UMIACS-TR-94-40 December, 1992
CS-TR-3250 Revised March, 1994

Static Analysis of Upper and Lower Bounds on Dependences and Parallelism

William Pugh
pugh@cs.umd.edu
Institute for Advanced Computer Studies
Dept. of Computer Science

David Wonnacott
davew@cs.umd.edu
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

Abstract

Existing compilers often fail to parallelize sequential code, even when a program can be manually transformed into parallel form by a sequence of well-understood transformations (as is the case for many of the Perfect Club Benchmark programs). These failures can occur for several reasons: the code transformations implemented in the compiler may not be sufficient to produce parallel code, the compiler may not find the proper sequence of transformations, or the compiler may not be able to prove that one of the necessary transformations is legal.

When a compiler extract sufficient parallelism from a program, the programmer extract additional parallelism. Unfortunately, the programmer is typically left to search for parallelism without significant assistance. The compiler generally does not give feedback about which parts of the program might contain additional parallelism, or about the types of transformations that might be needed to realize this parallelism. Standard program transformations and dependence abstractions cannot be used to provide this feedback.

In this paper, we propose a two step approach for the search for parallelism in sequential programs: We first construct several sets of constraints that describe, for each statement, which iterations of that statement can be executed concurrently. By constructing constraints that correspond to different assumptions about which dependences might be eliminated through additional analysis, transformations and user assertions, we can determine whether we can expose parallelism by eliminating dependences. In the second step of our search for parallelism, we examine these constraint sets to identify the kinds of transformations that are needed to exploit scalable parallelism. Our tests will identify conditional parallelism and parallelism that can be exposed by combinations of transformations that reorder the iteration space (such as loop interchange and loop peeling).

This approach lets us distinguish inherently sequential code from code that contains unexploited parallelism. It also produces information about the kinds of transformations that will be needed to parallelize the code, without worrying about the order of application of the transformations. Furthermore, when our dependence test is inexact, we can identify which unresolved dependences inhibit parallelism by comparing the effects of assuming dependence or independence. We are currently exploring the use of this information in programmer-assisted parallelization.

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

Static Analysis of Upper and Lower Bounds on Dependences and Parallelism

William Pugh
Univ. of Maryland
and
David Wonnacott
Univ. of Maryland

Existing compilers often fail to parallelize sequential code, even when a program can be manually transformed into parallel form by a sequence of well-understood transformations (as is the case for many of the Perfect Club Benchmark programs). These failures can occur for several reasons: the code transformations implemented in the compiler may not be sufficient to produce parallel code, the compiler may not find the proper sequence of transformations, or the compiler may not be able to prove that one of the necessary transformations is legal.

When a compiler extract sufficient parallelism from a program, the programmer extract additional parallelism. Unfortunately, the programmer is typically left to search for parallelism without significant assistance. The compiler generally does not give feedback about which parts of the program might contain additional parallelism, or about the types of transformations that might be needed to realize this parallelism. Standard program transformations and dependence abstractions cannot be used to provide this feedback.

In this paper, we propose a two step approach for the search for parallelism in sequential programs: We first construct several sets of constraints that describe, for each statement, which iterations of that statement can be executed concurrently. By constructing constraints that correspond to different assumptions about which dependences might be eliminated through additional analysis, transformations and user assertions, we can determine whether we can expose parallelism by eliminating dependences. In the second step of our search for parallelism, we examine these constraint sets to identify the kinds of transformations that are needed to exploit scalable parallelism. Our tests will identify conditional parallelism and parallelism that can be exposed by combinations of transformations that reorder the iteration space (such as loop interchange and loop peeling).

This approach lets us distinguish inherently sequential code from code that contains unexploited parallelism. It also produces information about the kinds of transformations that will be needed to parallelize the code, without worrying about the order of application of the transformations. Furthermore, when our dependence test is inexact, we can identify which unresolved dependences inhibit parallelism by comparing the effects of assuming dependence or independence. We are currently exploring the use of this information in programmer-assisted parallelization.

Categories and Subject Descriptors: D.3.4 [**Software**]: Compilers/Optimization

General terms: Automatic Parallelization, Compilation, Optimization, Array Data Dependence Analysis, Presburger Arithmetic

Additional Key Words and Phrases: Omega test, Dependence Relation

---

1. INTRODUCTION

Array data dependence testing algorithms detect ordering constraints among references to an array. These ordering constraints are used in parallelizing and vectoriz-

---

This work was supported by a Packard Fellowship and by a NSF PYI award, CCR-9157384.

ing compilers, to rule out program transformations that would change the semantics of the program [AK87, Ban88, Wol89]. If the dependence information is inexact, the compiler must act conservatively, rejecting some transformations because they violate a constraint that may or may not be real.

Determining array data dependences is at least as hard as checking for integer solutions to linear constraints, which is an NP-complete problem. Traditionally, conservative array data dependence tests, such as Banerjee's inequalities, have been used in practice. These algorithms never fail to predict a dependence where one exists, but they may also predict dependences that don't exist. When a test reports a dependence that does not actually exist, it is said to report a *false dependence*. The term false dependence is also used to for dependences that exist but can be eliminated through program transformations [MHL91a]. False dependences can prevent useful parallelization.

Recent work [WT92, MHL91b, IJT91, Pug92] has suggested that exact integer programming methods can be made efficient enough for use in production compilers. However, even tests that use exact integer programming methods produce false dependences. False dependences can arise due to non-linear terms, conditional dependences and array kills [MHL91a].

One way to determine the exact data dependences for a program is to instrument the program so that, when run, it produces an exact list of the data dependences that occurred during an execution [MHL91a, Lar93, PP93]. On the basis of these dependences, we can also calculate the critical path of the program for that execution. While this approach is useful, it has several drawbacks:

—the instrumented program may run substantially slower than the original,

—we only get information about dependences exhibited during a particular execution, and

—calculating the critical path of an execution may not give sufficient information about how to exploit the parallelism or how to remove existing bottlenecks.

Even if we have exact information about dependences, the amount of parallelism may not be obvious. If a loop does not carry any dependences, the loop can obviously be run in parallel. However, the presence of loop-carried dependences does not always keep us from running some iterations in parallel. All of the loops in Figure 1 carry dependences, but there is exploitable parallelism in each of them. Advanced optimization techniques may be able to expose the parallelism in some of these cases, but other cases exist that are even more difficult. The KAP preprocessor on the KSR is able to find the parallelism in Example 3 and 5. It also asks the user questions to determine if Example 7 is parallel. If the preprocessor is given permission to reorder reductions, it can find the obvious parallelism in Example 8. However, if it is not allowed to reorder reductions, it is unable to find the non-obvious parallelism in this example. The KAP preprocessor doesn't even suggest to the user that parallelism might exist in Examples 1, 2, 4, or 6.

When automatic techniques cannot expose the parallelism, finding and exploiting it can involve large amounts of programmer effort. For example, substantial parallelism exists in all of the Perfect Club Benchmarks codes [B+89], although existing automatic techniques are able to find only a small portion of it [EHLP91, Eig92, Eig93]. Exposing this parallelism required a careful, manual examination of all the dependences that appeared to prevent parallelism, and the

```
for i := 1 to n do
  a(i) := a(1) + b(i) + a(n)
```

Example 1

```
for i := 1 to n do
  a(i) := a(n-i)
```

Example 2

```
for i := 1 to n do
  for j := 1 to n do
    a(i,j) := (a(i,j-1)+a(i-1,j))/2
```

Example 3

```
for i := 1 to n-1 do
  for j := 2 to n do
    a(i,j) := (a(n-i,n)+a(i,j-1))/2
```

Example 4

```
for i := 1 to n do
  b(i) := a(i)*a(i)
  c(i) := b(i-1)*b(i)
```

Example 5

```
for i := 1 to n do
  s := 0
  for j := 1 to i-1 do
    s := s + a(i,j)*b(j)
  b(i) := b(i) - s
```

Example 6

```
for i := 1 to n do
  a(i) := a(i-p)+b(i)
```

Example 7

```
for i := 1, n do
  for j := 1, i do
    x(j) := x(j)+val(i,j)*v(i)
    x(i) := x(i)+val(i,j)*v(j)
```

Example 8

Fig. 1. Examples of Non-Obvious Parallelism

development and manual application of new program transformations that have not yet been automated. It is therefore helpful to identify those parts of the program that might contain parallelism that standard techniques have not uncovered, so that we can direct our efforts there.

Unfortunately, determining which sections are provably sequential is difficult, and does not follow directly from standard techniques for finding parallelism (much as co-NP-complete problems are harder in practice to solve than NP-complete problems). Standard dependence abstractions (such as distance/direction vectors) cannot be used to prove that a code segment is sequential. With existing methods, the only way to prove that a code segment is sequential is to conduct an exhaustive search of transformation sequences that might transform the program into parallel form. Since the search space can be infinite, this is not feasible.

In addition to getting feedback about sections of code that might be parallel, we also wish to get feedback about the types of additional information or transformations that might be required to exploit the parallelism in these sections.

1.1 Our approach

The Omega test [Pug92] has evolved into a set of routines for manipulating Presburger formulas. Presburger formulas are those expressions that can be built using linear constraints over integer variables and the usual logical connectives and quantifiers [KK67, Coo72]. In [Pug92], we describe how to check a conjunction of constraints for solution and symbolically eliminate existential quantified variables. In [PW92b], we give methods to eliminate redundant constraints from a conjunction and verify formulas of the form $\forall \vec{x}, (\exists \vec{y} \text{ s.t. } P(\vec{x}, \vec{y})) \Rightarrow (\exists \vec{z} \text{ s.t. } Q(\vec{x}, \vec{z}))$, where P and Q are conjunctions of linear constraints. In [PW93], we give methods for simplifying expressions of the form

$$C_0 \wedge \neg(\exists V_1 \text{ s.t. } C_1) \wedge \neg(\exists V_2 \text{ s.t. } C_2) \wedge \ldots$$

where the C_i's are conjunctions of linear constraints and the V_i's are sets of variables. These capabilities allow us to simplify arbitrary Presburger formulas.

In this paper, we apply the techniques from our previous work to perform a two part search for parallelism. In the first part, we construct several sets of constraints that describe, for each statement, which iterations of that statement can be executed concurrently. By constructing constraints that correspond to different assumptions about which dependences might be eliminated through additional analysis, transformations and user assertions, we can determine whether we can expose parallelism by eliminating dependences. In the second step of our search for parallelism, we try to determine how to transform the program to exploit scalable parallelism. We look for conditional parallelism, and try to identify the kinds of iteration reordering transformations that could be used to produce parallel loops.

To build the constraints describing the parallelism among statement iterations, we need accurate data dependence information. In Section 3, we describe techniques for calculating dependence information statically using *dependence relations*, a dependence abstraction that includes complete information about the iterations that participate in the dependence and the conditions on the symbolic constants for which the dependence exists. The dependence relation can be thought of a description of the set containing all pairs of statement iterations that are linked by a dependence. In all cases that do not include complex control flow or non-affine terms, the dependence relation describes the dependence exactly. To analyze this relation in an efficient manner, we use methods described in [Pug92], [PW92b], and [PW93].

When calculating dependence information, we have three options:

—Should we allow reductions (such as a summation) to be done in an arbitrary order or should we enforce the original reduction order? (Section 3.4)

—Should we ignore or respect dependences that arise only from the reuse of memory and not from the flow of values? (Section 3.5)

—Should we compute upper or lower bounds on a dependence that we cannot analyze exactly due to circumstances such as non-linear subscripts? (Section 3.6)

By considering all eight combinations of these options, we can see how additional analysis and/or transformations such as storage-breaking transformations and recognition of reductions can effect the dependences of the program (and thereby the amount of available parallelism).

We use the terms *value-based dependence* and *memory-based dependence* to distinguish dependences that arise from the flow of values from those that arise due to the re-use of storage. Calculating value-based flow dependences is substantially more difficult than calculating memory-based dependences. Many algorithms for value-based dependences for arrays can be applied only to code that does not contain any control flow constructs other than normalized `for` loops, and in which all subscripts and loop bounds are linear. Even within this domain, all previous methods [Bra88, Fea88, GS90, Ros90, Rib90, Fea91, PW92b, Li92, MAL92, May92, MAL93, DGS93] only calculate an upper bound on the value-based flow dependences, although one of these methods [Fea91] fails to give exact bounds only in extremely pathological cases. The method we describe in Section 3.5 is exact within this limited domain, and can be applied to programs outside this domain. The technique proposed by [Mas94] utilizes much of the technology developed in [PW93], and is also exact.

In this paper, we are concerned only with parallelism that grows as the size of the iteration space grows. Thus, we search for opportunities to execute different iterations of a statement concurrently, rather than opportunities to execute different statements concurrently. We do not address the question of finding parallelism that requires run-time synchronization, nor find parallelism that can only be obtained by changing the calculations performed in individual statements. For example, we do not detect the fact that a sequential sorting algorithm can be replaced by a parallel sorting algorithm.

The dependence relations describe constraints on the order in which we may execute the iterations of the statements. There are three basic ways to achieve parallel execution: First, we may be able to eliminate some of these constraints, via further dependence testing, relaxing the order of associative reductions, or performing variable expansion, privatization, or renaming. Second, we may be able to identify conditions on the symbolic constants for which a dependence does not exist. Finally, we may transform the code so that it traverses the iteration space in a different order (via loop interchange, iteration space splitting, or other transformations).

In Section 4, we show how to construct, for each statement in the program, a single relation that includes all dependences between iterations of that statement, taking into account chains of dependences through other statements. We can compare the parallelism allowed by two sets of dependences, to see if one is more restrictive than the other. By identifying reduction operations and using only the lower bound on the value-based flow dependences, we can produce an upper bound on the parallelism between iterations of a statement. We then test this upper bound to see if it is inherently sequential. If it is, we stop and mark the statement as sequential.

For statements that are not provably sequential, we can compare the parallelism available when dependence testing is adjusted for:

—original order vs. arbitrary order reductions, and

—memory-based vs. value-based dependences,

—lower bounds vs. upper bounds for dependences arising from non-linear references

These comparisons provide information about whether privatization, re-ordering of reductions, or further dependence testing will increase parallelism.

In Section 5, we describe a number of tests to subdivide the iteration space to allow parallel execution. These tests search for a way to partition the iterations of the statement so that either the partitions can be run in parallel threads or so that the iterations within each partition can be run in parallel. These tests provide insight into the types of iteration space transformations that are needed to expose parallelism. Our tests are heuristics, and may miss some possible partitioning. However, they will find any partitioning that could be obtained by a combination of loop interchange (including imperfect loop interchange), distribution, alignment or peeling, index set splitting to handle crossing dependences or enable imperfect loop interchange, statement reordering, unimodular loop transformations, and the introduction of run-time dependence tests.

We do not attempt to generate a transformation that exploits the parallelism found. For a single statement in isolation, this code generation would not be difficult. However, it is unclear how to combine the results from analyzing each of several statements into a transformation that simultaneously exploits the parallelism in all of the statements.

In Section 6, we show the results of applying our techniques to the codes in Figure 1 and a larger, more realistic example. In Section 7, we give an evaluation of the performance of our techniques for evaluating value-based dependences. We discuss related work in Section 8, comment on needed future work in Section 9, describe how to obtain source code for our implementation in Section 10, and offer concluding comments in Section 11.

2. CAUSES OF FALSE DEPENDENCES

In a study [MHL91a, May92] of Perfect Club Benchmark programs **QCD**, **MDG**, **ARC2D** and **TRFD**, Dror Maydan et. al. found that a total of 170 loop carried value-based flow dependences were expressed during execution of the programs on the standard input data. They also found that even a test that could solve the "affine memory disambiguation" problem exactly would find over one thousand false loop carried flow dependences. The major reasons for false dependences are (largely based on Maydan's analysis):

Data flow. Although two statements refer to the same memory location, intervening writes always occur between the source and sink of the dependence, and therefore there is no data flow along the dependence. In other words, there is a memory-based dependence, but not a value-based dependence, between the statements. This frequently arises when a work array is reused in each iteration of a loop. Memory-based dependences can often be eliminated via storage-dependence breaking transformations such as renaming, expansion and privatization. We address value-based flow dependences in Section 3.5.

Conditional. The dependence may exist only under certain conditions. In some cases, advanced symbolic analysis could determine that these conditions are impossible. In others, the conditions depend on input data and cannot be predicted at compile-time. It may be possible to eliminate these dependences through the use of run-time dependence tests and/or user interaction and assertions. We address conditional dependencies in Section 3.3.

Non-linear. The dependence may involve subscripts or loop bounds containing non-linear expressions such as `i*c+j`, `p(i)`, or a variable that cannot be expressed as an affine function of the loop indices and constants. It may be possible to eliminate some of these false dependences by using methods such as [McK90, Mas92]. However, these methods generally require user interaction and assertions. We describe methods for computing upper and lower bounds for dependences involving non-linear references in Section 3.6.

Reduction. We normally recognize a flow dependence between two successive updates to an array. If these updates use an associative and commutative operation such as addition, we may wish to allow them to be reordered or even done in parallel. To do so, we recognize the dependence between these two references as a reduction dependence, which does not impose an ordering constraint. In a review of Maydan's experimental data, we found that most of the 170 loop-carried value-based flow dependences could be classified as reduction dependences. We discuss reduction dependences in Section 3.4.

Dependence Abstractions. Traditional abstractions of data dependence, such as dependence difference summaries[1], do not include information about which loop iterations participate in the dependence. Systems using these abstractions must behave as though a dependence exists between all pairs of iterations separated by that difference, even if the dependence exists only on a subset of the iteration space. For example, the dependence differences for Example 2 are $1 \ldots \lfloor \mathtt{n}/2 \rfloor$, but this information does not reflect the fact that all dependences are from the first half of the iteration space to the second half. We address more precise dependence abstractions in Section 3.2.

Reductions and dependence abstractions were not cited as a cause of false dependences in [MHL91a], since that study only examined whether or not dependences occurred, not whether they actually prevented parallelism.

3. DEPENDENCE ANALYSIS

For the methods we describe to apply, we must be able to determine which loops and conditionals control the execution of each statement. This can be done in a straightforward manner for code that uses only structured **if**'s and loops for control flow. Our techniques can be applied to any single procedure written this way. We do not address the issue of inter-procedural analysis and transformation.

We recognize affine induction variables and use this information and forward substitution to try to recognize each subscript and loop bound as a affine function of the loop indices and loop-independent variables. If the expressions in the subscripts, loop bounds, and branching conditions are affine functions of the loop indices and loop-independent variables, and the loop steps are known constants, we can represent the dependence exactly.

The requirement that **if** conditionals be affine functions of the loop indices and loop-independent variables is overly restrictive, and additional work is needed to handle realistic conditionals (see Section 9).

We can handle `min`, `max`, `div` and `mod` operations in subscripts and loop bounds. Some of these, such as a `max(n,m)` as the upper bound of a loop, can only be represented by a disjunction. In such cases, dependence testing is more complicated but still exact.

3.1 Dependence Difference Summaries

With traditional data dependence analysis, each possible dependence is described by a summary of the possible dependence difference between the two references. These summaries give a constant value for the dependence difference when it is constant, and otherwise summarize the possible signs of the dependence difference.

[1]A dependence difference is equivalent to the more traditional "dependence distance" when loops are normalized, as they are in all examples in this paper. The term "dependence distance" is not uniquely defined for unnormalized loops – see [Pug93] for details.

3.2 Dependence Relations

The first step we take to improving the accuracy of our data dependence information is to represent dependences with an abstraction that is more accurate than a dependence difference summary. Dependence difference summaries do not describe which iterations of the statement are involved in the dependence, and do not describe the effect of the values of symbolic constants. We therefore represent dependences with dependence relations [Pug91]. A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that represent the values of the loop indices at the source and sink of the dependence and the values of the symbolic constants (e.g., n in Examples $1-8$). The notation we use in the constraints is adapted from [ZC91]:

$A, B, \ldots$	Refers to a specific array reference in a program
$\mathcal{I}, \mathcal{I}', \mathcal{I}'', \ldots$	An iteration vector that represents a specific set of values of the loop variables for a loop nest.
$[A]$	The set of iteration vectors for which A is executed
$A(\mathcal{I})$	The iteration of reference A when the loop variables have the values specified by $\mathcal{I}$
$A(\mathcal{I}) \stackrel{sub}{=} B(\mathcal{I}')$	The references A and B refer to the same array and the subscripts of $A(\mathcal{I})$ and $B(\mathcal{I}')$ are equal.
$A(\mathcal{I}) \ll B(\mathcal{I}')$	$A(\mathcal{I})$ is executed before $B(\mathcal{I}')$
`Sym`	The set of symbolic constants (e.g., loop-invariant scalar variables)

The dependence relation below describes exactly the iterations and values of symbolic constants for which $A(\mathcal{I})$ and $B(\mathcal{I}')$ refer to the same element of the array, and $A(\mathcal{I})$ is executed before $B(\mathcal{I}')$ (i.e., it describes the memory-based dependence from A to B).

$$\{ \mathcal{I} \to \mathcal{I}' \mid \mathcal{I} \in [A] \wedge \mathcal{I}' \in [B] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \wedge A(\mathcal{I}) \stackrel{sub}{=} B(\mathcal{I}') \}$$

For example, the flow dependence involving the array `b` in Example 6 is described by the direction vector $(+)$, and the dependence relation:

$$\{ [\mathrm{i}] \to [\mathrm{i}',\mathrm{j}'] \mid \overbrace{1 \le \mathrm{i} \le \mathtt{n}}^{\mathcal{I}\in[A]} \wedge \underbrace{1 \le \mathrm{j}' < \mathrm{i}' \le \mathtt{n}}_{\mathcal{I}'\in[B]} \wedge \overbrace{\mathrm{i} < \mathrm{i}'}^{A(\mathcal{I})\ll B(\mathcal{I}')} \wedge \underbrace{\mathrm{i} = \mathrm{j}'}_{A(\mathcal{I})\stackrel{sub}{=}B(\mathcal{I}')} \}$$

This can be simplified to $\{ [\mathrm{i}] \to [\mathrm{i}',\mathrm{j}'] \mid 1 \le \mathrm{i} = \mathrm{j}' < \mathrm{i}' \le \mathtt{n}\}$.

3.3 Conditional Dependences

It may be the case that two array accesses can refer to the same memory location only if certain conditions hold on the symbolic constants. For example, the flow dependence in Example 7 is described by the dependence relation:

$$\{[\mathrm{i}] \to [\mathrm{i}'] \mid 1 \le \mathrm{i} < \mathrm{i}' = \mathrm{i} + \mathtt{p} \le \mathtt{n}\}$$

If `p` is less than 1 or greater than $\mathtt{n} - 1$, the dependence does not exist. This information is accurately captured by the dependence relation.

3.4 Reduction Dependences

Consider the code fragment in Example 8. Traditional data dependence analysis will detect flow, output, and anti-dependences between these two statements. We can easily recognize these statements as update statements. If we are willing to treat machine arithmetic as if it were associative and commutative, we can change the order in which the updates are done. There are even a number of ways in which update operations can be done in parallel (e.g., each processor computes a local sum, and then the local sum's are combined in a short sequential (or even $\log p$) step).

We can either automatically recognize update operations constructed with traditional operators, or let the programmer use special operators or directives to indicate a commutative and associative update. A dependence between two compatible updates is termed a *reduction dependence*, and does not impose an ordering constraint. If we do not recognize reduction dependences, we treat an update as a read followed by a write.

3.5 Value-based Dependences

We can calculate value-based flow, output, and anti-dependences, but in this paper we are not concerned with the latter two. There is a value-based flow dependence between an iteration of a write $A(\mathcal{I})$ and an iteration of a read $C(\mathcal{I}'')$ if and only if $C(\mathcal{I}'')$ reads the value that was written by $A(\mathcal{I})$. For this to occur, $A(\mathcal{I})$ and $C(\mathcal{I}'')$ must access the same element of the array, and that element must not be overwritten between $A(\mathcal{I})$ and $C(\mathcal{I}'')$. If the element is overwritten by $B(\mathcal{I}')$, we say $B(\mathcal{I}')$ *kills* the dependence from $A(\mathcal{I})$ to $C(\mathcal{I}'')$. Let B_1, B_2, ..., B_p be the array writes that might overwrite the element (note that A might be included in the list of B_q's). The value-based flow dependence from A to C is described by the relation:

$$\begin{aligned}\{\ \mathcal{I} \rightarrow \mathcal{I}''\ \mid\ & \mathcal{I} \in [A] \wedge \mathcal{I}'' \in [C] \wedge A(\mathcal{I}) \ll C(\mathcal{I}'') \wedge A(\mathcal{I}) \stackrel{sub}{=} C(\mathcal{I}'') \\ & \wedge\ \forall q, 1 \le q \le p, \neg\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B_q] \wedge A(\mathcal{I}) \ll B_q(\mathcal{I}') \ll C(\mathcal{I}'') \\ & \qquad\qquad \wedge\ B_q(\mathcal{I}') \stackrel{sub}{=} C(\mathcal{I}'')\ \}\end{aligned}$$

The conditions under which B_q kills the dependence imply that there must be memory-based dependences from $A(\mathcal{I})$ to $B_q(\mathcal{I}')$, from $A(\mathcal{I})$ to $C(\mathcal{I}'')$, and from $B_q(\mathcal{I}')$ to $C(\mathcal{I}'')$. We can therefore simplify our relation by leaving out any q for which we can show this is not the case.

For example, consider the flow dependence from the write at line 2 to the read at line 7 in Example 9. The dependence difference summaries that will be used to construct the relation for this dependence are shown to the right of the code.

We build the dependence relation by expanding the outer quantification of $\forall q \ldots$ (since we can determine all possible B_q's statically). The relation will have two "kill" terms (B_q's): the writes at lines 4 and 5 (there is no kill term for the write on line 2 because there is no self output dependence for this write). The unsimplified version of the relation is:

```
1: for i := 1 to 2*n do
2:    a(i) := ...
3:    for j := 1 to n-1 do
4:       a(2*j)   := ...
5:       a(2*j+1) := ...
6:    endfor
7:    ... := a(i)
8: endfor
```

Example 9

```
flow    2: a(i)     --> 7: a(i)     (0)
output  2: a(i)     --> 4: a(2*j)   (0+)
flow    4: a(2*j)   --> 7: a(i)     (0+)
output  2: a(i)     --> 5: a(2*j+1) (0+)
flow    5: a(2*j+1) --> 7: a(i)     (0+)
```

Some dependence difference summaries for Example 9

$$
\begin{aligned}
\{\ [\mathrm{i}] \to [\mathrm{i}''] \mid\ & \overbrace{1 \le \mathrm{i} \le 2\mathtt{n}}^{\mathcal{I}\in[A]} \land \overbrace{1 \le \mathrm{i}'' \le 2\mathtt{n}}^{\mathcal{I}''\in[C]} \land \overbrace{\mathrm{i} = \mathrm{i}''}^{A(\mathcal{I})\ll C(\mathcal{I}'')} \land \overbrace{\mathrm{i} = \mathrm{i}''}^{A(\mathcal{I})\overset{sub}{=}C(\mathcal{I}'')} \\
\land\ & \underbrace{\begin{array}{l}
\neg(\ \exists[\mathrm{i}',\mathrm{j}']\ \text{s.t.}\ (1 \le \mathrm{i}' \le 2\mathtt{n} \land 1 \le \mathrm{j}' \le \mathtt{n}{-}1) \land (\mathrm{i} \le \mathrm{i}' \land \mathrm{i}' \le \mathrm{i}'') \\
\qquad \land\ (2\mathrm{j}' = \mathrm{i}'')\ ) \\
\land\ \neg(\ \exists[\mathrm{i}',\mathrm{j}']\ \text{s.t.}\ (1 \le \mathrm{i}' \le 2\mathtt{n} \land 1 \le \mathrm{j}' \le \mathtt{n}{-}1) \land (\mathrm{i} \le \mathrm{i}' \land \mathrm{i}' \le \mathrm{i}'') \\
\qquad \land (2\mathrm{j}'{+}1 = \mathrm{i}'')\ )\ \}
\end{array}}_{\forall q, 1\le q\le p, \neg\exists \mathcal{I}'\ \text{s.t.}\ \mathcal{I}'\in[B_q]\land A(\mathcal{I})\ll B_q(\mathcal{I}')\ll C(\mathcal{I}'')\land B_q(\mathcal{I}')\overset{sub}{=}C(\mathcal{I}'')}
\end{aligned}
$$

Simplifying such expressions are difficult. Some of the problems are described in Section 3.5.1. By using techniques described in [PW93], we simplify the above expression to:

$$\{\ [\mathrm{i}] \to [\mathrm{i}''] \mid\ (1 = \mathrm{i} = \mathrm{i}'' \le \mathtt{n}) \lor (1 \le \mathrm{i} = \mathrm{i}'' = 2\mathtt{n}) \lor (1 \le \mathrm{i} = \mathrm{i}'' \le 2\mathtt{n} \land \mathtt{n} = 1)\ \}$$

Thus, we have discovered that there is a dependence from the first write of Example 9 to the read only during the first iteration and last iteration (if $\mathtt{n} = 1$, there are only 2 iterations). With our current implementation, this simplification requires 5 milliseconds on a Sun SparcStation 10-51.

The above equation for value-based dependence analysis is best thought of as a denotational specification of what we compute. Using the techniques for value-base dependence analysis described in [PW92a, PW94] as a prepass, followed by the techniques described here, decreases the total cost of value-based dependence analysis by a factor of $0.9 - 1.75$ while not changing what is computed. In [PW93], we describe a number of techniques that decrease the cost of value-based dependence analysis by a factor of $1.1 - 2.9$ and eliminate the need to perform a prepass using the techniques described in [PW92a, PW94].

3.5.1 *Simplifying formulas containing negation.* When performing array kill analysis, we have to simplify formulas of the form:

$$\ldots \lor (C_0 \land \neg(\exists V_1 \text{ s.t. } C_1) \land \ldots \land \neg(\exists V_n \text{ s.t. } C_n)) \lor \ldots$$

Here, the C_i's are conjunctions of linear constraints, and the V_i's are (possibly empty) sets of variables. Techniques described in our previous papers [Pug92, PW92b] allow us to eliminate existentially quantified variables, check for the feasibility of a conjunction of constraints, and perform other simplifications, but these techniques do not address negation. There are two problems involved in simplifying formulas containing negations:

—We must transform a formula into disjunctive normal form in order to verify the existence of solutions. A straightforward transformation of a formula containing negation into disjunctive normal form may lead to a huge explosion in the number of terms.

—If a negated term $\exists V_i$ s.t. C_i represents non-convex constraints (e.g., $\exists\alpha$ s.t. $x = 5\alpha$), we cannot directly evaluate the negation. When standard Fourier-Motzkin variable elimination cannot eliminate an integer variable exactly, we can sometimes eliminate the variable exactly by introducing quasi-linear constraints (constraints containing floor and ceiling operators) [AI91]. For example, $(\exists\alpha \text{ s.t. } x = 5\alpha) \equiv \lceil x/5 \rceil \leq \lfloor x/5 \rfloor$. In these cases, negation is easy to apply (e.g., $\neg(\lceil x/5 \rceil \leq \lfloor x/5 \rfloor) \equiv (\lceil x/5 \rceil > \lfloor x/5 \rfloor)$).
Although we can always eliminate one variable this way, we may not be able to eliminate multiple variables this way, since we may not be able to apply Fourier-Motzkin variable elimination to a set of constraints containing floor and ceiling operators.

In [PW93], we give a partial (but effective) solution to the first problem and a complete and exact solution to the second.

3.6 Non-linear Dependences

If a structured routine does not meet the conditions under which we can produce an exact dependence relation, we can create relations that give lower and upper bounds on the dependence. We do so by using linear approximations for the nonlinear conditions. If a nonlinear constraint appears positively, replacing it with **False** gives a lower bound and replacing it with **True** gives an upper bound. However, we would like to obtain tighter bounds than that. We describe methods for obtaining tighter lower bounds for non-linear equality constraints.

Given a non-linear equality constant, we attempt to transform it into something of the form $\sum_p f_p(a_p) = \sum_p f_p(b_p)$, where the a_p's and b_p's are affine expressions. In Example 10, we can transform $\mathrm{i} * \mathrm{c} + \mathrm{j} = (\mathrm{i}' - 1) * \mathrm{c} + \mathrm{j}'$ into $\{f_1(x) = x, f_2(x) = \mathrm{c} * x, a_1 = \mathrm{j}, a_2 = \mathrm{i}, b_1 = \mathrm{j}', b_2 = \mathrm{i}' - 1\}$ and $\{f_1(x) = x, f_2(x) = \mathrm{c} * x, a_1 = \mathrm{j}, a_2 = \mathrm{i}, b_1 = \mathrm{j}' - \mathrm{c}, b_2 = \mathrm{i}'\}$. For each such transformation found, let L be $\forall p, a_p = b_p$. Since the f_p's are functions, we know that $L \Rightarrow \sum_p f_p(a_p) = \sum_p f_p(b_p)$.

For a non-linear equality constraint E, let $L_1, L_2, \ldots L_q$ be the constraints arising from the transformations found for E (typically q will be 1 or 2). If E appears positively in a dependence relation, we can replace E with $L_1 \vee L_2 \vee \cdots \vee L_q$ and obtain a lower bound on the dependence relation. For the memory-based flow dependence in Example 10, this leads to the a lower bound of:

$$\{[\mathrm{i},\mathrm{j},\mathrm{k}] \to [\mathrm{i}',\mathrm{j}',\mathrm{k}'] \mid \mathrm{i}' = \mathrm{i} + 1 \wedge 1 \leq \mathrm{i} < \mathtt{n} \wedge 1 \leq \mathrm{j} = \mathrm{j}' \leq \mathtt{m} \wedge 1 \leq \mathrm{k} = \mathrm{k}' \leq 10\}$$

$$\cup \{[\mathrm{i},\mathrm{j},\mathrm{k}] \to [\mathrm{i}',\mathrm{j}',\mathrm{k}'] \mid \mathrm{j}' = \mathrm{j} + \mathrm{c} \wedge 1 \leq \mathrm{i} = \mathrm{i}' \leq \mathtt{n} \wedge 1 \leq \mathrm{j} < \mathrm{j}' \leq \mathtt{m} \wedge 1 \leq \mathrm{k} = \mathrm{k}' \leq 10\}$$

```
for i := 1 to n do
  for j := 1 to m do
    for k := 1 to 10 do
      ... := a((i-1)*c+j, k)
      a(i*c+j, k) := ...
```

Example 10

```
for i := 1 to n do
  for j := 1 to m do
    a(p(j)) := ...
    ... := a(p(j))
```

Example 11

To build the upper bound for this dependence, we replace the nonlinear constraint with **True**, yielding:

$$\{[\mathrm{i},\mathrm{j},\mathrm{k}] \rightarrow [\mathrm{i}',\mathrm{j}',\mathrm{k}'] \mid 1 \leq \mathrm{i} \leq \mathrm{i}' \leq \mathtt{n} \wedge 1 \leq \mathrm{j},\mathrm{j}' \leq \mathtt{m} \wedge 1 \leq \mathrm{k} = \mathrm{k}' \leq 10\}$$

$$\cup \ \{[\mathrm{i},\mathrm{j},\mathrm{k}] \rightarrow [\mathrm{i}',\mathrm{j}',\mathrm{k}'] \mid 1 \leq \mathrm{i} = \mathrm{i}' \leq \mathtt{n} \wedge 1 \leq \mathrm{j} < \mathrm{j}' \leq \mathtt{m} \wedge 1 \leq \mathrm{k} = \mathrm{k}' \leq 10\}$$

When E appears negatively, as it might when computing value-based flow dependences, we substitute $L_1 \vee L_2 \vee \cdots \vee L_q$ to obtain an upper bound on the dependence relation. Thus, this technique can be used to improve the accuracy of upper bounds on value based dependences, as well as the accuracy of lower bounds on memory based dependences. In Example 11, this allows us to determine that there is no loop-carried value-based flow dependence between the two array references.

3.7 Interactions

If a program contains only linear terms and we do not recognize reduction dependences, we can eliminate all dependences other than value-base flow dependences by a sequence of storage-dependence breaking transformations (e.g., expansion, privatization, and renaming) [Fea88, Fea91]. That is, we can completely separate value flow and memory usage issues for these programs.

However, parallel reductions and/or non-linear terms may prevent us eliminating all storage-based dependences.

3.7.1 *Non-linear terms.* If a program contains non-linear terms, the upper bound on the value-based flow dependences is not always sufficient to ensure that the original semantics can be preserved. Specifically, when we cannot determine statically which of several writes provides a value that is read, we must still rely on the overwriting of memory to ensure that the correct value is received by the read. To do this, we ensure that any pair of writes that could both supply a value to a single read are executed in their original order, and we do not allow any storage-dependence breaking transformations that would prevent them from overwriting each other.

More formally, we find the we must respect an memory-based output dependence from $A(\mathcal{I})$ to $B(\mathcal{I}')$ (where A and B may be the same statement) if there exists a read $C(\mathcal{I}'')$ such that the value read in $C(\mathcal{I}'')$ may be provided by both $A(\mathcal{I})$ and $B(\mathcal{I}')$. This is only possible if our dependence information is an inexact upper bound on the value-based flow dependences.

We must respect a memory-based anti-dependence from $B(\mathcal{I}')$ to $C(\mathcal{I}'')$ if there exists a write $A(\mathcal{I})$ such that there is a value-based flow dependence from $A(\mathcal{I})$ to $B(\mathcal{I}')$ and we must respect the output dependence from $A(\mathcal{I})$ to $C(\mathcal{I}'')$.

For example, consider the code in Example 12. Even if we were to perform all possible privatization, expansion, and renaming, we would still be required to execute all of the iterations of S1 sequentially, allowing them to potentially overwrite each other, before allowing any iterations of S3 to execute.

```
for i := 1 to n do
  a(p(i)) := ...          /* S1 */
  for j := 1 to n do
    b(j) := ...           /* S2 */
    c(j) := a(j) + b(p(j)) /* S3 */
  endfor
endfor
```

Example 12: Nonlinear terms affecting value-based dependence analysis

The dependences in Example 12 are:

From	To	Type	Dependence Relation
S_1	S_3	Value – Flow	$\{[i] \rightarrow [i', j'] \mid 1 \le i \le i' \le n \wedge 1 \le j' \le n\}$
S_1	S_1	Memory – Output	$\{[i] \rightarrow [i'] \mid 1 \le i < i' \le n\}$
S_2	S_1	Memory – Anti	$\{[i, j] \rightarrow [i'] \mid 1 \le i < i' \le n \wedge 1 \le j \le n\}$
S_2	S_3	Value – Flow	$\{[i, j] \rightarrow [i, j'] \mid 1 \le i \le n \wedge 1 \le j \le j' \le n\}$
			$\cup\{[i, j] \rightarrow [i+1, j'] \mid 1 \le i < n \wedge 1 \le j', j \le n\}$
S_2	S_2	Memory – Output	$\{[i, j] \rightarrow [i', j] \mid 1 \le i < i' \le n \wedge 1 \le j \le n\}$
S_3	S_2	Memory – Anti	$\{[i, j] \rightarrow [i, j'] \mid 1 \le i \le n \wedge 1 \le j < j' \le n\}$
			$\cup\{[i, j] \rightarrow [i', j'] \mid 1 \le i < i' \le n \wedge 1 \le j', j \le n\}$

Using the calculations above, we see that we will need to enforce all of the output and anti dependences involving statement S_1, but none of the output and anti dependences involving S_2. In this case, we need to respect *all* dependences in the memory-based dependence relations, but in general, we may only need to respect a subset of the dependence pairs.

3.7.2 *Reductions.* We may also need to enforce some anti-dependences when we perform value-based analysis in the presence of reorderable reductions. Consider a value-based reduction dependence from $A(\mathcal{I})$ to $C(\mathcal{I}'')$ for which there is also a value-based flow dependence from $A(\mathcal{I})$ to $B(\mathcal{I}')$ and a value-based anti-dependence from $B(\mathcal{I}')$ to $C(\mathcal{I}'')$. Since $A(\mathcal{I})$ and $C(\mathcal{I}'')$ must update the same memory location, we cannot eliminate the anti-dependence. If we wish to find the set of dependences that cannot be eliminated with expansion and renaming, we must include the value-based flow dependences, the dependences listed in Section 3.7.1, and these anti-dependences.

This is similar to the situation for non-linear dependences, so we do not give a detailed example here.

3.8 Full Analysis of Only Some Dependences

Our methods for finding parallelism examine dependences from one iteration of a statement to another iteration of the same statement. We therefore consider cycles in the statement graph, where the statements are nodes and each feasible dependence relation is treated as a directed edge. Dependences that are not part of a cycle can be ignored, and we need not use expensive techniques to analyze them.

Given our initial, upper bound on dependences (calculated using standard techniques such as [Pug92]), we calculate the strongly connected components (SCC's) of the statement graph. A dependence is in a cycle if and only if its endpoints are in the same SCC.

When determining which flow dependences to perform value-based analysis on, we can base the SCC on the flow dependences only. If we use a method such as [PW92b] that eliminates some dependences that are not value-based, we can base the SCC on only the flow dependences that might be value-based. In the case where non-linear terms may prevent us from calculating value-based flow dependences exactly, we may also need to consider some output and anti dependences (see Section 3.7).

4. COMPUTING SELF-DEPENDENCE INFORMATION

The dependence relation between array accesses A and B gives conditions under which $A(\mathcal{I})$ must precede $B(\mathcal{I}')$. Conversely, if $A(\mathcal{I})$ and $B(\mathcal{I}')$ are not connected in the transitive closure of the graph of a set of control and data dependence relations, those dependences do not prevent us from executing them concurrently. Thus, dependence relations provide exact information about which iterations of which statements can be executed concurrently. In this paper, we only consider parallelism that grows as the size of the iteration space grows. Thus, we will look only for parallelism between the iterations of each statement.

We start by calculating the transitive self-dependence relation, or $TSDR$, for each statement. This relation will contain all the ordering constraints between iterations of the statement, including constraints arising from transitive dependences through other statements. A TSDR D for a statement describes the parallelism among the iterations of the statement, as it gives the conditions under which any pair of iterations $\mathcal{I}$ and $\mathcal{I}'$ can be executed concurrently: $([\mathcal{I}] \rightarrow [\mathcal{I}']) \notin D$.

4.1 Computing the Self-Dependences of a Statement

In previous work, we described methods for computing the transitive self-dependences of a statement [Pug91]. Unfortunately, no complete method is possible for computing an exact, closed form for the transitive closure of an arbitrary dependence relation (it is equivalent to adding multiplication to Presburger, which makes the theory undecidable). As described in [Pug91], it is possible to complete an exact transitive closure for dependences in certain special forms. For example, we can compute the exact transitive closure of any dependence relation that can be represented exactly by a direction/distance vector.

In [Pug91], we were only concerned with computing a lower bound on the transitive closure. Here, we discuss how to compute both an upper and lower bound on the transitive closure. To compute D^+ (the transitive closure of D), we find relations D^L and D^U such that $D^L \subseteq D \subseteq D^U$ and D^L and D^U are in special forms that can be closed exactly. We use these to obtain lower and upper bounds on D^+:

$$(D^L)^+ \subseteq D^+ \subseteq (D^U)^+$$

If these lower and upper bounds are identical, we know we have computed D^+ exactly. In other cases, we treat our bounds on the transitive closure as if they had arisen from a nonlinear term during dependence analysis.

In the abstract, computing a transitive closure appears to be very hard and/or expensive in the worst-case. When we analyze even large programs, we find that if we allow reductions to be reordered and consider only value-based flow dependences, the strongly connected components are typically very small, and computing

```
a(1) = b(1)
for i := 2 to n do
  a(i) := a(i-1) + b(i)
```

Example 13: Sequential Code

the transitive closure is easy and fast (less than 0.5 seconds on a Sun Sparc 10-51). However, when we analyze large programs using memory-based dependences and original order for reductions, we find that strongly connected components containing 20-30 statements are not unusual, and SCC's of over 100 statements can arise in practice. In these cases, many of the dependences derive from scalar variables and are typically "to all future iterations". Our current implementation does not utilize fast, special case methods to handle these cases. As we gain more experience with analyzing large programs, we expect to refine our techniques to allow us to handle these cases efficiently.

4.2 Comparing Transitive Self-Dependence Relations

In Section 3, we discussed several choices that can be made during the calculation of a single dependence relation: we can use upper or lower bounds for nonlinear constraints, we can calculate value or memory-based dependence relations, and we may wish to recognize reduction dependences. We can use the different kinds of individual data dependence relations to calculate several different TSDRs for a given statement. We can define a partial ordering on TSDRs, based on the sets of ordering constraints they describe:

$$D' \subseteq D \text{ iff } \forall \mathtt{Sym}, \mathcal{I}, \mathcal{I}', (\mathcal{I} \rightarrow \mathcal{I}' \in D') \Rightarrow (\mathcal{I} \rightarrow \mathcal{I}' \in D)$$

Note that, if $D' \subseteq D$, and D allows two iterations of the statement to be executed in parallel, D' also allows them to be executed in parallel.

We have some a priori knowledge of the ordering of different TSDRs for a statement: We will not add to the set of ordering constraints by (a) using value-based flow dependences rather than all memory-based dependences, (b) recognizing reductions, or (c) using a lower bound rather than an upper bound on an individual dependence.

For any statement S, we can calculate D_S^α, a TSDR based on the upper bound on memory-based flow, output, and anti-dependences we compute without recognizing reductions, and D_S^ω, a TSDR based on our lower bound on the value-based flow dependences we find after recognizing reductions.

We say that D_S^α is a lower bound on the parallelism of the iterations of statement S, since it gives conditions under which we must be able to execute a pair of iterations concurrently: if $([\mathcal{I}] \rightarrow [\mathcal{I}']) \notin D_S^\alpha$, no dependence prevents us from executing $\mathcal{I}$ and $\mathcal{I}'$ concurrently. Note that it may still be quite difficult to transform the program into a form that makes use of this parallelism; we will address this point in Section 5.

If $([\mathcal{I}] \rightarrow [\mathcal{I}']) \in D_S^\omega$, we cannot reorder $\mathcal{I}$ and $\mathcal{I}'$ without disrupting the flow of information in the program. Thus, we say that D_S^ω gives an upper bound on the parallelism of the iterations of S. We can use D_S^ω to prove that the iterations of S must execute sequentially, by checking:

$$\neg\exists \mathcal{I}, \mathcal{I}', \texttt{Sym} \text{ s.t. } \mathcal{I}, \mathcal{I}' \in [S] \wedge S[\mathcal{I}] \ll S[\mathcal{I}'] \wedge (\mathcal{I} \rightarrow \mathcal{I}') \notin D_S^{\omega}$$

If this is true, as it is for Example 13, and for the third assignment in Example 6, then there are no values for the symbolic constants such that there exist two statement iterations that can be run in parallel. In such cases, we classify the statement as "inherently sequential" and stop analyzing it.

4.3 Parallelism through Dependence Breaking

If a statement is not inherently sequential, we test to see whether we can increase parallelism between its iterations by breaking dependences with variable privatization, reduction re-ordering, or gathering additional information about nonlinear dependences. If $D_S^{\omega} = D_S^{\alpha}$, we know the aforementioned changes do not effect the transitive dependences, and are therefore not needed to expose the parallelism. Otherwise, we can calculate six other TSDRs for S, using various combinations of value or memory-based dependence testing, recognizing or not recognizing reductions, or finding upper or lower bounds on the basic dependence relations. We can then determine which dependence-breaking transformations have an effect on the transitive dependences.

It would be possible to attempt to produce a scalar quantification the amount of parallelism we find (for example, by finding the length of a critical path through the dependence relation). However, we do not address that issue in this paper.

5. IDENTIFYING USEFUL REORDERING TRANSFORMATIONS

A TSDR contains information about which iterations of a statement can be executed in parallel. To make use of the parallelism, we must identify conditions under which large groups of iterations can be run in parallel. We therefore attempt to partition the iterations of the statement such that

—the partitions can be executed in parallel, or

—the iterations within each partition can be executed in parallel.

The nature of the partitions often provides information about the kinds of reordering transformations needed to exploit the parallelism.

In this paper, we only consider partitions that provide parallelism that grows as the size of the iteration space grows. For example, if all dependence differences in a loop are multiples of 2, we could partition the iterations of the statement into the even and odd iterations (which can be executed in parallel). We would not consider this partitioning since it gives only a factor of 2 parallelism. Additional techniques to search for such parallelism could be added to our scheme.

All of our tests have the property that the amount of parallelism they detect is a nondecreasing function of the self-dependences of the statement being analyzed. We can therefore use them to investigate the amount of scalable parallelism that is exposed by the different dependence breaking techniques of Section 4.2. If $D'_S \subset D_S$, but the tests in this section give the same results for D'_S and D_S, the additional parallelism in D'_S either does not grow with the iteration space or is extremely difficult to exploit.

We analyze the parallelism in a statement S by applying the following steps to each unique TSDR for S that contains parallelism:

(1) First, we compute the dependence differences represented by the TSDR, and see if unimodular loop transformation methods can expose parallelism (Section 5.1).
(2) Next, we try peeling the iteration space to simplify the dependences (Section 5.2).
(3) We then search for parallelism in the core iterations of the peeled statement using unimodular loop transformation methods (Section 5.1).
(4) Finally, for each possible loop permutation, we can check for conditional parallelism (Section 5.3). If desired, directional peeling can be applied (Section 5.3.1).

5.1 Using Unimodular Loop Transformation Techniques

We can hand the self-dependences off to a module that looks for a unimodular loop transformation that will expose the most parallelism [Ban90, WL90, KKB92]. This module can report whether the self-dependences allow for coarse-grain parallelism, and if not, whether the self-dependences allow for fine-grain parallelism.

Standard unimodular techniques will find any parallelism that can be exposed using loop interchange, loop skewing and loop reversal. Since we consider each statement separately (using transitive self-dependences), we will also find parallelism that can be only exposed using loop distribution, statement reordering, and loop alignment [Pug91]. This check will find the parallelism when applied to D^α for the statement in Example 3. It will also find parallelism when applied to the transitive closure of the value-based flow dependences for the statements that assign to `s` in Example 6.

5.2 Dependence Peeling

Unimodular loop transformation techniques do not include transformations that split or peel the iteration space. We can determine that such transformations are necessary by examining the effects of removing, from a TSDR D_S, the iterations that do not have both incoming and outgoing dependences. This can be calculated as

$$\hat{D_S} = \{\ \mathcal{I}' \rightarrow \mathcal{I}'' \mid (\mathcal{I} \rightarrow \mathcal{I}') \in D_S \wedge (\mathcal{I}' \rightarrow \mathcal{I}'') \in D_S \wedge (\mathcal{I}'' \rightarrow \mathcal{I}''') \in D_S\ \}$$

The iterations that have no incoming dependences can be executed in parallel before the "core" iterations, and those without any outgoing dependences can be executed after the core iterations. If the number of core iterations is small (or zero), this may be sufficient. Also, it may be easier to exploit the dependences among the core iterations than the dependences across the entire set of iterations (i.e., it may be easier to find parallelism in $\hat{D_S}$ than D_S).

The dependence relation for Example 1 is:

$$\{ [\mathrm{i}] \rightarrow [\mathrm{i}'] \mid (\mathrm{i} = 1 \wedge 2 \leq \mathrm{i}' \leq \mathtt{n})\ \vee\ (1 \leq \mathrm{i} < \mathtt{n} \wedge \mathrm{i}' = \mathtt{n})\ \}$$

The peeled version of this dependence relation is empty, so we know that parallelism can be exposed in this example by loop peeling or splitting.

The dependence relation for Example 2 is:

$$\{[\mathrm{i}] \rightarrow [\mathrm{i}'] \mid 1 \leq \mathrm{i} < \mathrm{i}' \leq \mathtt{n} \wedge \mathrm{i}' = \mathtt{n} - \mathrm{i}\}$$

Once again, the peeled version of this dependence relation is empty, so we know that parallelism can be exposed in this example by loop peeling or splitting (splitting in this example).

We could peel the core iterations again, although we do not believe this is likely to be useful.

5.3 Directional/Conditional Parallelism

The conditions under which a dependence D_S exists are:

$$\exists \mathcal{I}, \mathcal{I}' \text{ s.t. } \mathcal{I} \rightarrow \mathcal{I}' \in D_S$$

Unfortunately, any self-dependence of a statement in a loop with symbolic bounds is conditional, since the statement will not execute (and the dependence not exist) if the bounds are empty. Asking the user about all such conditional dependences is not likely to be useful.

We could determine the conditions under which a dependence exists despite the presence of multiple iterations of each enclosing loop. Given a vector $\Delta\mathcal{I}$ of the number of required iterations of each enclosing loop, we find conditions under which

$$(\exists \mathcal{I}, \mathcal{I}' \text{ s.t. } \mathcal{I}, \mathcal{I}' \in [S] \wedge \mathcal{I} + \Delta\mathcal{I} \leq \mathcal{I}') \Rightarrow (\exists \mathcal{I}, \mathcal{I}' \text{ s.t. } \mathcal{I} \rightarrow \mathcal{I}' \in D_S)$$

While this test avoids the "false alarm" for loops with symbolic bounds (such as Example 13), it shares one other problem with the formulation above: it only finds conditions under which we can completely eliminate all dependences in D_S. We also need to find conditions that allow one or more parallel loops despite the existence of some dependences.

Therefore, we look for conditional parallelism in one direction at a time. Let d be a vector indicating the direction we are testing for parallelism (i.e., let d be $(0,1)$ to search for parallelism in the inner of two loops). This would correspond to a loop over the possible values of $d^\top\mathcal{I}$. Values of d such as $(1,1)$ correspond to skewed loops. We calculate D_S^d, the dependences that would be carried by an outer loop over $d^\top\mathcal{I}$, as:

$$D_S \cap \{\ \mathcal{I} \rightarrow \mathcal{I}' \mid d^\top\mathcal{I} < d^\top\mathcal{I}'\ \}.$$

Select W to be the minimum number of iterations a loop must have in order for running it in parallel to be profitable. The conditions under which the loop would carry dependences whenever it contains enough iterations to be profitable to run in parallel are:

$$(\exists \mathcal{I}, \mathcal{I}' \text{ s.t. } \mathcal{I}, \mathcal{I}' \in [S] \wedge \mathcal{I} + Wd \leq \mathcal{I}') \Rightarrow (\exists \mathcal{I}, \mathcal{I}' \text{ s.t. } \mathcal{I} \rightarrow \mathcal{I}' \in D_S^d)$$

If we can verify that these conditions are false (by user assertion or a run-time check), then we can execute the iterations in a parallel loop over $d^\top\mathcal{I}$ whenever it is profitable to do so.

5.3.1 *Directional Peeling.* For statements with multidimensional iteration spaces and complex self-dependences, it may be useful to perform directional peeling. To perform directional peeling in direction d, we separate the statement iterations into initial, core and final iterations by applying the techniques of Section 5.2 to D_S^d. The initial and finial sections can be executed by a parallel loop over $d^\top \mathcal{I}$.

This method appears to be of limited utility. The only example we have found where it would be useful is Example 4, and variations on it. For Example 4, we find the i loop can be divided into initial ($1 \leq \mathrm{i} \leq \lfloor(\mathtt{n}-1)/2\rfloor$), core ($\mathrm{i} = (\mathtt{n}-1)/2$, if i is odd), and final ($\lceil(\mathtt{n}-1)/2\rceil \leq \mathrm{i} \leq \mathtt{n}-1$) sections.

5.3.2 *Buried Directional Parallelism.* In searching for directional parallelism, we are simply checking if d can be used as an outer parallel loop. If an outer sequential loop was placed around the statement, the outer loop might carry enough dependences so that parallelism can be found in inner loops. Given a sequential outer loop, we can easily calculate which self-dependences are not carried by the outer loop.

There are two problems with searching for directional parallelism:

—We do not know what d's to use. We would not be able to consider all possible combinations of loops that include loop skewing (e.g., using $d = (1, -1)$ or $d = (3, 2)$).

—If we wish to search for inner loop parallelism, we not only have to consider all possible parallel loops, we have to consider all sequential loops that might surround it.

These problems pose limitations on the usefulness of this technique. However, this technique should be usable for considering all permutations of loops and searching for both inner and outer parallelism (there are only 6 permutations to try if the statement is nested 3 deep).

5.4 Transforming Scalar Calculations

Although our techniques do not, in general, detect cases in which code can be parallelized by performing a different set of operations, we can test to see if more advanced scalar induction variable detection might expose parallelism. Strongly connected components of the statement graph that consist entirely of assignments to scalar variables often reflect either a reduction or an induction variable and further scalar analysis may suggest a way of computing the values in parallel.

5.5 Additional Tests

The list above is not a complete list of tests that might be applied. The only cases we know of not checked by the above tests are when the GCD of the dependence distances is greater than 1 (e.g., if all dependence distances are even, we can execute the odd and even iterations in separate, parallel threads) and when all dependence distances are large (e.g., if all dependence distances are greater than 20, we can strip mine the loop by a factor of 20 and run the inner loop in parallel). Checks for these cases could be easily added to the above tests; we have omitted them since they rarely offer a significant amount of parallelism.

6. EXAMPLES

Figure 1 gave eight examples which contain parallelism that would be overlooked by a simple test for loops that do not carry any dependences. In this section we show that our techniques expose the parallelism in all of these examples. Unless otherwise noted, we found the parallelism in D^α (i.e., the upper bound on the memory-based dependences, without recognizing reductions).

In Examples 1 and 2, we find that the dependence peeling test from Section 5.2 splits each statement's iterations into two groups, each of which can be executed in parallel. The parallelism in Example 3 can be found with unimodular loop transformation techniques (Section 5.1).

Example 4 was contrived to foil our initial set of parallelism tests, and it does so: despite the fact that the iterations of this single statement are not provably sequential, the tests in Sections 5.1 and 5.2 fail to detect any parallelism, even when applied to D^ω. The directional peeling test of Section 5.3.1 does, however, indicate that we can expose parallelism by peeling iterations of the `i` loop.

The two statements in Example 5 can be trivially parallelized after the loop is distributed. Neither statement in this example is involved in a dependence cycle, so the parallelism is evident at the first step of our procedure. No additional dependence testing is done, and none of the parallelism tests are performed.

Example 6 demonstrates the importance of applying the tests to the different sets of dependences. When we apply the tests to D^α, we do not find parallelism. When we recognize reduction dependences and omit them from the dependences passed to the parallelism tests, we find that the statement updating `s` can be run in parallel (the test from Section 5.1 or a simple test of which loops carry dependences will detect this parallelism). Parallel reductions would be required to exploit this. If we instead use value-based flow dependences, the test from Section 5.1 will find that the iterations of the statement that initializes `s`, and those of the statement that update `s`, can be run in parallel. Scalar expansion, loop distribution, and imperfect loop interchange would be required to exploit this parallelism. Since we have not made any attempt to quantify the amount of parallelism, we cannot automatically choose between these alternatives, and present both possibilities to the user. Note that this would be necessary even if we did have a measure of the amount of parallelism, as the choice should also depend on other factors, such as the cost of allocating memory to expand the scalar `s`.

Assume we instruct the test for directional parallelism that a loop must have 10 iterations in order for it to be worthwhile to run the loop in parallel. In Example 7, the test for directional/conditional parallelism finds that the loop can be run in parallel if $\mathtt{n} \geq 10 \Rightarrow (\mathtt{p} \leq -\mathtt{n} \vee \mathtt{p} = 0 \vee \mathtt{n} \leq \mathtt{p})$. If we use only value-based flow dependences, we find that the loop can be run in parallel if $n \geq 10 \Rightarrow (\mathtt{p} \leq 0 \vee \mathtt{n} \leq \mathtt{p})$. The $\mathtt{n} \geq 10 \Rightarrow \cdots$ part of the formula is not particularly useful here; its main purpose is to prevent false alarms. It would also be possible to partition this code into `p` different loops, which could be executed in parallel threads, giving `p`-fold parallelism. Our current set of parallelism tests do not catch this.

For Example 8, if we recognize reduction dependences, then there are no other dependences in this code and both loops can be run in parallel. Even if we do not recognize reduction dependences, parallelism can be found in both statements by the unimodular parallelism test. This seems surprising, as it is difficult to see how this parallelism can be achieved, but Figure 2 shows code that achieves it. This transformation can be identified and generated automatically by the system described in [KP93].

```
forall i := 1 to n do
  for j := 1 to i-1 do
    x(i) := x(i)+val(i,j)*v(j) /* s2 */
  endfor
  x(i) := x(i)+val(i,i)*v(i)   /* s1 */
  x(i) := x(i)+val(i,i)*v(i)   /* s2 */
  for j := i+1 to n do
    x(i) := x(i)+val(j,i)*v(j) /* s1 */
  endfor
endfor
```

Fig. 2. A transformed version of Example 8

```
Can you verify that
  * there is no loop-carried value-based flow dependence
    from 37:RL(K+4) to 43:RL(K-5)?

If so, it is possible to parallelize all of the statements in the
1:do I and 2:do J loops. The following steps must be performed:
  * Privatize/Expand scalars: KC, ...
  * Privatize/Expand arrays: XL, RS, RL
  * Perform parallel reductions on scalar: VIR
  * Perform parallel reductions on array: FX
```

Fig. 3. Example dialog generated for INTERF extract

6.1 Extended Example: INTERF

We now describe an application of our techniques to the routine `INTERF` from the Perfect Club benchmark `MDG`. This routine contains substantial parallelism that is not exploited by current compilers [EHLP91]. Figure 4 shows a condensed version of this code after structuring by VAST-90 (from Pacific-Sierra Research) and automatic induction variable recognition. We have omitted statements that are easily parallelizable or equivalent to other statements that were left in. As our techniques do not address inter-procedural analysis, we have performed two inter-procedural optimizations by hand: First, we performed inline substitution of a call to the subroutine `CSHIFT`, which defines `XL(1:14)` (lines 3-16 of Figure 4). Second, we changed the variable `NATOMS` to the constant `3` (`NATOMS` is always `3`, but this is only evident with inter-procedural analysis).

The techniques described in this paper can be used to produce a dialog such as the one in Figure 3. This dialogue immediately focuses attention on the one dependence that requires analysis by the programmer, and summarizes the transformations that must be applied to exploit the parallelism if this dependence is indeed false. The computations that go into this dialog are described here:

The lower bound and upper bound on value-based flow dependences for `XL` and `RS` are identical: none are loop-carried. For the array `RL`, the upper bound gives a loop carried value-based flow dependence from `37:RL(K+4)` to `43:RL(K-5)`. The non-linear terms make it impossible to determine which `(I, J)` iteration of the write provides the value, and thus we must execute the writes in their original order, and preserve the relative ordering of these writes and reads (as per Section 3.7). These constraints force completely sequentially execution of the `(I, J)` iterations of lines

```
SUBROUTINE INTERF(X, Y, Z, FX, FY, FZ, XM, YM, ZM, VIR)
DO I = 1, NMOL1
   DO J = I + 1, NMOL
      XL(1) = ...
      XL(2) = ...
      XL(3) = ...
        ...
      XL(14) = ...
      DO CI = 1,14
        IF ( D_ABS(XL(CI))>BOXH ) THEN
          XL(CI) = XL(CI)-D_SIGN(BOXL,XL(CI))
        ENDIF
      END DO

      KC = 0
      DO K = 1, 9
         RS(K) = XL(K)*XL(K) + ...
         IF (RS(K) .GT. CUT2) KC = KC + 1
      END DO
      IF (KC .NE. 9) THEN
         IF (RS(1) .LT. CUT2) THEN
            VIR = VIR + RS(1)*...
         ENDIF
         DO K = 2, 5
            IF (RS(K) .LT. CUT2) THEN
               VIR = VIR + RS(K)*...
            ENDIF
            IF (RS(K+4) .LE. CUT2) THEN
               RL(K+4) = SQRT(RS(K+4))
               VIR = VIR + RS(K+4)*...
            ENDIF
         END DO
         IF (KC .EQ. 0) THEN
            DO K = 11, 14
               FTEMP = AB2*EXP((-B2*RL(K-5)))/RL(K-5)
               VIR = VIR + FTEMP*RS(K-5)
               RS(K) = XL(K)*XL(K) + ...
               RL(K) = SQRT(RS(K))
               VIR = VIR + RS(K)*...
            END DO
         ENDIF

         FX(3*I-1) = FX(3*I-1) + ...
         FX(3*J-1) = FX(3*J-1) - ...
         FX(3*I-2) = FX(3*I-2) + ...
         FX(3*I)   = FX(3*I)   + ...
         FX(3*J-2) = FX(3*J-2) - ...
         FX(3*J)   = FX(3*J)   - ...
      ENDIF
   END DO
END DO
```

Fig. 4. Excerpts from SUBROUTINE INTERF of MDG benchmark

		Sun SPARCstation 10-51 Execution time (millisecs)			
			Dependence Analysis		
			Memory-based	Value-based	
From	Code	f77 -c -O2	All dependences		In Cycles
[Fea91]	across	300	1.5	2.7	2.1
	burg	400	5.5	31	26
	relax	200	1.6	8.7	5.2
	gosser	400	2.3	23	14
	choles	300	2.7	11	9.3
	lanczos	1000	11	41	35
	jacobi	900	150	300	250
[MAL93]	extract from ocean	200	4.5	6.8	5.1
Perfect	TFRD: olda; simplified	1700	26	320	180
NASA	btrix	4100	170	310	240
NAS	cfft2d1	800	16	150	130
Kernels	cholsky	1400	31	73	40
	emit	1500	17	79	38
	gmtry	1900	13	58	38
	vpenta	1800	96	98	64

Table I. Timed Programs

`37` and `43`. However, the lower bound on the value-based flow dependences allows parallel execution of both the `I` and `J` loops. Therefore, we need to ask the user to verify the lower bound. Determining that this lower bound is accurate is rather subtle, and we do not expect automatic techniques to be able to verify this in the foreseeable future.

If we do not recognize the reduction operations applied to **`VIR`** and **`FX`**, the lower bound on value-based flow dependences indicates parallelism inhibiting dependences. Thus, we must recognize these reductions to parallelize this routine. When we calculate a lower bound on the memory-based dependences, we find that they force completely sequential execution. To find parallelism, we must apply techniques that allow us to eliminate these dependences.

7. TIME REQUIRED FOR VALUE-BASED DEPENDENCE ANALYSIS

Paul Feautrier's prototype implementation of his method for exact value-based dependence analysis is very slow (one program (`jacobi`) of 50 lines he examined took 82 seconds to analyze on a "low-end SPARC station"). An effort is underway to build an efficient implementation Feautrier's method; it is unclear how much improvement will be obtained. If Feautrier's initial results were indicative of the cost of exact value-based dependence analysis, the methods described in this paper may not be practical. Fortunately, we have reason to believe the cost is much less than Feautrier's initial results suggested.

While we do not have a complete implementation of the methods described in this paper at this time, we are able to measure the time required to compute value-based dependences for Feautrier's examples and some other benchmark codes. Table I lists the benchmarks that we used, and compares these analysis times with the time needed to compile and optimize the routine for a Sun SPARCstation 10-51. The table shows the time required to analyze all the memory-based dependences and the value-based flow dependences. The time shown for value-based dependence analysis does not include the memory-based analysis time, though the memory-based dependences were used during value-based analysis. The value-based dependence analysis was done using the optimizations described in [PW93]. The times shown in this table include analysis of both arrays and scalars, because we need to have a dependence relation for each possible dependence when we construct the transitive self-dependence relations (as described in Section 4.1). For value-based dependences, we also show the time required to analyze just the dependences that might be in cycles.

These results reassure us that it will be practical to apply these techniques to many significant, real world problems. We have not yet fully implemented in techniques in [PW93], so we have been unable to evaluate our performance on codes that stress our methods for handling negation.

8. RELATED WORK

In our first paper on array data dependence analysis [Pug92], we describe a set of algorithms (the Omega test) that can be used to check for the existence of integer solutions to sets of linear constraints and calculate the "shadow" of a set of linear constraints (i.e., eliminate existential quantified variables). For example, by creating a set of constraints for a data dependence, adding variables for the dependence differences, and finding the shadow of these constraints on the dependence difference, we find the possible values for the dependence difference. We then show how these techniques can be used to perform efficient analysis of memory-based array data dependences.

In our second paper on dependence analysis [PW92b], we extend the Omega test with efficient techniques for removing redundant constraints and checking when one set of constraints implies another. We give techniques that can identify some dependences as being not value based. However, these techniques do not identify all non-value based dependences and work on dependence differences, not dependence relations.

In [PW93], we give full descriptions of our techniques for simplifying formulas containing negation (as mentioned in Section 3.5.1). We also compare the performance of the technique described in this paper for analysis of array kills with that of [Fea91, MAL93].

In other papers [Pug91, KP93], we describe a unified framework for reordering transformations. Within this framework, we discuss methods for checking the legality of a transformation, generating transformations, and producing transformed code corresponding to a reordering transformation. As part of this work [Pug91], we describe dependence relations and techniques for computing their transitive closure.

There have been a number of papers on improving the accuracy of memory-based array data dependence analysis [KKP90, LC90, WT92, GKT91, MHL91b]. Some of these methods provide ways of recognizing when their results are exact, but do not describe any methods for computing lower bounds on dependences when they are not exact.

There are a number of papers on techniques to analyze array kills and value-based array data dependence [Bra88, Fea88, GS90, Ros90, Rib90, Fea91, Li92, MAL92, May92, MAL93, DGS93, Mas94]. Of these, only the technique we describe here and that of Feautrier [Fea88, Fea91] and of Maslov [Mas94] are complete over the domain of affine expressions and no control flow other than loops. The techniques described by Feautrier have only been implemented in a prototype form, and our techniques appear to be 40-75 times faster than Feautrier's prototype. Maslov's techniques are formulated using the lexigraphical maximum idea of Feautrier, but operate in a lazy manner and so are more efficient. Maslov's techniques achieve speed comparable to ours, and utilize our algorithms for manipulating and analyzing Presburger formulas. Both our methods and Maslov's methods handle more complicated control flow than Feautrier, such as `if`'s with non-affine guards, although inexactly.

The methods described by [MAL93] improve on the efficiency of [Fea91] but do not work for certain cases handled by [Fea91] and this paper.

The quast's(Quasi-Affine Search Trees)/Last-Write-Trees constructed by [Fea91, MAL93] may contain infeasible paths. To enable compile-time transformations such as privatization, it is necessary to determine which of these paths are feasible. Determining which of the paths are feasible requires checking the feasibility of a problem such as:

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \wedge \neg N_1 \wedge \neg N_2 \wedge \cdots \wedge \neg N_m$$

where the P_i's are the conditions for the nodes where we take the true branch and the N_i's are the conditions for the nodes where we take the false branch. Each of these conditions is a conjunction of linear constraints, and may include non-convex constraints (e.g., constraints such as "i is even" specified using wildcards or quasi-linear constraints). Converting these expressions into disjunctive normal form would be infeasible for many real problems. The methods we describe in [PW93] should control this blow-up. In addition to the blow-up problem, the methods described by [Fea91] and [MAL93] fail when forced to negate certain pathological cases of non-convex constraints. Our methods can handle these (although some of the pathological cases will create performance problems).

Our methods for array data-flow analysis, like those of [Bra88, Rib90, Fea91, MAL92, Voe92a, Voe92b, MAL93], are based on extending standard array dependence analysis methods to analysis the flow of values rather than the reuse of memory. Array data-flow dependence analysis methods such as [GS90, Ros90, Li92, DGS93] are based on extending scalar data-flow analysis methods to arrays. In general, the later approach deals better with control flow but the former approach gives more information about which iteration is dependent on which iteration (as opposed to simply summarizing which loops carry the dependence).

Both Feautrier [Fea88, Fea91] and Maydan, Amarasinghe and Lam [MAL93] describe how to use value-based dependence information for array expansion or privatization. Amarasinghe and Lam [AL93] describe ways to use value-based dependence analysis in analyzing and generating code for distributed memory multicomputers.

There have been a number of recent papers [Lar93, PP93, MHL91a] describing the results of instrumenting programs so that during a run on sample data, information about the actual dependences and available parallelism are collected. However, we do not know of any other work on static analysis of upper bounds on parallelism.

9. FUTURE WORK

There are two substantial obstacles to using these techniques in industrial strength environments: our inability to handle arbitrary control flow and procedure calls.

In these cases, we will likely have to resort to computing upper and lower bounds.

In order for the techniques described here to be applicable, we need to be able to identify well-defined loops so that the dependence relations can be well defined. If the loops are well defined, it is relatively easily to calculate upper and lower bounds on memory-based dependences. Calculating upper and lower bounds on value-based flow dependences appears more difficult.

For procedure calls, we can either effectively inline the procedure call, or we can calculate dependences based on summary information of what the call kills and uses. Since this summary information will probably not be exact, we will need to calculate appropriate upper and lower bounds (and the summary information will need to contain both may and must information for both kills and uses).

10. IMPLEMENTATION STATUS AND BENCHMARK AVAILABILITY

The techniques described here are being implemented in our extended version of Michael Wolfe's `tiny` tool [Wol91], which is available for anonymous ftp from `ftp.cs.umd.edu:pub/omega`. The programs analyzed in Section 7 come from a set of benchmark programs for comparing the performance and coverage of algorithms for analyzing value-based flow dependences between array references. Send email to `omega@cs.umd.edu` to receive a copy of the benchmarks and be added to the dataflow benchmarks mailing list.

11. CONCLUSIONS

If a routine's array subscripts, loop bounds, and branching conditions are affine functions of the loop indices and constants, we can compute a dependence relation that describes the dependence between any two array accesses exactly. Otherwise, we can compute both a lower and an upper bound on each dependence. The traditional upper bound can be used to conclude that code can be run in parallel (that is, to put a lower bound on the parallelism). The lower bound on value-based flow dependences can be combined with our tests for parallelism to provide an upper bound on the parallelism in a given algorithm. By comparing the parallelism found in the upper and lower bounds, we can determine whether more exact dependence information about non-linear terms would be useful in exposing parallelism.

The method we describe for value-based dependence analysis is complete and exact within the domain of programs with affine subscripts, loop bounds, and branching conditions, and works outside that domain. All other techniques had been incomplete over this domain, although Feautrier's is incomplete only for pathological cases. Value-based dependence analysis is essential for enabling array privatization/expansion and is very useful in analyzing and generating code for distributed memory multicomputers.

Our tests for parallelism, when applied to various classes of dependences, will detect any (possibly conditional) parallelism that can be exposed by any combination of the following transformations: scalar or array privatization, expansion, and renaming, parallel reduction, (possibly imperfect) loop interchange, distribution, alignment or peeling, index set splitting, statement reordering, unimodular loop transformations, introduction of run-time dependence tests, and scalar induction variable replacement. When conditional parallelism is detected and the conditions are affine functions of the symbolic constants, the conditions are provided in the dependence relation.

The parallelism tests often provide insight into which transformations are necessary to exploit the parallelism in the code. In cases where automatic techniques cannot parallelize a program, the techniques described here will allow a user or compiler to concentrate effort on those sections of code where there is hope of exposing parallelism.

12. ACKNOWLEDGMENTS

Thanks to Wayne Kelly and Vadim Maslov for their comments on the paper and assistance on the implementation. The referees comments were important in improving this paper. This work was supported by a Packard Fellowship and by a NSF PYI award, CCR-9157384.

REFERENCES

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

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

[AL93] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[B+89] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[Ban88] Utpal Banerjee. An introduction to a formal theory of dependence analysis. *Journal of Supercomputing*, 2(2):133–149, 1988.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 192–219, Irvine, CA, August 1990.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelism. In *Proc of 1988 International Conference on Supercomputing*, pages 407–417, July 1988.

[Coo72] D. C. Cooper. Theorem proving in arithmetic with multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. American Elsevier, New York, 1972.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of 4 Perfect benchmark programs. In *Proc. of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991.

[Eig92] Rudolf Eigenmann. Toward a methodology of optimizing programs for high-performance computers. CSRD Rpt. 1178, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1992.

[Eig93] R. Eigenmann. Towards a methodology of optimizing programs for high-performance computers. In *International Conference on Supercomputing*, pages 27–36, July 1993.

[Fea88] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, pages 429–441, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.

[GKT91] G. Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[GS90] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software – Practice and Experience*, 20:133–155, February 1990.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proc. of the 1991 International Conference on Supercomputing*, pages 244–253, June 1991.

[KK67] G. Kreisel and J. L. Krevine. *Elements of Mathematical Logic*. North-Holland Pub. Co., 1967.

[KKB92] K. G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hieracical parallel machines in polynomial time. In *Proc. of the 1992 International Conference on Supercomputing*, pages 82–92, July 1992.

[KKP90] X. Kong, D. Klappholz, and K. Psarris. The I test: A new test for subscript data dependence. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[Lar93] James R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):812–826, October 1993.

[LC90] L. Lu and M. Chen. Subdomain dependence test for massive parallelism. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[Li92] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of the 1992 International Conference on Supercomputing*, pages 313–322, July 1992.

[MAL92] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Data dependence and data-flow analysis of arrays. In *5th Workshop on Languages and Compilers for Parallel Computing (Yale University tech. report YALEU/DCS/RR-915)*, pages 283–292, August 1992.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM '93 Conf. on Principles of Programming Languages*, January 1993.

[Mas92] Vadim Maslov. Delinearization: an efficient way to break multiloop dependence equations. In *ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, San Francisco, California, June 1992.

[Mas94] Vadim Maslov. Lazy array data-flow dependence analysis. In *ACM '94 Conf. on Principles of Programming Languages*, January 1994.

[May92] Dror Eliezer Maydan. *Accurate Analysis of Array References*. PhD thesis, Computer Systems Laboratory, Stanford U., September 1992.

[McK90] Kathryn S. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report RICE COMP TR91-162, Dept. of Computer Science, Rice University, December 1990.

[MHL91a] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Effectiveness of data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1991.

[MHL91b] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[PP93] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis. In *International Conference on Supercomputers*, July 1993.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *1991 International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[Pug93] William Pugh. Definitions of dependence distance. *Letters on Programming Languages and Systems*, September 1993.

[PW92a] William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 140–151, San Francisco, California, June 1992.

[PW92b] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[PW94] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, 1994. To appear.

[Rib90] Hudson Ribas. Obtaining dependence vectors for nested-loop computations. In *Proc of 1990 International Conference on Parallel Processing*, pages II–212 – II–219, August 1990.

[Ros90] Carl Rosene. *Incremental Dependence Analysis*. PhD thesis, Dept. of Computer Science, Rice University, March 1990.

[Voe92a] Valentin V. Voevodin. *Mathematical Foundations of Parallel Computing*. World Scientific Publishers, 1992. World Scientific Series in Computer Science, vol. 33.

[Voe92b] Vladimir V. Voevodin. Theory and practice of parallelism detection in sequential programs. *Programming and Computer Software (Programmirovaniye)*, 18(3), May 1992.

[WL90] M. E. Wolf and M. Lam. Maximizing parallelism via loop transformations. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.

[Wol91] Michael Wolfe. The tiny loop restructuring research tool. In *Proc of 1991 International Conference on Parallel Processing*, pages II–46 – II–53, 1991.

[WT92] M. J. Wolfe and C. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

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