# Deferred Data-Flow Analysis : Algorithms, Proofs and Applications

Shamik D. Sharma† Anurag Acharya‡ Joel Saltz⋆

† Department of Computer Science
University of Maryland, College Park

‡ Department of Computer Science
University of California, Santa Barbara

⋆ Institute of Advanced Computer Studies
Uversity of Maryland, College Park

shamik@cs.umd.edu, acha@cs.ucsb.edu, saltz@cs.umd.edu

**Abstract**

Loss of precision due to the conservative nature of compile-time dataflow analysis is a general problem and impacts a wide variety of optimizations. We propose a limited form of *runtime* dataflow analysis, called deferred dataflow analysis (DDFA), which attempts to sharpen dataflow results by using control-flow information that is available at runtime. The overheads of runtime analysis are minimized by performing the bulk of the analysis at compile-time and deferring only a summarized version of the dataflow problem to runtime. Caching and reusing of dataflow results reduces these overheads further.

DDFA is an interprocedural framework and can handle arbitrary control structures including multi-way forks, recursion, separately compiled functions and higher-order functions. It is primarily targeted towards optimization of *heavy-weight operations* such as communication calls, where one can expect significant benefits from sharper dataflow analysis. We outline how DDFA can be used to optimize different kinds of heavy-weight operations such as bulk-prefetching on distributed systems and dynamic linking in mobile programs. We prove that DDFA is safe and that it yields better dataflow information than strictly compile-time dataflow analysis.

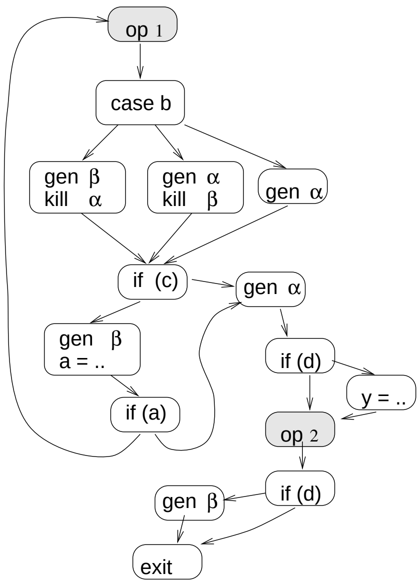
## 1 Introduction

Compile-time dataflow analysis combines information from all execution paths that a program could possibly take. The analysis is conservative, because at runtime, a program will follow only one of these (possibly infinite) execution paths. For example, consider 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 [18]. 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. Figure 1 provides an illustration. In this case, a compile-time analysis indicates that neither of the remote variables $\alpha$ or $\beta$ 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) ($\alpha, \beta$ or $\{\alpha, \beta\}$) could be prefetched.

Loss of precision due to the conservative nature of compile-time dataflow analysis is a general problem and impacts a wide variety of optimizations. This problem particularly influences optimization of *heavy-weight* operations such as bulk-prefetch, garbage-collection, runtime-compilation or remote procedure calls.

```
a = b = c = d =  α=  β=  3;
repeat  {
    prefetch ( ? )
    switch (b) {
       case 1 :  β  =  α –1    ; break;
       case 2 :  α  =  β –1    ; break;
       case 3:   x  =  α  -1   ; break;
    }
    if  (c == 0)   break;
    x = β;   a-- ;
} until ( ! a) ;
b = α;
if  ( d)  y = 5 ;
prefetch ( ? )
if  ( d)  a  =  β  ;
exit (0);
```

(a)




(b)

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 $\alpha$ and $\beta$. At each `prefetch` instruction, we would like to prefetch those remote variables that will be read along all outgoing paths. (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`. We will use this CFG fragment as a running example.

These operations are expensive ; avoiding or combining even a small number of them has the potential of providing significant benefit. The obvious alternative to compile-time dataflow analysis, namely runtime analysis, can provide good dataflow information but could add too much runtime overhead to be worthwhile. In this paper, we propose a hybrid between compile-time analysis and runtime analysis, called deferred dataflow analysis (DDFA). DDFA performs most of its analysis at compile-time and uses additional control-flow data that becomes available at runtime to *stitch* together the dataflow information that was collected at compile-time.

DDFA divides the task of flow analysis into two phases - a compile-time phase and a runtime phase. The compile-time phase, called the *builder*, analyzes the control-flow graph to identify forks[1] whose direction can be determined at specific points in program execution. These *predictable* forks are used to divide the control-flow graph into *regions* such that each region contains at most one such fork. The builder then performs dataflow analysis on each region independently and summarizes the result in the form of a *summary transfer function*. The summary transfer function for a region describes how the region affects dataflow attributes flowing through it. The second phase, called the *stitcher* is invoked whenever a heavy-weight operation is encountered during program execution. It checks values of program variables to predict future

[1]Forks are program-points from which control can flow in more than one direction - e.g. conditionals, procedure returns, switch statements, higher-order functions.

control-flow directions and computes the final dataflow results by stitching together summary functions from regions that may be encountered in the future. The final dataflow results can be used by a runtime system to optimize heavy-weight operations.

The key idea of DDFA is to divide dataflow analysis into a compile-time phase and a runtime phase and to use runtime control-flow information to improve precision of compile-time analysis. Two-phase techniques using summary functions have have previously been used for *interprocedural* flow analysis by others (e.g. Sharir et al [21] and Duesterwald et al [7]). In these techniques, the first phase computes a summary function for each procedure and the second phase applies these functions to obtain interprocedural dataflow properties. These techniques perform both phases at compile-time; they use summary functions as a mechanism to avoid reanalysis of a procedure at every call-site. DDFA differs from these techniques in three important ways. First and foremost, DDFA applies these functions at runtime. It is therefore important to construct compact representations for these summary functions, so that results of the compile-time phase can be efficiently passed to the runtime phase. Second, DDFA computes summary functions for *regions*, which are not necessarily procedures - this introduces complications as regions can overlap while procedures cannot. Third, DDFA computes multiple summary functions for each region and uses control-flow information available at runtime to choose between these functions.

In this report, we present an interprocedural DDFA framework that is applicable for arbitrary control structures including multi-way forks, recursion, separately compiled functions and higher-order functions. We present algorithms for construction of *region* summary functions and for composition and application of these functions. We limit our application of DDFA to solving *backward* flow problems in this report.

The report is structured as follows. We begin our presentation by providing an overview of the DDFA approach in Section 2, using intuitive descriptions for key terms and concepts. Section 3 provides a more rigorous definition of these terms and lists all our assumptions. Section 4 describes the details of the basic DDFA framework for *intra*procedural analysis. It has two parts – the first part describes the compile-time *builder* which constructs summary functions for regions while the second part describes the *stitcher* which composes and applies these functions at runtime. The running example is used to demonstrate how DDFA optimizes bulk-prefetches and to describe the data-structures necessary to pass information from compile-time to run-time. In section 5, we prove that DDFA is safe and that its results are at least as good as the compile-time meets-over-all-paths solution. Section 6 extends the basic intraprocedural DDFA framework to handle interprocedural analysis, separately compiled code, higher-order functions and dynamic merging of heavy-weight operations. Section 7 outlines several application scenarios in which DDFA can be used to optimize heavy-weight operations. We limit our application of DDFA to solving *backward* flow problems in this report. In section 8 we outline a variant of DDFA for solving forward dataflow problems and discuss more aggressive techniques for extracting control-flow information at runtime.

## 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. These terms are defined more rigorously in Section 4.

As mentioned in the introduction, DDFA is targeted towards optimization of heavy-weight operations (such as bulk-prefetch, garbage-collection, runtime compilation, remote procedure calls etc). We will refer to these operations 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 $op_1$ consists of the nodes that are reachable from $op_1$ without passing through another op-node or program-exit node. $op_1$ 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 nodes.

Each op-domain has zero or more *forks*. Forks are program-points from which control can flow in more

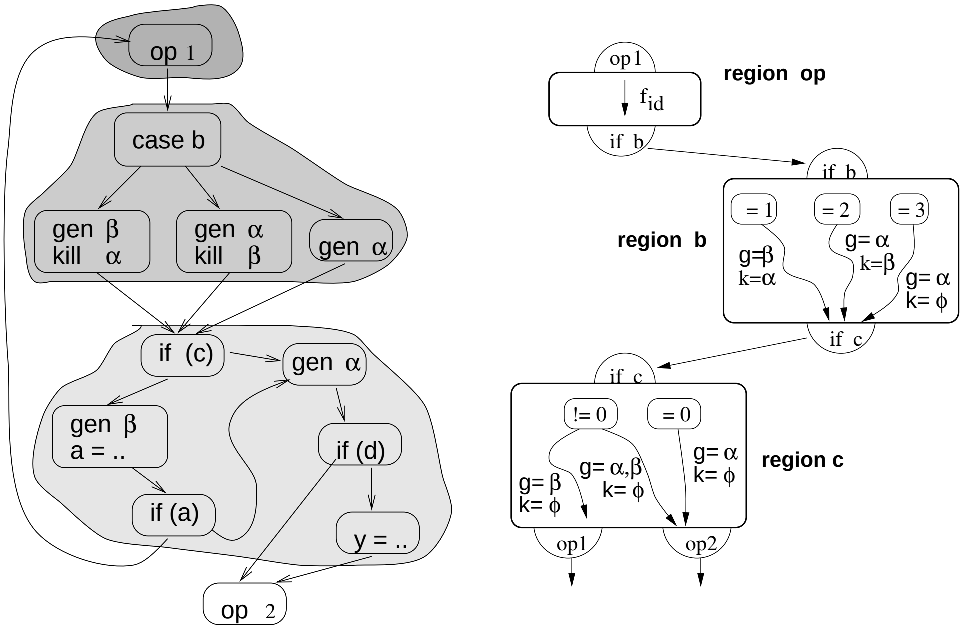



Figure 2: The three shaded areas show the *lp*-regions of the first op-domain in the running example. The figure on the right shows the summary functions for each *lp*-region. The gen and kill sets are represented by `g` and `k` respectively.

than one direction - e.g. conditionals, procedure returns, switch statements, calls to unknown functions. A fork is said to be *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 (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. Corresponding to each fork, there is at least one other program point, referred to as a *fork-determinant*, which determines which of the multiple control-flow alternatives the fork will take. For example, the fork-determinants for a `if` statement would be the reaching definitions of the variable being tested.[2] In other words, a fork is said to be *predictable* at an op-node if there is no path from the op-node to the fork-point that contains a fork-determinant. Forks that are both *lossy* and *predictable* are said to be *lp*-forks. Because *lp*-forks are forks at which a conservative analysis loses information *and* whose control-flow direction can be predicted, it is potentially beneficial to defer the meet operation for them. In figure 2, 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.

Each *lp*-fork induces an *lp*-region. An *lp*-region is the collection of nodes that can be reached from an lp-fork without passing through an *lp*-fork or any of the exits of the op-domain. An *lp*-region may contain zero or more 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 2 show the *lp*-regions for the running example.

The first stage of DDFA, performed at compile-time, 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 intuitive definitions presented above for op-domains and *lp*-regions indicate how they

[2]Fork-determinants can similarly be defined for other types of forks.

can be constructed; more rigorous definitions are provided in section 3. Note that, for op-domains that straddle procedure boundaries, care needs to be taken to preserve the calling context when computing the set of nodes reachable from an op-node.

The second stage, performed at compile-time, analyzes each *lp*-region and produces one or more *summary* transfer functions, $\phi(\text{fork-direction})$, one function for each direction the *lp*-fork at the entry to the region can take. [3] Each function summarizes the backward dataflow operations that would occur if control was restricted to flow in a particular direction.

The third stage is performed on-demand at runtime 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 from the op-node to exit and applies the corresponding summary functions (in reverse order as we are doing backward propagation). 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 performing iteration may not be expensive as the number of *lp*-regions in a op-domain is expected to be small (much less than the number of basic-blocks).

The final stage, performed at runtime, uses the data-flow information computed by the stitcher to make optimization decisions. This stage is application-specific. For conciseness, we do not present detailed examples in this abstract. We do, however, sketch how DDFA can be used for *bulk-prefetching*, our running example.

## 3 Terminology and Background

This section defines the terms used in describing the DDFA framework and lists all our assumptions. The casual reader may choose to skip directly to subsection 3.4 which summarizes the important terms and assumptions.

In the following discussion 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 assume 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 6, we will extend the framework to include interprocedural analysis, separately compiled code, higher-order functions and allow dynamic merging of dataflow analyses across op-domains.

### 3.1 Op-domains

A program is represented as a directed *flow graph*. A flow graph is denoted as $G = (R, N^*, E^*, X^*, O^*)$. $R$ is the root of the graph and represents the entry-point of the program. $N^*$ is a set of nodes that represent program statements and $E^*$ is a set of edges that represents the transfer of control among these statements. $X^* \subseteq N^*$ is a set of program-exit nodes. These nodes have no outgoing edges. $O^* \subseteq N^*$ is a set of nodes that are specially marked as *op-nodes*; these represent program points at which the program performs some heavy-weight operation that we want to optimize. The entry point $R$ is also marked a dummy op-node, so $R \in O^*$.

[3] Actually, for reasons described later, we may have more than one summary function for each fork-direction. Specifically, each summary function $\phi$ is parameterized by a (fork-direction, region-exit) pair.

For each node $n \in N^*$, $pred(n) = \{m \mid (m,n) \in E^*\}$ and $succ(n) = \{m \mid (n,m) \in E^*\}$ denote the set of immediate predecessors and successors of $n$. A sequence of nodes $p = (n_1, n_2, .....n_k)$ is said to be a *valid path* if for $1 \leq i \leq k$, $(n_i, n_{i+1}) \in E^*$. The *length* of the path is the number of edges in it ($k - 1$ in the above notation). For any pair of nodes $m, n$, $paths(m,n)$ is the set of *all* paths in graph $G$ leading from $m$ to $n$. $paths(m,n) = \phi$ if there is no path from node $m$ to node $n$. If there are cycles in the flow graph, then the number of paths in $paths(m,n)$ may be infinite.

Each op-node $op_i \in O^*$ induces an *op-domain*. The op-domain induced by $op_i$, is denoted as $D_i$. $D_i$ is is a subgraph of $G$ and contains all nodes that are reachable from $op_i$ without passing through another op-node. More precisely, $D_i = (op_i, N_i, E_i, X_i)$ where :

- $op_i \in O^*$ is the op-node that induced domain $D_i$.
- $N_i = \{n \mid n \in succ(op_i) \text{ or } \exists\, (op_i, m_1, \ldots, m_k, n) \in paths(op_i, n),\ \ m_1, \ldots, m_k \notin O^*\}$
  $N_i$ is the set of nodes in the op-domain.
- $E_i = \{(m,n) \mid m, n \in N_i, (m,n) \in E^*\}$ is the set of edges in the op-domain.
- $X_i = \{x \mid x \in N_i \cap (O^* \cup X^*)\}$ is the set of exit nodes for the op-domain. These nodes are called domain-exits.

When the domain being analyzed is clear from the context, we will drop subscripts and denote it as $D = (op, N, E, X)$. We use $pred_D(n)$ and $succ_D(n)$ to denote the nodes that immediately precede and succeed node $n$ and which lie within domain $D$. $paths_D(m,n)$ represents the set of paths between nodes $m$ and $n$ that lie completely within the domain.

It is worth pointing out two properties of op-domains that follow from the definition. (1) An op-node does not necessarily dominate all the other nodes in its op-domain. A node $m$ in $G$ may be reachable from two different op-nodes and therefore get included in both op-domains. This means that op-domains can *overlap*. (2) During execution, control will always be associated with a *unique* op-domain. Control gets associated with a new op-domain only when it encounters an op-node. By definition, control cannot encounter a new op-node within the current op-domain.

We assume that the dataflow information that influences each op-node is limited to the information propagated up from within its own op-domain. This assumption allows us treat op-domains independently of each other ; each op-domain can be treated a separate flow-graph for the purpose of dataflow analysis. Later, in section **??** we will relax this assumption and collect dataflow information across multiple op-domains.

### 3.2 Dataflow Frameworks

The *framework* of a dataflow problem is a pair $(L, F)$, where $L$ is a complete meet semi-lattice of attribute information and $F$ is a set of transfer functions $F : L \rightarrow L$. Following standard terminology, $L$ has 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. any descending chain in the lattice $L$ is of finite height.

We assume that the set $F$ is closed under functional composition ($\circ$) and point-wise functional meet ($\wedge$). [4] That is, for any pair of functions $f_1, f_2 \in F$, we must have $f_1 \circ f_2 \in F$ and $f_1 \wedge f_2 \in F$. $F$ also contains two special functions — an identity function $f_{id}$ , and a constant function $f_\top$ . For all $x \in L$, $f_\top(x) = \top$ and $f_{id}(x) = x$. A framework that satisfies these assumptions is said to be a *closed* framework [12]. In a closed framework, the functions in $F$ form a semi-lattice with $f_\top$ at the top of the function-lattice and the

[4]Functional-meet ($\wedge$) is different from the commonly used attribute-meet ($\sqcap$). Functional-meet operates on the function lattice, $\wedge : F \times F \rightarrow F$, while attribute-meet operates on the attribute-lattice $\sqcap : L \times L \rightarrow L$. Ideally, $(f_1 \wedge f_2)(x) = f_1(x) \sqcap f_2(x)$.

other functions partially ordered as follows : for $f_1, f_2 \in F$, $f_1 \geq f_2$ iff $f_1(x) \geq f_2(x)$ for all $x \in L$. [5] We also assume that every function in the set $F$ is monotone and distributive, i.e. $\forall f \in F, x, y \in L$, $x \geq y \Rightarrow f(x) \geq f(y)$ (monotone) and $\forall f \in F$, $f(x \sqcap y) = f(x) \sqcap f(y)$ (distributive). Assuming a monotone and distributive closed framework is fairly common in dataflow analysis; most elimination algorithms [2, 10, 20, 19] also make the same assumptions.

A (backward) dataflow problem for an op-domain can be described by a tuple P $= (L, F, D, M)$. Here, $L$ and $F$ together represent a closed distributive framework as discussed earlier. $D = (op, N, E, X)$ is the op-domain being analyzed and $M : E \rightarrow F$ is a mapping that associates each edge of the domain's flow graph with a transfer function from $F$. The transfer function $f_{(n,m)}$ associated with edge $(n, m)$ represents the change of relevant data attributes propagated backwards from the entry of block $m$, up through node $n$ to the entry of block $n$. Given $M$, $F$ can be reduced to the smallest set which contains $M$, $f_{id}$, $f_\top$, $f_\perp$ and which is closed under functional composition and functional meet. This ensures that $F$ is distributive iff $M$ is distributive.

A *solution* to the dataflow problem P is a map $\Theta : N \rightarrow L$ which maps each node $n$ of the flow graph to an element of the attribute lattice, i.e. $\Theta(n)$ represents the dataflow information that can be asserted at node $n$. It is always safe to assert less information, i.e. re-mapping a node to a lower lattice element is safe ; hence, there are many safe solutions possible for any dataflow problem.

A maximal safe static solution is given by the *meet-over-all-paths* (MOP) solution, $\Upsilon : N \rightarrow L$ which is abstractly defined as follows : For each $n \in N$,

$$\Upsilon(n) \;\; = \;\; \sqcap \; \{ \; f_p(\perp) \mid p \in paths(n, x), \quad x \in X \} \tag{1}$$

Here $f_p = f_{(n,m_1)} \circ f_{(m_1,m_2)} \circ \ldots \circ f_{(m_k,x)}$ for the path $p = (n, m_1, m_2, \ldots, m_k, x)$. If $p$ is null, then $f_p$ is defined to be the identity map $f_{id}$.

Computing the MOP solution $\Upsilon(n)$, for node $n$, requires propagating information backwards along every possible path from $n$ to an exit node. The attributes collected from all these paths are then combined by performing an attribute-meet ($\sqcap$). In graphs with cycles there may be an infinite number of paths to consider; hence, equation ( 1) does not suggest a practical algorithm. Instead, most practical dataflow algorithms try to approximate the MOP solution by solving the following set of "local-propagation" equations.

$$\begin{aligned} \Phi(x) &= \perp && \forall x \in X \\ \Phi(n) &= \sqcap_{m \in succ(n)} f_{(n,m)}(\Phi(m)) && \forall n \in (N - X) \end{aligned} \tag{2}$$

This system of equations describes the relations between attributes at adjacent basic blocks and uses them to propagate information backwards starting with $\perp$ at the exit nodes. An iterative algorithm suggested by Kildall [14] is commonly used to solve this system of equations; it yields a maximal fixed-point solution which can be shown to match the MOP solution when the all transfer functions are distributive. Notably, the iterative algorithm, like most other dataflow algorithms, computes $\Phi(n)$ for each and every node in $N$ even though we only need $\Phi(op)$, the dataflow attributes of the op-node. [6]

**Partitionable frameworks:** A dataflow framework $(L, F)$ is *partitionable* [23] if we can split the framework into a finite number of "independent" frameworks $(L_i, F_i)$, each inducing a separate dataflow problem, and obtain the solution to the original problem simply by grouping all the individual solutions together. For

[5] DDFA assumes a well-founded (finite-height) function-lattice $F$. If $L$ is finite, then $F$ will also be finite (and thereby well-founded). If $L$ is infinite (but of finite height), then $F$ may or may not be well-founded. See [21] for more details.

[6] The observation that we only need to obtain the dataflow solution only for op-nodes is significant — DDFA expends "runtime-effort" to compute safe results only for a few important nodes - the op-node and the *lp*-fork nodes in the op-domain.

example, the standard framework for available expressions analysis is partitionable into sub-frameworks each of which determines the availability of a single expression. Use-def chaining and live variable analysis are other examples of partitionable problems. Constant copy propagation is an example of a problem that is not partitionable; it requires simultaneous analysis of all program variables. In the classical "bit-vector" approach, partitionable problems correspond to transfer functions that operate on each bit-position independently.

**1-related frameworks:** A framework $(L, F)$ is 1-related [21] if it is partitionable and each partitioned framework $F_i$ consists only of constant functions and identity functions. In fact, it can be shown that the only constant functions allowed in $F_i$ are $f_\top$ and $f_\perp$. 1-relatedness is characteristic of problems in which there exists at most one point along each control flow path which can affect the data being propagated. For example, say $p = (s_1, s_2, \ldots, s_k)$ is an execution path, and let $j$ be the smallest index ($j \leq k$) such that $f_{(s_{j-1}, s_j)}$ is a constant function. Clearly $f_p = f_{(s_{j-1}, s_j)}$ ; i.e. the net effect of the entire path $p$ can be summarized by the transfer function of edge $(s_{j-1}, s_j)$. In the "bit-vector" approach, 1-relatedness implies that transfer functions have two properties : (1) a bit $b$ cannot be included in the `gen` and `kill` sets at the same time (i.e. $\texttt{gen} \cap \texttt{kill} = \phi$) [7] and (2) bits are modified independently of each other (from partitionability).

A large number of classical dataflow problems, such as available-expressions, liveness, use-def-chaining etc. can be characterized as having 1-related frameworks. We assume that the dataflow problem being tackled by DDFA has a 1-related framework. While this assumption is not necessary, it simplifies matters a great deal, especially by ensuring that the summary functions we generate can be compactly represented using `gen` and `kill` sets.

### 3.3 *lp*-forks and *lp*-regions

A node is labeled as a *fork* if it has more than one successor. A fork $v$ is said to be *lossy* with respect to a dataflow solution $\Phi$ if it receives different the information propagating up to $v$ from its successors is not the same along every path. In such cases, we must make the minimal assertions that hold true along every path, leading to loss of information. Formally, a node $v$ is lossy if $\exists u \in succ(v)$ such that $\Phi(u) \neq \sqcap_{w \in succ(v)} \Phi(w)$

Each fork has a variable which determines the successor node to which control will be transferred after the fork is executed. This variable, called the predicate, will be defined before the fork is reached. The set of nodes that contain reaching definitions of the predicate are called fork-determinants. A fork $v$ is said to *predictable* at an op-node $op$ if and only if there is no fork-determinant in $paths_D(op, v)$. Note that a fork's predictability is *with respect to an op-node*. For instance, fork $v$ may be included in two different domains, induced by $op_1$ and $op_2$. Fork $v$'s direction may be unpredictable at $op_1$ because a fork-determinant lies in $paths_{D_1}(op_1, v)$, but it may be predictable at $op_2$ because there is no determinant in $paths_{D_2}(op_2, v)$.

Forks that are both lossy and predictable are called *lp*-forks. The set $L_D$ contains all the *lp*-forks in domain $D$. The op-node of the domain is also marked as a dummy *lp*-fork and included in $L_D$.

Just as op-nodes induce domains inside flow graphs, similarly *lp*-forks induce *lp*-regions within op-domains. Recall that an op-domain begins at an op-node and is bounded by other op-nodes (or program exit points). Similarly, a region begins at a *lp*-fork node and includes all nodes that can be reached from that fork without passing through another *lp*-fork or exiting of the domain. Thus, each region is bounded by *lp*-fork nodes and domain-exit nodes. Formally, region $R_v = (v, N_v, E_v, X_v)$ where :

- $v \in L_D$ is a *lp*-fork that induces the region.

[7] This can be achieved in most cases by splitting a basic block.

- $N_v = \{n \mid n \in succ_D(v)$ or $\exists (v, m_1, ...., m_k, n) \in paths_D(v, n), \;\; m_1, \ldots, m_k \notin L_D\}$.
  is the set of nodes in region $R_v$.

- $E_v = \{(m, n) \mid \; m, n \in N_v, (m, n) \in E\}$ is the set of edges between nodes in the region.

- $X_v = \{x \mid \; x \in N_v \cap (L_D \cup X)\}$ is the set of exit nodes for the region.

A region $R_v$ can be uniquely identified by the *lp*-fork node $v$ that induces it. In the following discussion we will often use a fork-node's identifier to refer to the corresponding region. Thus, the phrase, "the region $v$" should be interpreted as $R_v$.

$paths_v(m, n)$ denotes the set of all paths from node $m$ to $n$ that lie completely within the region $v$. The successors of a *lp*-fork node, are called its *predictable successors*, and are denoted by $W_v = \{w \mid v \in L_D, (v, w) \in E\}$. When execution reaches the op-node, a simple check of $v$'s predicate can determine the direction of fork $v$. Similar predictions are made for all other *lp*-forks in the op-domain. The predicted control-flow edges for all *lp*-forks are summarized by the *prediction map* $\lambda : L_D \to N_v$. If $v \in L_D$ then $\lambda(v) \in W_v$ such that the fork-direction $(v, w)$ is predicted by a check of $v$'s predicate.

Each region has a set of predecessor and successor regions; $pred_v = \{y \mid v \in X_y\}$ and $succ_v = \{y \mid y \in X_v\}$. These predecessor and successor sets are based on the (conservative) assumptions that control may flow from a fork to any of its predictable successors. At runtime, given a prediction map $\lambda$, the predecessor-successor relationships can be narrowed by eliminating those entries that are no longer possible. These narrowed predecessor and successor sets are denoted as $pred_v(\lambda)$ and $succ_v(\lambda)$ respectively; $pred_v(\lambda) = \{y \mid \; y \in pred_v, \;\; w = \lambda(y)$ and $paths_y(w, v) \neq \phi\}$ and $succ_v(\lambda) = \{y \mid \; y \in succ_v, \;\; v = \lambda(w)$ and $paths_v(w, y) \neq \phi\}$

### 3.4 Summary

Figure 3 summarizes the most important terms defined in the previous subsections that will be used again in the following sections. The assumptions that have been made are listed below.

- The lattice of data-flow attributes is well-founded (of finite-height).

- The framework is closed, monotone and distributive; i.e. all the transfer functions are monotone and distributive and it is possible to compose them and perform functional-meet operations on them. Further, the framework is partitionable and 1-related. These assumptions allow us to represent summary functions compactly and manipulate them efficiently using bitmaps.

- The heavy-weight operation at an op-node depends only on dataflow attributes propagated from other nodes that lie within the same op-domain. We will relax this assumption later, and allow optimizations across multiple op-domains.

- If a fork-node is designated as being predictable with respect to an op-node, then its fork-direction can be determined when control is at the op-node. The fork-direction will remain the same as long as control remains within the same op-domain (the direction may change the next time the op-node is encountered).

## 4 The Basic Framework

The basic DDFA framework is a two-phase algorithm. The first phase, the *builder*, is performed at compile-time. It constructs summary transfer functions for each region. The second phase, the *stitcher*, is performed

| | |
|---|---|
| $(L, F)$ | $L$ is a lattice of attributes, with top ($\top$) and bottom ($\bot$) elements and a meet ($\sqcap$).<br>$F$ is a set of functions on $L$, closed under functional-meet ($\wedge$) and composition ($\circ$).<br>$F$ contains an identity function ($f_{id}$) and a constant function ($f_\top$). |
| $D = (op, N, E, X)$ | Op-domain $D$ induced by op-node $op$,<br>with nodes $N$, edges $E$, domain-exit nodes $X$. |
| $L_D$ | Set of *lp*-forks<br>The lossy and predictable forks in op-domain $D$. |
| $R_v = (v, N_v, E_v, X_v)$ | Region induced by *lp*-fork $v$,<br>with nodes $N_v$, edges $E_v$, region-exit nodes $X_v$ |
| $W_v$ | Successors of *lp*-fork node $v$<br>These nodes are called *predictable successors*. |
| $\lambda$ | The *prediction map*<br>Control-flow directions determined at runtime for all *lp*-forks in the domain |
| $succ_v(\lambda)$ | Narrowed successor regions<br>Regions that can succeed $v$, given $\lambda$ |
| $pred_v(\lambda)$ | Narrowed predecessor regions<br>Regions that can precede $v$, given $\lambda$ |

Figure 3: A summary of important terms

on-demand at runtime whenever control reaches an op-node. It checks the predicates of all *lp*-forks in the op-node's domain and uses this control-flow information to select and apply summary functions. The following subsections describe the builder and the stitcher in detail.

### 4.1 The *builder* — constructing summary functions at compile-time

In this subsection, we describe how the *builder* constructs summary transfer functions for *lp*-regions.

The term $\psi_v(n, m)$ is used to denote a function that summarizes the effect of dataflow operations along all paths between nodes $n$ and $m$ in region $v$. For each region, we construct a set of these summary functions, $\mathbf{\Psi}_v = \{\psi_v(w, x) \mid \; w \in W_v, x \in X_v\}$. $\psi_v(w, x)$ 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 non-linear equations describe how $\psi_v(w, x)$ can be constructed. For each $x \in X_v$, we have a separate set of equations :

$$\begin{aligned} \psi_v(x, x) &= f_{id} \\ \psi_v(n, x) &= \bigwedge_{(n,m)\in E_v} (f_{(n,m)} \circ \psi_v(m, x)) \qquad \forall n \in (N_v - \{x\} - \{v\}) \end{aligned} \tag{3}$$

$\psi_v(n, x)$ for node $n$ can be obtained by taking the summary function of each successor node $m$ ( $\psi_v(m, x)$ ) composing it with the transfer function for node $n$ ($f_{(n,m)}$) and then using functional-meet ($\wedge$) to combine the summary function with similar summary functions obtained from other successors. These equations are analogous to the propagation equations in Eqs. ( 2), except that they operate on functions, not dataflow attributes. We can solve Eqs. ( 3) by starting with $\psi^0_{(n,x)} = f_\top$ for each $n \in (N_v - \{x\} - \{v\})$ and iterating to obtain new approximations to the $\psi$'s until convergence. Note that Eqs. ( 3) do not compute a summary function for the fork-node itself. We do not compute $\psi_v(v, x) = \bigwedge \psi_v(w, x)$ because the meet would be lossy. Instead, we defer the meet operation until runtime. Once the successor (say $w_i$) of the fork-node $v$ is known the meet can be eliminated by simply assigning $\psi_v(v, x) = \psi_v(w_i, x)$

There are several pragmatic problems that arise when constructing summary functions. Eqs. ( 3) manipulate transfer functions directly, instead of just applying them on elements of $L$. The $\psi$'s that are generated may not have a finite representation if $L$ is infinite. Even when $L$ is finite, the space required to encode these functions may be excessive. As pointed out by Sharir et. al. [21], the summary-function approach belongs to the class of elimination algorithms for solving dataflow problems since it uses functional compositions and functional meets in addition to functional applications. All such elimination algorithms [2, 10, 20, 19] face similar problems. In practice, these algorithms are limited to cases where the functions in $F$ possess some compact and simple representation, in which meets and compositions of elements of $F$ can be easily calculated, and in which $F$ is a well-founded semi-lattice. Fortunately, this class of problems is quite large and includes the classical "bit-vector" dataflow problems. In particular, the important subclass of *1-related* problems (introduced earlier in section 3) always have compact representations for summary functions. Further, it is easy to perform functional composition and functional meets for 1-related problems; we show how this can be achieved using `gen` and `kill` sets.

Each basic block $i$ has a 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.[8]

For the DDFA framework to work, we must show how to perform the functional-meet and functional-composition of these transfer functions using $g$ and $k$ sets.

**Composition of gen-kill sets:** If $f_1 = (g_1, k_1)$ and $f_2 = (g_2, k_2)$, then the composite function $F = f_1 \circ f_2$ is defined as $F = (g, k)$, where $g = (g_1 \cup (g_2 - k_1))$ and $k = (k_1 \cup (k_2 - g_1))$. $F$ summarizes the effects of applying $f_2$ on any attribute and then applying $f_1$ on the result. Intuitively, the `gen` set of $F$ 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 the composite function $F = f_1 \wedge f_2$ is defined as $F = (g, k)$, where $g = (g_1 \cap g_2)$ and $k = (k_1 \cup k_2)$

We also need to constructively show that there exists a set $F$ which (i) contains all the transfer functions, $f_\top$ and $f_{id}$ (ii) is closed under functional composition and functional meet (iii) forms a function-lattice of bounded depth and (iv) contains only distributive functions. These properties are actually quite simple to show. We define $f_{id} = (g = \phi, k = \phi)$ , and $f_\top = (g = U, k = \phi)$, where $U$ represents a bitmap with all bits set (i.e. all attributes are generated). If the set of attributes in $L$ is finite then the power set of these attributes is also finite. This limits the possible number of distinct $g$ and $k$ sets ensuring that the number of possible functions is finite. Hence, $F$ can be constructed and is closed. Since $F$ is finite, the height of its lattice is bounded. To show distributivity of $F$ we need to show that distributivity is preserved under functional-meet and functional-composition. This can be shown quite easily (follows from the fact that $f_i(x) = (x - k_i) \cup g_i$).

### 4.1.1 Data structures and Example

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 in the CFG, we keep a data-structure called an *op-domain table*. An op-domain table contains one entry record for every region within the op-domain.

[8] 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, our symbol $g$ represents the standard `kill` set and the symbol $k$ represents the `gen` set.

| region | predicate check | fork dir. | $\Omega$ | pred-ecessors |
|---|---|---|---|---|
| OP | $\times$ | $\times$ | Cache | $\times$ |
| b | b = ? | ? | ? | ? |
| c | c = ? | ? | ? | ? |

Op-domain-table ( $op_1$ )

| fork directions. ( $\lambda$ ) | $\Omega$ |
|---|---|
| ( b = 2, c = 0 ) | { $\alpha$ } |
| ( b = 1, c != 0 ) | { $\beta$ } |

Cache

| (dir, exit) | { gen }, | { kill } |
|---|---|---|
| b = 1, c | { $\beta$ } | { $\alpha$ } |
| b = 2, c | { $\alpha$ } | { $\beta$ } |
| b = 3, c | { $\alpha$ } | { } |

Region-table b

| (dir, exit) | { gen }, | { kill } |
|---|---|---|
| c = 0, op2 | { $\alpha$ } | { } |
| c != 0, op1 | { $\beta$ } | { } |
| c != 0, op2 | { $\alpha$, $\beta$ } | { } |

Region-table c

Figure 4: Data-structures created by the builder and passed to the runtime stitcher for the op-domain in the running example. The fields marked by **?** are to be filled in by the runtime stitcher. The cache has been pre-filled by the builder for two of the paths.

These records contain the following fields: (1) a pointer to a *region-table* for each region, (2) a pointer to the code that must be executed to determine the direction of each region's *lp*-fork, (3) scratch space for storing the fork-direction (4) scratch space for storing intermediate dataflow results computed during stitching, and (4) a place-holder for the set of predecessors $pred_v(\lambda)$ for each region. The predecessor-set cannot be computed at compile-time because it depends on the prediction map $\lambda$ (the predicted fork-directions). Each region table contains summary functions indexed by (fork-direction, region-exit) pairs. Not every region-exit is reachable from every fork-direction, so the region tables are stored in a sparse format.

Figure 4 illustrates these data structures for the running example. It shows the op-domain table for the first op-node and the region-tables corresponding to the *lp*-forks in the first op-domain. The region-table for the op-node is not shown because the region has just one node and does not need summary functions. The second region (region b) is induced by the fork at `switch(b)`. The switch forks in three directions all of which to the same region-exit. Hence, there are three entries in region-table c, one for each (fork-direction, region-exit) pair. As region b does not contain any forks, its summary functions can be constructed simply by composing the `gen` and `kill` sets of the nodes along each of the three possible paths. The third region (region c) is created by the `if(c)` conditional. This conditional forks in two directions `c=0` and `c!=0` and has two region-exits $op_1$ and $op_2$. The region has three entries, not four, because there is not path from the `c=0` fork-direction to exit $op_1$. Region c contains two forks ( `if(a)` and `if(d)`). To construct functions that can summarize the effect of these forks, we need to perform functional-meet operations as specified in Eqs.( 3). Figure 5 demonstrates the construction of one of these summary functions — $\psi_c(\mathtt{c!=0}, op_2)$.

The sparse format of the region-tables encodes an important piece of information that is not clearly shown by the simple tabular structures used in Figure 4. The builder analyzes each region to determine which fork-directions can lead to which region-exits. This information is not thrown away – it is kept in the form of parameterized successor sets for each region : $succ_v(w_i) = \{y \mid \ y \in L_D, w_i \in W_v, paths_v(w_i, y) \neq \phi\}$ At runtime, when the prediction-map $\lambda$ is available, computing the narrowed successor sets is merely a

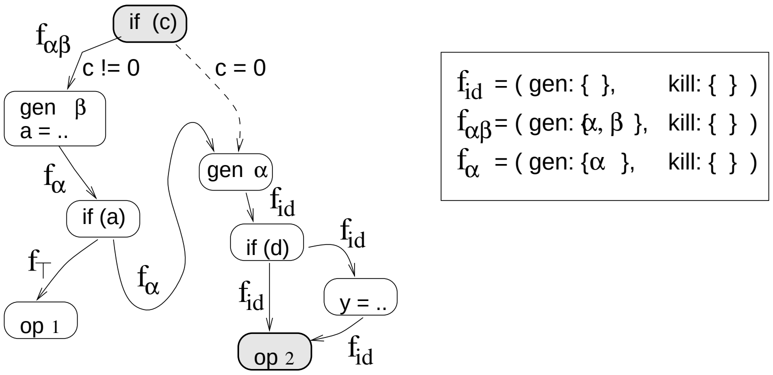



Figure 5: Construction of the summary function for (`c!=0`, $op_2$) entry of region-table `c`. The builder starts by assigning $f_{id}$ to exit node $op_2$, and $f_\top$ to all other nodes. The iterative algorithm propagates these values backwards until convergence. Edges in the figure have been labeled with the final $\psi$'s assigned to their targets. The final result is $\psi_c(\texttt{c!=0}, op_2) = f_{\alpha,\beta}$. Note that when computing $\psi_c(\texttt{c!=0}, op_2)$, values are not propagated along fork-direction (`c = 0`) and node $op_1$ is treated like a non-exit node and is initialized to $f_\top$, not $f_{id}$.

matter of selecting one of these parameterized successor sets, i.e. $succ_v(\lambda) = succ_v(w_i)$ where $w_i = \lambda(v)$.

The builder also generates a cache structure for holding previously computed dataflow properties. This cache is associatively indexed by the prediction-map $\lambda$. It may not be feasible to store the dataflow results for all combinations of fork-directions. In such cases we can use a hash-function that hashes the prediction-map to an entry in the cache and employ a simple cache-replacement policy such as random or fifo. The compiler can prime the cache by statically computing the dataflow properties for the "likeliest" prediction-maps. For instance, in Figure 4, the cache has been pre-filled with dataflow results for two different sets of fork-directions – (b=2,c=0), and (b=1, c!=0).

### 4.2 The *stitcher* — applying summary functions

This section describes the *stitcher*, the runtime phase of DDFA. The *stitcher* is invoked every time control reaches an op-note. On each invocation, it checks the predicates of every *lp*-fork in the domain and determines their directions. The prediction-map selects appropriate summary functions from region-tables which are then used to propagate dataflow information backwards to the op-node.

We pause to make some observations about regions and their summary functions. A region is like a big basic block and its summary function $\Psi$ is analogous to a block's transfer function. However, this analogy is incomplete. Basic blocks are single-entry single-exit units while regions have multiple exits. Moreover, each of the predictable successors of a region's *lp*-fork can be viewed as a potentially different entry point (with only one entry is feasible at any time). Thus regions are equivalent to (possibly overlapping) multiple-entry, multiple-exit hyper-blocks. The *builder* has prepared for this complexity by keeping separate summary functions for each (entry, exit) pair, i.e. $\Psi_v = \{\psi_v(w, x) \mid w \in W_v,\ x \in X_v\}$. In addition, the builder has also prepared parameterized successor sets $\{\ succ_v(w_1), succ_v(w_2), \ldots\}$, where $succ_v(w_i)$ represents the set of regions that can succeed region $v$ if the fork-direction were known to be to be $(v, w_i)$.

The stitcher is invoked every time execution reaches an op-node. It checks the predicate corresponding to each *lp*-fork in the op-domain $D = (op, N, E, X)$. The results of these checks constitute the prediction-map $\lambda$ which maps each *lp*-fork node to one of its successors. These fork-directions will remain true as

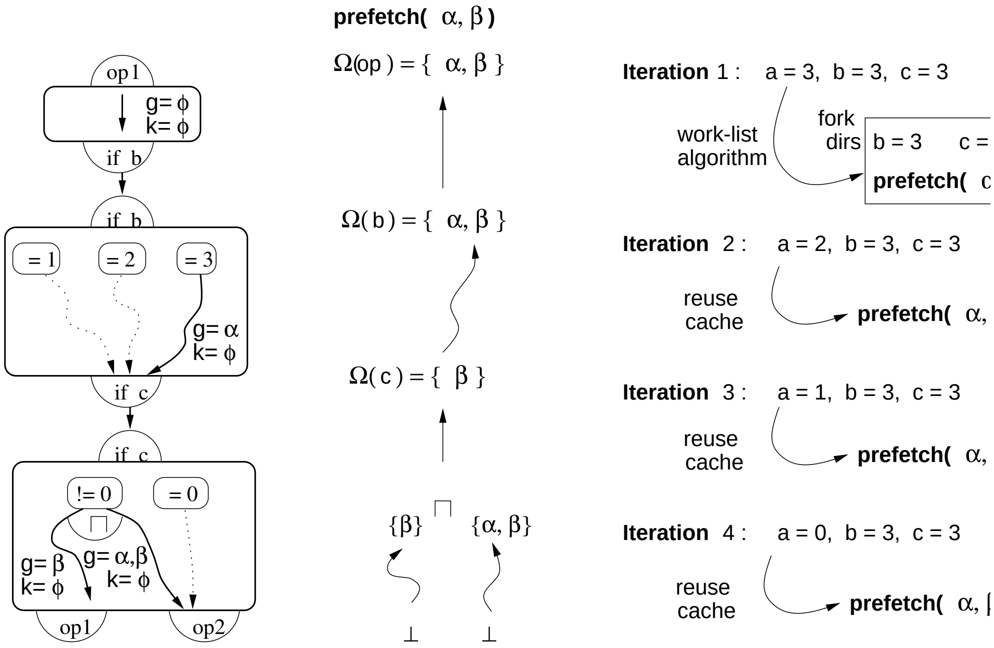



Figure 6: Operation of the stitcher on the running example. The summarized graph is shown on the left with the edges selected by the prediction map $\lambda$ = (`b = 3, c != 0`) marked in bold. The work-list algorithm performs iterative analysis on this summarized graph, reaching the fix-point values shown alongside. The result, $\{\alpha, \beta\}$, is used to prefetch data and is also cached. The cached result is reused in the next three iterations because fork-directions do not change.

long as control remains in the op-domain; they may change direction the next time control encounters an op-node. Once we have the prediction-set narrowing the successor relationships between regions is trivial; this is merely a matter of selecting the correct parameterized successor set i.e. $succ_v(\lambda) = succ_v(w)$ where $w = \lambda(v)$.

We can now describe the backward flow equations that must be solved by the stitcher in order to obtain dataflow information at the op-node of the domain.

$$\begin{aligned} \Omega(x) &= \quad \bot && \forall x \in X \\ \Omega(v) &= \bigsqcap_{u \in succ_v(\lambda)} \psi_v(\lambda(v), u)(\,\Omega(u)\,) && \forall v \in L_D \end{aligned} \tag{4}$$

Here, $\Omega(v)$ represents the dataflow properties at the *lp*-fork $v$. The $\Omega$'s of adjacent regions are related to each other in exactly the same way that adjacent basic-blocks are related to each other in Eqs. ( 2). Except for the fact that we are using summary functions of the form $\phi_v(w, u)$ instead of transfer functions of the form $f_(n, m)$, the two sets of equations are equivalent. It is worth noting that Eqs.(4) do not yield dataflow information at every node in the control-flow graph. They compute $\Omega(v)$(s only for $v \in L_D$, i.e. the *lp*-fork nodes in the domain. As we are only interested in $\Omega(op)$ and $op \in L_D$, this does not create a problem.

The solution procedure for Eqs.( 4) has been adapted from Kildall's iterative algorithm. Initially $\Omega^0 = \top$ for each $v \in L_D$. We iteratively re-apply Eqs. ( 4) to obtain new approximations to the $\Omega$s until convergence. The next subsection outlines an implementation of this iterative algorithm, which incorporates *caching* of dataflow results. Caching is very important because it allows dataflow results to be reused — this can substantially reduce runtime overheads.

### 4.2.1 Stitcher algorithm and Example

Figure 7 presents the stitcher algorithm. The stitcher consists of four parts, marked as statements S1 through S4. The first statement (S1), checks the predicates of all *lp*-forks in the domain and determines their directions. The second statement (S2), checks the cache to see if dataflow results for this particular set of directions are cached. If so, the cached results can be used directly to prefetch data (S4). If the results are not cached, an iterative work-list algorithm is invoked (S3). This algorithm, shown on the right in Figure 7, is similar to standard iterative work-list algorithms used for dataflow analysis except that it operates on regions, instead of basic blocks. The work-list contains tuples of the form `(r, x, In)`, which denotes attributes `In` flowing into region `r` through exit node `x`. In each iteration, a tuple is removed from the work-list and the transfer function $\psi_r(\lambda[r], x)$ is applied to `In`, the incoming attributes. The result of this function-application is combined with region $r$'s old dataflow attributes using the meet operator $\sqcap$. The result of the meet is used to update region $r$'s attributes and is also fed to its predecessors by adding new tuples into the work-list. This process is repeated until convergence.

Figure 6 illustrate the operation of the stitcher using the running example The program iterates through the `repeat-until` loop four times, so the `prefetch()` instruction inside the loop will be invoked four times.

In the first iteration through the loop, the values of `b` and `c` are `3` and `3` respectively, so $\lambda$, the prediction map is `(b = 3, c != 0)`. Assuming the builder has primed the cache with the entries shown in Figure 4, `ChkCache()` will fail, and the work-list algorithm will be invoked. The work-list is initialised with tuples $\{(c, op_2, \perp), (c, op_1, \perp)\}$, corresponding to the two exits of the op-domain. All regions are initially assigned the dataflow attribute $\top$. Tuples are then removed from the work-list and transfer functions are applied. For example, after removing the first tuple $(c, op_2, \perp)$, the algorithm will apply $\psi_r(c! = 0, op_2)$ to $\perp$ and get $\{\alpha, \beta\}$. This causes $\Omega(c)$ to be updated to $\{\alpha, \beta\}$ and a new tuple $(b, c, \{\alpha, \beta\})$ is inserted into the work-list. The algorithm will reach fix-point in two iterations — the final assignments are shown in Figure 6. The result, $\Omega(op) = \{\alpha, \beta\}$ is fed to the prefetcher which uses this information to bulk-prefetch both remote variables. This dataflow result is also placed in the cache. When the stitcher is reinvoked in the next three iterations of the loop, it will find that the fork-directions are the same, i.e. `(b = 3, c != 0)`. The dataflow results will be picked up from the cache and the prefetcher will prefetch $\{\alpha, \beta\}$ on each of the following iterations.

## 4.3 Pragmatics: space and time

The primary space requirement is for storing the summary functions for each *lp*-region. We note that only very loose bounds can be placed on the number of directions that a fork can take. For example, a switch statement could fork into very many directions (order of $N$, the number of CFG nodes) and different summary function would need to be stored for each of these directions. We bound the number of fork-directions to a small number $W$ — when the number of fork-directions gets exceeds $W$, we perform a functional-meet on all the extra directions and store a single summary function for all of them. The number of exit nodes in a *lp*-region is bounded by the maximum number of *lp*-regions in an op-domain (say $L$). Thus the number of summary functions to be stored in a *lp*-region is bounded by $W \times L$. Each op-domain can have $L$ *lp*-regions so the total number of summary functions stored within an op-domain can be as large as $W \times L \times L$. Note that this is a very conservative estimate. It assumes that control can reach every exit of an *lp*-region from any of the fork directions. It also assumes that every *lp*-region connects to every other *lp*-region (a circumstance that would require a very convoluted set of mutually recursive functions or inter-connecting jump-tables). Even so, we note that $L$ is the number of predictable fork points within an op-domain, which we expect to be a reasonably small number; hence, the number of summary functions that need to be stored for the *lp*-region should not be excessive. It is also possible to bound the number of *lp*-regions in a op-domain by specifying

| **Helper Functions** | **Functionality** |
|---|---|
| `succ(r, λ)` | regions succeeding region `r`, given fork-direction $\lambda$ |
| `pred(r, λ)` | regions preceeding regon `r`, given fork-direction $\lambda$ |
| `SetPredecessors(r)` | prepares pred(r,$\lambda$) sets by transposing pre-computed succ(r, $\lambda$) sets |
| `G(r,dir,exit)` | gen component of summary-function $\psi_r$(dir,exit) |
| `K(r,dir,exit)` | kill component of summary-function $\psi_r$(dir,exit) |
| `remove(list)` | returns (and removes) a region from worklist |
| `ChkPredicate(r)` | checks predicate to get direction of fork `r` |
| `ChkCache(opd,dir)` | checks if dataflow results are cached |
| `StoreCache(opd,dir,flow)` | stores dataflow results in cache |
| `SetPrefecthList(flow)` | fills datastructures used by the prefetcher |

```
function stitcher(domain *opd)

 S1:  foreach region r
        λ[r] ← ChkPredicate(r)

 S2:  Ω ← ChkCache(opd,λ)

 S3:  if ( not found in cache )
        SetPredecessors()
        Ω ← work(opd, λ)
        StoreCache(opd,λ,Ω)

 S4:  SetPrefetchList (Ω)
end
```

```
function work(opd, λ)
 foreach region r
    out[r] ← ⊤
    foreach exit x in succ(r, λ)
       if x.type = Op
          wkList ← (r,x,⊥)

while (wkList ≠ φ)
    (r,x,In) ←  remove(wkList)
     oldOut ← out[r]
     tmp  ← G(r,λ[r],x) ∪
             (In - K(r,λ[r],x))
     out[r] ← out[r] ⊓ tmp
     if (out[r] ≠ oldOut)
        foreach region p in pred(r,λ)
           wkList ← wkList ∪
                         (p,r,out[r])
returns out[Op]
end
```

Figure 7: The stitching algorithm. The workList contains tuples of the form `(r, x, In)`, which denotes attributes `In` flowing into region `r` through exit node `x`. In each iteration, a tuple is removed from the work-list and the appropriate transfer function applied; this process is repeated until convergence.

the maximum number of *lp*-forks that should be checked at the op-node. All other *lp*-forks would then be considered unpredictable and would be completely analyzed at compile-time. This flexibility allows the DDFA framework to trade accuracy for runtime space as well as execution time. We would like to point out that sacrificing accuracy by bounding the number of fork-directions and *lp*-regions in an op-domain does not make DDFA unsafe, it only reduces the benefits of having future control-flow information.

The primary concern regarding time is that the stitcher executes an iterative algorithm every time control reaches an op-node. In this regard, we point out two alleviating factors. We note that the results of the runtime analysis at an op-node are *cached*. The cache entries are uniquely identified by the fork-directions for all the *lp*-forks within the op-domain. The cache of dataflow results need not be large. A two-entry cache should suffice for most cases as it takes care of the common case where one path is more taken more frequently taken than others. A similar technique, called *record-replay* [18] has been shown to work well for optimizing communication in irregular parallel programs. Even in cases where cache-reuse is low, overheads can still be limited. The stitcher operates on the high-level *region-graph*, which is much smaller than the original control-flow graph. In the worst case, we can bound the time allowed to the stitcher. If this time is exceeded without reaching a fix-point, the stitcher is stopped and we can fall back on the results of compile-time analysis (which can be stored in the cache for use as a fall-back option).

# 5 Proofs

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 no worse than a compile-time meet-over-all-paths solution. Our proofs are largely adapted from the proofs of the two-phase interprocedural analysis used in [21].

We begin by defining a few terms that will be used in the proofs. The domain under consideration is $D = (op, N, E, X)$, using terminology of section 3. We need only consider a single op-domain because DDFA treats domains independently of each other. The domain contains a set of *lp*-fork nodes $L_D$ and each of these fork-nodes induces an *lp*-region $R_v = (v, N_v, E_v, X_v)$. A sequence of nodes $p = (n_1, n_2, .....n_k)$ is said to be a *valid path* if all its edges are in the domain, i.e. $1 \leq i \leq k$, $(n_i, n_{i+1}) \in E$. The *length* of the path is the number of edges in it ($k-1$ in the above notation). $paths(m, n)$ is the set of *all* paths leading from $m$ to $n$. A path $p = (n_1, n_2, \ldots n_k)$ can be *decomposed* into sub-paths (or *segments*) — the decomposed path is denoted as $p = (n_1, n_2, \ldots, n_a \| n_a, n_{a+1}, \ldots n_b \| n_b, n_{b+1}, \ldots \| \ldots \| \ldots n_k)$. where $n_a, n_b, \ldots$ etc. are the segmentation-points. Our notation includes the segmentation-points twice in the decomposed path. A *region-segment* is a segment that begins at a *lp*-fork node and ends at one of the exit nodes of the region induced by that fork. Formally, segment $s = (n_0, n_1, .....n_k)$ is a region-segment if $n_0 \in V_D$ and $n_k \in X_{n_0}$ and and for $1 \leq i \leq k$, $n_i \notin V_D$.

**Path Decomposition Lemma:** 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. [9]

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

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 satifies the property $x_i = v_{i+1}$. If these constraints are satisfied then $S \in paths(v_0, x_k)$.

Decomposability and uniqueness are related properties and can be proven together. Note that the path-decomposition lemma constrains the path $p$'s start node $v_0$ to be a *lp*-fork node and the exit $x$ to be a domain-exit node (and therefore an exit-node of atleast one region). If there are no *lp*-forks in the path

[9] There are subtle differences between our path decomposition lemma and a similar lemma used in [21]. 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 much more intuitive because every CFG node of belongs to only one procedure while nodes in our region-segments can belong to multiple regions.

$p = (v_0, \ldots, x)$, then $p$ itself is a region-segment and the entire path is the trivially unique solution. If the path does contain other *lp*-forks inside it, then a decomposition can be obtained by segmenting the path at *lp*-fork node. This is a valid decomposition as each segment $s_i = v_i \ldots v_{i+1}$ has $v_i \in V_D$ and $v_{i+1} \in X_{v_i}$ and all the intermediate edges are not *lp*-forks and therefore in $E_{v_i}$. This decomposition in region-segments is unique. Any other choice of segmentation points would result in atleast one segment having a *lp*-fork node in the middle; such a segment cannot be a valid region-segment. The "existence" property, follows directly from the fact that compositions of valid paths yield other valid-paths as long as the end of the first path is the start of the second path. This conditions are true – each region-segment is a valid path and the end of one segment is the start of the next segment.

## 5.1 Termination

We need to show that both the *builder* and the *stitcher* algorithms terminate.

Termination of the *builder* follows from the following observations.

1. The summary function of a node never rises (in the function-lattice) from one iteration to the next; i.e. $\psi^{i+1} \leq \psi^i$. This follows from monontonicity of transfer functions in $F$.
2. In each iteration of the builder one of three things must happen : (1) the algorithm terminates because the work-list is empty or (2) the summary function of atleast one node in the region changes — i.e. $\exists n \in N_v$ such that $\psi^{i+1}(n, x) \leq \psi^i(n, x)$ or (3) the size of the work-list is reduced. Only possibility (2) increases the size of the work-list. However, this possibility can happen only a bounded number of times, because there are a finite number of nodes in a region and the depth of the function-lattic $F$ is bounded.

The *stitcher* phases terminates because :

1. The dataflow attribute assigned to a region never rises from one iteration to the next; i.e. $\mathbf{\Omega}^{i+1}(v) \leq \mathbf{\Omega}^i(v)$. This follows from monontonicity of summary functions.
2. In each iteration of the stitcher one of three things must happen : (1) the algorithm terminates because the work-list is empty or (2) the dataflow attribute of atleast one region changes — i.e. $\exists v \in l_D$ such that $\mathbf{\Omega}^{i+1}(v) \leq \mathbf{\Omega}^i(v)$ or (3) the size of the work-list is reduced. Only possibility (2) increases the size of the work-list. However, this possibility can happen only a bounded number of times, because there are a finite number of region in a domain and the depth of the attribute-lattice $L$ is bounded.

We are now ready to prove the other two results.

## 5.2 DDFA is safe

For any feasible execution path $p$, from $op$ to an exit node $x \in X$ the dataflow properties that flow up the path would be ideally determined by the composition of the transfer functions for each node along the path. That is, if path $p = (op..., x_0 \| v_1, ...x_1 \| ... \| v_j...x)$ then $f_p(op) = f_x \circ \ldots \circ f_{v_j} \circ \ldots \circ f_{v_1} \circ \ldots f_{op}(\perp)$. To prove safety, we need to show that $\Omega(op)$ computed by DDFA will be a conservative approximation of $f_p(op)$. We need to prove that :

$$\Omega(op) \quad \leq f_p(op) \quad \forall p \in paths(op, x), x \in X \tag{5}$$

We outline the steps of the proof below.

1. Every region-segment $(v_i, w_i, \ldots, x_i)$ in the unique decomposition of path $p$, will have a corresponding summary function $\psi_i(w_i, x_i)$ constructed by the builder.

2. $\psi_i(w_i, x_i)$ is a safe approximation for the composition of transfer functions of nodes that lie in the $i$-th region-segment of path $p$ ; i.e. $\psi_i(w_i, x_i) \leq f_{w_i} \circ \ldots \circ f_{x_i}$. This is shown by an inductive proof on the segment-length, using the safety property of functional-meet and functional-composition. After this step, we know that the builder is safe.

3. For every segmentation-point $v_i \ldots x_i || v_{i+1} \ldots$ in path $p$, there will be a corresponding backward propagation of attributes from region $v_{i+1}$ to region $v_i$. This essentially translates to the requirement that $v_{i+1} \in succ_{v_i}(\lambda)$ which is directly based on the safety of the predictability analysis (for which we use safety of reaching-definitions analysis). After this step we know that the region-graph used by the stitcher includes the path $p$.

4. The iterative stitching algorithm on the region-graph is safe. This requires an inductive proof on the number of region-segments in path $p$.

### 5.3 DDFA matches Meet-over-all-paths solution

In this section, we show that DDFA's solution is at least as good as a compile-time meet-over-all-paths (MOP) solution ($\Upsilon$). That is we need to show that :

$$\Upsilon(op) \leq \Omega(op) \tag{6}$$

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 at least as good as a MOP solution. Consider a crippled version of DDFA (called p-DDFA, for poor-man's DDFA) that is identical to DDFA except that the stitcher does a poorer job of selecting which summary functions to apply. Unlike DDFA's stitcher which selects functions based on fork-direction as well as the exit node, p-DDFA selects based only on the exit node. This may activates a larger number of summary functions at every region, all of which are applied to the incoming flow attributes. The results are combined by a meet operation ($\sqcap$). Let us denote the results obtained by p-DDFA as $\Omega^*(op)$ We can show that p-DDFA can perform as well as a $MOP$ analysis, i.e :

$$\Upsilon(op) \leq \Omega^*(op) \leq \Omega(op) \tag{7}$$

First, we introduce an alternative way of obtaining the dataflow results at the op-node, called BOA (builder-only approach). The BOA approach ignores all *lp*-forks and treats the entire op-domain as a single region. It computes a summary function (multiple functions if there are multiple exits) for the entire domain. BOA's stitcher is much simpler than DDFA's stitcher. If the domain has a single exit, then there will be only one big summary function for the entire domain; the stitcher can get the dataflow result for the op-node by merely applying this function to $\perp$. If the domain has multiple domain-exits then there will be a different summary function corresponding to each exit. Each of these summary functions is applied to $\perp$ and the results are combined using $\sqcap$.

**Lemma 1** *p-DDFA's solution is at least as good as a builder-only solution (BOA).*

The proof of Lemma 1 is based on the "existence" property of the path-decomposition lemma. Any sequence of region-segments used by p-DDFA's stitcher is a valid path. This path must also be considered

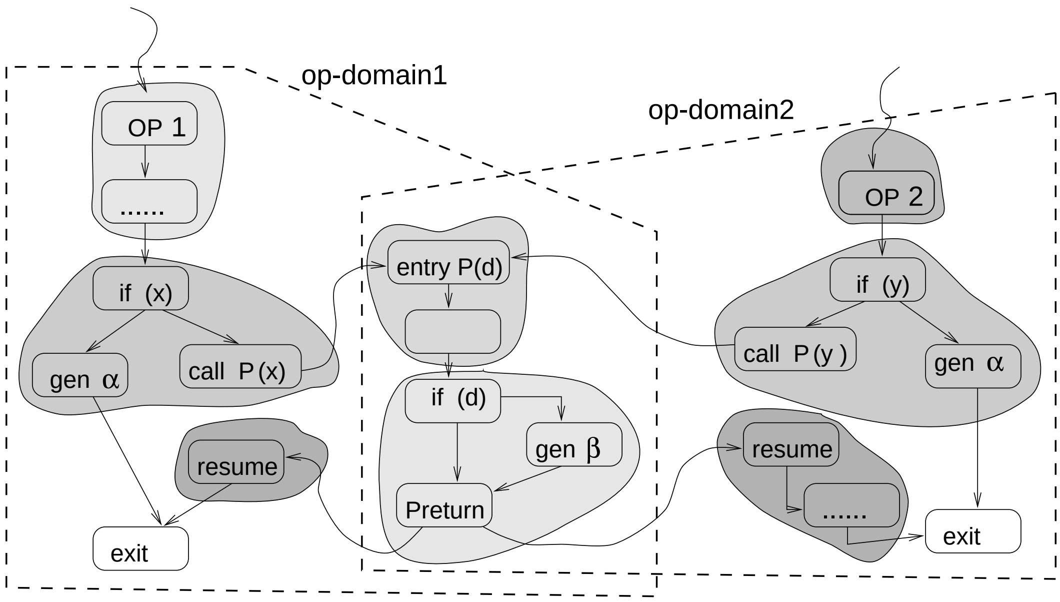



Figure 8: Interprocedural Example. Procedure P is called from two sites. At both sites, the fork inside P can be predicted. As a result, the op-domains for the two sites can share the common region's data-structures.

by the BOA builder when computing the domain's summary function. An inductive proof (on the number of segments in any such path) can be used to show that the domain summary function will be no better than the p-DDFA approach of iterating on the region-graph.

Note than BOA is computing the domain's summary function for the entire op-domain using an iterative process. We can show that the iterative computation of the summary function is just as good as a functional meet over all paths (FMOP) computation of the summary function.

**Lemma 2** *The BOA solution is at least as good as a FMOP solution.*

This part is relatively easy - 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 [14].

**Lemma 3** *The FMOP solution is at least as good as the MOP solution.*

This step shows that operating on the functional domain does not hurt the quality of the solution. For this we just cite previous work on summary functions, in particular Theorem 7-3.4 in [21].

From the Lemmas stated above we have : DDFA $\geq$ p-DDFA $\geq$ BOA $\geq$ FMOP $\geq$ MOP.

# 6 Extensions to the basic framework

In this section, we extend the basic framework to incorporate interprocedural analysis, higher-order functions and separately compiled functions. We also describe how the runtime analysis can be modified to look further ahead in the execution – the goal being to optimize over multiple op-domains.

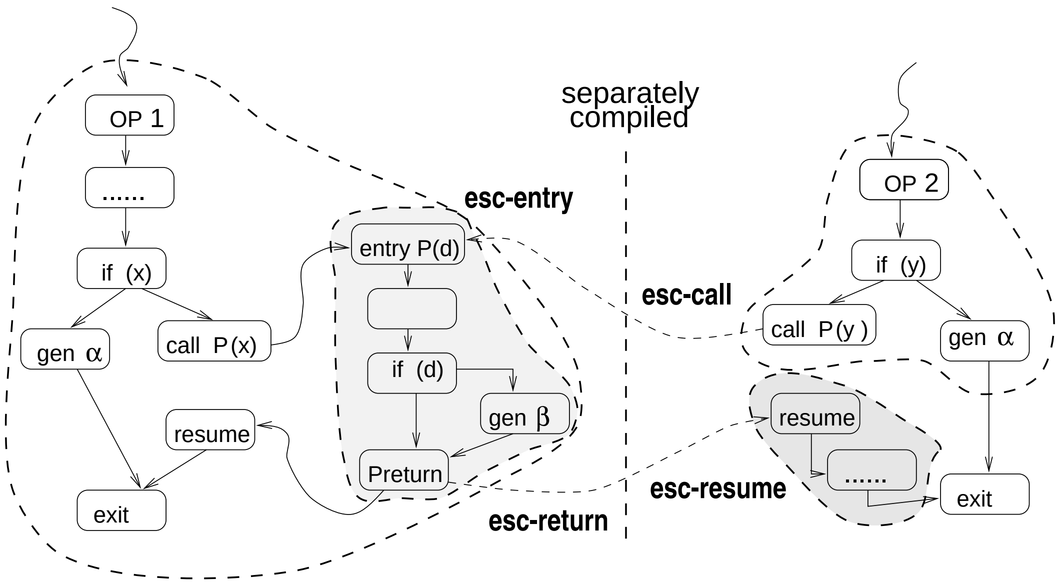



Figure 9: Separately Compiled Code. Procedure P is called from the separately compiled code on the right, as well as from within its own module. The shaded regions show the new op-domains that would be created to handle the `esc-call`.

### 6.1 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 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. Control that arrives at a procedure via a call-node can only leave the procedure via the matching resume-node. Therefore, 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 the procedure can be a part of a large number of op-domains. Since a region table is generated for every region in an op-domain, this can result in a large number of replicated region-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 [13] and suggest a similar solution. For each op-domain that extends into a procedure, we mark each fork within the procedure as *lp* (lossy and predictable) 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 that 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.

To increase the possibility of sharing, we ensure that *lp*-regions don't straddle procedure boundaries. We do this by marking entry-nodes and resume-nodes as dummy *lp*-fork nodes. Since these forks have only one child, they are always predictable – we've just imposed a false lossiness onto them.[10] Figure 8(a) illustrates the interprocedural representation and sharing of regions. In the figure, the regions of procedure P are shared by both op-domains.

### 6.2 Separately compiled code and higher-order functions

The primary problem that has to be dealt with when extending DDFA to separately compiled code and higher-order functions is that the complete control-flow graph is not available at compile-time. This makes it impossible to perform a precise predictability analysis, and correctly mark of forks as *lp*-forks. In these situations, we have to make some conservative approximations which may lose precision across call-sites but preserve it on each side of the boundary. The strategy we have chosen is to insert new *op-nodes* at every control-flow point at which control may arrive from an unknown place. This strategy assumes that it is possible to safely insert new op-nodes (e.g. it is safe to insert new bulk-prefetch calls).

We refer to calls to separately compiled code and to unknown functions as *esc-calls*. The corresponding resume-nodes are marked as *esc-resumes*. Similarly, we mark entry- and return-nodes of *escaping* functions (functions that can be called from unknown sites) as *esc-entries* and *esc-returns*. See Figure 9 for an example. To handle the fact that a function may have both known and unknown call-sites, we analyze two versions of escaping functions – a non-escaping version and an escaping version. The non-escaping version is analyzed as usual and allows op-domains to straddle procedure boundaries. For the escaping version, we place an new op-node at the esc-entry node. We also place an op-node at the esc-resume node corresponding to sites that may call unknown functions. No dataflow information can flow across these call-sites. During runtime $\perp$ is passed up from these call-nodes. Figure 9 shows an example of separately-compiled function. The shaded regions show the new op-domains that would be created to handle the *esc-calls*. Higher-order function calls are treated in exactly the same manner.

### 6.3 Dynamic merging of op-domains

DDFA restricts the domain of influence of each operation to a single op-domain. 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 form boundaries of predictability. However, this boundary is not always strict. There may very well be situations where *all* the fork-directions predictable at an op-node $op_2$ could also be predicted at one of its preceding op-node (say, $op_1$). These circumstances can be detected at compile-time using simple reaching-definitions analysis. In these cases, $op_2$ is designated a half-op node (or *hop-node*) with respect to $op_1$. The runtime stitcher when invoked at $op_1$, can detect that control may reach $op_2$. On detecting that $op_2$ is a hop-node with respect to $op_1$, the stitcher can continue processing $op_2$'s domain-table. by checking $op_2$'s fork directions and adding $op_2$'s regions to its own. The hopping process continues until control reaches an exit that is not a hop-node.

For instance, in our running example (Figure 1), the *lp*-fork (`if(d)`) inside $op_2$'s domain is predictable even at $op_1$. When the stitcher at $op_1$ detects that control can exit the loop and reach $op_2$ it stitches in $op_2$'s *lp*-region to its own set of regions and prefetches the combined result. In the example, the prefetch instruction is already prefetching ($\{\alpha, \beta\}$), so dynamic merging does not make a difference.

[10]Marking a non-lossy node as lossy does not hurt DDFA's correctness or the quality of the results; it only increases the number of regions that must be analyzed at runtime, thereby increasing space and time overheads.

```
proc grep_on_hostA( )
  clean_&_go("A");
  A_fopen();
   .......
return

proc grep_on_hostB()
  clean_&_go("B");
  B_fopen();
   .....
return

proc clean_&_go(hostname)
     finalize_state();
     go(hostname);
return
```

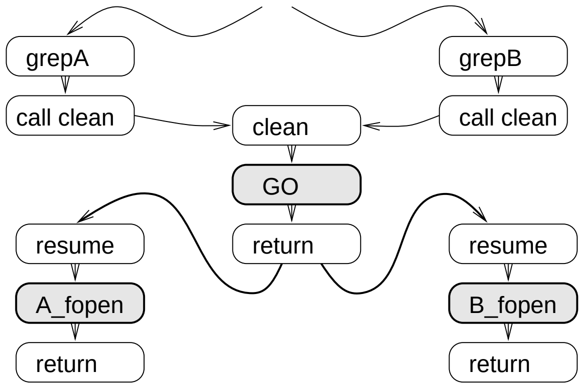



Figure 10: A simple program that cannot be successfully linked using a conservative approach. After the migration at `go`, the linker at the new site would try to bind `A_fopen` as well as `B_fopen` because both are deemed reachable. Since these functions are defined on different sites, the linker would fail. Checking the calling-context at runtime avoids this problem.

# 7 Other Applications of Deferred Data-Flow Analysis

This paper has used bulk-prefetching as a running example for DDFA. In this section we briefly outline other dataflow problems where DDFA is applicable.

## 7.1 Compiler-directed Dynamic Linking for Mobile Programs

The problem of dynamic linking for mobile programs is described in [1]. In mobile programs, there are migration points (called `go(newhost)` instructions) at which the execution is suspended on the current host and is resumed at a named host. After migration, the mapping between program names and local procedures have to be re-established. The problem is to determine the names that the mobile program can be refer to while on a particular host and to dynamically link these in at the migration point. A simple compiler-based solution would be to collect all names that can possibly be referenced between one migration point and the next and to link these names when the program migrates. While this approach is safe, it can be overly restrictive and can result in link-failures for "reasonable" mobile programs. For example, in Figure 10, the call to `go` is inside the function `clean_and_go`, which is called from procedures `grep_on_A()` and `grep_on_B()`. Ignoring the calling context (which will be known at runtime) would the linker to link in both `A_fopen` and `B_fopen`. Since these two `fopen` functions are defined on different hosts, the linker would fail. To tackle this problem, Acharya et. al. [1] propose a compiler-directed technique that uses the runtime call-stack to prune the set of future control-flow paths. The linker links in only those names that may be reached along these pruned paths. For the example in Figure 10, the algorithm can check the call-stack at runtime and determine the direction of the return-fork. This allows it to link in only one of the `fopen`, and prevent a link-failure.

The solution proposed in [1] only handles return-forks. Deferred data-flow analysis can improve the solution presented in [1] by taking all predictable conditionals into account while pruning future paths. and yields a smaller set of names to link in.

Figure 11 illustrates the differences between the static solution, the call-stack algorithm of [1] and the DDFA solution. The mobile program shown in the figure will migrate twice. It migrates from `home` to `newsite` where it performs a host-specific operation (`fgets`) after which it migrates back to `home`. The first call to `go()` is made through the call-sequence A $\rightarrow$ B $\rightarrow$ `go`, and the second call is is made through the call-sequence A $\rightarrow$ D $\rightarrow$ B $\rightarrow$ `go` or A $\rightarrow$ U $\rightarrow$ B $\rightarrow$ `go`. The corresponding flow graph is shown

```
proc A (newsite)
home = hostname();
call B(newsite);
if(newsite.os == dos)
   call D();
else
   call U();
process(data);
return
```

```
proc B(site)
  cleanUp();
  go(site);
return
```

```
proc U()
  unix_fgets(data);
  call B(home);
return
```

```
proc D()
  dos_fgets(data);
  call B(home)
return
```

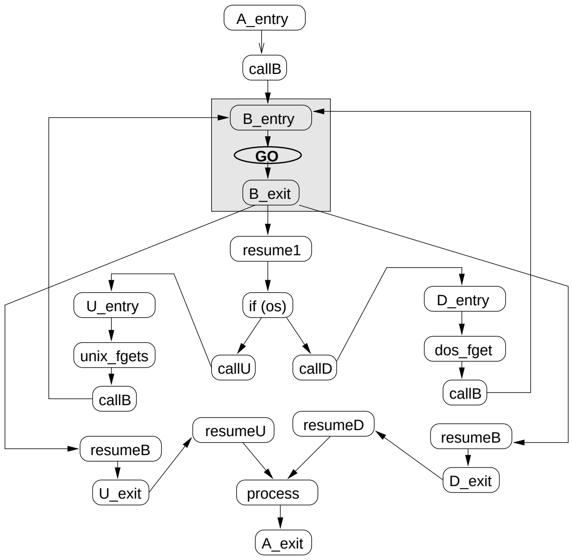



Figure 11: The figure on the left shows a mobile program that migrates migrates from `home` to `newsite`, performs a host-specific operation (`fgets`) at `newsite` and then migrates back to `home`. The figure on the right shows the program's interprocedural flow graph. Static compile-time collection of link-sets for the `go` call would require linking in both `unix_fgets` and `dos_fgets` causing a link-failure. An algorithm that inspects the call-stack at runtime and uses it to figure out the calling-context can avoid a link-failure at the second `go` but would still fail at the first `go`. DDFA can predict the direction of the conditional-fork, allowing it to safely link in only one version of `fgets`.

alongside. The dataflow problem being solved is very simple, merely requiring that all names reachable from a `go` call be collected. The transfer function for each basic block has a `gen` set that contains all the names referred to in that basic-block. Names are never killed, so `kill` sets are empty. The meet-operator is set-union.

Using standard compile-time dataflow analysis, we find that the *link-set*, i.e. the names that can be reached from the `go`-node are : $N = \{$ `U`, `D`, `B`, `dos_fgets`, `unix_fgets`, `process` $\}$. The compile-time algorithm cannot distinguish between the first call to `go()` and the second, so the dynamic linker will try to link in the set same set $N$ at both migration-points. Since the functions `dos_fgets` and `unix_fgets` are not available on the same machine, the linker will fail on both occasions.

The *call-stack* algorithm of [1] can do better than strictly static analysis. It can distinguish between the first `go` and the second by inspecting the call-stack at runtime. Its runtime stitcher will create two different sets of names. For the first `go`, we have the name-sets: $N_1 = \{$ `U`, `D`,`B`, `dos_fgets`, `unix_fgets` $\}$ and $N_2 = \{$ `process` $\}$. Unlike the static algorithm, $N_2$ is a feasible link-set; unfortunately however, the first link-set is not, so there will be link-failure after the first `go`.

DDFA can tackle not only the return-fork at `B_exit` but also the conditional `if(site.os)` as well. The name-set created by the DDFA runtime stitcher for the first `go` call will be based on the value of `site.os`. If `site.os = dos`, then $N_1 = \{$ `D`, `B`, `dos_fgets` $\}$, otherwise $N_1 = \{$ `U`,`B`, `unix_fgets` $\}$. Both these link-sets are feasible for the linker, so the linker will not fail.

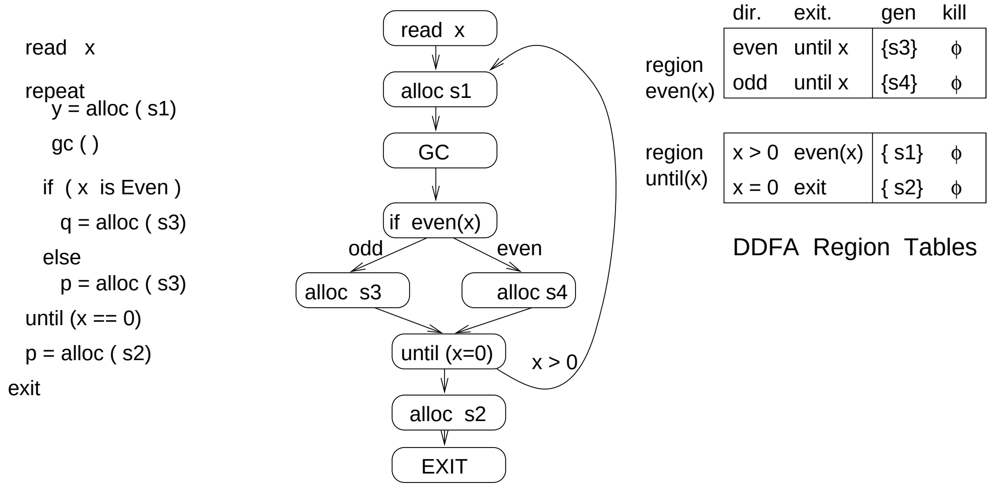



Figure 12: Garbage collection example. A strictly compile-time algorithm would estimate memory required after the `gc()` call to be $max(s_3, s_4) + max(s_1, s_2)$. DDFA can use runtime control flow information (i.e the value of `x`) to improve the estimate to $select(s_3, s_4) + select(s_1, s_2)$. The region-tables computed by the builder for the two *lp*-forks are shown alongside.

### 7.2 Explicit Garbage Collection

Garbage collection presents a great difficulty for programs that have real-time requirements. Garbage collection may be trigerred at any memory allocation point and take unpredictable amounts of time. To limit this variability, many garbage-collected languages allow the garbage-collector to be turned off unless it is explicitly invoked. Explicit garbage-collection is cumbersome because it leads to frequent over-cautious collections of small bits of garbage even when there is a lot of free space available. Compiler-support can help by providing the garbage-collector with an estimate of the amount of memory that will be requested before the next `gc()` call. The garbage-collector will perform garbage-collection only if the estimate is more than the amount of available memory. Otherwise, garbage-collection can be deferred until the next call to `gc` thus aggregating garbage-collection operations dynamically. However this is not an ideal solution. Different execution paths may have different memory requirements and static analysis has to make worst-case estimates. Further, the size of each memory allocation request may be a runtime variable. For example, consider the code fragment in Figure 12. Each iteration of the repeat-until loop makes an explicit call to the garbage-collector `gc()`. The compile-time estimate of memory required between any two calls to `gc()` is $E = max(e_1, e_2) + max(e_3, e_4)$, where $e_i$ = compile-time-estimate($s_i$).

DDFA can improve the memory requirement estimate in two ways. First, it uses runtime information to eliminate infeasible control-flow paths. In the example of Figure 12, runtime knowledge of the value of `x` can tell the stitcher exactly which allocation calls will be made, effectively replacing the $max$ operator used in the compile-time estimate with a $select$ operator. Second, the runtime stitcher may have precise values of $s_1, \ldots, s_4$, which can be used to improve the estimate of memory requirements.

# 8 Discussion

### 8.1 Forward DDFA

Our presentation of DDFA was limited to solving backward flow problems. This is because DDFA's 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* in the CFG. We could build similar regions and domains. A notable difference is that in forward frameworks the domain of influence of an operator is executed before the operator is reached. This implies that there are no non-predictable joins, for there is nothing to predict. The path has already been executed, so the points of entry and exit into regions can be logged by 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 tell the runtime stitcher which transfer functions to compose.

### 8.2 Predictability of forks

We have described the DDFA predictability analysis in only very broad terms. What is predictable and what is not it depends on how much computation one is willing to do to check for a condition's value. In the extreme case, every conditional could be made predictable by slicing out all the code required to determine the condition-variable and pushing it inside the runtime stitcher. Obviously, how much computation should be performed just for the sake of prediction depends on the particular dataflow problem being solved. Our approach uses a very conservative form of predictability. In our approach there should be no definitions to the condition-variable along a path from the op to the condition — if there is such a definition, we allow simple renames of one variable to another or assignment to a constant. In all other circumstances, the condition variable is considered unpredictable. Return forks tend to easily predictable because return destinations are stored from the call-stack and are not usually manipulable (i.e. no setjmp/longjmp). We have also taken a very simplistic approach towards exploiting correlations between fork-directions. It may not be necessary to check every *lp*-fork's direction on reaching an op-domain because one fork's direction may imply another fork's direction. Similarly, a fork that is designated as non-predictable may actually be predictable depending on some other fork-directions.

## 9 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. [9] allow programmers to construct and manipulate templates explicitly. Others, [6, 11, 16], generate templates automatically. These efforts are concerned with generating optimized code based on runtime values, so they focus on lower-level optimizations like load-elimination, loop-unrolling, static branch-elimination etc. Like these techniques, DDFA is a two-phase approach, with a compile-time phase that "sets things up" for 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 heavy-weight operation.

Our use of summary functions have been adapted from previous research on interprocedural analysis, in particular Sharir et. al. [21] and Duesterwald et. al. [8]. An important distinction is that we apply these summary functions at runtime. Another important distinction is that we use summary functions for regions instead of procedures; this introduces additional complications as regions overlap while procedures do not.

Standard algorithms for dataflow analysis assume that every sequence of edges that leads from a node to an exit-node constitutes a feasible execution path. All these paths are included in the meet-over-all-paths solution. However, in practice, not every control-flow path is actually feasible. Edges are often correlated to each other — for example, Mueller et. al. [17], has 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 [12], 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. It should be possible to combine both techniques — we have not yet investigated this angle.

There has been considerable research recently in branch prediction [15, 22] and profiling [3]. These techniques generally give probabilistic information about control-flow directions. DDFA could use such information to guide its caching policies.

The technique of deferring collection of dataflow information until runtime was used in a limited form (only procedure return forks) by Acharya et. al. in [1] to handle the problem of dynamic linking in mobile programs. DDFA is more general as it handles all types of forks.

# 10 Conclusions

The DDFA framework uses runtime control-flow information to sharpen compile-time dataflow analyses. We have shown how this framework can be used for optimizing various heavy-weight operations, including bulk-prefetching and dynamic linking of mobile programs. Runtime overheads are kept low by performing most of the analysis at compile-time; at runtime, knowledge of future control-flow can be used to simply stitch compile-time results together -time to get sharper dataflow information. Overheads are further reduced by caching dataflow results and reusing them. We have outlined how the basic DDFA framework can be extended to include interprocedural analysis, separately compiled code, higher-order functions and for analysis across multiple heavy-weight operations. In this report, we have limited our focus to backward dataflow problems. In section 8, we outlined a very simple variant of DDFA for forward problems, which logs the execution-path and uses this log to compute dataflow information for the traversed path.

In describing the DDFA framework we have assumed placement of heavy-weight operations is fixed. In some scenarios, the placement may not be so sacrosanct. In such cases, performance could be improved if one were allowed to move these operations in conjunction with deferred dataflow analysis.

# 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] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–147, 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] 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.

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

[9] 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.

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

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

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

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] J. Reif and R.E. Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, February 1982.

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

[21] 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.

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

[23] F.K. Zadeck. Incremental dataflow analysis in a structured program editor. In *SIGPLAN Symposium on Compiler Construction*, 1984.