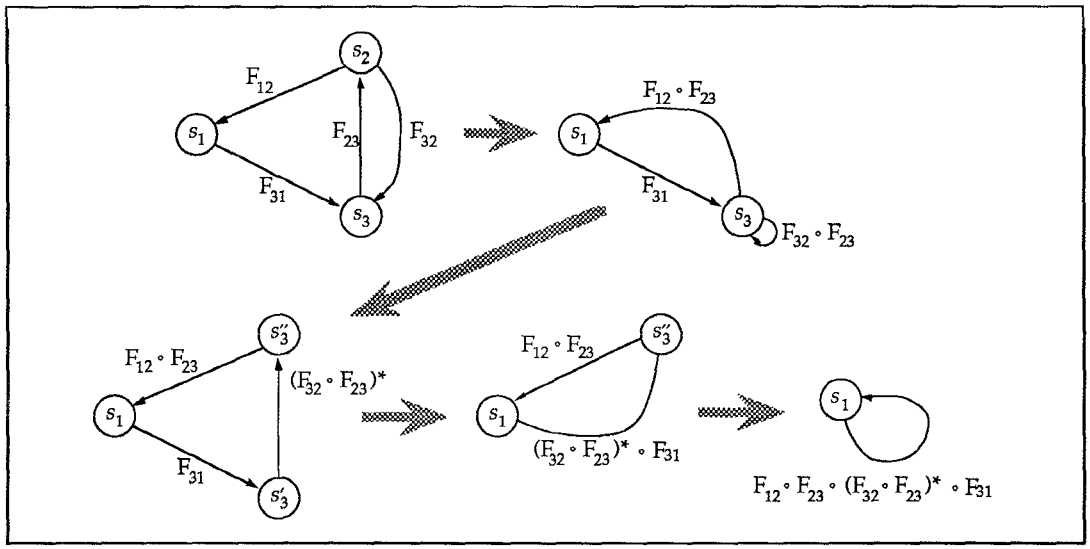

Figure 2 – Computing exact self-dependencies.

of all the edges from p to q. There may be edges from a vertex back to itself.

We can find the complete set of self-dependencies for a statement by repeatedly trimming this graph until only the vertex for the statement of interest in left. We first remove any vertices not in the same strongly connected component as the vertex of interest. We then perform the following process until only the vertex of interest remains. (*Note:* the process we use is very similar to the process of deriving a regular expression for a NFA).

If there are two edges going between the same two vertices (in the same direction), we replace them with a single edge labeled with the union of the dependencies of the two original edges. If a vertex x has a single self-edge labeled with a dependence F, we replace x with two new vertices, x' and x'', such that all edges that went to x now go to x', all edges that left x now leave from x'' and there is an edge from x' to x'' labeled with the dependence F^*. When we eliminate a vertex with no self-edges, we generate a new edge for each combination of incoming and outgoing edges, labeled with the compositions of the edges. Figure 2 shows an example of computing the exact self-dependencies for s_1. Note that s_1 is not a feedback vertex, so we are required to compute transitive closures.

Note that since we are only nominating candidate schedules, we can use approximations that report false negatives (approximations that might report no dependence exists when in fact one does exist). At this stage, approximations that produce a small number of false negatives are preferable to approximations that produce a larger number of false positives. This is particularly important since we do not currently have effective and general methods for computing F^* (we can approximate F^* for any F as $I \cup F \cup F \circ F \cup F \circ F \circ F$; for certain special forms, exact methods for computing F^* are described in Appendix 1).

Computing self-dependence distances

If F describes all the self-dependences of a statement s_p on itself, then $dd = range(F \doteq I)$ is the set of all self-dependence distances for s_p. If T_p is a candidate timing function for s_p, then

$$(T_p \backslash dd \;/([t]\ (t \leq 0)))$$

is empty iff T_p is a legal timing function for s_p. If it is not empty, we can determine a specific constant self-dependence distance of s_p that is violated by T_p.

Nominating, testing and transferring candidate schedules

We enumerate possible schedules, starting with schedules with the smallest 1-norm (the 1-norm is the Manhattan metric; the sum of the absolute values of the coefficients). Note that constant offsets in timing functions cancel out and can be ignored when examining self-dependencies. This enumeration process takes into account the limitations imposed by the specific constant self-dependence distances of s_p seen so far. For example, assume the schedule for $s_p[i, j]$ is of the form $T_p[i, j] = \alpha_p i + \beta_p j + c_p$. We might test the schedule $T_p[i, j] = j$ and be told that it is illegal because of a self-dependence $s_p[i, j]\ \delta\ s_p[i+1, j-1]$. Given this information, we know to nominate only schedules such that $\alpha_p - \beta_p > 0$.

Once we have a schedule that appears to satisfy all transitive dependencies for s_p, we check the schedule against reduction and broadcast dependencies. If it passes those tests, we transfer the schedule to the other statements in the SCC. Assume we have decided on a candidate schedule for s_p and we wish to determine the constraints this produces on the schedule for s_q. Let F be a relation that describes the dependence $s_p\ \delta\ s_q$ and let G be the relation that describes the dependence $s_q\ \delta\ s_p$. We then have

$$(T_p \circ F^{-1})[x] < T_q[x] < (T_p \circ G)[x].$$

From this information, we can often derive the only allowable form for the schedule of s_q. If this does not provide us with enough information, we can combine it with enumeration methods.

Testing and Aligning Schedules

Now that we have developed a set of candidate schedules for all statements, we can test and align them. To do this, we simply check that the schedules imply that if $s_p[x]\ \delta\ s_q[y]$, then $T_p[x] < T_q[y]$ and if $s_p[x]\ \delta^R\ s_q[y]$ or $s_p[x]\ \delta^B\ s_q[y]$, then $T_p[x] \neq T_q[y]$. Note that we allow each schedule to include a constant offset. When only considering self-dependencies, the constant offsets drop out. When comparing dependencies between two different statements, the constant offsets become very important. As will be seen, allowing fractional offsets is sometimes essential to aligning schedules[4]. In Section 4, we will see that fractional

[4] Actually, we don't need fractional offsets. Given a schedule with fractional offsets, we can obtain an equivalent (but longer) schedule by multiplying by an appropriate constant. For example, rather than using the schedules $T_1[i] = i+k+0.5$ and $T_2[i, j] = i+j$ in Example 1, we could use the schedules

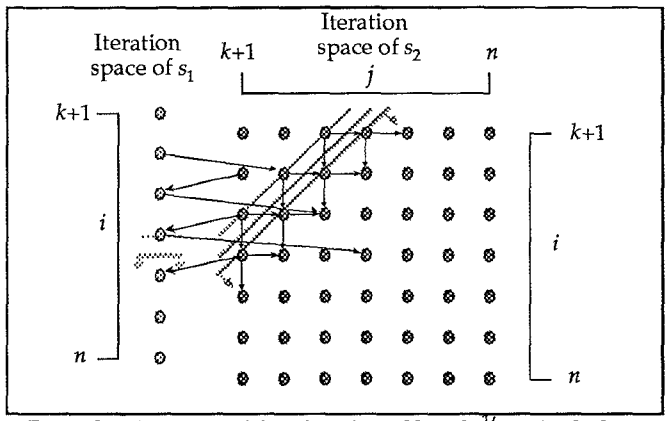

Figure 3 – Illustration of the schedules $T_1[i] = i+k+½$ and $T_2[i, j] = i+j$ for Example 1, with some of the more constraining dependencies for this example. Each wavefront shows the set of tasks performed at a particular moment of time. The figure shows the wavefronts at t equal to 2k+4, 2k+4½ and 2k+5. The wavefronts for the two iteration spaces are synchronized. In this schedule, s_1 gets executed as early as possible.

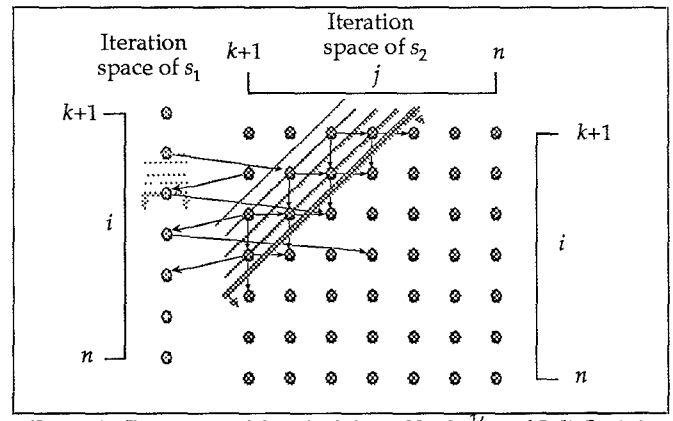

Figure 4 – Illustration of the schedules $T_1[i] = 2i-½$ and $T_2[i, j] = i+j$ for Example 1, along with some of the more constraining dependencies for this example. The wavefronts shown are for t equal to 2k+3½, 2k+4, 2k+4½ 2k+5 and 2k+5½. In this schedule, s_1 gets executed as late as possible.

offsets in schedules produce the effects of statement reordering, and the integer parts of the constant offset produce the effects of loop fusion and splitting.

We test and align schedules by generating all constraints imposed by direct dependencies and seeing if there is some set of constant offsets that can be given to the schedules so as to satisfy the constraints. Since different constant offsets produce different code, we produce a simplified set of constraints that allows us to examine the different possibilities for constant offsets.

Note that this is the one point where we are forbidden to use any approximation to the true dependencies that gives false negatives (that fails to report a dependence where there actually is one). If we used approximate methods previously, we will catch any potential problems here.

Example 1, part 2 – scheduling

Before considering transitive dependencies, we first examine direct self-dependencies and see how much they restrict the candidate schedules. Let $T_1[i] = \alpha_1 i + \gamma_1 k + c_1$ and $T_2[i, j] = \alpha_2 i + \beta_2 j + \gamma_2 k + c_2$. Without loss of generality, we assume $c_2 = 0$. We enumerate possible schedules for s_2, with the following results:

Try $T_2[i, j] = i$: fails, due to dependence distance of (0,1)
(restricts schedules to ones in which $\beta_2 > 0$)
Try $T_2[i, j] = j$: fails, due to dependence distance of (1,0)
(restricts schedules to ones in which $\alpha_2 > 0$)
Try $T_2[i, j] = i+j$: succeeds

While we could examine the transitive self-dependencies for s_1 to restrict candidates for a schedule for s_1, we illustrate our method of transferring schedules. We use the dependencies between s_1 and s_2 to transfer the schedule:

Dependence	*Inferred schedule constraint*	*range*
$s_1[i] \delta s_2[i, i]$	$T_1[i] < T_2[i, i] = 2i$	$k+1 \le i \le n$
$s_1[i] \delta s_2[i+1, i]$	$T_1[i] < T_2[i+1, i] = 2i+1$	$k+1 \le i < n$
$s_1[i] \delta s_2[i, i+1]$	$T_1[i] < T_2[i, i+1] = 2i+1$	$k+1 \le i < n$
$s_2[i, k+1] \delta s_1[i+1]$	$i+k = T_2[i-1, k+1] < T_1[i]$	$k+1 \le i \le n$
$\therefore$	$i+k < T_1[i] < 2i$	$k+1 \le i \le n$

The only solutions to these constraints with integer coefficients are $T_1[i] = i+k+c_1$, $0 < c_1 < 1$ or $T_1[i] = 2i+c_1$, $-1 < c_1 < 0$. These schedules are shown graphically in Figures 3 and 4. We consider using the first (motivated by the fact that all other things being equal, it is better to have unary coefficients). Having a candidate set of schedules, we check the schedules against all the direct dependencies. For our example, we have already checked the schedules against most of the dependencies while transferring s_2's schedule to s_1. We find that the schedules $T_1[i] = i+k+½$ and $T_2[i, j] = i+j$ are legal with respect to all the dependencies.

Example 2, part 2 – scheduling

The transitive self-dependencies for Example 2 are:

$s_1[i, j] \delta s_2[i+2, j+2] \delta s_1[i+2, j+3]$
$s_1[i, j] \delta s_2[i+2, j+2] \delta s_1[i+3, j+1]$
$s_2[i, j] \delta s_1[i, j+1] \delta s_2[i+2, j+3]$
$s_2[i, j] \delta s_1[i+1, j-1] \delta s_2[i+3, j+1]$

These dependencies give us a fair degree of freedom in scheduling this code. We consider the case where we know m is small compared to n and therefore would prefer to parallelize the i loop. This corresponds to schedules of $T_1[i, j] = j+c_1$ and $T_2[i, j] = j+c_2$ (that both meet our nomination criteria). Without loss of generality, we assume $c_2 = 0$. We now have to check that these schedules are feasible. Since we already have a schedule for both dependencies, we simply have to check that we can align the two schedules, based on the cross-statement dependencies:

$s_2[i, j-1] \delta s_1[i, j] \Rightarrow T_2[i, j-1] < T_1[i, j] \Rightarrow j-1 < j+c_1$
$s_2[i-1, j+1] \delta s_1[i, j] \Rightarrow T_2[i-1, j+1] < T_1[i, j] \Rightarrow j+1 < j+c_1$
$s_1[i-2, j-2] \delta s_2[i, j] \Rightarrow T_1[i-2, j-2] < T_2[i, j] \Rightarrow j-2+c_1 < j$

This set of constraints simplifies to $1 < c_1 < 2$ that can be satisfied with $c_1 = 1½$, giving the schedules $T_1[i, j] = j+1½$ and $T_2[i, j] = j$.

5. Code rearrangement

Once we have a schedule, we produce new code that simply moves through time, executing code as needed. To do this, we determine the minimum and maximum scheduled time for each statement. If all statements have the same minimum and maximum scheduled time, we can simply generate a single loop over time. Otherwise, we break time down into several

$T_1[i] = 2i+2k+1$ and $T_2[i, j] = 2i+2j$. However, the former schedule produces good code directly, while additional transformations would be required to produce good code from the later schedule.

segments such that during a single segment, one set of statements is scheduled.

We use loops to advance through time by whole units. Within the body of a time loop with loop variable t_{unit}, we might execute all statements that execute at times t for $t_{unit} \le t < t_{unit}+1$, or we could choose to execute all statements that are supposed to execute at times t for $t_{unit} - 1 < t \le t_{unit}$. Picking different rounding methods for fractional times effects the code that is generated. Executing statements scheduled for a unit interval around a whole time unit simply produces several distinct groups of statements to be executed in order. Statements scheduled to execute at the same moment can be executed in parallel (or in any order).

Once we have selected rounding methods for fractional times, we must generate the appropriate loops to execute all of the iterations of a statement that must be performed at any moment. Consider a schedule $T_p[i, j] = \alpha i + \beta j + \gamma k + c$. Let $T_p'[i, j] = \alpha i + \beta j$ be the variable portion of the schedule and let $g = \gcd(\alpha,\beta)$. If $g > 1$, then s_p will only be executed every g time steps, and appropriate code must be generated. Regardless of whether $g > 1$, consider the following matrix:

$$Q = \begin{bmatrix} \alpha/g & \beta/g \\ a & b \end{bmatrix}$$

$$\begin{bmatrix} \alpha/g & \beta/g \\ a & b \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} t/g \\ s \end{bmatrix}$$

We will use s as the loop variable for our non-temporal loop. We pick a and b to be integers such that $\mathrm{Det}[Q] = \pm 1$ (so that Q is unimodular [Ban90]). Therefor, Q is invertible:

$$\begin{bmatrix} \alpha/g & \beta/g \\ a & b \end{bmatrix}^{-1} \begin{bmatrix} t/g \\ s \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}$$

Performing this calculation and substituting the results into the loop bounds for i and j gives appropriate loop bounds for t/g and s.

Similar techniques are used for schedules involving more than two loop variables, although extending a schedule involving more than two loop variables into a unimodular matrix can be more difficult. Consider the schedule $T_q[i,j,k] = 6i+10j+15k$. This can be extended to a unimodular matrix as:

$$\begin{bmatrix} 6 & 10 & 15 \\ 1 & 1 & 0 \\ 0 & 1 & 4 \end{bmatrix}$$

To extend a row $[a_1, a_2, ..., a_m]$ with a gcd of 1 into a unimodular matrix, we can use a row with a 1 in position L and 0 elsewhere to transform the problem into finding a unimodular extension of $[a_1, a_2, ..., a_{L-1}, a_{L+1}, ..., a_m]$. If there exist two components of the row that have a gcd of 1, we use this technique to reduce the problem to finding a unimodular extension of those two components. If no such pair exists, we reduce the problem down into finding a unimodular extension of a row of three elements, and use more elaborate techniques that can find a unimodular extension of a row of three elements.

The Omega test's ability to perform simplification is used extensively in calculating the lower and upper bounds of the resulting loops.

Example 1, part 3 – code rearrangement

Remember that $T_1[i] = i+k+\frac{1}{2}$, so we need to execute $s_1[(t+\frac{1}{2})-k-\frac{1}{2}]$ at time $t+\frac{1}{2}$ for all t such that $k+1 \le (t+\frac{1}{2})-k-\frac{1}{2} \le n$. Similarly, $T_2[i, j] = i+j$, so we need to execute $s_2[t-\alpha, \alpha]$ at time t for all t and α such that $k+1 \le t-\alpha, \alpha \le n$. Simplifying these bounds, we find that we execute $s_1[t-k]$ at $t+\frac{1}{2}$ for all t, $2k+1 \le t \le n+k$ and we execute $s_2[t-\alpha, \alpha]$ at time t for all t and α s.t. $2k+2 \le t \le 2n$ and $k+1, t-n \le \alpha \le t-(k+1), n$. During the whole unit time interval around t_u, we decide to execute all statements that execute at time $t_u - 1 < t \le t_u$, rather than $t_u \le t < t_u+1$. This realignment allows $s_1[(t\frac{1}{2})-k-\frac{1}{2}]$ to execute at $t-\frac{1}{2}$ during the period $2k+2 \le t \le n+k+1$ (which aligns the start times for s_1 and s_2). This generates the following code:

```
for t := 2*k+2 to n+k+1 do
    s1[t – 1– k]
    forall α := max(k+1, t–n) to min(n, t–k–1) do
        s2[t – α, α]
for t := n+k+2 to 2*n do
    forall α := max(k+1, t–n) to min(n, t–k–1) do
        s2[t – α, α]
```

We find we can resolve the min and max computations statically, producing:

```
for t := 2*k+2 to n+k+1 do
    s1[t – 1– k]
    forall α := k+1 to t–k–1 do s2[t – α, α]
for t := n+k+2 to 2*n do
    forall α := t–n to n do s2[t – α, α]
```

Substituting back the bodies of the statements and renaming α to j produces:

```
for t := 2*k+2 to n+k+1 do
    B[t–1–k, t–1–k] := B[t–2–k, k+1] * B[k, k]
    forall j := k+1 to t–k–1 do
        B[t–j, j] := B[t–j, j] + B[t–j–1, j] + B[t–j, j–1]
                   + B[t–j, k]*B[k, j]
for t := n+k+2 to 2*n do
    forall j := t–n to n do
        B[t–j, j] := B[t–j, j] + B[t–j–1, j] + B[t–j, j–1]
                   + B[t–j, k]*B[k, j]
```

Example 2, part 3 – code rearrangement

For example 2, the schedules are $T_1[i, j] = j+1\frac{1}{2}$ and $T_2[i, j] = j$. This tells us that we must execute $s_1[\alpha, (t+\frac{1}{2})-1\frac{1}{2}]$ at time $t+\frac{1}{2}$ for $4 \le t \le m$ and execute $s_2[\alpha, t]$ at time t for $3 \le t \le m-1$. This produces the following code.

```
forall α := 3 to n do s2[α, 3];
for t := 4 to m–1 do
    forall α := 3 to n do s2[α, t];
    forall α := 3 to n do s1[α, t–1];
forall i := 3 to n do s1[α, m–1];
```

Substituting back the bodies of the statements produces:

```
forall i := 3 to n do b[i, 3] := g(c[i–2, 1]);
for t := 4 to m–1 do
    forall i := 3 to n do b[i, t] := g(c[i–2, t–2]);
    forall i := 3 to n do c[i, t–1] := f(b[i, t–2], b[i–1, t]);
forall i := 3 to n do c[i, m–1] := f(b[i, m–2], b[i–1, m]);
```

Note that the tightness of the schedule did not allow us any flexibility in aligning the schedule. Consider what would happen if s_2 in the original code was different:

```
    for i := 3 to n do
      for j := 4 to m–1 do
s1:       c[i, j] := f(b[i, j–1], b[i–1, j+1]);
s2:       b[i, j] := g(c[i–2, j–3]);
```

Using schedules of $T_1[i, j] = j+c_1$ and $T_1[i, j] = j$ would produce alignment constraints of $1 < c_1 < 3$, which would allow us to choose $c_1 = 2$. Since the fractional parts of c_1 and $c_2 (=0)$ are

equal, the computations for s_1 and s_2 can be performed in parallel. This would allow us to produce the code (for clarity, the original statements have *not* been substituted back):

```
forall α := 3 to n do s2[α, 4]
forall α := 3 to n do s2[α, 5]
for t := 6 to m–1 do
  forall α := 3 to n do
    cobegin
      s2[α, t]
      s1[α, t–2]
    coend
forall α := 3 to n do s1[α, m–2]
forall α := 3 to n do s1[α, m–1]
```

6. The relation between schedules and optimizations

Schedules of the form $T_p[i, j] = i+c_p$ or $T_p[i, j] = j+c_p$ correspond to parallelizing s_k along j and i respectively. Schedules of the form $T_p[i, j] = i+j+c_p$, $T_p[i, j] = i+2j+c_p$ or $T_p[i, j] = i-j+c_p$ correspond to parallelization using different wavefronts or loop skewing. Fractional offsets in schedules produce the effects of statement reordering, and the integer parts of the constant offset produce the effects of loop fusion and splitting.

7. Completeness Results

We claim that a transformation equivalent to *any* transformation that can be obtained via some combination of

- statement reordering,
- loop interchange,
- loop fusion,
- loop skewing,
- loop splitting,
- loop reversal,
- loop alignment,
- loop distribution and
- loop parallelization

will be found by our optimization methods. Our methods can find optimizations that no combination of the above techniques will find (as was seen in Example 1 and 2), but it is difficult to describe the complete set of optimizations our techniques might find.

We should note one type of code transformation our current methods cannot produce: a parallel loop with a loop body containing sequential code. For example, given the program

```
for i := 1 to n do
    for j := 2 to m do
        a[i, j] := a[i, j] + a[i, j–1]
```

our methods could not produce the optimization below.

```
forall i := 1 to n do
    for j := 2 to m do
        a[i, j] := a[i, j] + a[i, j–1]
```

However, our methods could produce the code

```
for j := 2 to m do
    forall i := 1 to n do
        a[i, j] := a[i, j] + a[i, j–1]
```

that, except for synchronization overhead, has the same running time. We are looking at ways to overcome this limitation. For the moment, we assume that a latter step of the optimization process may look at moving sequential code inside of parallel loops.

THEOREM 1. Let P be any program consisting solely of looping constructs and conditional assignments. Let P' be any optimized version of P, subject to the following constraints:

- The program P' consists of sequential loops, parallel loops, cobegin/coend constructs and conditional assignments such that the body of each parallel loop contains no sequential code.
- Each conditional assignment statement $s_x[i_1, i_2, \ldots, i_m]$ in P appears once in P' as $s_x[f_1, f_2, \ldots, f_m]$, where $f_1, f_2, \ldots, f_m$ are affine functions of the loop indices surrounding the occurrence of s_x in P'.
- The execution of P' involves executing the same iterations of the same statements as P, in some order that respects the ordering implied by the dependencies between the statements and ordering between the iterations of the statements in P.

Then the methods described in this paper can derive P' from P.

PROOF: The restrictions imposed on P' allow us to direct derive an affine schedule for the time at which each statement $s_x[i_1, i_2, \ldots, i_m]$ is executed (involving nested time iff s_x appears in nested sequential loops). Therefore P' specifies a complete set of schedules for the statements of P. Since P' must respect the ordering imposed by the dependencies of P, the schedule for P' must respect those dependencies. Therefore, our methods could derive the schedule that produces P'. □

THEOREM 2. If P' is derived from P by some combination of statement reordering, loop interchange, loop fusion, loop skewing, loop splitting, loop reversal, loop distribution and loop parallelization, and P' doesn't contain any sequential code inside of parallel loops, then P' meets the restrictions described in Theorem 1.

PROOF: None of these optimizations duplicate statements, and each of them replaces the indices with an affine function of the original indices (possibly the identity function). Since the composition of a series of affine functions is an affine function, the statements in P' must be obtained by replacing the original loop indices with affine functions of the indices. Since any optimization that did not respect the ordering require by P would be a faulty optimization, P' must respect the ordering of the iterations of P imposed by the dependencies.

COROLLARY 1. If P' is derived from P by some combination of loop optimizations discussed in Theorem 2, our methods can derive a program P'' from P that has the same inherent parallelism as P'.

PROOF: Given P', we can derive a program P'' that contains no sequential code inside of parallel loops by performing loop distribution and loop interchange to move parallel loops inward. We are assured that we can do these transformations because no dependencies can exist between different iterations of a parallel loop. The program P'' has the same inherent parallelism as P', and by Theorems 2 and 1, our methods can derive P''. □

8. Related work

Methods for performing individual optimizations have been described by a long list of authors, including [Ban79, AK87, Nico87, Wol89, LYZ89].

Whitfield and Soffa [WS90] looked at the interactions between optimizations and the order in which they should be performed. The only loop optimizations they considered were loop interchanging and loop fusion. (They also consider strip-mining and loop unrolling, which are optimizations that would

be applied after the optimizations in this paper were performed). They found that loop interchange should be performed before loop fusion, although it is not clear this would still hold if they allowed loop splitting.

Hudson Ribas [Rib90] discusses techniques for manipulating dependencies that are similar to ours. His techniques can be used to calculate the dependencies between two statements, but he does not provide methods that allow the wide range of operations on dependencies supported by our work[5].

Wavefront methods

The idea of equating code-rearrangement with finding schedules for an iteration space has been examined by several authors [Lam74, Rib90, Wol90, Ban90]. However, previous approaches have been limited in their ability to deal with imperfectly nested loops and complicated dependencies.

The intent of Banerjee's paper [Ban90] is very close to our own. However, Banerjee only discusses perfectly nested double loops with constant distance vectors, and only considered transformations that can be obtained by combinations of loop reversal, loop interchange and loop skewing.

Scheduling of recurrence equations

The problem of finding a set of schedules for statements that respect the dependencies between the statements is closely related to the topic of parallel scheduling of recurrence equations. Doing this for a set of uniform recurrence equations was closely studied in [KMW67]. An example of a set of k–dimensional *uniform* recurrence equations is

$$a_1(x) = f_1(a_2(x - w_1), a_3(x - w_2))$$
$$a_2(x) = f_2(a_1(x - w_3), a_2(x - w_4), a_3(x - w_5), a_3(x - w_6))$$
$$a_3(x) = f_3(a_2(x - w_7)),$$

where x is an integer k-vector and $w_1, \ldots, w_7$ are constant integer k-vectors. These equations are uniform because all equations are defined over the same number of dimensions and because each the value of each equation at a point x only depends on equation values located at constant offsets from x. Recurrence equations are only defined at integer coordinates. Generally, we wish to determine the values of the recurrence equations at the integer coordinates within a convex polyhedron of k–space and any needed values outside that area are supplied as input.

Determining a schedule for a single uniform recurrence equation was solved in [KMW67]. Determining schedules for a set of two or more uniform recurrence equations is more difficult [KMW67, Chen86, RK88]. The recent work is in the context of attempting to compile uniform recurrence equations into a form suitable for execution on processor arrays.

We solve an even more difficult problem, because our dependencies are not uniform. In particular, different recurrence equations (i.e., statements) are defined on different number of dimensions (i.e., are at different nesting depths) and the locations from which $a_i(x)$ might need values are not restricted to be constant offsets from x.

There is a direct relation between a set of dependencies and a set of recurrence equations that require the same schedule. For example, a schedule that satisfies the set of dependencies

$$s_2[i, j-1] \; \delta \; s_1[i, j], \; s_2[i-1, j+1] \; \delta \; s_1[i, j], \; s_1[i-2, j-2] \; \delta \; s_2[i, j]$$

can be satisfied if and only if the same schedule can be used to compute the recurrence equations

$$a_1(x) = f_1(\, a_2(x - \langle 0, 1 \rangle), a_2(x - \langle 1, -1 \rangle) \,)$$
$$a_2(x) = f_2(\, a_1(x - \langle 2, 2 \rangle) \,)$$

[5] Such as the ability compose dependencies, which is essential to scheduling imperfected nested loops.

9. Conclusion

In this paper, we have described uniform methods for performing loop optimizations. These techniques automatically consider all possible combinations of standard optimizations (such as parallelization, loop fusion, loop splitting, loop interchange and loop skewing) and additional optimizations. This allows our techniques to find ways of parallelizing code that could not be parallelized by previous techniques. Besides being of practical use, our techniques hold the promise of providing a simpler language for discussing loop optimizations.

We have shown that our techniques allows us to easily parallel two programs that are difficult or impossible to parallelize using standard techniques.

These techniques can easily be adapted for single assignment programming languages with lazy arrays. In this setting, only a single write can be performed to any array location and dependencies always go from a write to reads of the location.

There are some kinds of loop optimization, such as node splitting [Poly88] and having sequential code inside of parallel loops, that cannot be described by our current methods. We are currently exploring methods of extending our techniques.

References

[AK87] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form". ACM TOPLAS, Vol. 9, No. 4, Oct. 1987, 491-542.

[Ban79] U. Banerjee, "Speedup of ordinary programs," *Ph. D Thesis, Report No. UIUCDCS-R-79-989*, Dept. of Computer Sciences, University of Illinois at Urbana-Champaign, 1979.

[Ban88] U. Banerjee, "Dependence Analysis for Supercomputing," *Kluwer Academic Publishers, 1988.*

[Ban90] U. Banerjee, "Unimodular Transformations of Double Loops," Proc. of 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA, August 1990.

[Chen86] Marina Chen, "A Design Methodology for Synthesizing Parallel Algorithms and Architectures," *Journal of Parallel and Distributed Computing*, page 461-491, 1986.

[DE73] G.B. Dantzin and B.C. Eaves, Fourier-Motzkin Elimination and Its Dual, *Journal of Combinatorial Theory (A)* 14 (1973), 288-297.

[Grun90] D. Grunwald, "Data Dependence: The λ Test Revisted," *1990 International Conference on Parallel Processing*, Penn State Univ. Press, 1990, pages II-220 – II223.

[KKP90] X. Kong, D. Klappholz and K. Psarris, "The I Test: A new test for subscript data dependence," *1990 International Conference on Parallel Processing*, Penn State Univ. Press, 1990, pages II-204 – II211.

[KMW67] R.M. Karp, R.E. Miller and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM* Vol 14, pp563-590, 1967.

[Lam74] L. Lamport, "The Parallel Execution of DO Loops," *Comm. of the ACM*, Vol. 17, No. 2, 1974.

[LYZ89] Z. Li, P.C.Yew and C.Q. Xhu, "Data dependence analysis on multi-dimensional array references," *Proc. of ACM 1989 Int. Conf. on Supercomputing*, Crete, Greece, July 1989.

[Nico87] A. Aiken and A. Nicolau, "Loop Quantization, or unwinding done right," *Proceedings of the 1987 ACM International Conference on Supercomputing*, Springer-Verlag LNCS 298, May 1987.

[Poly88] C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[Pug91] William Pugh. *The Omega Test: a fact and practical integer programming algorithm for dependence analysis*, Tech. Report #91-50, Institute for Advanced Computer Studies, Univ. of Maryland, College Park, April 1991.

[Ribas90] Hudson Ribas, Obtaining Dependence Vectors for Nested-Loop Computations, 1990 International Conference on Parallel Processing.

[RK88] S.K. Rao and T. Kailath, "Regular Iterative Algorithms and their Implementation on Processor Arrays," *Proc. of the IEEE*, Vol 76, No. 3, pp 259-269, March 1988.

[Wol89] Michael Wolfe, *Optimizing Supercompilers for Supercomputers*, The MIT Press, 1989.

[Wol90] Michael Wolfe, Massive Parallelism through program Restructuring, Symposium on Frontiers on Massively Parallel Computation, College Park, MD, October, 1990

[WS90] D. Whitfield and M.L. Soffa, "An Approach to Ordering Optimizing Transformations," *Proc. of the ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, Seattle, March 1990, pages 137-146.

[WT90] Michael Wolfe and Chau-Wen Tseng. *The Power Test for Data Dependence*, Tech. Report CS/E90-015, Oregon Graduate Institute, August 1990.

[ZC91] Hans Zims and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*, ACM Press, 1991.

[ZY90] Z. Li and P.C. Yew, "Some Results on Exact Data Dependency Analysis," *Languages and Compilers for Parallel Computing*, Edited by Gelernter, Nicolau and Padua, MIT Press, 1990, pages 374–401.

Appendix 1: Implementation of Tuple Relations

In this section, we describe our implementation of operations on tuple relations and sets. A relation is represented by the union of a list of simple relations. A simple relation is represented by a set of linear equalities and inequalities involving the input variables, the output variables, symbolic constants and wildcards.

We do this because in general we cannot produce a closed form for the union of two relations. All operations distribute across union, so this does not pose any fundamental problems.

In the following, we use $C(F)$ to denote set of constraints for a relation or set F. We also use $\vec{s}$ to denote the set of input variables for a relation or a set, $\vec{t}$ to denote the set of output variables for a relation and $\vec{\alpha}$ or $O(\beta,^{\rightarrow})$ to denote a set of additional wildcard variables. Given a set of constraints C, we use $C[\vec{s} \rightarrow \vec{\alpha}]$ to denote the set of constraints produced by renaming $\vec{s}$ to $\vec{\alpha}$. In the following, F and G denote relations, S and T denote sets.

$$C(F^{-1}) \equiv C(F)\ [\vec{s} \rightarrow \vec{t}, \vec{t} \rightarrow \vec{s}]$$
$$C(F \circ G) \equiv C(F)[\vec{s} \rightarrow \vec{\alpha}]\ \&\ C(G)[\vec{t} \rightarrow \vec{\alpha}]$$
$$C(F \cap G) \equiv C(F)\ \&\ C(G)$$
$$C(F \setminus S) \equiv C(F)\ \&\ C(S)$$
$$C(F / S) \equiv C(F)\ \&\ C(S)[\vec{s} \rightarrow \vec{t}]$$
$$C(F \dot{+} G) \equiv C(F)[\vec{t} \rightarrow \vec{\alpha}]\ \&\ C(G)[\vec{t} \rightarrow \vec{\beta}]\ \&\ \vec{t} = \vec{\alpha} + \vec{\beta}$$
$$C(F \dot{-} G) \equiv C(F)[\vec{t} \rightarrow \vec{\alpha}]\ \&\ C(G)[\vec{t} \rightarrow \vec{\beta}]\ \&\ \vec{t} = \vec{\alpha} - \vec{\beta}$$
$$C(S \cap T) \equiv C(S)\ \&\ C(T)$$
$$C(\text{domain}(F)) \equiv C(F)[\vec{t} \rightarrow \vec{\alpha}]$$
$$C(\text{range}(F)) \equiv C(F)[\vec{s} \rightarrow \vec{\alpha}, \vec{t} \rightarrow \vec{s}]$$

For operations that introduce additional wildcard variables, the Omega test is used to simplify the sets of constraints (possibly eliminating some of the additional wildcard variables).

The above methods involve slightly more work than is needed. By keeping the constraints is specially ordered form, we can do part (but not all) of the work involved in simplifying the results of operations such as composition by using matrix multiplication. However, the above is a concise description of what gets calculated, if not necessarily how it gets calculated. Using more complicated representations can give speed improvements of a factor of 2 or so.

Transitive Closure

It is often difficult or impossible to derive a simple, closed form for the transitive closure of a dependency. Fortunately, in the situations in which we need to compute the closure of a dependency, we can use approximations that produce false positives and/or false negatives. It is often more efficient in practice to approximate F^* as some finite sequence such as

$$F^* \approx I \cup F \cup (F \circ F) \cup (F \circ F \circ F) \cup (F \circ F \circ F \circ F)$$

Exact computation of a closed form for the closure of a dependency

We calculate a closed form for the transitive closure of a dependency only when the dependence involves a constant dependence distance. More specifically, assume the constraints for a dependence F can be represented as shown below (where P contains only inequality constraints and $\vec{c}$ is a vector of constant offsets):

$$C(F) \equiv \vec{t} = \vec{s} + \vec{c}\ \&\ P$$

In this case, the constraints for the transitive closure of F are:

$$F^* = I \cup F^+$$
$$C(F^+) \equiv \vec{\alpha} = \vec{s} + \vec{c}\beta\ \&\ \beta \geq 0$$
$$\&\ \vec{t} = \vec{\alpha} + \vec{c}\ \&\ P[\vec{t} \rightarrow \vec{s} + \vec{c}]\ \&\ P[\vec{s} \rightarrow \vec{\alpha}, \vec{t} \rightarrow \vec{\alpha}+\vec{c}]$$

Appendix 2: The Omega test

The Omega test [Pug91] can be used to decide if a set of linear equalities and inequalities has an integer solution. Also, it can be used to simplify an integer programming problem by eliminating wildcard variables. The full details of the Omega test are described in [Pug91]. We give a few details of the important aspects here. Although the Omega test can take exponential time, it appears to take reasonable time in practice (no more than several milliseconds to decide or simplify the problems that arise in practice).

Simplifying equality tests

An equality test is a test of the form $a + b^T x = 0$. We calculate g, the GCD of b. If a is not evenly divisible by g, there is no way to satisfy the constraint. Otherwise, we divide a and b by g.

Simplifying and Tightening inequality tests

An inequality test is a test of the form $a + b^T x \geq 0$. We calculate g, the GCD of b and replace a and b with $\lfloor a/g \rfloor$ and b/g.

Eliminating wild cards constrained by linear equalities

From a set of linear equality involving a wild card, we may be able to find a valid substitution for the wild card that will allow us to eliminate it. Methods similar to the generalized GCD test can be used [Ban88, WT90], as can other methods we have devised that appear simpler in practice [Pug91].

Checking for redundant or impossible inequalities

We can simplify the inequality tests by checking when a combination of some inequality constraints imply or contradict another inequality constraint. For example, $x \geq 5$ implies $x \geq 3$ and is contradicted by $-x \geq -4$. In such situations, we can eliminate the implied constraint or decide that the entire system is infeasible if we find an impossible constraint.

We could determine all redundant or impossible inequalities using integer programming techniques, but this probably would not be cost effective We can efficiently check to see if an inequality is implied or contradicted by any single or pair of other dependencies. In practice, we have found it sufficient to check each pair of dependencies to see if they contradict each other or if one of the pair is redundant.

Eliminating unused wild cards constrained by linear inequalities

If a wildcard is involved in inequalities tests but not equality tests and all the coefficients of the wildcard are unary, we elim-

inate the variable by performing exact integer Fourier-Motzkin variable elimination [DE73, WT90, Pug91] on it.

Testing if a relation or set is vacuous

We can use the Omega test [Pug91] to determine if an integer programming problem has a solution. The Omega test applies all the techniques described above, plus special techniques for use when there are no variables that can be eliminated via exact integer Fourier-Motzkin variable elimination [DE73, WT90, Pug91].

If a set of constraints has a solution, we use an adaptation of the Omega test that returns an actual solution. A further adaptation forces the Omega test to return some sample solution other than (0,0,...,0), or tell us that (0,0,...,0) is the only solution.

Appendix 3: Nested time

To consider less highly parallel schedules, we must consider nested time. Normally, we require that if $s_p[x]\ \delta\ s_q[y]$, then $T_p[x]$ is strictly less than $T_q[y]$. We relax this by allowing $T_p[x]$ to be at most $T_q[y]$. This schedules one level of time. At any moment t produced by this schedule, many tasks may be scheduled to be executed, some of which must be performed before others. We handle this by producing a set of dependencies involving only dependencies between statement iterations scheduled to execute at the same moment. We use those dependencies and our standard techniques to derive an inner schedule for each statement, giving a second time index.

Although we could attempt to parallelize a program by using many levels of nested time, using at most two levels of time seems appropriate if we wish to parallelize the code.

It should be clear that using nested time corresponds to using nested serial loops in the final program.

Appendix 4: Iterative smoothing example

This example concerns a program for iterative smoothing. The program is automatically transformed into the standard red-black algorithm for iterative smoothing.

Original program

```
for k := 1 to p do
   for i := 2 to n-1 do
      for j := 2 to m-1 do
s1[k,i,j]     a[i, j] := (a[i, j-1] + a[i-1, j] + a[i+1, j] + a[i, j+1] )/4;
```

Dependencies:

$s_1[k,i,j]\ \delta\ s_1[k',i,j+1]$ $\quad 1 \le k \le k' \le p\ \&\ 2 \le i \le n-1\ \&\ 2 \le j < m-1$
$s_1[k,i,j]\ \delta\ s_1[k',i+1,j]$ $\quad 1 \le k \le k' \le p\ \&\ 2 \le i < n-1\ \&\ 2 \le j \le m-1$
$s_1[k,i,j]\ \delta\ s_1[k',i,j-1]$ $\quad 1 \le k < k' \le p\ \&\ 2 \le i \le n-1\ \&\ 3 \le j \le m-1$
$s_1[k,i,j]\ \delta\ s_1[k',i-1,j]$ $\quad 1 \le k < k' \le p\ \&\ 3 \le i \le n-1\ \&\ 2 \le j \le m-1$

Finding a schedule

Our schedule is of the form $T_1[k,i,j] = \alpha k + \beta i + \gamma j$

Try $T_1[k,i,j] = k$: fails due to dd distance of (0,0,1)
Try $T_1[k,i,j] = j$: fails due to dd distance of (0,1,0)
Try $T_1[k,i,j] = i+j$: fails due to dd distance of (1,0,-1)
Try $T_1[k,i,j] = 2k+i+j$: succeeds

Finding an appropriate code transformation

We first find a unimodular extension of schedule:

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

This extensions requires us to use a forall loop on k and i. The loop bounds we derive are:

$6 \le t \le 2p+n+m-2$
$1 \le k \le n$
$(2+t-m-n)/2 \le k \le (t-4)/2$
$2 \le i \le n-1$
$1+t-m-2k \le i \le t-2-2k$

Generated Code

This analysis leads to the following generated code:

```
for t := 6 to 2*p+n+m-2 do
   for k := max(1, ⌈(2+t-m-n)/2⌉) to min(⌊(t-4)/2⌋, p) do
   for i := max(2, 1+t-m-2*k) to min(t-2-2*k, n-1) do
      s1[k, i, t - 2*k - i]
```

Appendix 5: QR decomposition via Givens Rotations Example

This problem involve the reduction of a matrix to upper triangular form via Givens Totations. We assume our language has been extended with simultaneous assignment statements and functions that can return multiple values.

```
   for c := 1 to n-1 do
      for r := n-1 downto c do
s1[c, r]     cosTheta, sinTheta :=
                computeRotation(a[r+1, c], a[r, c])
             for k := c to n do
s2[c, r, k]     a[r, k], a[r+1, k] :=
                   cosTheta * a[r, k] + sinTheta * a[r+1, k],
                   cosTheta * a[r+1, k] - sinTheta * a[r, k]
```

To produce good code for this problem, we would need to perform scalar expansion on cosTheta and sinTheta and local to each loop and transform this code into:

```
   for c := 1 to n-1 do
      for r := n-1 downto c do
s1[c, r]     cosTheta[r,c], sinTheta[r,c] :=
                computeRotation(a[r+1, c], a[r, c])
             for k := c to n do
s2[c, r, k]     a[r, k], a[r+1, k] :=
                   cosTheta[r,c]*a[r, k] + sinTheta[r,c]*a[r+1, k],
                   cosTheta[r,c]*a[r+1, k] - sinTheta[r,c]*a[r, k]
```

Dependencies

This example contains the dependencies shown in Table 4.

Finding a Schedule

For this example, we decide to attempt to produce code that avoids broadcast "dependencies". We first nominate a schedule for s_2:

try $T_2[c,r,k] = c$: fails, due to dd distance of (0,-1,0)
try $T_2[c,r,k] = -r$: fails, due to dd distance of (1,0,0)
try $T_2[c,r,k] = c-r$: fails, due to dd distance of (1,1,0)
try $T_2[c,r,k] = 2c-r$: fails,
   due to broadcast dd distance of $(0,0,\neq 0)$
try $T_2[c,r,k] = 2c-r+k$: succeeds

Direct dependencies	*range*	from	to
$s_2[c, r, k]\ \delta\ s_2[c', r-1, k]$	$1 \le c \le c' \le k \le n\ \&\ c' < r < n$	s_2 U a[r, k]	s_2 U a[r+1, k]
$s_2[c, r, k]\ \delta\ s_2[c', r, k]$	$1 \le c < c' \le k \le n\ \&\ c' \le r < n$	s_2 U a[r+1, k]	s_2 U a[r+1, k]
$s_2[c, r, k]\ \delta\ s_2[c', r, k]$	$1 \le c < c' \le k \le n\ \&\ c' \le r < n$	s_2 U a[r, k]	s_2 U a[r, k]
$s_2[c, r, k]\ \delta\ s_2[c', r+1, k]$	$1 \le c < c' \le k \le n\ \&\ c' \le r+1 < n$	s_2 U a[r+1, k]	s_2 U a[r, k]
$s_1[c, r]\ \delta\ s_2[c, r, k]$	$1 \le c \le k \le n\ \&\ c \le r < n$	s_1 U theta	s_2 R theta
$s_2[c, r, k]\ \delta\ s_1[k, r]$	$1 \le c < k \le n\ \&\ c \le r < n$	s_2 U a[r, k]	s_1 R a[r, k]
$s_2[c, r, k]\ \delta\ s_1[k, r+1]$	$1 \le c < k \le n\ \&\ c \le r < n$	s_2 U a[r+1, k]	s_1 R a[r, k]
$s_2[c, r, k]\ \delta\ s_1[k, r-1]$	$1 \le c < k \le n\ \&\ c \le r < n$	s_2 U a[r, k]	s_1 R a[r+1, k]
Broadcast "dependencies"			
$s_2[c, r, k]\ \delta^B\ s_2[c, r, k']$	$1 \le c \le r \le n\ \&\ c \le k \le n\ \&\ c \le k' < n$	s_2 R theta	s_2 R theta

Table 4 - Dependencies for QR Decomposition via Givens Rotations

Next, we transfer this schedule to s_1. In doing so, we use the simplification abilities of the Omega test to remove wild card variables (k and c') from the constraints:

dependence	constraint
$s_1[c, r]\ \delta\ s_2[c, r, k]$	$T_1[c, r] < 2c-r+k,\ c \le k$
	$T_1[c, r] < 3c-r$
$s_2[c', r-1, c]\ \delta\ s_1[c, r]$	$2c'+c-r+1 < T_1[c, r],\ c' < c$
	$3c-r-1 < T_1[c, r]$
$s_2[c', r, c]\ \delta\ s_1[c, r]$	$2c'+c-r < T_1[c, r],\ c' < c$
	$3c-r-2 < T_1[c, r]$
$s_2[c', r+1, c]\ \delta\ s_1[c, r]$	$2c'+c-r-1 < T_1[c, r],\ c' < c$
	$3c-r-3 < T_1[c, r]$

This gives $T_1[c, r] = 3c-r-\frac{1}{2}$, and we round fractions so as to execute $t_{unit}-1 < t \le t_{unit}$ in one iteration of the time loop.

Finding an appropriate code transformation

We find that we can perform appropriate unimodular extensions of the schedule by looping on c for s_1 and on c and k for s_2. For s_2, we obtain the following loop bounds:

$4-n \le t \le 2n-1$

$1, t-n \le t \le n-1, (n-1+t)/3$

$c, t-c \le k \le n-1+t-2c, n-1$

For s_1, we obtain the following loop bounds:

$4-n \le (t+\frac{1}{2}) \le 2n-2$

$1, (t+\frac{1}{2})/2 \le c \le (n-1+t+\frac{1}{2})/3, n-1$

Generated Code

```
for t := 4 - n to 2n - 1 do
    forall c := max(1, ⌈t/2⌉) to min(⌊(n - 1 + t)/3⌋, n-1) do
        s1[c, 3c - t]
    forall c := max(1, t - n) to min(⌊(n - 1 + t)/3⌋, n-1) do
        forall k := max(c, t - c) to min(n - 1 + t - 2c, n) do
            s2[c, 2c+k-t, k]
```

Scheduling, transformation and resulting code, Take 2

Without worrying about broadcast dependencies, we would settle on schedules of $T_2[c, r, k] = 2c-r$ and $T_1[c, r] = 2c-r-\frac{1}{2}$. This would have produced the following code

```
for t := 3 - n to n - 1 do
    forall c := max(1, t) to min(⌊(n - 1 + t)/2⌋, n-1) do
        s1[c, 2c - t]
    forall c := max(1, t) to min(⌊(n - 1 + t)/2⌋, n-1) do
        forall k := c to n do
            s2[c, 2c-t, k]
```