Dynamic Parallelization of Modifications to Directed Acyclic Graphs

Lorenz Huelsbergen
Bell Laboratories*
lorenz@research.bell-labs.com

Abstract

We describe an interprocedural technique, called *dynamic resolution*, for the automatic parallelization of procedures that destructively manipulate dynamic DAGs. Dynamic resolution dynamically detects shared data and correctly coordinates access to this data at run time. In pointer-unsafe languages (*e.g.*, C), dynamic resolution requires programmer identification of acyclic data structures and the use of dynamic resolution's macros for pointer manipulations; parallelization is then automatic. In pointer-safe languages (*e.g.*, ML), cyclicity can often be automatically inferred by the compiler and parallelization via dynamic resolution is completely automatic.

This paper empirically studies the performance of dynamic resolution. Our study reveals that dynamic parallelization can readily outperform optimized sequential C programs on fast contemporary hardware. In particular, implementations of two general problems (DAG rewrite and in-place list quicksort) using dynamic resolution and three processors already outperform their sequential counterparts. Dynamic resolution is the first technique that can automatically and effectively parallelize DAG rewrite and list quicksort. For both problems, the absolute performance of dynamic resolution steadily improves as processors are added. We find that in the presence of some shared structure, dynamic resolution can still offset its run-time overheads.

1 Introduction

Parallelization of irregular computations involving mutable dynamic data structures is difficult for programmers and compilers alike. Programmers must reason about shared substructures in a program's dynamic data to correctly synchronize parallel access to shared state; parallelizing compilers, moreover, must statically infer structure sharing and produce a safe evaluation and synchronization schedule [7, 17]. The difficulty lies in the dynamic nature of irregular computations—shared structure appears (and disappears) dynamically. Hence, the available parallelism, and its attendant synchronization requirements, necessarily vary during program execution.

This paper describes the design and implementation of an interprocedural dynamic-parallelization technique called *dynamic resolution*. Dynamic resolution (**dr**) can automatically parallelize program procedures that destructively manipulate directed acyclic graphs (DAGs). The **dr** technique dynamically detects and dynamically schedules potentially conflicting DAG modifications; it preserves the program's sequential semantics by *resolving* conflicts at run time.

Static pointer analyses [4, 14, 9, 17, 7, 28, 15] can not provide precise parallelization information in the presence of dynamic structure sharing—such analyses must conservatively assume that *if* sharing can occur, then it *always* occurs. Only a dynamic approach can parallelize programs that *may* share structure when in fact they *do not* dynamically share structure. Dynamic resolution is a run-time approach that finds and exploits parallelism necessarily obscured by all static approaches.

We have implemented dynamic resolution for two general DAG problems (destructive DAG rewrite and destructive quicksort) on a fast contemporary shared-memory multiprocessor. Dynamic resolution is the first technique that can automatically and effectively parallelize these problems. Given a small number of processors (two or three), dynamic resolution already outperforms fully-optimized sequential C code *even though* dynamic resolution requires run-time support. We furthermore observe continued performance improvements as processors (up to our machine's maximum) are added to the problem.

To be completely automatic, dynamic resolution requires language support in the form of strong typing and garbage collection (such as provided by ML [22, 23]); such support ensures pointer integrity, *i.e.* pointers can never point to unallocated storage. However, with minor programmer intervention, **dr** may also be used with C programs. For C, the programmer must first statically identify the program's acyclic and cyclic data since **dr** can only parallelize procedures that oper-

*Address: 700 Mountain Ave. Murray Hill, NJ 07974. Phone: (908) 582 4628

To Appear in Proceedings of 1996 Parallel Architectures and Compilation Techniques.

ate on acyclic structures. Secondly, the C programmer must use **dr**'s C pointer-manipulation macros when constructing or modifying dynamic data. That is because **dr** must dynamically associate sharing information with heap-allocated data.

The purpose of this paper is threefold. First, we describe the compile-time analyses (static component) and the run-time support (dynamic component) required for dynamic resolution in the context of pointer-safe ML.[1] Second, we empirically evaluate dynamic resolution on two nontrivial DAG problems: destructive DAG rewrite and destructive (in place) list quicksort. Experimental comparisons of the **dr** versions of these problems to their sequential C and unsafe[2] parallel C versions reveal **dr**'s run-time costs. With respect to unsafe parallelization, run-time overhead is low and readily surmountable (with eight processors, overhead is 13% for DAG rewrite and 17% for list sort). Third, we study the impact of sharing on dynamic resolution. It is our assumption that sharing occurs in programs, but does so only infrequently. Note that static techniques must conclude that if sharing *can* occur, then it *always* occurs whereas **dr** adapts to the actual sharing present at run time. Empirical evidence suggests that **dr** can tolerate some sharing (4% of nodes shared) and still outperform a sequential implementation.

The next section is an overview of the problem dynamic resolution addresses; it provides the main example used throughout the paper. Section §3 provides definitions and notation. Section §4 explains the idea underlying dynamic resolution. Sections §5 and §6 respectively describe dynamic resolution's static and dynamic components. Section §7 describes the implementations and reports results. Related work is in §8.

2 Overview

Parallel evaluation of program expressions that read (`get`) and modify (`set`) shared data—data that multiple expressions may concurrently access—must prevent read/write and write/write conflicts from violating the sequential semantics of the program. A program's data-sharing characteristics, however, depend on the program's dynamic data structures which often depend on the program's input. Not surprisingly, dynamic data structures are difficult to precisely analyze at compile time [15, 3, 18, 28, 10, 13, 4].

For example, a compiler may statically deduce that a list l of mutable items (called reference values in ML) *may* contain the same element a more than once (thereby sharing a). This forces the compiler to perform operations on individual elements of l sequentially because, at compile time, it is not known when (at run time) or where (in l) such shared elements exist. For a given dynamic instance of l, however, l's elements may be disjoint so that their concurrent access and modification is safe. Furthermore, even if *some* elements of l are identical (shared), others can still be safely modified concurrently if sharing detection and expression scheduling are dynamic. Dynamic resolution performs such sharing detection and expression scheduling at run time.

The `incnode` function of Figure 1 further illustrates the problem and serves as the example of automatic parallelization using dynamic resolution. The `incnode` function operates on dynamic data of the `tree` datatype. Function `incnode`'s single parameter has type `int tree`; that is, internal nodes contain integer reference values in addition to two subtrees. When supplied a leaf node, the `incnode` function does nothing. Otherwise, when supplied an internal node, `incnode` first increments the integer reference value at that node and then recursively descends into the node's `left` and `right` subtrees.

The sequential semantics of the ML language requires that all modification (with `set`) of a reference value r by the expression `(incnode left)` occur before expression `(incnode right)` accesses r. Similarly `(incnode right)` may not `set` r until `(incnode left)` completes its last access of r. Parallel evaluation of `(incnode left)` and `(incnode right)` is however safe when `(incnode left)` and `(incnode right)` access disjoint sets of reference values; *i.e.*, when the dynamic data bound to `left` and `right` do not share structure. Static detection of this parallelism, however, requires the compiler to ascertain whether (and where) sharing exists in `incnode`'s argument.

Static extraction of parallelism from `incnode` is difficult because the `tree` datatype can be used to construct directed cyclic graphs and DAGs as well as trees;[3] *e.g.*, the expression

```
let val n = Node(ref 0,n1,n2)
in Node(ref 0,n,n) end
```

creates a DAG with sharing using the `tree` datatype. Figure 2 depicts valid arguments[4] to `incnode` with and without sharing: a tree and a DAG (the one constructed in the above `let` expression). A naive parallel version of `incnode` that simply evaluates `(incnode left)` and `(incnode right)` concurrently without coordinating internal-node accesses cannot ensure correct results in the references. Because of race conditions, concurrent `get` and `set` operations to shared structure may produce indeterminate values. With naive parallel evaluation, for example, `incnode` applied to the ○ node in the DAG of Figure 2 may produce indeterminate results since expressions can concurrently modify the same reference values—the reference value in and below the ● node.

Even when a data structure contains sharing, it is still possible to (dynamically) discover and utilize parallelism in expressions that access portions of the structure that are not shared; *e.g.*, `incnode` can safely modify the nodes of disjoint trees that are subgraphs *within* a

[1] Our exposition of dynamic resolution uses ML because it abstracts pointer operations and is hence a succinct vehicle for describing dynamic resolution's static analyses. Our experiments, on the other hand, use C because the dynamically-parallel versions produced by **dr** can be meaningfully compared to optimized sequential implementations (also in C).

[2] An *unsafe* program may exhibit indeterminate behavior in the presence of sharing.

[3] Although it may be the programmer's intent to only construct trees with the α `tree` datatype, a compiler must consider *all* structures that a datatype can produce. Hendren [9] and Hummel *et al.* [14, 13] describe programmer annotations for dynamic datatype definitions that express such intent to the compiler.

[4] Cyclic structures cannot be arguments to `incnode` since ML's type system prohibits the introduction of a cycle into a structure of type `int tree`.

```
datatype α tree = Leaf | Node of (α ref * α tree * α tree)

fun incnode Leaf = ()
  | incnode (Node(x,left,right)) = (set x (1 + (get x)); incnode left ; incnode right)
```

Figure 1: The tree datatype and the incnode function. A tree is a Leaf or a Node. Leaf is a nullary constructor; Node is a ternary constructor. A Node contains a mutable reference of type α and two subtrees. Note that DAGs can also be constructed from this datatype. Dynamic resolution can safely evaluate expressions (incnode left) and (incnode right) in parallel since it detects conflicts, due to potential sharing in incnode's argument (of type int tree), dynamically.

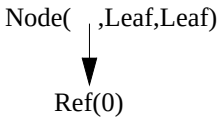
Figure 2: Possible structures of type α tree.

DAG (such as the structure below the ● node in Figure 2). Since static methods that approximate the structure of a program's dynamic data can, in general, only do so imprecisely, it is possible to design a program using incnode that a given static technique cannot parallelize: incnode applied to a DAG whose size and shape (*i.e.*, connectivity) exceeds the static technique's limit of precise approximation (see §8). As another example of such a program, consider the tree and DAG of Figure 2 both reaching an application of incnode via a conditional whose predicate is statically unknown—in this case, static techniques forgo parallelism in incnode since they must conservatively approximate incnode's argument as always containing shared nodes.

The dynamic-resolution technique of this paper can automatically extract parallelism from incnode. We believe incnode to be representative of a large class of database-like functions.

3 Preliminaries

ML datatype constructors build *dynamic values*; that is, reference values, tuples, and recursive data structures created with (non-nullary) data constructors are values that reside in dynamically-allocated storage in the program's *heap*. Denote the heap as $\mathcal{H}$. Implementations represent a program's dynamic values as *nodes* in $\mathcal{H}$. A node $h \in \mathcal{H}$, representing a dynamically-allocated value, contains basic values directly (*e.g.*, integers and nullary constructors) and *links* to other nodes in $\mathcal{H}$. For example, the expression `Node(ref 0,Leaf,Leaf)`, given the `tree` datatype of Figure 1, creates the structure



in $\mathcal{H}$ that consists of two nodes and one link. The heap $\mathcal{H}$ is a directed graph with nodes as its vertices and links as its edges. A node h's in-degree, *in-degree*(h), is the number of links incident on h.

Definition 1 (Simple Node) *A node $h \in \mathcal{H}$ is a* simple *node if in-degree*$(h) \leq 1$.

Definition 2 (Join Node) *A node $h \in \mathcal{H}$ is a* join *node if in-degree*$(h) > 1$.

Join nodes will serve as indicators of potentially-shared dynamic data.

Definition 3 (Path) *A* path *of length n in $\mathcal{H}$ is a sequence of nodes, $\langle h_1, \ldots, h_n \rangle \in \mathcal{H}$ where $n \geq 1$, such that $\forall i$, $1 \leq i < n$, there exists a link from h_i to h_{i+1}.*

Denote the existence of a path from $h \in \mathcal{H}$ to $h' \in \mathcal{H}$ as $h \Longrightarrow h'$. The nonexistence of a path from h to h' is noted $h \not\Longrightarrow h'$. If $h \Longrightarrow h'$, then node h is said to *reach* node h'.

Definition 4 (Simple Path) *A* simple path *of length n in $\mathcal{H}$ is a sequence of nodes, $\langle h_1, \ldots, h_n \rangle \in \mathcal{H}$ where $n \geq 1$, such that $\forall i$, $1 \leq i < n$, there exists a link from h_i to h_{i+1}, and $\forall i$, $1 \leq i \leq n$, node h_i is simple.*

Denote the existence of an simple path from $h \in \mathcal{H}$ to $h' \in \mathcal{H}$ as $h \longrightarrow h'$. The notation $h \not\longrightarrow h'$ denotes that no such path exists. If $h \longrightarrow h'$, then node h is said to *simply reach* node h'.

The relations $\Longrightarrow$, $\not\Longrightarrow$, $\longrightarrow$, and $\not\longrightarrow$ collectively comprise the *reaching relations* for nodes.

Definition 5 (Acyclic Node) *A node $h \in \mathcal{H}$ is an* acyclic node *if all paths from h to h have length* 1.

That is, h is acyclic when it does not lie on a cycle in $\mathcal{H}$. Dynamic resolution's static component determines when a dynamic value is always represented by an acyclic node.

Identification of the free variables of an expression that can bind dynamic values or functions will also be

necessary. As usual, let FV(e) be the set of free variables in e. The *free dynamic variables* of an expression e are:

$$\mathrm{FDV}(e) = \{\, x \in \mathrm{FV}(e) \mid x \text{ can bind a dynamic value} \,\}$$

In ML, an identifier's type indicates whether it can bind dynamic values. The *free function variables* of an expression e are:

$$\mathrm{FFV}(e) = \left\{\, f \in \mathrm{FV}(e) \mid f \text{ can have type } \tau \to \tau' \right\}$$

That is, a free variable f in e is a free function variable if it can be used as a function (*i.e.*, can be applied). Finally, characterize a function f as *true* if all dynamic values accessible in f are either created in f or are parameters to f. Otherwise, f is *untrue*.

Definition 6 (True Function) *A function* $f \equiv (\lambda x.e)$ *is a* true *function if* $\mathrm{FDV}(f) = \emptyset$ *and if* $\forall g \in \mathrm{FFV}(f) \setminus \{f\}$ *the function* g *is a true function.*

That is, f is a true function when f does not contain free dynamic variables and does not apply free functions that contain free dynamic variables. For example, in the function definition

```
fun f (x::xs) =
    let fun g y = (y+1)::xs
    in g x end
```

f is true since $\mathrm{FDV}(f) = \emptyset$ and $\mathrm{FFV}(f) = \{::, +\}$, where f denotes the λ-abstraction bound to f. The infix list constructor (::) and integer addition (+) are true functions. Function g is an untrue function since it accesses the dynamic value bound to xs (*i.e.*, $\mathrm{FDV}(g) = \{\mathtt{xs}\}$, where g is the λ-abstraction bound to g).

4 Dynamic Resolution Property

In this section we describe the basic idea underlying dynamic resolution.

To safely evaluate two expressions e and e' that update a dynamic data structure (*e.g.*, a DAG) in parallel, it is necessary to identify the dynamic data that is potentially reachable by both expressions, and to correctly coordinate the accesses to this data. Initially, evaluation of the two expressions can proceed in parallel with e having priority over e' in the following sense. Upon detection of an access to *any* shared data by e', all further evaluation occurs sequentially; *i.e.*, e' must suspend[5] on an access to shared data and may not restart until e completes. Suspending e' on access to shared data is a means of preserving the language's sequential semantics. Note that in the absence of shared data, dynamic resolution will evaluate both expressions completely in parallel.

The detection of shared data and the coordination of the accesses to this data (*i.e.*, deciding which expression to suspend) occurs dynamically. A dynamic-resolution compiler can automatically insert code into the program text to detect potential sharing at run time; the **dr** run-time system governs which expressions may access shared data. Static analysis is used to select expressions for parallel evaluation whose shared reachable data can always be detected at run time. This analysis relies on the following property concerning paths and nodes.

[5] When a processor suspends an expression's evaluation it need (and must) not idle, but should rather evaluate other runnable threads.

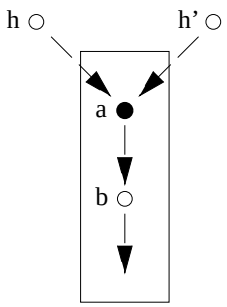


Figure 3: The nonexistence of simple paths from nodes h to h' and from h' to h imply that the shared structure reachable from h and h' (boxed region) is always guarded by a join node (node a). Dynamic resolution detects potentially sharing at run time by detecting join nodes.

Property 1 *Let* h, h' *be nodes in heap* $\mathcal{H}$. *If* $h \not\longrightarrow h'$ *and* $h' \not\longrightarrow h$, *then for all* $h'' \in \mathcal{H}$ *such that* $h \Longrightarrow h''$ *and* $h' \Longrightarrow h''$, *the following relations hold:* $h \not\longrightarrow h''$ *and* $h' \not\longrightarrow h''$.

That is, if all paths from h to h' and from h' to h contain a join node, then all paths from h or h' to any shared node h'' (accessible from both h and h') must contain a join node. This property enables the static selection of program expressions for which all shared data can be detected dynamically.

Figure 3 illustrates the above property. If it is known that node h cannot simply reach h' (and vice versa), then all shared structure reachable from h and h' is always delimited by a join node (node a in the diagram). Note that simple nodes (*e.g*, node b) as well as join nodes may be shared; however, evaluation of an expression will always traverse a join node before encountering a shared simple node, thereby providing a means for detecting sharing dynamically.

Statically, dynamic resolution locates program identifiers that always bind nodes $h, h' \in \mathcal{H}$ such that the above property ($h \not\longrightarrow h' \wedge h' \not\longrightarrow h$) holds. Suppose that the only dynamic values accessible to expression e are those reachable from h. Similarly, suppose that the only dynamic values accessible to expression e' are those reachable from h'. Furthermore, assume e and e' are candidates for parallel evaluation, but potentially conflict (due to read/write or write/write conflicts). If the sequential semantics requires evaluation of e before e', then e and e' may be safely evaluated in parallel with the following restriction: e' may not access any join node until e completes (e, however, may access all—join or simple—nodes that it can reach).

When e and e' do not share structure (*e.g.*, $\nexists h'' \in \mathcal{H}$ such that $h \Longrightarrow h'' \wedge h' \Longrightarrow h''$) then it is possible for e and e' to completely evaluate in parallel with dynamic resolution. Otherwise, evaluation of e' must suspend upon access to a join node—a node potentially shared

with e—until e's evaluation completes. Note that in the presence of sharing, *some* of the evaluation of e' may still be concurrent with that of e.

Dynamic resolution's static component identifies program identifiers that satisfy the conditions of the above property, and uses this information to select expressions for parallel evaluation. The dynamic component detects join nodes and dynamically schedules (suspends and restarts) expressions as necessary.

5 Static Component

Here we first informally describe **dr**'s static component. Sections §5.1–§5.4 supply the technical detail.

Informally, the goal of dynamic resolution's static component is to find two expressions e and e' whose safe parallel evaluation is impeded by `set` operations to dynamic data potentially shared by both expressions. The static component ensures that all shared nodes reachable by e and e' can be detected dynamically. That is, it infers if the **dr** property holds. For such expressions, access to shared data can be detected and correctly coordinated at run time.

Static **dr** parallelization occurs at the function level. For a function f, the static component first identifies the data constructors in f's patterns[6] that always (dynamically) bind acyclic nodes (§3). Static classification of a datatype constructor as acyclic (*i.e.*, it only matches acyclic nodes) enables—in turn—static inference of the reaching relations among a pattern's variables. In particular, static classification of a data constructor as acyclic allows the static inference (§5.2) of strong (*i.e.*, $\not\rightarrow$) reaching relations among the constructor's variables. Such reaching relations permit **dr** parallelization because shared structure accessible from these variables can be dynamically detected by **dr**'s dynamic component (§6). Given such reaching relations, expressions are statically selected and restructured (§5.3) for concurrent **dr** evaluation. Finally, the static component places checks into the program that examine a node's status (join or simple) in expressions that can access its contents (§5.4).[7]

We first describe how to statically determine whether a data constructor in a pattern matches only acyclic nodes, and then how to use this information to infer the reaching relations among a function's variables. Lastly, we describe how to select candidate **dr** expressions and where, in the program text, to place the checks that detect sharing.

5.1 Data-Constructor Classification

A **dr** compiler must statically classify data constructors in patterns as cyclic or acyclic depending on whether the nodes that the constructor dynamically matches can lie on cyclic structures in the heap. Acyclic constructors admit **dr** parallelization; cyclic constructors inhibit **dr** parallelization because the shared structure reachable from a cyclic constructor's variables can not always be dynamically detected. For simplicity, we first assume all patterns in the program contain at most one data constructor—this restriction is relaxed below (§5.2). The form of such a pattern is

$$p \equiv C(x_1, \ldots, x_n)$$

where C is a data constructor and the x_i, $0 < i \leq n$, are variables[8] that are bound when p is matched. For example, the pattern `Node(x,left,right)` of the `tree` datatype (Figure 1) contains the data constructor `Node` and variables `x`, `left` and `right`.

For a pattern p of the form above, **dr**'s static component classifies p's constructor C as cyclic or acyclic. We describe two possible methods of attaining this classification: from static type information (inferred automatically in ML) or from programmer-supplied assertions.

5.1.1 Classification From Static Type Information

Identification of a datatype constructor in pattern p as acyclic is often possible from p's type. In a call-by-value language, cyclic data structures arise only from the re-assignment of a reference value that resides in a dynamic data structure. Furthermore, to introduce a cycle, the contents of this reference value must be a dynamic value; *i.e.*, the reference value must have a dynamic-value type. A pattern's type, therefore, indicates whether the data it can match contains reference values. Hence, type information can identify a pattern's constructors that always match acyclic nodes.

For example, the pattern $p \equiv$ `(Node(x,left,right))` in the `incnode` function (Figure 1) has type `int tree` since the contents of `x` is used in an integer addition. Pattern p's dynamic variables (`left` and `right`) also have type `int tree`. This type information insures that p always dynamically matches an acyclic node in the heap (*i.e.*, p is acyclic) since the reference values in a structure of p's type can only contain integers.

5.1.2 Classification From Programmer Assertion

In a language with polymorphic datatypes (as here), static determination of whether a constructor only builds acyclic nodes is not always possible. Constructors in patterns that cannot be classified as acyclic inhibit parallelization with dynamic resolution because the com-

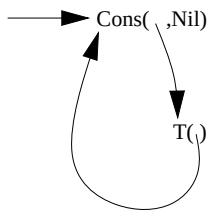


Figure 4: A cyclic list constructed with the conventional Cons constructor.

[6] Patterns (see, *e.g.*, [23]) match dynamic values against datatype constructors, constants, and variables. A pattern gives information about the reaching relations among its variables; it is a *positional* notation that reveals the position of a pattern's variable with respect to the pattern's other variables and constructors.

[7] Note that the contents of a node can be accessed only by matching (*deconstructing*) it in a pattern.

[8] The language's constants (*e.g.*, integers) may also appear in patterns. However, since they are not dynamic values they cannot reach shared data and hence require no special treatment.

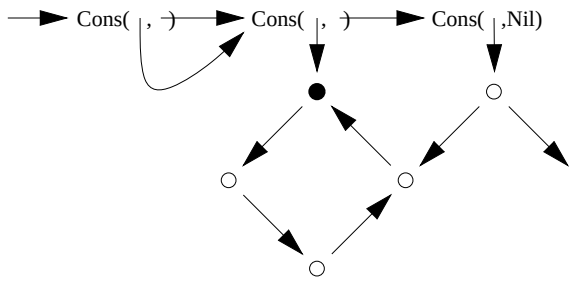


Figure 5: An acyclic list suitable for **dr**. An element of an acyclic list may reach tail elements, list elements may themselves be cyclic structures, and multiple list elements may reach shared structure.

piler will not be able to infer strong (*i.e.*, $\not\longrightarrow$) reaching relations for the variables of cyclic constructors (§5.2 below). For example, the `Cons` constructor of the conventional `list` datatype declaration

```
datatype α list = Nil | Cons of (α * α list)
```

can create cyclic nodes. The program

```
datatype t = T of t list ref | S

let val x = T (ref [S])
    val (T y) = x
in set y [x]; get y end
```

returns a list l whose single element (of type `t`) contains a reference value with contents l (*e.g.*, the list in Figure 4). A compiler cannot generally infer that the list `Cons` constructor matches acyclic nodes. For example, l is a valid argument to the standard `map` function (see, *e.g.*, [23] for its definition)—accordingly, `map`'s pattern does not contain acyclic constructors, and dynamic resolution cannot parallelize the `map` function.

A programmer-supplied assertion can be used to identify acyclic constructors in the presence of polymorphism. Programmers are typically aware of cyclic data since precautions must be taken when traversing it—lists, tuples, trees, and DAGs can often be identified as acyclic by the programmer. We introduce the `acyclic` qualifier for programmer assertion that a datatype's constructors are used only to create acyclic nodes.

Declaration of the acyclic list datatype

```
acyclic datatype α list' =
                  Nil' | Cons' of (α * α list')
```

states that the list nodes constructed with the (acyclic) `Cons'` constructor will not lie on a cycle in the heap. This restricts the spine of a list thus constructed from containing cycle nodes. Elements of an acyclic list, however, may be cyclic structures; elements may also share structure (Figure 5). The list of Figure 4, however, is not a valid acyclic list since it violates the declaration of acyclic. Note that the compiler can not in general detect such violations; incorrect usage of an `acyclic` datatype can cause indeterminate program behavior.

The function `map'` of Figure 6 is an acyclic version of `map` that can only be applied to lists of type α `list'`. The dynamic-resolution technique can be applied here because `Cons'` may only bind acyclic nodes. Hence, the

```
fun map' f Nil' = Nil'
  | map' f (Cons'(x,xs)) = Cons'(f x,map' f xs)
```

Figure 6: The map' function for acyclic lists.

compiler can infer strong reaching relations for its variables (`x` $\not\longrightarrow$ `xs` and `xs` $\not\longrightarrow$ `x`). Even if the higher-order parameter `f` performs imperative `get` and `set` operations it may still be possible to evaluate the expressions in `map'` in parallel (*cf.* [21, 11]).

5.2 Reaching-Relation Inference

Static classification of the data constructors in patterns as acyclic allows the automatic inference of reaching relations among a pattern's variables. When data constructor C in pattern p is acyclic, the nodes dynamically bound to C's variables x_i, $0 < i \leq n$, cannot reach one another via simple paths. That is, when C is acyclic, the compiler can safely infer that $x_j \not\longrightarrow x_k$ for all pairs of C's variables x_j and x_k, where $0 < j,k \leq n$ and $j \neq k$. Proof of this follows. Let $h, h' \in \mathcal{H}$ denote the nodes bound to two of C's variables x_j and x_k (where $0 < j,k \leq n$ and $j \neq k$) when p matches dynamically. When h and h' are the same node ($h = h'$) then h (and h') are join nodes due to the two links from C's node. Alternately, when $h \neq h'$ a simple path cannot exist from h to h' (*i.e.*, $h \not\longrightarrow h'$). Suppose a simple path from h to h' exists. Node h' then has at least two links: one from C's node and one from the node preceding h' on the path from h to h' (this path cannot pass through C's node since C's node is acyclic; hence this path cannot use links from C's node). Since h' has at least two incident links, it must be a join node. This, however, contradicts the supposition. Therefore, a simple path cannot exist from h to h'. Similarly, a simple path cannot exist from h' to h (*i.e.*, $h' \not\longrightarrow h$).

Figure 7 depicts the relationship between an acyclic node a (corresponding to an acyclic constructor) and the nodes x_i and x_j directly accessible from a. If x_i can reach x_j via any path, then that path must contain a join node (x_j). Since the constructor node a is acyclic, the path from x_i to x_j cannot pass through a and hence cannot include the link from a to x_j.

Reaching relations that assert the nonexistence of simple paths between pattern variables enable dynamic resolution—sharing in the structure bound to these variables can be detected at run time because a join node is always encountered before an expression reaches any shared structure.

In Section 5.1 the program's patterns were restricted to contain at most one data constructor. Relaxing this restriction is straightforward and doing so admits nested data constructors in patterns. Without loss of generality, if the constructors C and C' in the pattern

$$p \equiv C(x_1, \ldots, x_n \text{ as } C'(y_1, \ldots, y_m))$$

are acyclic, the reaching relations

$$\begin{array}{ll} x_j \not\longrightarrow x_k & 0 < j,k \leq n \ \wedge \ j \neq k \\ x_j \not\longrightarrow y_k & 0 < j < n \ \wedge \ 0 < k \leq m \\ x_n \Longrightarrow y_k & 0 < k \leq m \end{array}$$

can be inferred. Any path from a variable x_j to a variable y_k cannot be simple because C is acyclic; however,

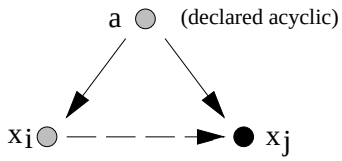


Figure 7: Node a is an acyclic data-constructor node. Nodes x_i and x_j are directly reachable—via a single link—from a. Any path from x_i to x_j is not simple because it always contains a join node (x_j). Such a path cannot use the link from a to x_j since a is acyclic. Black nodes are join nodes; gray nodes represent any (simple or join) node.

a simple path can exist from variable x_n to a variable y_k because the nodes (dynamically) corresponding to the constructors C, C', and to variables y_k may all be simple. This occurs, for example, when p matches an unshared tree.

5.3 Expression Selection

Static analysis propagates the reaching relations induced by a pattern into the pattern's scope. Static selection of expressions for parallelization with dynamic resolution then commences.

Two expressions, e and e', whose safe parallel evaluation is constrained only by read/write or write/write conflicts, are candidates for parallel evaluation using dynamic resolution if they meet three criteria:

1. $\forall x \in \mathrm{FDV}(e)$ and $\forall x' \in \mathrm{FDV}(e')$ the relations $x \not\rightarrow x'$ and $x' \not\rightarrow x$ hold.
2. $\forall f \in \mathrm{FFV}(e)$, f is a true function; and, $\forall f' \in \mathrm{FFV}(e')$, f' is a true function.
3. $\forall x \in \mathrm{FDV}(e)$, x does not contain untrue functions; and, $\forall x' \in \mathrm{FDV}(e')$, x' does not contain untrue functions.

The first criterion requires that all dynamic values bound to the free variables in e cannot reach, via a simple path, dynamic values bound to the free variables in e'. It thereby ensures that all shared data accessible to both e and e' can be detected dynamically §6. The second criterion restricts the functions in e and e' to not have access, through their free variables, to dynamic values other than those available in e and e' (due to the first criterion). The last criterion requires e and e' to not apply untrue functions contained in their accessible dynamic data; it prohibits access to (arbitrary) dynamic values through the free variables of higher-order untrue functions stored in dynamic data. A free dynamic variable's type indicates whether structure bound to it can contain functions.

The `incnode` function contains two expressions that can safely evaluate concurrently using dynamic resolution: $e \equiv$ `incnode left` and $e' \equiv$ `incnode right`. The pattern $p \equiv$ `(Node(x,left,right))` in `incnode` induces the set $\{x \not\rightarrow y \mid x, y \in \{\mathtt{x}, \mathtt{left}, \mathtt{right}\} \wedge x \neq y\}$ of relations for p's corresponding function body. Thus, since $\mathrm{FDV}(e) = \{\mathtt{left}\}$ and $\mathrm{FDV}(e') = \{\mathtt{right}\}$, expressions e and e' meet the first criterion. Furthermore, since e and e' do not apply untrue functions (`incnode` is a true function) and do not have access to data containing untrue functions (`left` and `right` cannot contain functions), expressions e and e' also meet the second and third criteria. Figure 8 reflects the selection of `(incnode left)` and `(incnode right)` for parallel evaluation *provided* that all shared data is dynamically detected and access to this data dynamically coordinated. This detection and coordination is performed by dynamic resolution's dynamic component (§6). The sequence separator $;_{||_{dr}}$ specifies parallel evaluation with sharing detection of the expressions it separates.

5.4 Check Placement

The last responsibility of dynamic resolution's static component is the identification, in the program text, of all heap-node accesses so that sharing can be dynamically detected. In particular, a check to determine if a node is a join node (and hence potentially accessible to other concurrent expressions) is placed immediately before a datum is deconstructed when it matches a datatype constructor (both cyclic and acyclic) in a pattern. Recall that only patterns can be used to deconstruct (access) a dynamic value's contents. Placing a check on every dynamic-value access ensures that sharing (*i.e.*, join nodes) is dynamically detected along any path in the dynamic data that the program may take. These checks examine the status (join or simple) of the node matching the constructor. In Figure 8, the (de)constructor $\overline{\mathtt{Node}}$ must check the status of the nodes it matches before it accesses any of their fields. The result (join or simple) of this check governs the program's subsequent behavior; the full dynamic operation of these checks is discussed below (§6.3).

6 Dynamic Component

Dynamic resolution's dynamic component detects join nodes in the heap at run time. It also maintains a *total order* of all concurrently-evaluating expressions which reflects the evaluation order required by the language's sequential semantics. An expression is dynamically scheduled for concurrent evaluation as a *thread*. Before access to potentially-shared data, an expression examines its position in the total order of threads to determine whether it may access the data or must wait for the evaluation of other expressions (threads earlier in the order) to complete.

6.1 Join-Node Detection

Reference counts are used to dynamically distinguish join nodes from simple nodes. The reference count of a node h counts the number of links from other nodes incident on h—thereby, reference counts reveal information about the heap's structure. A non-link pointer to a node h (*e.g.*, a variable bound to h) is not included in h's reference count because it does not reveal information about the connectivity of the data structure in which

```
fun incnodeDR Leaf = ()
  | incnodeDR (Node(x,left,right)) = (set x (1 + (get x)) ;
                                      (incnodeDR left ;||dr incnodeDR right))
```

Figure 8: The restructured incnode function. The expressions (incnodeDR left) and (incnodeDR right) are evaluated in parallel using dynamic resolution. An overlined constructor requires a check for sharing of the matched heap node before any access to its components.

h resides.[9] A node with a reference count of zero[10] or one is simple; a node with reference count > 1 is a join node. A join node is an indicator of potential sharing because concurrent threads may potentially access the same nodes from a join node. Therefore, coordination of accesses to join nodes is necessary to preserve the program's semantics.

If a thread has access to a simple node, no other thread has concurrent access to this node. Expression selection (§5.3), in cooperation with **dr**'s dynamic component, establishes this invariant. Recall that the static component selects expressions e and e' for parallel evaluation using dynamic resolution only when the evaluation of e and that of e' will always encounter a join node before reaching shared data accessible to either expression.

Building new data (*e.g.*, *consing* an element onto a list) increments reference counts. An assignment to a reference value increments the count of the (dynamic) value being assigned; assignment also decrements the count of the (dynamic) value being overwritten with the following proviso: reference counts are *sticky*—a reference count of two never changes. That is, a join node never becomes simple. Sticky reference counts circumvent the following problem: Suppose an expression e makes a local binding to the contents v of a dynamic reference value r and then reassigns r's value. This would violate the invariant that a simple node is accessible to at most one concurrent thread because the thread evaluating e has access to v (through the local binding) and—if reference counts are not sticky—another thread may now also have (uncoordinated) access to v since the assignment to r removes a link to v and can therefore make v simple. Not decrementing reference counts that are > 1 prevents a thread from inadvertently granting a concurrent thread access to its simple nodes.[11]

Atomicity is *not* necessary for the reference-count increment and decrement operations. This is because of the invariant that simple nodes are not concurrently accessible. Since the reference counts of join nodes are never decremented[12] (*i.e.*, join nodes never become simple), changing the reference count on a join node also requires no synchronization.

[9] The reference-counting scheme required by dynamic resolution is similar to that used for Deutsch-Bobrow *deferred reference counting* [5].

[10] A program can have access to a node with a reference count of zero through pointers (*e.g.*, from local variables) to that node since non-link pointers are not included in the node's reference count.

[11] [11] describes a method that uses the garbage collector to reconstitute a node's reference count that has become imprecise.

[12] Note that a reference count that is *greater* than the actual number of links incident on a node is conservative—such a count may indicate sharing where none exists, but it cannot admit incorrect uncoordinated access to a join node.

6.2 Parallel-Thread Linearization

Dynamic resolution's run-time system imposes a total order on the program's concurrently evaluating expressions (threads). A linked list of *thread descriptors* forms a *linearization* that implements this order on threads. This linearization dictates which thread may access join nodes and which threads must suspend on access to join nodes.

A thread descriptor has three fields: the thread, the thread's run state, and a pointer to the next thread descriptor. A thread can be in one of three run states: *active*, *suspended*, or *finished*. The **dr** run-time system also maintains a single global pointer to the head of the linearization (the *head* thread).

Threads are inserted into the linearization as follows. A thread t evaluating the expression $(e\ ;_{||_{dr}}\ e')$ creates a new thread t' to evaluate e'. Thread t continues with the evaluation of e. A descriptor is created for t' that is inserted into the list directly behind the descriptor of t in the linearization. This will force t to complete before t' may access mutable shared state. Upon creation, a thread descriptor's run-state field is set to *active*.

The linearization is a concurrent data structure—insertions of thread descriptors by different threads occur in parallel without synchronization. As such, the linearization does not unnecessarily sequentialize the program.

6.3 Expression Scheduling

The head thread in the linearization may freely access any node (join or simple) that it can reach. Non-head threads later in the linearization, however, must suspend on access to a join node since it—and all nodes accessible from it—are potentially shared with other concurrent threads. A thread t may not access a join node until it is the head thread; *i.e.*, until all prior threads have completed. On access to a join node a non-head thread sets the run-state field in the its associated thread descriptor to *suspended* and then suspends itself.[13] A non-head thread that completes without accessing a join node sets its descriptor's run-state field to *finished*. When the head thread completes, the next uncompleted (run state $\neq$ *finished*) thread in the linearization becomes the head thread. If this thread is suspended, it is restarted and may now access any join node it can reach—if it is computing, it continues to do so. Since the head thread always makes progress, deadlock cannot occur.

This scheduling scheme preserves the language's sequential semantics because, in an expression $(e\ ;_{||_{dr}}\ e')$, it allows e (and threads created by e) to access all data

[13] The processor that suspends a thread can now proceed to evaluate other (non-suspended) threads.

Incnode(Tree of 2^{21} Internal Nodes)

	Processors							
	1	2	3	4	5	6	7	8
SC	1.36	–	–	–	–	–	–	–
UP	1.96	0.97	0.67	0.52	0.48	0.42	0.39	0.36
DR	2.27	1.16	0.77	0.59	0.48	0.40	0.35	0.33

Quicksort(List of 10^6 Random Integers)

	Processors							
	1	2	3	4	5	6	7	8
SC	26.2	–	–	–	–	–	–	–
UP	29.6	18.5	18.2	15.1	13.6	11.6	11.1	10.7
DR	38.3	23.2	23.3	18.4	16.7	14.8	13.6	12.6

Figure 9: Timing results in seconds for incnode and quicksort. SC is sequential C, UP is unsafe-parallel C, and DR is dynamic resolution.

potentially shared with e' (and threads created by e') *before* it allows e' access to this data. In the absence of sharing, e and e' evaluate concurrently without synchronization under dynamic resolution.

7 Implementation and Results

We have implemented dynamic resolution for two problems coded in C—for the `incnode` program of this paper and for a destructive list quicksort—on an SGI Challenge machine with one gigabyte of shared memory and eight 150Mhz R4400 processors.

The programs were written mostly in C (compiled -O).[14] A few synchronization functions were written in assembler. We compare three versions of each program: a sequential version, an unsafe-parallel version that operates correctly only on data guaranteed not to contain shared structure, and a dynamic-resolution version that correctly handles sharing. For the sequential programs, recursive and non-recursive implementations were compared—the results of the faster implementation are reported here. The unsafe-parallel and the dynamic-resolution versions both use a simple work-queue scheme (*e.g.*, [27, 24]) to distribute work. Each processor has a thread queue into which it inserts the new threads that it creates. When a processor exhausts the work in its queue, it steals work from other processors if possible. The dynamic-resolution versions of the programs differ from the unsafe-parallel versions in three respects. The **dr** versions maintain reference counts on dynamic data, they linearize all threads using a linked-list of thread descriptors, and, when necessary, they conditionally suspend threads on access to potentially shared data (*i.e.*, they check reference counts). This extra work constitutes the overhead of dynamic resolution.

The first program measured is `incnode` (Figures 1 and 8) applied to a balanced tree with 2^{21} internal nodes. The absolute execution times in Figure 9 show that **dr** already outperforms the sequential version with only two processors. Speedup is plotted in Figure 11. With four processors **dr** achieves a speedup of 2.3. Furthermore, the additional overhead with respect to unsafe-parallel is only 13%. Note that with more than five processors dynamic resolution outperforms unsafe-parallel. This is because termination detection with **dr** is trivial. A **dr** computation is complete when the head of the linearization becomes empty. Unsafe-parallel, on the other hand, has no thread linearization so it must reach agreement that all processors have no more work—the time required to reach agreement seems to be a function of the number of processors.

Percentage of Shared Nodes

0.5%	1.0%	1.5%	2.0%	2.5%	3.0%	3.5%	4.0%
0.50	0.64	0.80	0.96	1.05	1.22	1.27	1.38

Figure 10: Timing results in seconds for incnode with varying degrees of sharing in the argument DAG on 6 processors.

Figure 10 contains the absolute execution times for the dynamic-resolution version of `incnode` applied to a DAG d of 2^{21} internal nodes using six processors. Sharing was simulated by selecting nodes of d at random and setting their reference count to two. Since the choice of which nodes to share can greatly influence performance, the reported numbers are the mean of ten trials, each selecting a different set of random nodes. We believe that these numbers will improve for the final version of the paper since our current implementation moves much more data than necessary when suspending a thread.

The second program we measured is a destructive (in place) quicksort for lists. This is another program in which dynamic resolution automatically finds expression-level parallelism.[15] The timing results for the three versions (sequential, unsafe-parallel, dynamic-resolution) are in Figure 9. Again, dynamic resolution outperforms the sequential program with two processors. At four processors, **dr** incurs overhead of 21% relative to the unsafe-parallel version; at eight processors, this overhead is 17%.[16]

It is interesting to note that dynamic-resolution overhead (from counting references, linearizing threads, and checking for shared data) is itself "parallel"; that is, its cost distributes over the available processors.

8 Related Work

Dynamic resolution was first suggested in [12] and further developed in [11]. The early ML implementations were however not effective since they did not outperform sequential C. In this paper we validate dynamic resolution as effective by demonstrating that C programs implemented with dynamic resolution surpass—in absolute performance—their optimized sequential counterparts.

[14] Although our measurements are of C programs, it is not clear that the requisite analyses for dynamic resolution's static component are practically feasible for dynamic resolution of C. In contrast, ML's pointer safety, strong typing, and pattern matching requires only simple, mostly local, analyses.

[15] See [11] for a listing of this program.

[16] For quicksort, unlike `incnode`, the unsafe-parallel version need not spend time reaching agreement for termination since the quicksort function returns a value (the sorted list). Unsafe-parallel can terminate as soon as this value is produced.

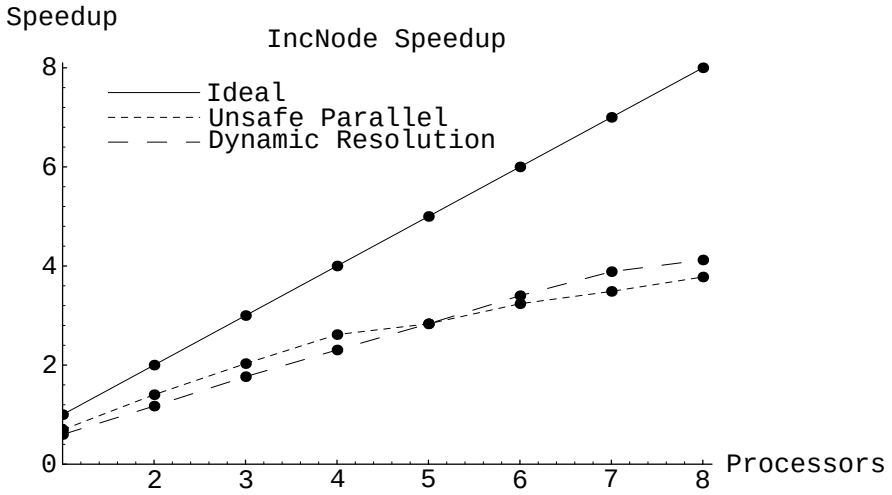


Figure 11: Parallel speedup for dynamic resolution and unsafe-parallel applied to a tree of depth 22. The text explains why dynamic resolution outperforms unsafe-parallel on more than five processors.

Work related to the parallelization of languages with dynamic data structure falls into two classes: static–dynamic techniques and solely static techniques.

Tinker and Katz propose to model a Scheme implementation as a database in their *ParaTran* system [26, 16]. Concurrent reads and writes are concurrent transactions under this model. Evaluation in *Paratran* proceeds optimistically. Upon dynamic detection of a conflict, the computation must be *rolled back* to a point where the linear access order is intact. Reversing large computations is expensive. By contrast, our dynamic-resolution technique suspends a conflicting expression and can immediately evaluate other pending work. The amount of run-time information required by dynamic resolution is also small (reference counts, thread descriptors) in comparison to the complex time-stamps *Paratran* maintains for heap objects. *Paratran* has not produced effective speed-up.

Lu [19], and Lu and Chen [20], describe run-time methods for parallelizing loops with indirect array accesses (in Fortran and C) and (restricted) pointer accesses (in C). Their methods pre-execute a loop nest at run time to find *data dependences* between program statements in the loop. The compiler, using static analysis, generates a *scheduler* for the loop's iterations. At run time, this scheduler dynamically records references to dynamic data and, using the reference patterns thus collected, allocates loop iterations to individual processors for parallel evaluation. Unlike dynamic resolution, their method does not handle procedure calls, modification of existing data structure links, or the allocation of new data. Their method also depends on extensive pointer analysis (*e.g.*, [17]) that has been shown to be expensive in practice [17].

Harrison's PARCEL system [7, 6] and Larus's Curare [17] seek parallelism in sequential Scheme programs using static analyses. Both systems compute bounded approximation information intended to allow parallelization of non-interfering imperative expressions. For large, irregular data, such bounded approximation leads to overly conservative parallelization—if sharing *can* occur dynamically, PARCEL and Curare assume that it *always* occurs. In the presence of sharing, therefore, some of the parallelism that dynamic resolution finds must elude these systems.

Hendren [9, 8] addresses the problem of parallelizing programs with recursive data structures with an algorithm for estimating the relationships between accessible nodes in a dynamic data structure. Relationships thus attained are then used to (statically) detect interference between program statements. Her analysis finds parallelism when it can statically determine that trees, rather than DAGs, always reach a given program point. Therefore, this analysis cannot discover parallelism in DAGs—the type of parallelism that dynamic techniques find. Hendren's analysis also detects when a set of *handles* (pointers) into a dynamic data structure cannot reach common structure. Relationships between handles are similar to the reaching relations that dynamic resolution obtains from pattern matching (§5.2). Hendren's analysis can potentially perform the task of dynamic resolution's static component in languages that do not support patterns.

Many approaches to the general problem of static pointer analysis have been designed (*e.g.*, [15, 3, 18, 10, 28, 4]). These approaches usually use a variety of bounded approximations (of the heap itself, of the store of heap variables, or a combination of the two). Bounded approximations are conservative—they must account for all possible configurations of the program's dynamic data. In contrast, dynamic techniques adapt to individual instances of individual dynamic structures at run time. Static pointer analyses also require expensive interprocedural analyses that curtail their practical use (*cf.* [17, 25]). With dynamic resolution, interprocedural information (*i.e.*, sharing information) dynamically propagates into functions at run time.

9 Conclusion

We have described a run-time technique called dynamic resolution. Dynamic resolution extracts parallelism from program procedures that destructively manipulate DAGs (that is, for procedures that modify a DAG's edges or update the contents of its vertices). Dynamic resolution is the first parallelization technique that can automatically and effectively parallelize the destructive DAG rewrite and destructive list quicksort problems. On a contemporaneously-fast shared-memory parallel machine, the run-time overhead of this technique is small relative to unsafe parallelization. Furthermore, dynamic resolution provides significant speedup over the sequential implementations.

Used in the context of a pointer-safe language such as ML, dynamic resolution can find such parallelism automatically while preserving the language's sequential semantics. In unsafe languages such as C, additional programmer assertions are necessary for safe dynamic resolution. We believe that dynamic parallelization in general, and dynamic resolution in particular, are viable approaches for language parallelization.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310. Association for Computing Machinery, June 1990.

[4] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conference on Programming Language Design and Implementation*, pages 230–241. Association for Computing Machinery, June 1994.

[5] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(7):522–526, July 1976.

[6] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[7] W. L. Harrison and D. A. Padua. PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp. In *International Conference on Supercomputing*, pages 527–538, July 1988.

[8] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, August 1990.

[9] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[10] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation*. Association for Computing Machinery, June 1989.

[11] L. Huelsbergen. *Dynamic Language Parallelization*. PhD thesis, University of Wisconsin–Madison, August 1993.

[12] L. Huelsbergen and J. R. Larus. Dynamic program parallelization. In *Lisp and Functional Programming*, pages 311–323. Association for Computing Machinery, June 1992.

[13] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[14] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Conference on Programming Language Design and Implementation*, pages 218–229. Association for Computing Machinery, June 1994.

[15] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Symposium on Principles of Programming Languages*, pages 244–256. Association for Computing Machinery, January 1979.

[16] M. Katz. ParaTran: A transparent, transaction based runtime mechanism for parallel execution of Scheme. Technical Report LCS/TR-454, MIT, July 1989.

[17] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley, Computer Science Division, May 1989.

[18] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Conference on Programming Language Design and Implementation*, pages 21–34. Association for Computing Machinery, June 1988.

[19] L. Lu. *Loop Transformations for Massive Parallelism*. PhD thesis, Yale University, November 1992.

[20] L. Lu and M. C. Chen. Parallelizing loops with indirect array references or pointers. In *Preliminary Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[21] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Symposium on Principles of Programming Languages*, pages 47–57. Association for Computing Machinery, January 1988.

[22] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[24] E. Mohr, D. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Lisp and Functional Programming*, pages 185–197. Association for Computing Machinery, June 1990.

[25] P. E. Pfeiffer. *Dependence-Based Representations for Programs with Reference Variables*. PhD thesis, University of Wisconsin-Madison, 1991.

[26] P. Tinker and M. Katz. Parallel execution of sequential Scheme with ParaTran. In *Lisp and Functional Programming*, pages 28–39, July 1988.

[27] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[28] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Symposium on Principles of Programming Languages*, pages 83–94. Association for Computing Machinery, January 1980.