Deferred Data-Flow Analysis

Shamik Sharma
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Anurag Acharya
Dept. of Computer Science
University of California
Santa Barbara, CA 93106

Joel Saltz
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Technical Report TRCS98-38
Dept of Computer Science
University of California, Santa Barbara

December 1998

Abstract

Loss of precision due to the conservative nature of compile-time dataflow analysis impacts a wide variety of optimizations. In this paper, we propose a general framework which combines compile-time analysis with limited runtime analysis to improve the precision of dataflow information at performance-critical points in the program. This technique, which we refer to as *deferred dataflow analysis (DDFA)*, performs most of the analysis at compile-time but defers the final stages of the analysis till runtime when additional information is available. We present algorithms for construction of *region* summary functions and for composition and application of these functions. Dividing the analysis in this manner raises two concerns: (1) is it possible to generate correct and compact summary functions for *regions*? (2) is it possible to correctly and efficiently compose and apply these functions at *run-time*? To address these concerns, we show that DDFA terminates, is safe and that its results are at least as good as a compile-time meets-over-all-paths solution. To address concerns regarding the size of the data-structures used to communicate information from the compiler to the runtime system, we present space requirements of DDFA in the context of three different backward data-flow analyses. To address concerns regarding the runtime overhead of DDFA and to demonstrate the ability of DDFA to optimize heavy-weight operations, we present results from the application of DDFA for two of these analyses.

1 Introduction

Compile-time dataflow analysis is inherently conservative as it combines information from all possible execution paths. Loss of precision due to the conservative nature of compile-time dataflow analysis is a general problem and impacts a wide variety of optimizations. The obvious alternative to compile-time dataflow analysis, namely runtime analysis, can provide accurate dataflow information. However, if runtime dataflow analysis is used for the *entire* program, it can add a large overhead which usually outweighs the advantage that can be gained using the increased precision. The performance of programs, however, increasingly depends on the performance of individual *heavy-weight* operations such as communication, disk I/O, just-in-time compilation, garbage-collection, interactions with the operating-system, interactions with other (possibly remote) programs etc. These operations are expensive; avoiding or combining even a small number of them (by using improved dataflow information) has the potential of providing significant benefit.

In this paper, we propose a general framework which combines compile-time analysis with limited runtime analysis to improve the precision of dataflow information at performance-critical points in the program. This technique, which we refer to as *deferred dataflow analysis (DDFA)*, performs most of the analysis at compile-time but defers the final stages of the analysis till runtime when additional information is available.

DDFA divides the task of dataflow analysis into three stages. The first two stages are performed at compile-time and third is performed at run-time. The first stage analyzes the control-flow graph to identify *forks* whose direction can be determined at the points of interest in the program. Forks are program-points from which control can flow in more than one direction - e.g. conditionals, procedure returns, switch statements, etc. It uses this information to divide the control-flow graph into *regions* such that each region contains at most one such fork. The second stage analyzes these regions and constructs a *summary transfer function* for each *region* describing how it affects dataflow attributes flowing through it. The third stage is performed on-demand when control reaches the points of interest. It uses available information about future control-flow to determine the set of reachable regions and completes the analysis by stitching together the summary transfer functions for these regions.

Two-phase techniques using summary functions have previously been used for *interprocedural* flow analysis (e.g. Sharir et al [26] and Duesterwald et al [8]). In these techniques, the first phase computes a summary transfer function for each procedure and the second phase applies these functions to obtain interprocedural dataflow properties. These techniques perform both phases at compile-time. DDFA differs from these techniques in three important ways. First, DDFA computes summary transfer functions for *regions*, which are not necessarily procedures - this introduces complications as regions can overlap while procedures cannot. Second, DDFA applies these functions at runtime, using runtime control-flow information to improve the precision of the dataflow information. It is therefore important to have compact representations for these functions, so that intermediate analysis results can be efficiently passed from the compiler to the runtime system. Third, DDFA computes multiple summary functions for each region and uses control-flow information available at runtime to choose between them.

We present algorithms for construction of *region* summary functions and for composition and application of these functions. Dividing the analysis in this manner raises two concerns: (1) is it possible to generate correct and compact summary functions for *regions*? (2) is it possible to correctly and efficiently compose and apply these functions at *run-time*? We address these concerns in two ways. To provide the theoretical underpinning for DDFA, we prove that it terminates, that it is safe and that its results are at least as good as a compile-time meets-over-all-paths solution. To address concerns regarding the size of the data-structures used to communicate information from the compiler to the runtime system, we present space requirements of DDFA in the context of three different backward data-flow analyses. To address concerns regarding the runtime overhead of DDFA and to demonstrate the ability of DDFA to optimize heavy-weight operations, we present results from the application of DDFA for two of these analyses.

Note, that DDFA is applicable for both forward and backward dataflow analyses. Given the limited space, we restrict ourselves, in this paper, to a description of DDFA for backward analyses. We briefly mention how DDFA can be extended to forward analyses in Section 5.

2 Overview of DDFA

In this section, we present a brief overview of DDFA including terms and concepts which are needed to follow the rest of the paper. For conciseness, we limit ourselves to intuitive description of terms; for details we refer the reader to [27].

As mentioned in the introduction, DDFA is targeted towards optimization of heavy-weight operations. These operations are referred to as *op*s and the nodes representing them in the control-flow graph as *op-node*s. Each op-node induces an *op-domain*. The op-domain corresponding to the operation O consists of the nodes that are reachable from O without passing through another *op*. O is referred to as the *entry-point* to the op-domain; the exits from an op-domain are either other op-nodes or program exit-points.

Each op-domain has zero or more *forks*. A fork is said to be *lossy* with respect to a particular dataflow analysis if all its incident edges in the control-flow graph do not have the same dataflow attributes (after fix-point has been reached). Meet operations at such forks result in loss of information. A fork is said to be *predictable* with respect to an op-node if the direction of the fork is always determined before execution reaches the op-node. Forks that are both *lossy* and *predictable* are said to be *lp*-forks. Since a conservative analysis loses information at *lp*-forks *and* their direction is known when control reaches the *op*-node in question, it is potentially beneficial to defer the meet operation for them.

Each *lp*-fork induces an *lp*-region. An *lp*-region is the collection of nodes that can be reached from an

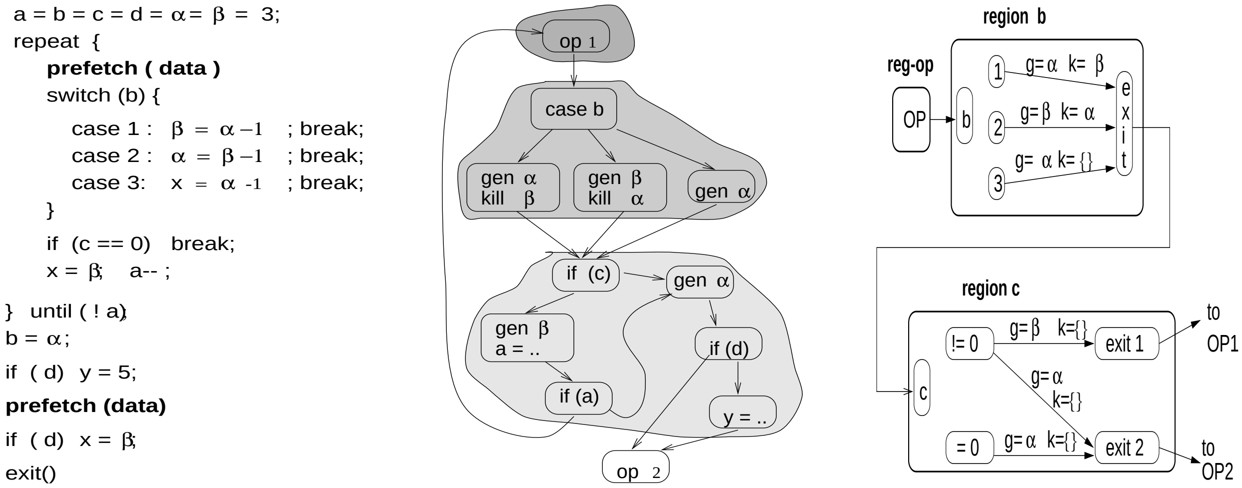

Figure 1: (a)A program fragment to illustrate loss of precision in conservative analysis of distributed shared memory code. The code accesses two remote variables α and β. (b) Control-flow graph for the program fragment in (a). The two op-nodes `op1` and `op2` in the CFG, correspond to the two calls to `prefetch`. The shaded regions represent the *lp*-regions in this code fragment. (c) Summary functions computed for each *lp*-region at compile-time. The gen and kill sets are represented by `g` and `k`.

lp-fork without passing through an *lp*-fork or any of the exits of the op-domain. An *lp*-region may contain other forks. These forks, however, are either not predictable or not lossy. Since we will not have better information about these forks at runtime, deferring the data-flow analysis for these forks will provide no advantage. The shaded regions in Figure 1 show the *lp*-regions.

Figure 1 provides an illustration using the problem of *bulk-prefetching* for distributed shared memory programs. Fetching data in small chunks can be expensive and prefetching data in bulk can significantly improve performance [24]. A commonly used conservative approach is to prefetch only the data that will *definitely* be required along all paths. This prevents needless communication, but may limit the effectiveness of prefetching. In this case, a compile-time analysis indicates that neither of the remote variables α or β is required along *all* paths from the first call to `prefetch()`. If, however, it were possible to determine which *one* of the paths would be actually taken (in a given iteration), the appropriate value(s) (α, β or $\{\alpha, \beta\}$) could be prefetched. In this example, only the forks corresponding to `case(b)` and `if(c)` are *lp*-forks; the fork at `if(a)` is lossy but not predictable (due to the assignment to `a`); the fork at `if(d)` is predictable but not lossy.

The first stage of DDFA identifies the op-nodes in a control-flow graph, constructs the op-domains, identifies the lossy and predictable forks within each op-domain and constructs the *lp*-regions.The second stage analyzes each *lp*-region and produces one or more *summary* transfer functions. The third stage is performed when control reaches the op-node at the entry to an op-domain. At this point, the directions of all *lp*-forks in the op-domain are known. This information is used to select the appropriate summary function for each *lp*-region in the op-domain. The runtime *stitcher* uses the information about the directions of *lp*-forks to determine the sequence of *lp*-regions that lie along the path that will be traversed and applies the corresponding summary functions. The links among the *lp*-regions of an op-domain may contain back-edges; this may require the stitcher to iterate. In such cases, one has the choice of trading off execution time and analysis precision by choosing between performing the iterations and falling back to using the conservative dataflow properties computed at compile-time. Note that iterating may not be expensive as the number of *lp*-regions in a op-domain is expected to be small.

3 The Basic DDFA Framework

In this section, we describe the basic DDFA framework. We first present the terminology and the background information. Next, we describe the compile-time algorithm for the construction of summary transfer functions (the *builder*). Finally, we present the stitching algorithm that uses runtime information to select and apply these functions (the *stitcher*). For now, we limit ourselves to intra-procedural analysis of a first-order language in which the only forks are conditionals that test a single variable. In addition, we make the assumption that dataflow analysis for each *op* is restricted to the corresponding op-domain; i.e. we do not try to merge information from multiple op-domains. These restrictions are for expositional simplicity. In section 5, we extend the framework to include interprocedural analysis and dynamic propagation of dataflow information across op-domains.

3.1 Terminology and Background

We characterize a global backward dataflow framework by the pair (L, F), where L is a lattice of attribute information and F is a semi-lattice of monotone and distributive transfer-functions from L to L. L is assumed to be a complete lattice with a partial order $(\leq)$, top $(\top)$ and bottom $(\perp)$ elements and a meet operator $(\sqcap)$. Also L is assumed to be well-founded, i.e. all chains in the lattice L are of finite height. F is assumed to be ordered as follows: for $g_1, g_2 \in F$, $g_1 \geq g_2$ iff $g_1(x) \geq g_2(x)$ for all $x \in L$. Further, we assume that F is the smallest set of functions acting on L which includes an identity function f_{id}, $f_\top$ and $f_\perp$[1] and is closed under functional composition $(\circ)$ and functional meet $(\wedge)$. We need to make these assumptions regarding F to construct the region's summary transfer function for a region, because flow analysis is carried out on transfer-functions rather than data attributes. Dataflow frameworks that satisfy these requirements are said to be *closed* [25]. Closed frameworks are common in dataflow analyses; in particular, all *elimination* algorithms [2, 11, 25] require closed frameworks.

Each op-domain is represented as a separate control-flow graph $G = (N, E, R, X)$ where N is the set of nodes, E is the set of edges, R is the root (i.e. the op-node), and X is the set of nodes at which control flows out of the op-domain (other op-nodes or program exit-nodes). It is possible that $R \subseteq X$ due to a back-edge (for example, see Figure 1). With each edge $(n, m) \in E$, we associate a transfer-function $f_{(n,m)} \in F$ which represents the change of data attributes from the entry of block m, up through node n to the entry of block n (backward-propagation). The sets $pred(n)$ and $succ(n)$ denote the immediate predecessors and successors of node n respectively.

A fork v is said to be *lossy* if the information propagating up to v from its successors is not the same along every path. Each fork has a predicate that determines which of the successor nodes control will be transferred to after the fork is executed. The set of nodes that contain reaching definitions of the variable tested in the fork-predicate are called fork-determinants. A fork v is said to *predictable* at the op-node *op* if there is no fork-determinant on any path between *op* and v. Forks that are both lossy and predictable are referred to as *lp*-forks. Note that, since op-domains can overlap, a fork-node may be contained in multiple op-domains. A lossy fork is lossy in every op-domain but the same fork may be predictable in one op-domain and not the other, as its fork-direction may be predictable at one op-node and not at the other.

Just as an op-node induces an op-domain in the control-flow of the program, an *lp*-fork induces an *lp*-region in every op-domain it is contained in. An *lp*-region within a given op-domain is identified by the *lp*-fork v that induces the region. (We use the identifier v to refer to the *lp*-fork as well as the corresponding *lp*-region). Each *lp*-region has its own subgraph $G_v = (N_v, E_v, R_v, X_v)$. The children of the *lp*-fork v are called *predictable children* and are belong to a set $W_v = \{m : (v, m) \in E_v\}$. We also define E^*_v to be the subset of E_v which contains all edges except those between the *lp*-fork v and its predictable children (W_v) (i.e. $E^*_f = E_f - \{(v, m) : (v, m) \in E_f\}$).

Each region also has a set of predecessor and successor regions: $succ_v$ is the set of all *lp*-regions that are successors of v whereas $succ_v(d)$ is the set of *lp*-regions that can be reached if the fork-direction is known to be d. Similarly, $pred_v$ is the set of actual predecessors of v whereas $pred_v(p)$ is the set of exit nodes of p at which control will flow from p to v.

[1] $\forall x, f_{id}(x) = x, f_\top(x) = \top, f_\perp(x) = \perp$

For each node $n \in N$, we define $paths(n, e)$ as the set of all paths within G which lead from a node n to an exit node $e \in X$. Similarly, within each *lp*-region, the $paths_v(n, e)$ contains all paths from between $n \in N_v$ and $e \in X_v$ that lie entirely within G_v.

3.2 Constructing summary functions (the *builder*)

In this section, we describe how the *builder* constructs the summary transfer function for each *lp*-region. We show that for most typical dataflow analyses, summary functions can be constructed efficiently and compactly.

For an *lp*-region v, the term $\phi_v(n, m)$ is used to denote a function that summarizes the effect of dataflow operations along all paths between nodes n and m in the region. For each region, we construct a set of these summary functions, $\Phi_v = \{\phi_v(w, x) \mid w \in W_v, x \in X_v\}$. $\phi_v(w, x)$ summarizes the dataflow operations along on all paths between w, a predictable successor of the fork node v and x, an exit node of the region. The following set of equations describe how $\psi_v(w, x)$ can be constructed. For each $x \in X_v$, we have a separate set of equations :

$$
\begin{aligned}
\psi_v(x, x) &= \quad f_{id} \\
\psi_v(n, x) &= \bigwedge_{(n,m) \in E_v^*} (f_{(n,m)} \circ \psi_v(m, x)) \qquad \forall n \in (N_v - \{v\})
\end{aligned} \tag{1}
$$

This set of non-linear equations operates over the function-lattice F. Since F is assumed to be closed under composition ($\circ$) and meet ($\wedge$), we can solve the system of equations by starting with : $\phi^0_{(n,x)} = f_\top$ for each $n \in (N_v - \{v\} - \{x\})$ and iterating to obtain new approximations to the ϕ's until convergence. Note that Eqs. (1) do not compute a summary function for the fork-node v itself. We do not compute $\phi_v(v, x) = \bigwedge \phi_v(w, x)$ because this lossy meet can be avoided by deferring it until runtime. When the successor (say w_i) of the fork-node v becomes known the meet can be eliminated by simply assigning $\phi_v(v, x) = \phi_v(w_i, x)$. In order to defer the meet until runtime, each *lp*-region stores $\phi_{(w,x)}$ for each $w \in W_v$ and each exit node $x \in X_v$ that is reachable from w. In the worst case, we may need to store $|X_v| \times |W_v|$ functions for each region v. [2]

For the above procedure to work efficiently, the transfer functions must allow compact representations and it must be possible to perform the composition and meet operators on the function-lattice. In general, these conditions need not hold. Fortunately, however, these conditions do hold for most commonly-used dataflow analysis frameworks. As pointed out in [26], the class of dataflow problems that allow compact and simple representations are the same class that allow elimination algorithms [2] (because these algorithms also need to perform functional meets and compositions). This family includes the classical "bit-vector" dataflow problems. In particular, there exists a common class of dataflow problems, called *1-related* problems, which always have compact representations and allow efficient manipulation [25]. For 1-related problems it is easy to perform function composition and functional meets; we show how this can be achieved using `gen` and `kill` sets.

Each basic block i in a control-flow graph G has an associated transfer function $f_i \in F$, that can be represented in terms of two sets - a set (g_i) that contains a list of attributes which are mapped to $\top$ by the block and a set (k_i) that lists the attributes which are mapped to $\perp$ by the block. Other attributes are assumed to be unaffected. g and k are supposed to correspond to the `gen` and `kill` sets respectively.[3]

Composition of gen-kill sets: If $f_1 = (g_1, k_1)$ and $f_2 = (g_2, k_2)$, then their composition $f_{12} = f_1 \circ f_2$ is defined as $f_{12} = (g, k)$, where $g = (g_1 \cup (g_2 - k_1))$ and $k = (k_1 \cup (k_2 - g_1))$. f_{12} summarizes the effects of applying f_2 on any attribute and then applying f_1 on the result. Intuitively, the `gen` set of f_{12} should contain all elements in the f_1's `gen` set as well as those in f_2's `gen` set that are not killed by f_1.

Meet of gen-kill sets: If $f_1 = (g_1, k_1)$ and $f_2 = (g_2, k_2)$, then their meet $f_{12} = f_1 \wedge f_2$ is defined as $f_{12} = (g, k)$, where $g = (g_1 \cap g_2)$ and $k = (k_1 \cup k_2)$

[2] We really only need to store entries for those (w, x) pairs which are connected in G_v (i.e. if $(w, x) \in paths_v(w, x)$).

[3] To be precise, this depends on the orientation of the lattice. There are cases where the intuitive mapping may be reverse. For instance in live-variable analysis, the null set is the top element in the lattice; hence, in that case, g represents the `kill` set and k represents the `gen` set.

region	predicate check	fork dir.	Ω	pred-ecessors
OP	×	×	Cache	×
b	b = ?	?	?	?
c	c = ?	?	?	?

Region-table (op_1)

fork directions (D).	φ
(b = 2, c = 0)	{ β }
(b = 1, c != 0)	{ α }

Cache

(dir, exit)	{ gen },	{ kill }
b = 1, c	{ α }	{ β }
b = 2, c	{ β }	{ α }
b = 3, c	{ α }	{ }

summary-function-table b

(dir, exit)	{ gen },	{ kill }
c = 0, op2	{ α }	{ }
c != 0, op1	{ β }	{ }
c != 0, op2	{α, β }	{ }

summary-function-table c

Figure 2: Data-structures created by the builder and passed to the runtime stitcher for the op-domain in the running example. The fields marked ? are to be filled in by the runtime stitcher. The cache has been pre-filled by the builder for two of the paths.

3.2.1 Data structures

The *builder* passes the results of its analysis to the *stitcher* via pre-initialized data-structures in the data segment of the compiled code. For each op-domain, it creates a *region*-table which contains one entry for every region within the op-domain. Each entry contains: (1) a pointer to a *summary-function* table for each region, (2) a pointer to the code that must be executed to determine the directions of the *lp*-forks in the region, (3) scratch space for intermediate dataflow attributes during stitching, and (4) a place-holder for the predecessor set of each region. The predecessor set cannot be computed accurately at compile-time because any given set of fork-directions may render several paths infeasible, changing the predecessor relationships. Each *summary-function* table is indexed by a (fork_direction, exit_node) pair and contains all summary functions for the region. Since summary functions exist only for a few of the (fork_direction,exit_node) pairs,[4] summary-function tables are stored in a sparse format. It also contains the successor information for every *lp*-region and every exit in a region (the columns labeled `exit` in the summary-function tables). Each entry corresponds to the parameterized successor set $succ_v(d)$ of region with the fork_direction d. This information is used by the stitcher to compute the corresponding predecessor sets ($pred_v(d)$).

Figure 2 illustrates these data structures. It shows the data structures generated for the example presented in figure 1. The region associated with the op-node is trivial and needs no summary functions. The region associated with `switch(b)` has three fork-directions all leading to the same exit. The summary functions corresponding to the three (fork_direction, exit_node) pairs in this region need only trivial functional compositions and no functional meet (no other forks in the region). Functional meet is needed for the `if(c)` region: eg., at `if(a)` $(\phi_{left} \wedge \phi_{right} = (g_{left} \cap g_{right}, k_{left} \cup g_{right}) = (\{\} \cap \{\beta\}, \{\} \cup \{\}) = (\{\beta\}, \{\}))$.

The builder also generates a cache structure for holding previously computed dataflow properties. This cache is associatively indexed on the settings of all the fork-directions in the op-domain (the directions are hashed together to yield a single unique key). This cache is used to exploit control-flow locality in loops. Quite frequently, a large number of iterations in a loop follow the same path in the loop body[24]. The compiler primes the cache by statically computing the dataflow properties for the “likeliest” choices for all fork-directions (statically guessed).

3.3 Applying summary functions (the *stitcher*)

In this section, we describe how the *stitcher* selects the set of summary functions to apply and how it composes these functions. We first describe the operation of the stitcher in terms of the data-flow equations it solves. Then we describe the stitcher algorithm and illustrate its operation with an example.

The stitcher is invoked every time execution reaches an op-node. It first applies the tests to compute the set of fork-directions, $D = \{d_1, ...d_k\}$, where d_v is the predicted (and guaranteed) direction for the *lp*-fork v. Direction is denoted by assigning d_v to the appropriate child node of fork v. Our definition of predictability implies that these fork-directions hold as long as control remains in the op-domain.

Note that each *lp*-region is a multiple-entry, multiple-exit hyper-block. The builder has already prepared for this complexity by keeping separate summary functions for each (entry, exit) pair. Once fork-directions

[4] A summary function is created for a (fork_direction,exit_node) pair iff the exit_node is reachable after taking the fork_direction.

are known at runtime, the stitcher determines the successor and predecessor sets of each region, $succ_v(D)$ and $pred_v(D)$. These are the regions that can succeed and precede region v, given the set of fork-directions D.

We can now describe the backward flow equations solved by the stitcher.

$$
\begin{array}{lll}
\Omega(x) = & \perp & \forall x \in X \\
\Omega(v) = & \sqcap_{u \in succ_v(D)} \phi_v(d_v, u)(\ \Omega(u)\ ) & \textit{if} \;\; v \;\; \textit{is a} \;\; \textit{lpfork}
\end{array}
\qquad (2)
$$

Each *lp*-region v takes dataflow attributes ($\Omega(u)$) from a successor *lp*-region u and determines which of its own exit-nodes the successor u connects to (there may be more than one connections from v to u). For each such exit-node x , we look up the appropriate summary function to apply $\phi_v(d_v, x)$. Applying this summary function provides the resultant dataflow attributes at the top of *lp*-region v due to the paths from fork v to fork u through the exit-node x. This process is repeated for other exit-nodes between v and u, and then continued for other successor *lp*-regions of v. The meet of all these dataflow attributes gives us $\Omega(v)$ dataflow attributes at *lp*-fork v. The process is iteratively continued until convergence.

Stitcher algorithm and example Figure 3 presents the stitcher algorithm. The stitcher consists of two parts. The first part handles the determination of fork-directions (using the code pointers in the op-domain table) and caching of previous results. The second part is a simple worklist algorithm (over regions instead of basic blocks). The primary difference from other worklist algorithms is that it recomputes the predecessor sets on each invocation. This is done since it is invoked each time with different fork-directions and a different set of exits may be reachable.

We illustrate the operation of the stitcher using the example in Figure 1. In this example, `prefetch()` is called three times (once per iteration). On the first iteration through the loop, the values of `b` and `c` are `3` and `3` respectively. Since the cache doesn't contain an entry for these values, the worklist algorithm is invoked. It first sets up the predecessor relationships - trivial as the region C has one predecessor region (region B) and connects to a single exit node of its predecessor. The algorithm selects C's summary function for fork-direction `(c != 0)` and B's summary function for direction 3. It applies C's function to $\perp$ and feeds the result ($\{\beta\}$) to B's function resulting in $\{\alpha, \beta\}$. The results of this analysis are then stored in the cache and supplied to the prefetcher. The next two iterations have the same fork-directions and these results can be used directly from the cache.

4 Soundness, Correctness and Accuracy

In this section, we provide the theoretical underpinnings for DDFA. We show that: (1) DDFA terminates; (2) DDFA is safe; and (3) DDFA computes a solution that is at least as accurate as a compile-time meet-over-all-paths solution. Our proofs are largely adapted from the proofs of the two-phase interprocedural analysis used in [26]. We first prove a lemma about properties of paths in op-domains. We shall use this lemma to show that DDFA is safe and that its results are at least as good as a meet-over-all-paths solution.

Path Decomposition Lemma: A *region-segment* is defined as a segment that begins at a *lp*-fork node and ends at one of the exit nodes of the region induced by that fork. Let p be a *valid path* that begins at a *lp*-fork node v_0 and ends at a domain-exit node $x \in X$. The lemma states that path p satisfies three properties.[5]

1. **Decomposability** : Path p admits a decomposition into *region-segments*, i.e., p can be decomposed into segments such that $p = (v_0 ..., x_0 \| v_1 ... x_1 \| ... \| v_j ... x)$ where $v_0, v_1, ..., v_j$ are *lp*-forks , $x_i \in X_{v_i}$ and $paths_{v_i}(v_i, x_i) \neq \phi$.

[5] There are subtle differences between our path decomposition lemma and a similar lemma used in [26]. These differences stem from the fact that regions can overlap while procedure calls are completely nested. Decomposition of a path into unique procedural segments is a conceptually simpler operation since every node in a control-flow graph belongs to only one procedure while a node in our region-segments can belong to multiple segments. See [27] for more details.

Helper Functions	Functionality
`succ(r,e)`	region succeeding `r` at exit `e`
`SetPredecessors(r)`	Prepares pred(r) and predExit(r,p) using fork-directions and the pre-computed succ. sets
`pred(r)`	set of predecessor regions of region `r`
`predExit(r,p)`	set of exits of `p` that lead to `r`
`G(r,dir,e)`	gen component of summary-function ϕ_r(dir,exit)
`K(r,dir,e)`	kill component of summary-function ϕ_r(dir,exit)
`remove(list)`	returns (and removes) a region from worklist
`Exits(r)`	set of exits of region `r`
`Dir(r)`	predicted direction of `r`'s fork
`CheckCache(opd,dir)`	Checks if dataflow results are cached
`StoreCache(opd,dir,flow)`	Stores dataflow results in cache

```
function stitcher(domain *opd)

   foreach region r
     direc[r] ← Dir(r)

  flow ← CheckCache(opd,direc)

  if ( not found in cache )
    SetPredecessors()
    flow ← workList(opd, direc)
    StoreCache(opd,direc,flow)

end
```

```
function workList (domain *opd, int *direc)
   foreach region r
     out[r] ← ⊤
      foreach exit e in Exits(r)
        if succ(r,e).type = Op
           wkList ← (r,e,⊥)

  while (wkList ≠ φ)
     (r,e,inflow) ←  remove(wkList)
     oldOut ← out[r]
     tmp  ← G(r,direc[r],e) ∪
              (inflow - K(r, dir[r],e))
     out[r] ← out[r] ⊓ tmp
     if (out[r] ≠ oldOut)
        foreach region p in pred(r)
          wkList ← wkList ∪
             (p,predExit(r,p),out[r])
  returns out[Op]
end
```

Figure 3: The stitching algorithm.

2. **Uniqueness :** For every valid path there is a unique decomposition into region-segments; i.e. there is one and only one way to decompose p into region-segments.
3. **Existence :** The converse is also true; any sequence of valid region-segments that are "correctly connected", constitute a valid path. More precisely, say S is a sequence of region-segments, $S = (s_0 \| s_1 \| \ldots \| s_k)$ where $s_i = v_i \ldots x_i$, $v_i \in V_D$, $x_i \in X_{v_i}$. Correctly connected means that the sequence satisfies the property $x_i = v_{i+1}$. If these constraints are satisfied then $S \in paths(v_0, x_k)$.

DDFA terminates: The *builder* terminates as each iteration causes the summary function of a region to move down the function lattice F ; $\phi^{i+1} \leq \phi^i$ is implied by monotonicity of transfer functions and termination follows because F is of bounded depth. The *stitcher* terminates as the summary functions computed for each region are also monotone (F being closed and monotone) and L is of bounded depth.

DDFA is safe: To show that DDFA is safe, we need to show that the builder generates safe summary functions and the stitcher combines them in a safe manner.

To show that the builder is safe, we use the path decomposition lemma and the safety of functional meets and composition in F. The path decomposition lemma states that for every feasible execution path P, from an op-node *op* to an exit node $x \in X_v$ can be decomposed into region-segments. The builder generates a summary transfer function for each segment in P by composition of transfer functions of each node in the segment. These summary transfer functions are safe as function composition and functional meet are safe in F (a simple inductive proof on path-length can be constructed using the safety of functional meet and composition).

To show that the stitcher combines these summary functions in a safe manner, we need to show that: (1) the stitcher selects summary functions for all the regions that control might flow through while it is within this op-domain; and (2) the stitcher combine the selected summary functions in a safe manner. To show the former, we need to show that for every cross-segment edge (x_i, v_{i+1}) in the decomposition of P, there is a corresponding cross-region edge included in the stitcher stage (i.e. region $v_i \in pred_{v_{i+1}}$ and $x_i \in pred_{v_{i+1}}(v_i)$). This follows directly from our definition of *pred*. As path P has a region-segment $(v_i, ..., x_i)$ it follows that x_i will be marked as a reachable exit from v_i. This will mark region v_{i+1} as a reachable successor of region v_i via exit x_i. Since *pred* is obtained by transposing the *succ* set, it follows that the stitcher will properly account for all cross-segment edges in P. To show that the stitcher combines the summary functions in a safe manner we use the safety of functional meet and composition in F. A simple inductive proof can be constructed using these safety properties. In this case, the induction is on the number of regions in the path instead of the number of nodes.

DDFA is at least as good as the compile-time meet-over-all-paths solution: The idea here is to show that if we did not use any runtime information (knowledge of control-flow decisions), the results of DDFA's two-phase analysis would still be just as good as a meet-over-all-paths (MOP) analysis. Consider a crippled version of DDFA (called pDDFA, for poor-man's DDFA) that is identical to DDFA except that the stitcher selects summary functions based only on the exit-node and not the (fork-direction,exit-node) pair as in DDFA. As a result, it loses precision and selects a larger number of summary functions than DDFA. We show that pDDFA can perform as well as a MOP analysis. The proof proceeds in three steps, of which only first step is new.

First, we use the existence property from the path-decomposition lemma and the safety of composition and functional meet in F to show that pDDFA is at least as good as a approach that we call BOA (builder-only approach). The BOA approach considers the entire op-domain as a single region and computes a summary function (multiple functions if there are multiple exits) for the entire op-domain. The stitcher for BOA is trivial - it merely applies each of these functions to $\perp$ and performs a meet on the results. The existence property of the path-decomposition lemma states that any sequence of segments used by pDDFA's stitcher is a valid path and must necessarily have been used by BOA's builder to compute the op-domain summary function. Given this and the safety of composition and functional meet a simple inductive proof can be constructed to show that op-domain summary-function computed by BOA at compile-time can be no better than the combined summary function generated by pDDFA at run-time (the induction is over the number of regions in the path).

Next, we argue that the iterative computation of the summary function by BOA is just as good as a functional meet over all paths (FMOP) computation of the summary function. Other than the fact that we are operating on the function lattice F, this step is identical to any normal dataflow analyses. Since F is distributive, monotone, bounded and closed under functional-composition and functional-meet, we can directly borrow proofs from Kildall [17].

Finally, we use previous results on summary functions (in particular, Theorem 7-3.4 in [26]) to argue that operating on the functional domain does not hurt the quality of the solution. In other words the FMOP solution is as good as the MOP solution. This completes the proof.

5 Extensions to the basic framework

In this section, we extend the basic DDFA framework described in the previous section by relaxing some of the restrictions. First, we extend the framework to interprocedural analysis. Second, we extend the analysis to consider multiple op-domains at once – the goal being to optimize over multiple op-domains. Third, we briefly describe how DDFA can be extended for forward data-flow analysis.

Interprocedural Analysis: To handle procedure calls, we extend the control-flow graph. We insert two nodes for each call-site – a *call-node* and a *resume-node*. For every procedure, we insert an *entry-node* and

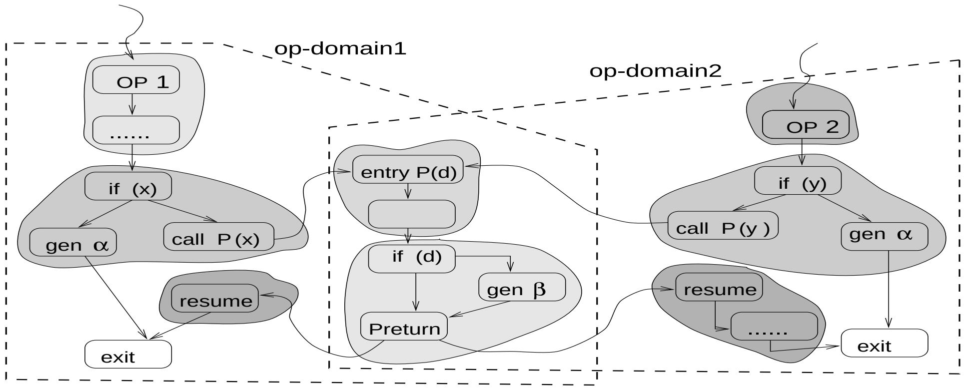

Figure 4: Interprocedural example. Procedure P is called from two sites. At both sites, the fork inside P can be predicted. As a result, the op-domains for the two sites can share the common region's data-structures.

and *return-node*. Each entry-node has incoming edges from all its call-nodes; each return-node has outgoing edges to the corresponding resume-nodes. A suitable binding function and its inverse are associated with the entry-node and the return-node of a procedure.

Extending DDFA to an interprocedural setting introduces two primary problems. First, not all paths in the control-flow graph are valid; care must be taken to preserve the calling context while creating op-domains that straddle procedure boundaries. Second, since procedures can be called from a large number of call-sites and since each call-site can be within a different op-domain, the regions corresponding to the code in a procedure can be a part of a large number of op-domains. Since a summary-function table is generated for every region in an op-domain, this can result in a large number of replicated summary-function tables.

Sharing the region-tables for common regions among different op-domains that reach a procedure is non-trivial – a fork within the procedure may be predictable at one op-node but not at another. We note that this problem is similar to context-sensitive binding-time analysis [16] and suggest a similar solution. For each op-domain that extends into a procedure, we mark each of its forks within the procedure as *lp* or *non-lp*. If two op-domains induce the same markings for all forks in a procedure then they can share the region analysis and data-structures for the entire procedure. Since the number of forks is likely to be small, we expect that the number of possible divisions will also be small, allowing significant sharing across op-domains. Furthermore, to increase the likelihood of sharing, we mark entry-nodes and resume-nodes as dummy *lp*-fork nodes – this ensures that *lp*-regions don't straddle procedure boundaries. Figure 4(a) illustrates the interprocedural representation and sharing of regions. In the figure, the regions of procedure P are shared by both op-domains.

Combining information across op-domains: The basic DDFA framework described in Section 3 restricts its analysis to individual op-domains. This is because a *lp*-fork that is predictable at one op-node may not be predictable at a previous op-node. Thus, op-domains mark boundaries of predictability. However, this boundary is not always strict – *all* the fork-directions predictable at an op-node (op2) may possibly also be predicted at one of its predecessors (say, op1). These circumstances can be detected at compile-time using a reaching-definitions analysis. In such cases, op2 is said to be a *half-op* node (*hop*) with respect to op1. The runtime stitcher when invoked at op1, checks its fork-directions to see if control could reach op2. If so, it checks whether op2 is a hop-node. If that is indeed the case, it performs the tests to determine op2's fork directions and combines the op2's summary functions with those of op1. The hopping process continues, until control reaches an exit that is not a hop-node with respect to op1. For instance, in Figure 1, op2 has only one *lp*-fork (`if(d)`) and this fork is predictable even at the first op. Hence, when op1's runtime stitcher detects that control may exit the loop and reach op2, it stitches in op2's *lp*-region to its own set of regions and analyses the composite result. To ensure that this analysis is not repeated when control reaches op2, we

Application	Lines of code	Number of Op-domains	Regions per domain min,max,avg	Space (in bytes)
`unst`	780	15	1,2,1.1	640
`moldyn`	1,562	9	1,2,1.1	400
`DSMC3d`	4,996	6	1,3,1.3	440
`schedule`	750	3	3,3,3	160
`scanner`	400	1	3,3,3	48
`XDataSlice`	28,000	94	1,4,1.3	7200

Table 1: Space requirements for DDFA for a set of applications.

add a hop-bit to each region-table. This bit is set by its predecessors and is reset when control reaches op2.

Forward DDFA: Our presentation of DDFA has focused on solving backward flow problems where its main power is its ability to predict directions of lossy forks that will be encountered in the future. Forward dataflow analysis frameworks suffer from an analogous loss of information at *joins* (program points at which control can arrive from multiple directions) in the control-flow graph. We can extend DDFA to deal with forward analyses relatively easily. The op-domains would be constructed in a similar fashion; a join is lossy with respect to a particular data-flow analysis if all its incident edges in the control-flow graph do not have the same dataflow attributes. There is, however, no notion of predictability; when control arrives at a join, there is no uncertainity about the direction it arrived from. The regions for forward analyses would be constructed backwards from the join-nodes and the points of entry and exit into regions would be logged (possibly using path-profiling techniques [3] adapted for profiling region entry and exit). At the op-node, a check of the region entries and exits would allow the runtime stitcher to select the transfer functions to compose.

6 Size of data-structures

The size of the data-structures used to communicate the summary transfer functions and other auxiliary information from the compiler to the runtime system is an important concern for the practical applicability of DDFA. To address this concern, we studied the space requirements of DDFA in the context of three different backward flow analyses: communication analysis for efficient message aggregation in parallel programs, call-graph analysis of mobile programs for safe dynamic linking [1] and analysis of programs that use the Hierarchical Data Format (HDF) library [13] for accessing disk-resident scientific datasets for optimizing disk accesses. We adapted each analysis to work within the DDFA framework and hand-simulated its operation on one or more applications.

Communication analysis for parallel programs: Message aggregation is an important optimization for parallel programs [14]. Backward data-flow analysis is used to collect the set of remote memory locations that could be accessed between calls to the communication library. For our study, we selected three irregular scientific programs which have been studied by several parallel processing researchers [20, 21, 24]: (1) `unst` which solves a computational fluid dynamics problem using an unstructured mesh; (2) `moldyn` which simulates molecular dynamics; and (3) `DSMC3d` which performs a three-dimensional simulation of a gas jet.

Call-graph analysis for mobile programs: Dynamic linking provides functionality that is necessary for secure flexible use of mobile programs but it introduces a new class of runtime errors - unbound procedure names. Call-graph analysis can identify of the set of procedures that the program could possibly refer to while executing on a host and can help ensure safe dynamic linking for mobile programs [1]. For our study, we selected two mobile programs: (1) `schedule`,[6] which manages a calendar; (2) `scanner`,[7] which visits a

[6] Part of the Mole Mobile Agent System v2.0 distribution. Available at *http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/release.html*

[7] Part of ARA Mobile Agent System distribution available at *http://www.uni-kl.de/AG-Nehmer/Projekte/Ara/Download/download_D.html*

set of web sites to collect information.

Analysis of programs that use the HDF library: The HDF library is used for managing and accessing disk-resident scientific datasets. Since scientific datasets are structured and this structure is visible to the HDF library, an analysis of the patterns in which HDF routines are called can be useful for aggregating disk accesses. For this study, we selected the NCSA `XDataSlice`[8] which provides a color imaging and data analysis tool for 3D, 32-bit floating-point scientific data.

Table 1 presents characteristics of the DDFA data-structures for each application and their space requirement. We note that the space requirements are small for these applications. Further, we note that the number of *lp*-regions per op-domain is usually small (the largest being 4). This indicates that the overhead due to the *stitcher* which traverses the graph of *lp*-regions of an op-domain at run-time is likely to be small.

7 Runtime overhead and effectiveness

To address concerns regarding the runtime overhead of DDFA and to demonstrate the ability of DDFA to optimize heavy-weight operations, we studied the operation of DDFA in the context of two of the backward flow analyses mentioned in the previous section: communication analysis for efficient message aggregation in parallel programs and call-graph analysis of mobile programs for safe dynamic linking [1]. We hand-simulated the compile-time aspect of DDFA and added code to individual applications to implement stitching and the optimizations.

Communication analysis for parallel programs: for this study, we selected two of the irregular scientific programs mentioned in the previous section – `unst` and `moldyn`. We started with an unmodified version of each application (with no message aggregation). To get as much message aggregation as possible, we used the extended DDFA framework described in section 5 which combines information from multiple op-domains.

Both these programs use iterative simulation algorithms that have an outer time-step loop encasing an inner compute-intensive loop which accesses data via indirect references. Each indirect reference requires a local/remote check at runtime to determine if the data has to be fetched from another processor. Communication is trigerred for off-processor references leading to many small messages. A DDFA-based technique can aggregate these messages by invoking the runtime stitcher when the first message is trigerred. At this point, the stitcher can precisely predict the results of all future local/remote checks in the loop (in the same iteration as well as future iterations) and can aggregate all these messages.

To evaluate the impact of this technique on a real machine, we ran each application on two processors of an IBM SP-2 running AIX 4.2 with “thin” nodes, 128MB of memory and a 40MB/s multi-stage switched interconnect. As shown in Table 2, DDFA is able to aggregate most of the messages occuring in the original programs. An inspection of the programs indicated that these are the minimum number of messages required for correctness.

Other techniques have also been proposed for message aggregation in irregular codes. Das et. al. [7] have used slicing to generate a preprocessing step called an *inspector* which performs communication analysis at runtime before entering the actual loop. These inspectors are invoked for every loop that has indirect references, whereas DDFA-based stitchers are trigerred only at the first communication call and will not be invoked if a loop happens to have only local references. On the other hand, treating the inspector as a separate preprocessing entity allows Das et. al. to hoist it out of inner loops and combine it with other inspectors.

Call-graph analysis for mobile programs: for this study, we selected both the mobile programs mentioned in the previous section – `schedule` and `scanner`. We started with the versions of these programs in the respective distributions. The `ops` in these programs are the migration points; goal of the analysis is to determine the smallest number of modules (Java classes for Mole, procedures for Tcl) needed while executing on a given site. These experiments were conducted on a Sun Ultra 1 (167MHz) running Solaris 2.6. Table 3 presents the results. For each migration point, it shows the minimum number of modules that would be linked at that point with and without DDFA. These programs have two or more modes of operations which

[8]Available at *ftp://ftp.ncsa.uiuc.edu/Unix/XDataSlice/XDS2.2/*

Application	Num of ops w/o DDFA (time for ops / total time)	Num of ops with DDFA (time for ops / total time)	Time for stitcher
`unst`	30092 (6.39s / 6.48s)	2 (0.02s / 0.28s)	0.24s
`moldyn`	57985 (16.03s / 16.13s)	7 (0.01s / 0.18s)	0.07s

Table 2: Runtime overhead and effectiveness of DDFA for message aggregation in irregular parallel programs. Both these programs are iterative; the numbers presented are *per time-step*.

Application	`op 1`		`op 2`		`op 3`		time for stitcher
	w/o DDFA	with DDFA	w/o DDFA	with DDFA	w/o DDFA	with DDFA	
`schedule`	23	11	23	23	23	11	0.001s
`scanner`	12	6	–	–	–	–	0.001s

Table 3: Runtime overhead and effectiveness of DDFA for safe dynamic linking in mobile programs.

they select between soon after they arrive at a new location. Just before completing execution at a location, they reset the mode and move to the next location. Since the value of the mode variable is known at the migration points, DDFA is able to accurately determine the number of modules needed.

These results indicate that DDFA is able to provide effective and usable information with low runtime overhead.

8 Related Work

Specializing programs at runtime with respect to runtime invariants has been studied in considerable detail. Typically, most dynamic compilation strategies focus on efficient *code generation* by delaying some parts of code generation until runtime (or an intermediate *specialization* time). The process usually involves generating parameterized code templates with holes in them; a runtime specializer plugs in these holes with runtime values and stitches the templates. Engler et al [10] allow programmers to construct and manipulate templates explicitly. Others [6, 12, 19], generate templates automatically. These efforts are concerned with generating optimized code based on runtime values, so they focus on low-level optimizations like load-elimination, loop-unrolling, static branch-elimination etc. DDFA generates information at compile-time, in a similar manner, for the use of a runtime stitcher. However, as our goal is to optimize heavy-weight operations, we focus on improving the precision of the *dataflow* analysis rather than generating optimized code. DDFA does not modify code, except to insert a call to the runtime stitcher at each operation.

More closely related to our work is the recent work on predicated array dataflow analysis by Hall et al [22] and on staged type inference by Shields et al [28]. The analysis framework proposed in [22] can generate low-cost runtime predicates to guard conditionally optimized code. Unlike DDFA, their dataflow analysis and optimization is performed completely at compile-time, with runtime predicates used to distinguish between different versions of optimized code. Stitching dataflow results at runtime gives DDFA more power. However, the predicate algebra presented in [22] is more sophisticated than ours - while DDFA is limited to control-flow tests, predicated dataflow analysis allows arbitrary program expressions as run-time predicates. Shields et al employ a technique similar to the one proposed in this paper to partition type inference into a compile-time component and a runtime component. Their goal is to seamlessly use a single type-system for static type-inference (where that is possible) as well as for expressions whose type is necessarily known only at runtime. In effect, staged type inference uses runtime information to improve the precision of typing whereas DDFA focuses on improving the precision of data-flow information at performance-critical points.

Our use of summary functions have been adapted from previous research on interprocedural analysis, in particular Sharir et al [26] and Duesterwald et al [9]. The main distinction between their approach and ours is that we apply these summary function at runtime.

DDFA needs to know about future control flow decisions in order to be effective. There has been considerable research recently in branch prediction [18, 29], profiling [3] and exploiting correlation between conditional branches. For example, Mueller et al [23] have reported the existence of significant amounts of correlation among conditional branches. Often these correlations can be detected at compile-time [4] and can be used to eliminate some paths during dataflow analysis, resulting in a solution that is sharper than the meet-over-all-paths solution. For example, Holley and Rosen [15], describe a *qualified* dataflow approach which exploits edge correlations to sharpen dataflow analysis; Bodik et al present a similar technique in [5]. Unlike these approaches, DDFA does not try to detect edge correlations at compile-time. Instead, DDFA tries to eliminate paths at runtime, by using runtime control-flow information.

The technique of delaying the collection of dataflow information was used in a limited form (only for procedure return forks) in [1] to handle the problem of dynamic linking in mobile programs. DDFA is more general as it handles conditional-forks as well.

9 Conclusions

In this paper, we have proposed a general framework which combines compile-time analysis with limited runtime analysis to improve the precision of dataflow information at performance-critical points in the program. We present algorithms for both the compile-time and runtime parts of the technique. To provide the theoretical underpinning for DDFA, we showed that it terminates, that it is safe and that its results are as good as a compile-time meets-over-all-paths solution. To address concerns regarding the size of the data-structures used to communicate information from the compiler to the runtime system, we presented space requirements of DDFA in the context of three different backward data-flow analyses. Our results indicate that the size of these data-structures is small. To address concerns regarding the runtime overhead of DDFA and to demonstrate the ability of DDFA to optimize heavy-weight operations, we presented results from the application of DDFA for two of these analyses. Our results indicate that DDFA is able to provide effective and usable information with low runtime overhead.

References

[1] A. Acharya and J. Saltz. Dynamic Linking for Mobile Programs. *Mobile Object Systems, J. Vitek and C. Tschudin (eds), Springer Verlag Lecture Notes in Computer Science*, 1997.

[2] F. Allen and J. Cocke. A program dataflow analysis procedure. *CACM*, 19(3), March 1976.

[3] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of MIRCO-29*, 1996.

[4] R. Bodik, R. Gupta, and M. Souffa. Interprocedural Conditional Branch Elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–58, June 1997.

[5] Ratislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Fifth ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1997.

[6] C. Consel and F. Noel. A General Approach to Run-time Specialization an its Application to C . In *POPL'96*, pages 145–56, 1996.

[7] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[8] E. Duesterwald, R. Gupta, and M.L. Soffa. Reducing the cost of dataflow analysis by congurence partitioning. In *Fifth International Conference on Compiler Construction*, number 786 in Lecture Notes on Computer Science, pages 357–373. Springer Verlag, April 1994.

[9] E. Duesterwald, R. Gupta, and M. Souffa. Demand-driven computation of interprocedural dataflow. In *Proceedings of POPL*, 1995.

[10] D. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C : A Languages for High-Level, Efficient and Machine-Independent Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 131–144, January 1996.

[11] S.L. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. In *J. ACM*, 1976.

[12] B. Grant, M. Mock, M. Phillpose, C. Chambers, and S. Eggers. Annotation-Directed Run-Time Specialization in C. In *PEPM'97*, 1997.

[13] The NCSA hierarchical data format home page. *http://hdf.ncsa.uiuc.edu/*, 1998. HDF is a library and platform independent data format for the storage and exchange of scientific data.

[14] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machine s. *Communications of the ACM*, 35(8):66–80, August 1992.

[15] L. Howard Holley and Barry Rosen. Qualified Dataflow Problems. In *POPL*, 1980.

[16] L. Hornof and J. Noye. Accurate Binding-Time Analysis for Imperative Languages: Flow, Context and Return Sensitivity. In *PEPM'97*, 1997.

[17] G. Kildall. A unified approach to global program analysis. In *First ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, January 1973.

[18] A. Krall. Improving semi-static branch-prediction by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[19] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–48, 1996.

[20] D.J. Mavriplis, R. Das, R.E. Vermeland, and J. Saltz. Implementation of a parallel unstructured Euler solver on shared and distributed memory architectures. In *Proceedings Supercomputing '92*, pages 132–41, 1992.

[21] B. Moon and J. Saltz. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 176–83, 1994.

[22] S. Moon, M. Hall, and B. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the International Conference of Supercomputing*, July 1998.

[23] F. Mueller and D.B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[24] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *PPOPP*, pages 68–79, 1995.

[25] Barry Rosen. Monoids for Rapid Data-Flow Analysis. In *POPL*, 1978.

[26] M. Sharir and A. Pnueli. *Two approaches for interprocedural data flow analysis*, chapter in book titled Program Flow Analysis : Theory and Applications, pages 189–234. Prentice-Hall, edited by S. Muchnik and N.D. Jones, 1981.

[27] Shamik Sharma, A. Acharya, and J. Saltz. Deferred data-flow analysis : Algorithms, proofs and applications. Technical report, University of Maryland, 1997. Available at ftp://hyena.cs.umd.edu/pub/shamik/html/papers/TR.ddfa.ps.Z.

[28] M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, 1998.

[29] C. Young, N. Gloy, and M. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd ISCA*, June 1995.