In the Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, California, June 17–19, 1992, pages 249–260. Copyright ©1992 ACM (see notice below).

Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs

Laurie J. Hendren*
School of Computer Science
McGill University

Joseph Hummel†
Alexandru Nicolau‡
Dept. of Information and Computer Science
UC-Irvine

Abstract

Even though impressive progress has been made in the area of optimizing and parallelizing programs with arrays, the application of similar techniques to programs with pointer data structures has remained difficult. In this paper we introduce a new approach that leads to improved analysis and transformation of programs with recursively-defined pointer data structures.

Our approach is based on a mechanism for the Abstract Description of Data Structures (ADDS), which makes explicit the important properties, such as dimensionality, of pointer data structures. Numerous examples demonstrate that ADDS definitions are both natural to specify and flexible enough to describe complex, cyclic pointer data structures.

We discuss how an abstract data structure description can improve program analysis by presenting an analysis approach that combines an alias analysis technique, *path matrix* analysis, with information available from an ADDS declaration. Given this improved alias analysis technique, we provide a concrete example of applying a software pipelining transformation to loops involving pointer data structures.

1 Introduction and Motivation

One of the key problems facing both optimizing and parallelizing compilers for imperative programming languages is *alias analysis*, that is, detecting when two distinct memory accesses may refer to the same physical memory location. Alias analysis is a critical component of such compilers, and the effectiveness of many compiler analysis techniques and code-improving transformations rely upon accurate alias analysis. Given the current trend towards vector, RISC, and superscalar architectures, where optimizing compilers are even more important for efficient execution, the need for more accurate analysis will grow.

Scientific codes have often been the target of optimizing and parallelizing compilers. Such codes typically use arrays for storing data, and loops with regular indexing properties to manipulate these arrays. A good deal of work has been done in the area of analysis and transformation in the presence of arrays and loops, and as a result numerous techniques have been developed; for example, invariant code motion, induction variable elimination, loop unrolling, and vectorization [Lov77, Kuc78, DH79, PW86, AK87, ASU87, ZC90], along with various instruction scheduling strategies such as software pipelining [RG82, AN88a, AN88b, Lam88, EN89]. Unfortunately, codes that utilize dynamically-allocated pointer data structures are much more difficult to analyze, and therefore there has not been as much progress in this area. This is problematic, since numerous data structures in imperative programs—linked-lists and trees for example—are typically built using recursively-defined pointer data structures. Furthermore, such data structures are used not only in symbolic processing, but also in some classes of scientific codes (e.g. computational geometry [Sam90] and the so-called *tree-codes* [App85, BH86]).

The goal of this paper is to: (1) investigate why the analysis of pointer data structures is more difficult than the analysis of array references, (2) suggest a new mechanism for describing pointer data structures, and (3) show how better descriptions of such data structures lead to improved analysis and transformation of imperative programs.

*This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.
†Direct correspondence to this author, jhummel@ics.uci.edu.
‡This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

1.1 The Problem

To motivate the problem, consider the two code fragments shown in Figure 1[1]. The upper code fragment adds the element `b[j]` to each element of array `a`. Since arrays have the property that `a[i]` and `a[j]` refer to different locations if $i \neq j$, the compiler can immediately determine from this code fragment that each iteration references a different location of `a`. Further, arrays are usually statically allocated, and it is straightforward to determine that the arrays `a` and `b` are different objects. Thus any reference to `a` will never refer to the same location as `b[j]`, implying that `b[j]` is loop invariant. These observations allow the compiler to perform a number of transformations, including (1) loading `b[j]` into a register before the loop begins, and (2) transforming the loop by perhaps unrolling or applying software pipelining.

```
for i = 1 to N
  a[i] = a[i] + b[j];


while p <> NULL
 { p->data = p->data + q->data;
   p = p->next;
 }
```

Figure 1: Arrays versus linked-lists.

Now consider the lower code fragment in Figure 1, which operates on a linked-list. This code traverses a list `p`, and at each step adds the value `q->data` to the node currently being visited. In this case, the properties of the structure are not obvious from the code fragment. For example, unless we can guarantee that the list is acyclic, we cannot safely determine that each iteration of the loop refers to a different node `p`. This makes the application of loop transformations difficult. Secondly, since the nodes are dynamically allocated, it is much more difficult to determine if `p` and `q` ever refer to the same node. Hence the compiler will be unable to detect if `q->data` is loop invariant. As a result, even though the code fragment performs a similar function to the array-based version, traditional optimizing and parallelizing compilers will be unable to apply similar transformations.

1.2 Previous Work

There are currently three approaches for dealing with the problem of alias analysis in the presence of pointer data structures: (1) assume the most conservative case, that all pointers and nodes are potential aliases, (2) analyze other parts of the program in an attempt to automatically discover the underlying properties of the data structures (and hence the relationship between all pointers and nodes), or (3) require code annotations.

Although approach (1) is far more common (perhaps with (3) as a backup), there has been significant work on approach (2). One class of solutions has been the development of advanced alias analysis techniques (also called structure estimation techniques) that attempt to statically approximate dynamically-allocated data structures with some abstraction. The most commonly used abstraction has been *k-limited* graphs [JM81], and variations on *k-limited graphs* [LH88a, LH88b, HPR89, CWZ90]. The major disadvantage of these techniques is that the approximation introduces cycles in the abstraction, and thus list-like or tree-like data structures cannot be distinguished from data structures that truly contain cycles. The work by Chase et al. [CWZ90] has addressed this problem to some degree; however, their method fails to find accurate structure estimates in the presence of general recursion. This is a serious drawback since programs with recursively-defined data structures often use recursion as well. Another method, *path matrix* analysis, was designed to specifically deal with distinguishing tree-like data structures from DAG-like (shared) and graph-like (cyclic) structures [HN90, Hen90]. This analysis uses the special properties exhibited by tree-like structures to provide a more accurate analysis of list-like and tree-like structures even in the presence of recursion. However, it has the disadvantage that it cannot handle cyclic structures (even if the cyclic nature would not hamper optimizing or parallelizing transformations). Related approaches include those based on more traditional dependence analysis (e.g. [Gua88], which assumes that structures do not have cycles) and abstract interpretation techniques (e.g. Harrison in [Har89] presents a technique designed for list-like structures commonly used in Scheme programs, while more recently he has been developing a uniform mechanism for handling both symbolic data and arrays using the notion of generalized iteration space [Har91]). In general, even though each of these methods work for certain classes of pointer data structures, they undoubtably fail in the presence of general, possibly cyclic structures and recursion.

Code annotations represent a compromise, since the programmer can specify what the compiler cannot determine. However, code annotations are often difficult to use, due to the varied kinds of information that must be conveyed to the compiler. For example, the programmer may need to specify the data dependencies [Lar89], the transformation to apply [CON], or the "distinctness" of data [KKK90]. Also, the compiler typically takes these annotations on blind faith, and so is unable to warn the programmer if a coding change invalidates an annotation; this is true for [Lar89, CON],

[1] Treat a, b, p, and q as local variables, where a and b are arrays of size N and p and q are pointers.

while in [KKK90] it was suggested that the system could perform dynamic checks provided that the *programmer* uniquely "tags" each node. Finally, such annotations must be repeated throughout the program to maximize performance.

2 Our Approach

Based on our past experience of developing alias analysis techniques for tree-like structures, and the failure of other techniques to find accurate information for more general structures, we believe that a lack of appropriate data structure descriptions is the most serious impediment to the further improvement of analysis techniques. After studying a wide range of imperative pointer data structures, we have developed an approach for describing what we feel are the important properties of such structures, important in the sense of enabling numerous optimizing and parallelizing transformations. Current imperative programming languages provide no mechanisms for expressing this kind of information[2].

Our approach, called **ADDS**, provides the programmer with a mechanism for the **A**bstract **D**escription of **D**ata **S**tructures. ADDS is a minor addition to most imperative programming languages, and was designed to:

- be simple for the programmer to use,
- minimize and localize program annotation,
- describe complex pointer data structures, and
- enable powerful transformations.

The properties expressed through such data structure descriptions are used to increase the accuracy and hence effectiveness of existing analysis techniques. This allows the application of powerful optimizing and parallelizing transformations, and may also lead to the development of new pointer-specific transformations.

Asking the programmer to specify some properties of his or her data structures should not be considered a radical change in our way of thinking about programming in imperative programming languages. In the pointer data structures domain, programmers already convey quite a bit of implicit information about their data structures. For example, consider the following two recursive type declarations:

```
type BinTree              type TwoWayLL
 { int       data;         { int       data;
   BinTree *left;            TwoWayLL *next;
   BinTree *right;           TwoWayLL *prev;
 };                        };
```

[2]It should be noted that Larus in [Lar89] discussed a similar approach for Lisp; however, his approach required code (not type) annotations, described only acyclic structures, and could not be used in code fragments which might modify the structure.

Even though these type declarations appear identical to the compiler (each declares a record with three fields, one integer and two recursive pointers), the naming conventions imply very different structures to readers of the program. In addition, each structure has some very nice properties which the compiler could exploit. A binary tree naturally subdivides into two disjoint subtrees that can be operated on in parallel. A two-way linked-list has the property that a traversal in the forward direction using only the `next` field never visits the same node twice (likewise for traversals using only the `prev` field); this property of never visiting the same node twice enables the parallelization of node processing along the list. The idea of ADDS is simply to make this implicit information explicit to the compiler. Positive side-effects may be increased human understanding of programs, and the compiler's ability to generate run-time checks to ensure proper use of dynamic data structures.

Note that Fortran 90 has taken a similar stance in its treatment of pointers to variables. Variables accessible through pointers must be explicitly declared as either pointers or targets [MR90]. This simple declaration greatly improves the accuracy of alias analysis in the presence of pointers.

However, also note that the presence of a description mechanism such as ADDS is not enough by itself. Side-effects in imperative programs often rearrange the components of a data structure, causing a temporary but intentional invalidation of the properties we wish to exploit. Application of optimizing or parallelizing transformations (that rely on these properties) during such a time would be incorrect, and intolerable. Hence some form of data structure validation analysis, beyond alias analysis, is necessary not only to ensure correctness, but to enhance debugging as well.

In the remainder of this paper we present our viewpoint that:

1. recursive pointer data structures, regardless of their overall complexity, typically contain substructures that exhibit regular properties that can be exploited for the purpose of program analysis and transformation, and
2. the ability to express these properties explicitly is an important first step towards the long-term goal of accurate and efficient program analysis in the presence of cyclic pointer data structures.

The organization of the paper is as follows. In sections 3 and 4 we address the problem of expressing the regular properties of recursive pointer data structures. In section 3 we present ADDS through a series of intuitive, increasingly complex examples, and in section 4 we define the properties of ADDS data structures more formally. In section 5 we discuss how the information provided by ADDS can be used to improve the analysis and transformation of programs; in particular, we demonstrate how ADDS enables the application of software pipelining to a list-traversal loop. Finally, in section 6 we present our conclusions.

3 ADDS - Abstract Description of Data Structures

The transformation of codes involving data structures requires knowledge about the properties exhibited by that structure, e.g. shape, size, and method of element access. With arrays, these properties are readily identifiable. The shape of an array is fixed and declared at compile-time, its size is known at the time of creation (and often at compile-time), and locations are referenced using integer indices. For example, given a statement S of the form `i = i + c` (`c` $\neq 0$), the compiler is guaranteed that `a[i]` refers to different elements of `a` before and after the execution of S. Contrast this with user-defined pointer data structures, in which none of these properties are made explicit. Unlike the array example, even a simple traversal statement T such as `p = p->next` will prevent the compiler from determining whether `p` denotes a different "element" after the execution of T, or the same element.

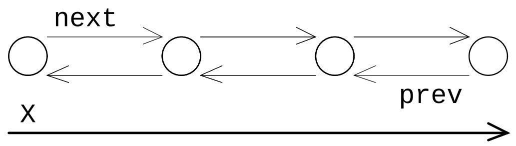
In this section we shall present our mechanism—ADDS—for describing what we feel are the important properties of recursive pointer data structures: shape, and a sense of direction. We begin with an intuitive summary of these properties, and then demonstrate through a series of increasingly complex examples how such properties are captured using ADDS. We postpone a more formal definition of ADDS to section 4.

3.1 Dimensions, Directions in Pointer Data Structures

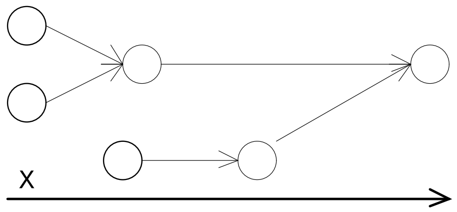
Suppose you have a recursive pointer data structure and you wish to describe its shape. Consider the structure in its general form, and select a node as the "origin"—it doesn't matter which node you choose, though some choices make more sense than others (e.g. the root of a tree versus a leaf). Next, think of your structure as having "dimensions," different paths emanating from the origin, with typically one dimension per path. Finally, select a node n other than the origin, and for each recursive pointer field f in n, decide which dimension f traverses and in which "direction." The "forward" direction implies traversing f moves one unit away from the origin, and "backward" implies traversing f moves one unit back towards the origin. A field is limited to traversing one dimension in only one direction[3].

[3] This restriction can be overcome by the programmer without too much difficulty.

By default, a structure has one dimension D, where it is assumed that all recursive pointer fields traverse D in an "unknown" direction. The idea is to override this default and provide more specific information. For example, we can specify that a singly linked-list has one dimension `X`, and one recursive pointer field `next` that traverses `X` in the forward direction. As illustrated below, a two-way linked-list is best described as having a single dimension `X`, with `next` traversing forward along `X` and `prev` traversing backward along `X`:



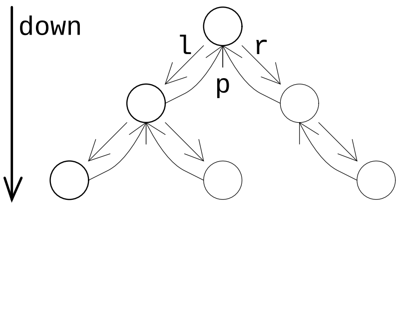
To differentiate such a list from a more general DAG-like list, e.g.



we introduce the notion of a field f traversing "uniquely forward," which implies that for any node n, at most one node n' ($n' \neq n$) points to n using f. Syntactically, the ADDS declaration of a two-way linked-list would then look like:

```
type TwoWayLL [X]
 { int      data;
   TwoWayLL *next is uniquely forward along X;
   TwoWayLL *prev is backward along X;
 };
```

A binary tree can be thought of as having two dimensions `left` and `right`, but for reasons soon to be apparent, we shall consider a binary tree as having only one dimension, `down`. The important property of a binary tree (and of trees in general) is that for any node n, all subtrees of n are disjoint. This information can be expressed by saying that the `left` and `right` fields "combined" traverse `down` in a uniquely forward manner. More formally, `left` and `right` exhibit the property that at most one `left` *or* one `right` points to any node n, but not both. The motivation for choosing one dimension to describe a binary tree is to support the notion of parent pointers, a field that refers from either a left or right child back to its parent. Pictorially, we view a binary tree with parent pointers as:

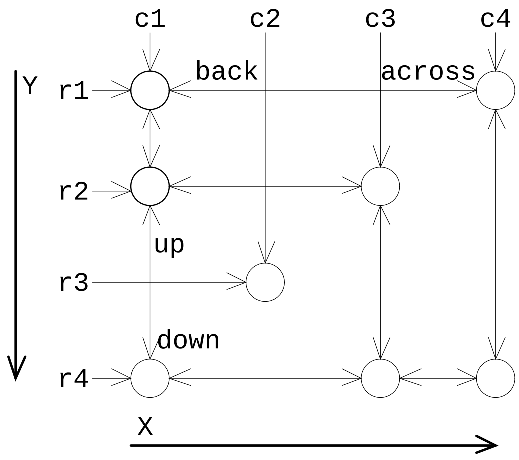


Its ADDS declaration would thus be:

```
type PBinTree [down]
 { int      data;
   PBinTree *left, *right
               is uniquely forward along down;
   PBinTree *parent
               is backward along down;
 };
```

Note that by declaring **left** and **right** together, we are expressing this notion of a combined traversal.

The flexibility of ADDS is illustrated by more exotic recursive pointer data structures. Typically such structures exhibit multiple dimensions, where dimensions are either "independent" (disjoint) or "dependent." For example, an *orthogonal list* [Sta80], used to implement sparse matrices, has two dependent dimensions X and Y (much like the two-dimensional array it represents):

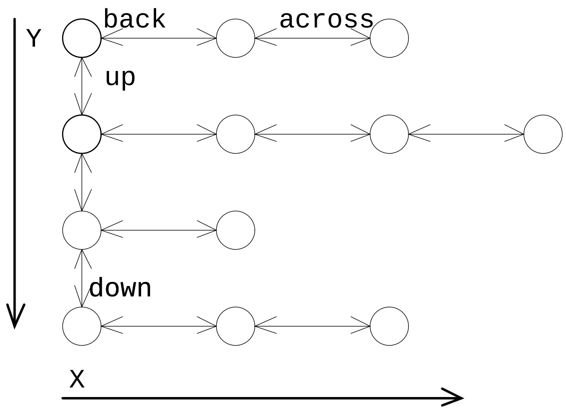


We say X and Y are dependent since one traversal along X and another traversal along Y may lead to a common node or substructure. For example, traversing along X from r4 and along Y from c3 may lead to the same node. However, even though the dimensions are dependent, notice that *orthogonal lists* still possess regular properties. For example, traversing forward along X, or forward along Y, is guaranteed never to visit the same node twice. Further, each row is disjoint, so that parallel traversals of different rows along X will never visit the same node (likewise for columns and the Y dimension). These properties are captured in the following ADDS declaration by declaring that the fields across and down are uniquely forward:

```
type OrthL [X][Y]
 { int    data;
   OrthL *across is uniquely forward along X;
   OrthL *back   is backward along X;
   OrthL *down   is uniquely forward along Y;
   OrthL *up     is backward along Y;
 };
```

Note that unless stated otherwise, dimensions (in this case X and Y) are considered dependent. Such conservative nature is intentional.

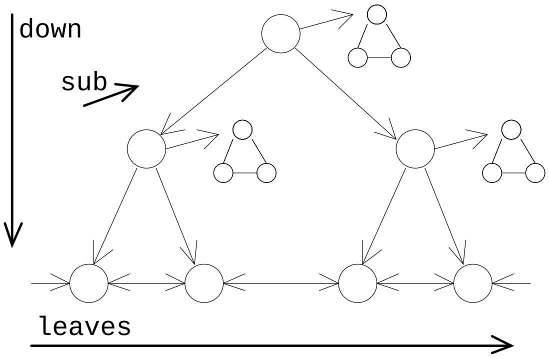
A similar data structure that has two independent dimensions is a list of lists. Consider the following which illustrates a list of lists, including back pointers along each dimension:



Note that each node may be accessed by a forward traversal along **either** the X dimension **or** the Y dimension, but not both; this is identical to the situation for binary trees. Hence X and Y are considered independent, which is conveyed using || in the following ADDS declaration:

```
type LoLs [X][Y] where X || Y
 { int   data;
   LoLs *across is uniquely forward along X;
   LoLs *back   is backward along X;
   LoLs *down   is uniquely forward along Y;
   LoLs *up     is backward along Y;
 };
```

An interesting three-dimensional structure that has both dependent and independent dimensions is the *two-dimensional range tree* [Sam90], used to answer queries such as "find all points within the interval x1...x2" or "find all points within the bounding rectangle (x1,y1) and (x2,y2)." As illustrated below, it is a binary tree of binary trees, where the leaves of each tree are linked together to form a two-way linked list:

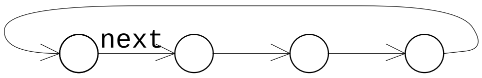


The dimensions down and leaves are dependent; each leaf node can be reached by a forward traversal along both the down dimension and the leaves dimension. However, observe that sub is independent of both down and leaves. That is, any node that can be accessed by a forward traversal along sub, cannot be accessed by a forward traversal along down nor along leaves. This leads to the following declaration:

```
type TwoDRT [down][sub][leaves]
                     where sub||down, sub||leaves
 { int     data;
   TwoDRT *left, *right
            is uniquely forward along down;
   TwoDRT *subtree is uniquely forward along sub;
   TwoDRT *next is uniquely forward along leaves;
   TwoDRT *prev is backward along leaves;
 };
```

Note that a union (variant record) can be used to save space by overlaying the storage for (left, right) with that of (next, prev). This can be done without a loss of expressive power.

Lastly, let us consider a common cyclic data structure, the circular linked-list:



This type of structure can be problematic, since a single field next is used for what are essentially two purposes, (1) traversing uniquely forward and (2) circling around. One declaration is thus:

```
type CirL [X]
 { int   data;
   CirL *next is unknown along X;
 };
```

This is equivalent to saying nothing at all (the default), and the unknown nature of next prevents the compiler from performing possible optimizing or parallelizing transformations (e.g. given the statement p = q->next the compiler must conservatively assume that p and q are aliases). One solution is to provide a more explicit declaration that more accurately reflects the properties of a circular list:

```
type CirL [X]
 { int   data;
   int   un_type;
   union
    { CirL *next is uniquely forward along X;
      CirL *around is unknown along X;
    } un;
 };
```

Obviously, this may require some rather clumsy coding practices. A second solution is to extend ADDS with another type of direction, "circular", and then declare next as traversing X in this direction:

```
type CirL [X]
 { int   data;
   CirL *next is circular along X;
 };
```

This solution is more desirable, since the declaration does not require a change in the coding of the program. However, one needs to know the length of the list in order to perform accurate alias analysis, which is generally unavailable at compile-time. This implies information must be collected and maintained at runtime, which is not considered in this paper.

3.2 Speculative Traversability

In all cases, a data structure declared using ADDS is required to be *speculatively traversable* [HG92]. This property allows one to traverse past the "end" of a data structure without causing a run-time error. It can be automatically supported by the compiler, and places no additional burden on the programmer (except good programming practices—e.g. in C, they must use the name NULL and not an arbitrary integer). This property is analogous to *computing* an array index outside the bounds of an array, but not actually using it. It is often useful when applying various optimizing and parallelizing transformations.

3.3 Summary of ADDS

In summary, ADDS is a technique for abstractly describing the important properties of a large class of useful data structures. Once known, these properties can be exploited by the compiler for analysis and transformation purposes. By default, a structure has one dimension D, and all recursive pointer fields traverse D in an unknown fashion. The programmer may refine this by describing (1) additional dimensions, (2) the interaction between dimensions, and (3) how the various fields traverse the dimensions. As illustrated with our examples, the choice of dimensions and directions is quite intuitive, and the use of ADDS does not place a heavy burden upon the programmer.

It should be noted that a programming language and its compilers could directly support data structures such as TwoWayLL and PBinTree via predefined, abstract data types (e.g. see [Gro91] for a tool that generates, from a context-free grammar description, ADTs for graph-like structures, and [Sol90] for an implementation of parallelizable lists). However, a quick survey of the literature (or a data structures text [Sta80]) would reveal a wide variety of important pointer data structures. Implementations for these structures can differ widely as well. Thus, instead of trying to predefine the most common types of pointer data structures (and constrain the implementation), we chose the opposite approach, namely to develop a technique which allows the programmer to define and implement *their own* data structures. We believe ADDS is flexible enough to describe the important properties of nearly all pointer data structures, while providing the compiler with the information necessary to enable powerful optimizing and parallelizing transformations.

4 Formal Properties of ADDS

In this section we present an overview of the formal properties of ADDS data structures. These definitions are not crucial to the understanding of the ideas, but they do show that our ADDS approach has some well-specified properties. For each definition we summarize the importance of the property being defined. If the reader is comfortable with the intuitive definitions presented in section 3, this section can be safely skipped.

Let N be a data structure declaration with n recursive pointer fields, $n \geq 1$. Let k be the number of programmer-specified dimensions over N, $1 \leq k \leq n$. The dimensions of N are denoted by $d_1, \ldots, d_k$, and the recursive pointer fields of N are denoted by $f_1, \ldots, f_n$. A recursive pointer field f may traverse along only one dimension d, and in only one of three directions: forward, backward, or unknown. pN denotes a dynamically-allocated node of type N, and $pN.f$ is either NULL[4] or denotes the dynamically-allocated node of type N reached by traversing the pointer stored in field f of pN. Traversing a series of non-NULL fields is denoted using a regular expression notation, e.g.

- $pN(.f)^2$ denotes the node $pN.f.f$,
- $pN(.f)^+$ denotes any one of the nodes $pN.f$, $pN.f.f$, $\ldots$,
- $pN((.f)+(.g))^*$ denotes any one of the nodes pN, $pN.f$, $pN.g$, $pN.f.f$, $pN.f.g$, $pN.g.f$, $pN.g.g$, $\ldots$.

For any such regular expression R, R denotes a list of nodes which may or may not contain duplicates. If this list is **run-time finite** (i.e. for all fields f in R, there exists a node pN' denoted by R such that $pN'.f$ is NULL), then there are no duplicates. Otherwise R denotes a **run-time infinite** list of nodes, in which case there must be duplicates.

All nodes pN can be thought of as forming a single data structure pDS of type N. pDS is considered **well-behaved** if the abstraction defined for N is valid. In cases where pDS is not well-behaved (e.g. pDS is under modification) the abstraction must be ignored at these points in the program[5]. The definitions presented here assume a well-behaved data structure.

Def 4.1: if N is **speculatively traversable**, then for all fields f and for all nodes pN, if $pN.f$ = NULL then $pN.f.f$ is a valid traversal and also yields NULL.
[Implication: legal to traverse past what would normally be thought of as the "end" of a structure.]

Def 4.2: if f traverses d in the **forward** direction, then for all nodes pN, the list of nodes denoted by $pN(.f)^*$ is run-time finite.
[Implication: f is acyclic, traversing f never visits a node twice.]

Def 4.3: if f traverses d in a **uniquely forward** direction, then Def 4.2 holds for f, and for all distinct nodes pN and pN', either

1. $pN.f = pN'.f =$ NULL, or
2. $pN.f \neq pN'.f$.

[Implication: f is acyclic and list-like, traversing f from different nodes never visits the same node.]

Def 4.4: if f traverses d in an **unknown** direction, then there may exist a node pN such that the list of nodes denoted by $pN(.f)^*$ is run-time infinite.
[Implication: f is potentially cyclic, traversing f is unpredictable and could visit a node twice.]

Def 4.5: if a field b traverses d in the **backward** direction, then there exists a field f that traverses d in the forward direction.

Def 4.6: if f traverses d in a uniquely forward direction and b traverses d in the backward direction, then for all nodes pN, either:

1. $pN.f =$ NULL,
2. $pN.f.b =$ NULL, or
3. $pN.f.b = pN$.

[Implication: traversing f then b forms a cycle, otherwise f or b is NULL.]

Def 4.7: if f and g ($f \neq g$) traverse d in a **combined uniquely forward** direction, then Def 4.3 holds for f, Def 4.3 holds for g, and for all distinct nodes pN and pN',

1. $pN.f \neq pN'.g$ or $pN.f = pN'.g =$ NULL, and
2. $pN.g \neq pN'.f$ or $pN.g = pN'.f =$ NULL.

[Implication: f and g are tree-like, they separate a structure into disjoint substructures.]

Def 4.8: if fields $f_1, f_2, \ldots, f_m$ ($2 < m \leq n$) traverse d in a combined uniquely forward direction, then for all f_i and f_j, $1 \leq i, j \leq m$ and $i \neq j$, Def 4.7 holds for f_i and f_j.
[Implication: generalization of Def 4.7—the m fields separate a structure into m disjoint substructures.]

Def 4.9: (a) let d_i and d_j be dimensions of N ($i \neq j$). If d_i and d_j are **independent**, then for any field f traversing d_i in the forward direction, for any field g traversing d_j in the forward direction, and for all distinct nodes pN and pN',

[4] Treat NULL as denoting a typeless node that is dynamically-allocated by the system at startup.

[5] The problem of validation is discussed in section 5.

1. $pN.f \neq pN'.g$ or $pN.f = pN'.g =$ NULL, and
2. $pN.g \neq pN'.f$ or $pN.g = pN'.f =$ NULL.

(b) Further, for any field uf traversing d_i in a uniquely forward direction and for any field b traversing d_i in the backward direction, either

3. $pN.uf =$ NULL, or
4. for all pN'' in the list denoted by $pN.uf(.g)^*$, $pN''.b = pN$ or $pN''.b =$ NULL.

[Implication: (a) d_i and d_j separate a structure into disjoint substructures, (b) uniquely forward/backward cycles hold across an independent dimension.]

Def 4.10: let d_i and d_j be dimensions of N $(i \neq j)$. If d_i and d_j are not independent, then they are **dependent** and Def 4.9 does not hold for d_i and d_j.
[Implication: traversal along d_i and d_j could lead to the same node/substructure.]

5 Analysis and Transformation using ADDS

The principal goal of ADDS is to improve the analysis of codes utilizing pointer data structures. As discussed in section 1.2, existing analysis-only approaches exhibit various limitations when faced with such structures. Our approach is to use the information available in the ADDS declarations to guide the analysis. This synergy between the abstract data structure descriptions and the analysis technique provides a more general and more accurate approach. For example, by using information about the dimensionality and direction of field traversals, the abstraction approximations are freed from estimating needless cycles (such as those formed by the forward and backward directions along the same dimension), and can therefore avoid making needless conservative approximations.

Let us make this idea more concrete. Analysis-only approaches begin with initial assumptions about what properties are important to estimate, and then tailor the approximation domain and analysis rules appropriately. For example, by choosing k-limited graphs as the approximation domain, one is forced to conservatively approximate non-cyclic structures with cycles, which implies the approximation cannot distinguish tree-like structures from cyclic ones. To further illustrate how the choice of analysis depends on the type of data structure under consideration, consider once again the following two data types, `BinTree` and `TwoWayLL`:

```
type BinTree              type TwoWayLL
 { int      data;          { int      data;
   BinTree *left;            TwoWayLL *next;
   BinTree *right;           TwoWayLL *prev;
 };                        };
```

Without additional information, these types will appear identical to the compiler, and thus the same sort of analysis must be applied. Clearly however, there are some analyses that will be more appropriate for one structure than the other; after all, binary trees exhibit much different properties than two-way linked-lists.

In the case of two-way linked-lists, the appropriate approximation for the paths between two nodes p and q is the number of `next` links from p to q. In addition, the analysis rules for traversing `next` links from q (`q = q->next`) should lengthen paths between p and q, while analysis rules for traversing `prev` links from q (`q = q->prev`) should shorten paths. Both the approximation domain and the appropriate rules for traversals can be easily inferred from an ADDS declaration that says (1) there is one dimension, (2) the field `next` traverses `uniquely forward`, and (3) the field `prev` traverses `backward`.

Now consider binary trees. In this case, the appropriate approximation of paths is that of going `left`, `right` or `down` (where `down` is a conservative approximation for going either `left` or `right`). Traversing along either the `left` or `right` fields lengthens paths; there is no field for which a traversal shortens a path. This information can be expressed by an ADDS declaration that which states that `left` and `right` traverse `uniquely forward` along the same dimension. This in turn can be used to build an appropriate approximation domain, and set of analysis rules, for binary trees.

Thus, the problem is that initial assumptions bias program analysis. If the analysis starts with very weak assumptions, then the resulting analysis will be overly conservative; this is the current situation with structure estimation techniques. If the initial assumptions are valid for only one class of data structures, then applying the analysis to other classes will lead to inaccurate and conservative analysis; this is the case of applying ordinary path matrix analysis to cyclic structures. Hence it appears crucial that in order to perform accurate analysis in the presence of a wide variety of data structures, we must begin with accurate information about the structure. This is the essence behind ADDS, and our approach.

In this section we show how the properties of ADDS data structures can be used to improve the accuracy of program analysis, and thus enable the application of powerful optimizing and parallelizing transformations. In particular, we present in section 5.1 a new approach combining ADDS with an existing analysis technique. Then in section 5.2 we demonstrate a new application of software pipelining made possible by our approach.

5.1 Program Analysis

As discussed earlier, the presence of a description mechanism such as ADDS is not enough in itself to enable optimizing and parallelizing transformations in the presence of pointer data structures. Imperative programs routinely rearrange components of such a structure, and it is during these points in a program that the abstraction (or parts thereof) must be ignored by the compiler. Otherwise transformations may be applied that are based on invalid assumptions, causing incorrect code generation. Hence some form of abstraction validation analysis is required to enable a given transformation; this is in addition to more traditional alias analysis which is needed to ensure that the transformation is safely applied.

Our approach to the analysis problem is a combined one, in which safe analysis techniques are used in conjunction with ADDS declarations. In particular, we are developing an approach to the static analysis of ADDS data structures that is an extension of *path matrix* analysis [HN90, Hen90], called *general path matrix* analysis. Path matrix analysis was originally designed to automatically discover and exploit the properties of acyclic data structures. With the help of ADDS, general path matrix analysis is capable of handling cyclic data structures as well.

General path matrix analysis computes, at each program point, a path matrix PM which estimates the relationship between every pair of live pointer variables; PM is thus a function of the current path matrix and the program statement under analysis. The entry $PM(p, q)$ denotes an explicit path or alias, if any, from the node pointed to by p to the node pointed to by q. The analysis does not attempt to express all possible paths between two nodes, since cyclic data structures would soon overwhelm the matrix. Instead, the paths explicitly traversed by the program are captured in the PM, while the remaining paths and aliases are deduced from the current state of the path matrix and the ADDS declarations.

5.1.1 Abstraction Validation Analysis

In order to validate an ADDS declaration, the effect of certain pointer statements on the path matrix must be compared with the original ADDS declaration. In particular, statements of the form `p->f = q` may change the shape of the data structure. This in turn may result in a violation of the declared abstraction. However, this is generally not an error in an imperative program, and so is not treated as one. Instead, we note that the abstraction is invalid at this point in the program, and we do not perform any transformations that rely on the validity of the necessary ADDS properties[6].

Though the actual process of validation is beyond the scope of this paper, the idea is as follows. The ADDS declaration is encoded as a series of relationships between the various pointer fields of the node. During analysis, if the path matrix ever denotes a relationship between two fields that is illegal, this part of the abstraction is deemed invalid and an entry is added to the path matrix encoding the violation. Later, if another program statement fixes the relationship between these two fields, the entry is removed and the abstraction is once again considered valid.

A common example of a temporary break in an abstraction is the moving of a subtree from one node to another within a binary tree. Here is a possible code fragment:

```
dest->left = src->left;
src->left  = NULL;
```

After analysis of the first statement, it is obvious that `src` and `dest` share a common subtree, even though this violates the disjointness property of a binary tree. However, the violation is immediately corrected, as is usually the case.

5.1.2 Alias Analysis

Alias analysis is best explained via example. Consider the following code fragment, in which the pointer variable `hd` denotes the head of a two-way linked-list of nodes. Each node represents a point in 2D space, and contains its x and y coordinates. The code shifts the origin from (0, 0) to (`hd->x`, 0) by subtracting the x-coordinate of this first point from all remaining points:

```
p = hd->next;
while p <> NULL
 { p->x = p->x - hd->x;
   p = p->next;
 }
```

	hd	p	p'
hd	$=$	$=?$	$=?$
p	$=?$	$=$	$=?$
p'	$=?$	$=?$	$=$

Alias Matrix AM

If the compiler fails to discover that `next` traverses a list in the forward direction (i.e. that Def 4.2 holds for `next`), then its analysis of the above code will be overly conservative—the compiler must assume that `next` is cyclic, and hence that `hd` and all values of `p` are potential aliases for the same node. The effect of this conservative alias analysis is summarized above in "alias matrix" form[7]. The p' entries are used to denote aliasing during the loop, where definite aliases are indicated by $=$ and possible aliases are indicated by $=?$. Observe that all the entries denote some form of aliasing. In particular, the entry $AM[hd, p']$ indicates that `hd` is a possible alias for the iterative values of `p`. As discussed in section 1.1, this prevents a number of useful loop transformations.

Now suppose the programmer declared his or her linked-list using the ADDS declaration `TwoWayLL`, as

[6] Of course, such a violation could in fact be an error. Warning the programmer, or providing a compiler switch to enable these warnings, would be a useful debugging tool.

[7] Please note that this is just a convenient way of expressing aliases, and is not the result of path matrix analysis.

given in section 3.1. Assuming general path matrix analysis determines that the structure abstraction is well-behaved (valid) at the start of this code fragment, the compiler can use the acyclic nature of the `next` field (from Def 4.2) to infer that the statement `p = p->next` never visits the same node twice. As a result, general path matrix analysis would produce the following path matrices, which denote (from top to bottom): just before the loop, after one iteration, and after the loop analysis has reached a fixed point. The important difference between these path matrices and the previous alias matrix AM is the replacement of the =? entires with more accurate information about paths.

	hd	p
hd	$=$	$next$
p		$=$

	hd	p	p'
hd	$=$	$next^2$	$next$
p		$=$	
p'		$next$	$=$

	hd	p	p'
hd	$=$	$next^+$	$next^+$
p		$=$	
p'		$next$	$=$

An entry in the path matrix like $next^+$ indicates a path of one or more `next` links; all paths are either explicitly encoded within an entry, implicitly encoded within other entries and the ADDS declarations, or both. All aliases are explicitly encoded, so an empty entry guarantees that the two pointers are not aliases. Thus, we see that the ADDS declaration and general path matrix analysis have captured the desired property in the PM necessary for performing numerous optimizing and parallelizing transformations, namely that `hd`, `p` and p' are never aliases.

5.2 Transformations

Optimizing and parallelizing transformations come in many forms, including:

- fine-grain transformations (e.g. improved instruction scheduling),
- loop transformations (e.g loop unrolling or software pipelining), and
- coarse-grain transformations (e.g. parallel execution of code blocks).

The application of such transformations typically requires accurate alias analysis. As shown in the previous subsection, ADDS and general path matrix analysis provide such accuracy. This will clearly aid in fine-grain transformations where dependency analysis is crucial. When transforming code that operates on a data structure, loop transformations typically require the structure to exhibit list-like properties, while coarse-grain transformations typically require tree-like properties. Such properties are all expressible using ADDS.

Earlier work [HG92] has shown the applicability of loop unrolling [DH79] on scalar architectures. For example, a simple loop to initialize every node of a linked-list showed 47% speedup on the MIPS architecture for a list of size 100 with 3-unrolling (see [HG92] for more details and timings). In this section we present an example of a more powerful loop transformation, software pipelining [RG82, AN88a, AN88b, Lam88, EN89]. Given the current trend towards machines supporting higher degrees of parallelism, e.g. VLIW and superscalar, this type of optimization offers larger speedups.

Once again, consider the code fragment discussed in the previous subsection 5.1.2, which manipulates a two-way linked-list of points. The following is an equivalent loop body written in pseudo-assembly code:

```
S1   if p==NULL goto done
S2   load   p->x,R1
S3   load   hd->x,R2
S4   sub    R1,R2,R3
S5   store  R3,p->x
S6   load   p->next,p
S7   goto   S1
```

As we saw earlier, if the compiler's analysis is overly conservative, then the compiler will incorrectly assume that `hd` and all values of `p` are potential aliases for the same node. In this case the data dependency graph shown in Figure 2 is constructed, with false loop-carried dependencies from S5 to S2 and S3. The result is that the loop appears to exhibit very little parallelism between iterations. However, as we also discussed, if the data structure is declared as a `TwoWayLL`, general path matrix analysis will be able to determine that in fact `hd` and all the iterative values of `p` are never aliases. This will eliminate these false dependencies, and the loop now appears as a sequence of nearly independent, parallelizable iterations.

To exploit this parallelism, we can apply software pipelining. First we need to minimize the effect of the loop-carried dependence from S6 to S1. By renaming, we can move S6 above S2, calling it S1.6, and then replace S6 with a copy statement. The result is the following semantically equivalent loop:

```
S1   if p==NULL goto done
S1.6 load   p->next,p‘
S2   load   p->x,R1
S3   load   hd->x,R2
S4   sub    R1,R2,R3
S5   store  R3,p->x
S6   move   p‘,p
S7   goto   S1
```

The next iteration of the loop can now begin as soon as S1.6 of the current iteration completes, allowing the

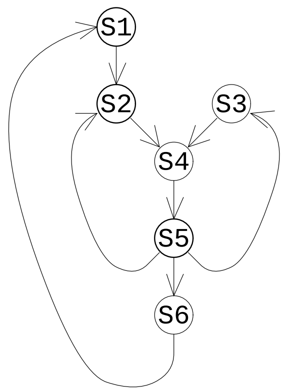


Figure 2: Dependency graph for pseudo-code.

loop bodies to nearly overlap in execution. But in addition, since the list must be speculatively traversable (by Def 4.1), it is safe to swap S1 and S1.6, further increasing the amount of available parallelism. Finally, since **hd** and **p** are never aliases, we can deduce that **hd->x** is loop invariant. Hence S3 can be moved outside and above the loop.

The code below incorporates the above-mentioned transformations, producing a semantically equivalent loop:

```
S0   load   hd->x,R2
S1   load   p->next,p‘
S2   if p==NULL goto done
S3   load   p->x,R1
S4   sub    R1,R2,R3
S5   store  R3,p->x
S6   move   p‘,p
S7   goto   S1
```

This loop can be pipelined as follows (the columns denote different iterations, rows denote statements executing in parallel):

```
S1
S2  S1
s3  S2  S1
S4  S3  S2  S1
S5  S4  S3  S2  S1
    S5  S4  S3  S2
        S5  S4  S3
            S5  S4
                S5
```

The boxed "statement" represents the new parallel pipelined body of the loop. As a result, the code exhibits a theoretical speedup of 5. Note that the copy statement S6 is removed as part of the pipelining process, via (enhanced) copy propagation [NPW91].

In general, software pipelining can lead to even larger speedups, depending on the characteristics of the loop body. Obviously, the actual speedup also depends heavily on the target machine's architecture.

6 Conclusions

As we have shown with numerous examples, many recursively-defined pointer data structures exhibit important properties which compilers can exploit for optimization and parallelization purposes. These properties are often known to the programmer—conveyed implicitly e.g. through the use of appropriate identifiers—and yet unavailable to the compiler. This lack of information hinders the accuracy of alias analysis and thus restricts the transformations that can be applied to codes using pointer data structures.

In this paper we have proposed an abstract description technique, ADDS, which allows the programmer to state such properties explicitly. The description of a recursive pointer data structure using ADDS is quite intuitive, and does not place an excessive burden on the programmer. By combining ADDS and general path matrix analysis, we demonstrated that the resulting approach enables more accurate and more general alias analysis, and hence the application of powerful optimizing and parallelizing transformations. As we have seen, software pipelining is one such transformation.

ADDS represents an important first step in our long-term goal of efficient compiler analysis of codes involving cyclic pointer data structures. ADDS is a simple extension to most any imperative programming language, and yet can lead to analysis and transformations that are otherwise not possible using traditional methods. With the increasing use of languages that support pointers and recursively-defined pointer data structures, the importance of such an approach will no doubt increase.

References

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[AN88a] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[AN88b] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*. Springer Verlag Lecture Notes in Computer Science No.300, March 1988.

[App85] Andrew W. Appel. An Efficient Program for Many-Body Simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[ASU87] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1987.

[BH86] Josh Barnes and Piet Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324:446–449, 4 December 1986. The code can be obtained from Prof. Barnes at the University of Hawaii.

[CON] CONVEX Computer Corporation. CONVEX C and FORTRAN Language Reference Manuals. 1990.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[DH79] J.J. Dongarra and A.R. Hinds. Unrolling loops in FORTRAN. *Software-Practice and Experience*, 9:219–226, 1979.

[EN89] Kemal Ebcioglu and Toshio Nakatani. A New Compilation Technique for Parallelizing Loops Loops with Unpredictable Branches on a VLIW Architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing. MIT-Press, 1989.

[Gro91] Josef Grosch. Tool support for data structures. *Structured Programming*, 12(1):31–38, January 1991.

[Gua88] Vincent A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[Har91] W. Ludwell Harrison III. Generalized iteration space and the parallelization of symbolic programs. In Ian Foster and Evan Tick, editors, *Proceedings of the Workshop on Computation of Symbolic Languages for Parallel Computers*. Argonne National Laboratory, October 1991. ANL-91/34.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, April 1990. TR 90-1114.

[HG92] Laurie J. Hendren and Guang R. Gao. Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structures. In *Proceedings of the 4th IEEE International Conference on Computer Languages (to appear, also available as ACAPS Technical Memo 28, McGill University)*, April 1992.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1):35–47, January 1990.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[JM81] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[KKK90] David Klappholz, Apostolos D. Kallis, and Xiangyun Kang. Refined C: An Update. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 331–357. The MIT Press, 1990.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations: Volume I*. Wiley, 1978.

[Lam88] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[Lar89] James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley, 1989.

[LH88a] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LH88b] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[Lov77] D.B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford University Press, 1990.

[NPW91] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and their impact on Fine-grain Parallelism. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[RG82] B. R. Rau and C. D. Glaeser. Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support. In *Proceedings of the 9th Symposium on Computer Architecture*, April 1982.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[Sol90] Jon A. Solworth. The PARSEQ Project: An Interim Report. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 490–510. The MIT Press, 1990.

[Sta80] Thomas A. Standish. *Data Structure Techniques*. Addison-Wesley, 1980.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.