Deferred Data-Flow Analysis

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

shamik@cs.umd.edu, acha@cs.ucsb.edu, saltz@cs.umd.edu
Contact Phone : 301-405-2756

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 improve precision by performing most of the analysis at compile-time and using additional information at runtime to *stitch* together information collected at compile-time. 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. Dividing the analysis in this manner raises two concerns: (1) is it possible to generate correct and compact summary functions for *regions*? (2) is it possible to correctly and efficiently compose and apply these functions at *run-time*? To address these concerns, we show that DDFA terminates, is safe and provides good results.

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 [15]. 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) (α, β or $\{\alpha, \beta\}$) could be prefetched.

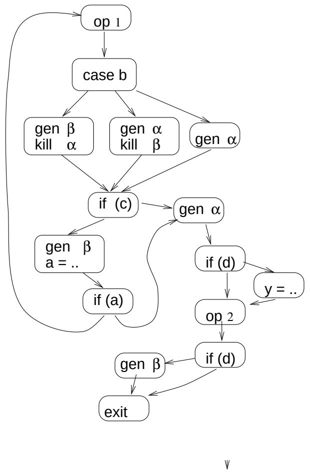
Loss of precision due to the conservative nature of compile-time dataflow analysis is a general problem and impacts a wide variety of optimizations. This is particularly important for optimization of *heavy-weight* operations such as bulk-prefetch, garbage-collection, runtime compilation or remote procedure calls. These operations are expensive, avoiding or combining even a small number has the potential of providing significant benefit. We propose a limited form of runtime dataflow analysis, called deferred dataflow analysis (DDFA), which attempts to improve precision by performing most of the analysis at compile-time and using additional information at runtime to *stitch* together information collected at compile-time. It divides the task of backward flow analysis into four stages. The first two stages are done at compile-time and the last two stages are done at run-time.

The first stage analyzes the control-flow graph to identify forks[1] whose direction can be determined at specific points in program execution. It uses this information to divide the control-flow graph into *regions* such that each region contains at most one such fork. The second stage analyzes these *regions* and constructs a *summary transfer function* for each *region* describing how it affects dataflow attributes flowing through it. The third stage is performed

[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.

```
a = b = c = d = α= β = 3;
repeat {
    prefetch ( data )
    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)
if ( d)  y = 5;
prefetch (data)
b = α;
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. (b) Control-flow graph for the program fragment in (a). We will use this fragment as a running example.

on-demand at particular points in the program execution. It uses available information about future control-flow to determine the set of reachable *regions* and completes the analysis for this set of *regions* using the corresponding summary transfer functions. The final stage uses the information computed by this analysis to make optimization decisions.

The key idea of DDFA is to divide the analysis into a compile-time phase and a runtime phase and to use runtime control-flow information to improve precision of analysis. Two-phase techniques have previously been used for *inter-procedural* flow analysis by others (e.g. Sharir et al [17] 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 the inter-procedural dataflow properties. DDFA differs from these techniques in three important ways. First, DDFA computes summary functions for *regions*, which are not necessarily procedures - this introduces an additional complication as regions can overlap and procedures cannot. Second, DDFA applies these functions at runtime. It is, therefore, very important to have compact representations for these functions. Third, DDFA computes multiple summary functions for each region and uses control-flow information available at runtime to choose between these summary functions.

In this paper, 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. Dividing the analysis in this manner raises two concerns: (1) is it possible to generate correct and compact summary functions for *regions*? (2) is it possible to correctly and efficiently compose and apply these functions at *run-time*? To address these concerns, we prove that DDFA terminates, is safe and its results are as good as a compile-time meets-over-all-paths solutions (for finite lattices with distributive transfer functions).

2 Overview of DDFA

In this section, we present a brief overview of DDFA including terms and concepts which are needed to follow the rest of the paper. For conciseness, we limit ourselves to intuitive descriptions of the terms; for details we refer the reader to [18].

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*.

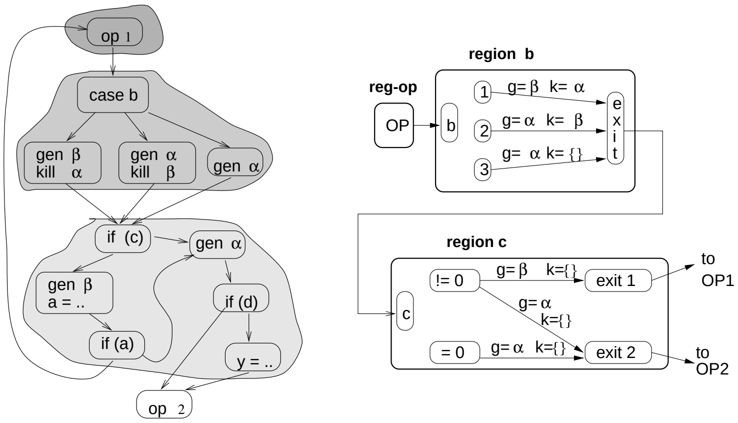


Figure 2: The three shaded regions show the *lp*-regions of this op-domain. The figure on the right shows the summary functions that would be computed for the region induced by fork `if(c)` .

The op-domain corresponding to the operation O consists of the nodes that are reachable from O without passing through another *op*. O is referred to as the *entry-point* to the op-domain; the exits from an op-domain are either other op-nodes or program exit-points.

Each op-domain has zero or more *forks*. Forks are program-points from which control can flow in more 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 lattice element. 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 *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* if there exists 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. Since *lp*-forks are forks at which a conservative analysis loses information *and* their 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. In this abstract, we do not describe this stage and assume that the op-domains and *lp*-regions have already been identified; we refer the reader to [18] for the details. The intuitive definitions presented above for op-domains and *lp*-regions indicate how they can be constructed. 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, $\phi(direction)$, one function for each direction the *lp*-fork at the entry to the region can take. 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 ap-

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

propriate 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 The Basic framework

In this section, we describe the basic DDFA framework. First, we present terms and definitions. Next, we describe the compile-time algorithm for the construction of summary transfer functions. Next, we present the stitching algorithm that uses runtime information to select and apply these functions. We then present the theoretical underpinnings for these algorithms including proofs that the summary functions generated by our algorithm are correct and that the stitcher is able to compose and apply these functions in a correct and efficient manner.

Here, we limit ourselves to intra-procedural analysis of a first-order language in which the only forks are conditionals that test a single variable. In addition, we make the assumption that the locations of the heavy-weight operations that are the target of optimization are fixed. These restrictions are for expositional simplicity. In section 4, we extend the framework to include interprocedural analysis, separately compiled code and higher-order functions. For conciseness, we defer the treatment of heavy-weight operations that can be repositioned to [18].

3.1 Terms and definitions

We characterize a global backward dataflow framework by the pair (L, F), where L is a lattice of attribute information and F is a semi-lattice of monotone and distributive transfer-functions from L to L. L is assumed to be a complete lattice with a partial order ($\leq$), top ($\top$) and bottom ($\perp$) elements and a meet operator ($\sqcap$). Further, it is assumed that L satisfies the finite descending chain property. F is assumed to be ordered ordered as follows : for $g_1, g_2 \in F$, $g_1 \geq g_2$ iff $g_1(x) \geq g_2(x)$ for all $x \in L$. We need to make these assumptions regarding F to construct the region's summary transfer function for a region, because flow analysis is carried out on transfer-functions rather than data attributes. In order to do this, we need to ensure that F is closed under functional composition ($\circ$) and functional meet ($\wedge$).[3] More specifically, we assume that F is the smallest set of functions acting on L which is closed under functional composition ($\circ$) and functional meet ($\wedge$). It contains an identity function (f_{id}) and a constant function ($f_\top$) ($f_\top(x) = \top$ for all x).

Each op-domain is represented as a separate control-flow graph $G = (N, E)$. G has a single root, op, which is the op-node that induces the op-domain and a set of exit nodes, $exits(G)$. It is possible that $op \in exits(G)$ due to a back-edge (for example, see Figure 2). With each edge $(n, m) \in E$, we associate a transfer-function $f_{(n,m)} \in F$ which represents the change of data attributes from the entry of block m, up through node n to the entry of block n (backward-propagation). The sets $pred(r)$ and $succ(r)$ denote the immediate predecessors and successors of node n respectively.

The *lp*-regions within each op-domain are identified by the *lp*-fork v that induces the region. (We use the identifier v to refer to the *lp*-fork as well as the corresponding *lp*-region). Each *lp*-region has its own subgraph $G_v = (N_v, E_v)$. The children of the *lp*-fork v are called *predictable children* and are belong to a set $W_v = \{m : (v, m) \in E_v\}$. We also define E_v^* to be a subset of E_v which contains all edges except those between the *lp*-fork v and its predictable children (W_v) (i.e. $E_f^* = E_f - \{(v, m) : (v, m) \in E_f\}$). The exits of the *lp*-region belong to the set $exits(G_v)$ (each exit is another *lp*-fork node or a node $n \in exits(G)$).

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

[3] As noted in [17], F may not be a bounded lattice, even if L is bounded. However, if L is finite, then F must also be finite, and therefore bounded. We limit ourselves to finite lattices.

3.2 Constructing summary functions (the *builder*)

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

We construct a summary function, $\phi(c, e) \in F$ for every predictable child $c \in W_v$ and exit node $e \in exits(G_v)$. This function summarizes the dataflow operations on all paths between c and e with the constraint that all these paths lie entirely within the *lp*-region. We do not include paths between n and e that leave the *lp*-region and return. The propagation functions for an *lp*-region can be constructed as follows :

$$\begin{aligned} \phi_v(e, e) &= f_{id} \quad e \in exits(G_v) \\ \phi_v(e, n) &= \bigwedge_{(n,m) \in E_v^*} (f_{(m,n)} \circ \phi_{(e_i,m)}) \end{aligned} \tag{1}$$

This set of non-linear equations operates over the function-lattice F, and not L. Since F is assumed to be closed under composition ($\circ$) and meet ($\wedge$), we can solve the system of equations by starting with : $\phi^0_{(e,n)} = f_\top$ for each $n \in N_f - \{e\}$ and iterating to obtain new approximations to the ϕ's until convergence.

Since the iterative process does not include the edges from the *lp*-fork v to its predictable children w_i, we still do not have a summary function for the fork node itself (i.e. we have $\phi_v(w, e)$, for all $w_i \in W_v$ but we do not have $\phi_v(v, e)$). We could get $\phi_v(v, e) = \bigwedge \phi_v(w, e)$ over all predictable children $w \in W_v$. However, we do not perform this meet because the *lp*-fork is lossy. Delaying this meet until runtime will allow us to predict the edge through which control will flow (say (v, w_r)). At runtime, we can simply *assign* $\phi_v(v, e) = \phi_v(w_r, e)$ and avoid performing a lossy meet. To achieve this, each *lp*-region stores $\phi_{(w,e)}$ for each $w \in W_v$ and each exit node $e \in exits(G_v)$ that is reachable from w. The summary function (ϕ_v) for the region is thus a set of summary functions $\phi_v = \{\phi_v(w, e) : w \in W_v e \in exits(G_v)\}$. In the worst case, each region's summary function can have $|exits(G_v)| \times |W_v|$ entries. However, we really only need to store entries for those (w, e) pairs which are connected in G_v (i.e. $(w, e) \in paths_v(w, e)$).

For the above procedure to work efficiently, it must possible to compactly represent the transfer functions and it must be possible to perform the composition and meet operators on the function-lattice. In general, these conditions need not hold. Fortunately, however, these conditions do hold for most of the typical dataflow analysis frameworks. As pointed out in [17], the class of dataflow problems that allow compact and simple representations are the same class that allow elimination algorithms [2] (these algorithms also need to perform functional meets and compositions). This family includes the classical "bit-vector" dataflow problems. In particular, there exists a common class of dataflow problems, called *1-related* problems, that always have compact representations and allow efficient manipulation [16]. For 1-related problems it is easy to perform functional composition and functional meets; we show how this can be achieved using `gen` and `kill` sets.

Each basic block i in G 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.[4]

For the DDFA framework to work, we must show how to perform the functional-meet and functional-composition of these transfer functions. We also need to show that there exists a set F which (i) contains at least G's transfer functions, (ii) is closed under the composition and meet operators that we define and (iii) is finite and therefore, of bounded depth.

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 $k = (k_1 \cup k_2)$

If the set of attributes A analyzed in the dataflow framework is finite then the power set A^* is also finite. This limits

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

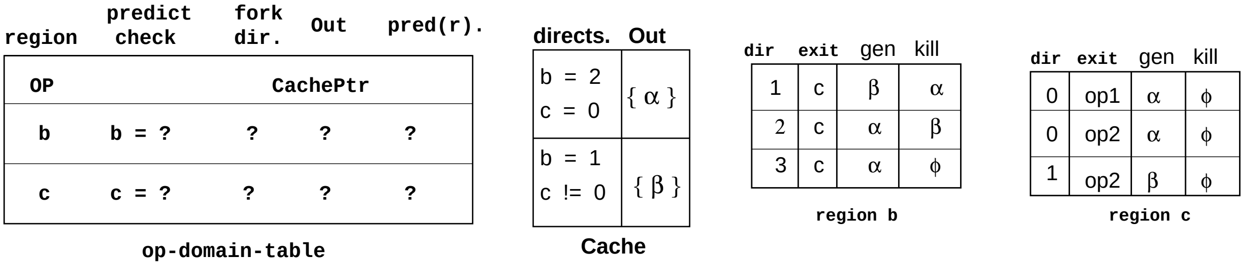


Figure 3: Data-structures created by the builder and passed to the runtime stitcher for the op-domain in the running example. The fields marked by **?** are to be filled in by the runtime stitcher. The cache has been pre-filled by the builder for two of the paths.

the number of distinct g and k sets ensuring that F is finite, of bounded depth and also closed under composition and meet. It is easy to show that every function in F is distributive (follows from the fact that $f_i(x) = (x - k_i) \cup g_i$).

3.2.1 Data structures

The *builder* passes the results of its analysis to the *stitcher* via pre-initialized data-structures in the data segment of the compiled code.

For each op-domain in the CFG, we keep a data-structure called an op-domain table. The op-domain table contains one entry record for every region within the op-domain. 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 for determining the direction of each region's fork, (3) scratch space for intermediate dataflow attributes during stitching, and (4) a place-holder for the predecessor set of each region. The predecessor set cannot be computed accurately at compile-time because any given set of fork-directions may render several paths infeasible, changing the predecessor relationships. Each *region* table is indexed by (fork-direction, exit) pair and contains all summary functions for the region. Since summary functions exist only for a few (fork-direction,exit) pairs, region tables are stored in a sparse format.

Figure 3 illustrates these data structures. It shows the data structures generated for the running example presented in figure 1. The region associated with the op-node is trivial and needs no summary functions. The region associated with `switch(b)` has three fork-directions all leading to same exit. The summary functions corresponding to the three (fork-direction, exit) pairs in this region need only trivial functional compositions and no functional meet (no other forks in the region). Functional meet is needed for the `if(c)` region: eg., at `if(a)` ($\phi_{left} \wedge \phi_{right} = (g_{left} \cap g_{right}, k_{left} \cup g_{right}) = (\{\} \cap \{\beta\}, \{\} \cup \{\}) = (\{\beta\}, \{\})$).

The builder also generates a cache structure for holding previously computed dataflow properties. This cache is associatively indexed on the settings of all the fork-directions in the region (the directions are hashed together to yield a single unique key). The compiler primes the cache by statically computing the dataflow properties for the "likeliest" choices for all fork-directions (statically guessed).

3.3 Applying summary functions (the *stitcher*)

In this section, we describe how the *stitcher* selects summary functions from the sequence of *lp*-regions that lie along feasible paths given a set of fork-directions and how it applies the corresponding summary functions. We first describe the operation of the stitcher in terms of the data-flow equations it solves. Then we describe the stitcher itself and illustrate its operation with an example.

The stitcher is invoked every time execution reaches an op-node. It first applies the tests corresponding to all *lp*-forks in the op-domain. At this point, it has a set of fork-directions, $D = \{d_1, \ldots d_k\}$, where d_i is the predicted (and guaranteed) direction for the *lp*-fork v_i. Our definition of predictability implies that these fork-directions hold as long as control remains in the op-domain; they may change direction the next time control reaches the op-node.

Note that each *lp*-region is like a big basic block with the *lp*-region's summary function being analogous to a block's transfer function. However, this analogy is incomplete. Basic blocks are single-entry single-exit units whereas an *lp*-region has a single entry but multiple exits. However, each of the predictable children of the fork can be viewed as a potentially different entry point (only one entry is feasible at any time). This makes each *lp*-region a multiple-entry, multiple-exit hyper-block. The builder has already prepared for this complexity by keeping separate summary functions for each (entry, exit) pair. In addition, the builder has also prepared parameterized successor sets: $succ_v$ is the set of all *lp*-regions that are successors of v whereas $succ_v(d)$ is the set of *lp*-regions that can be reached if the fork-direction is known to be d. Once the fork-directions are known, the stitcher uses these parameterized successor sets to determine the predecessor sets for this invocation of the stitcher. These sets are parameterized — $pred_v$ is the set of actual predecessors of v whereas $pred_v(p)$ is the set of exit nodes of p at which control will flow from p to v.

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

$$\begin{aligned} OUT(e) &= \quad \bot && \forall e \in exits(G) \\ OUT(r) &= \prod_{(s,m)\in T} \phi_r(d,m)(OUT(s)) && \\ & && \text{where } T = \{(s,m) : s \in succ_s(d_s), m \in pred_s(r)\} \end{aligned} \tag{2}$$

Here, $OUT(r)$ describes the dataflow properties at the node atop *lp*-region r. Each *lp*-region r takes dataflow attributes ($OUT(s)$) from a successor *lp*-regions s and determines which of its exit-nodes the successor s connects to (there may be more than one connections from r to s). For each such exit-node m , we look up the *lp*-region r's summary function and find the appropriate transfer function to apply $\phi_r(d,m)$. Applying this transfer function provides the resultant dataflow attributes at the top of *lp*-region r due to the paths from fork r to fork s through the exit-node m. This process is repeated for other exit-nodes between r and s, and then continued for other successor *lp*-regions of r. The meet of all these dataflow attributes gives us the dataflow attributes at fork r. The process is iteratively continued until convergence.

3.3.1 Stitcher algorithm and example

Figure 4 presents the stitcher algorithm. The stitcher consists of two part. The first part handles the determination of fork-directions (using the code pointers in the op-domain table) and caching of previous results. The second part is a simple worklist algorithm (over regions instead of basic blocks). The primary difference from other worklist algorithms is that it recomputes the predecessor sets on each invocation. This is done since regions can have multiple exits, many of which may not be reachable for this set of fork-directions.

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

3.4 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 D — when the number of fork-directions gets exceeds D, 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 R). Thus the number of summary functions to be stored in a *lp*-region is bounded by $D \times R$. Each op-domain can have R *lp*-regions so the total number of summary functions stored within an op-domain can be as large as $D \times R \times R$. 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 R is the number of predictable fork points within an op-domain, which we expect to be

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

```
function stitcher(domain *opd)

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

  flow ← CheckCache(opd,direc)

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

   Set_Prefetch_List (flow)
end
```

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

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

Figure 4: The stitching algorithm.

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 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 handles the case where there is one path that is more frequently taken than others (A similar technique, called *record-replay* [15] has been shown to work well for optimizing communication in irregular parallel programs). Even when the algorithm has to be executed, it operates on the meta-CFG which has *lp*-regions instead of basic blocks, and summary functions instead of transfer functions. We expect the size of the meta-CFG for individual op-domains to be small. In the worst case, we can limit the time spent in the stitcher. If this time is exceeded without reaching a fixpoint, the stitcher is stopped and we can fall back on the results of compile-time analysis.

3.5 Theoretical underpinnings

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. We only provide an intuitive outline of our arguments in this abstract and refer the reader to [18] for more details. Our proofs are largely adapted from the proofs of the two-phase interprocedural analysis used in [17].

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

Path Decomposition Lemma: Let P be a path from a node p_0 in the op-domain to an exit node $e \in exits(G)$. The lemma states that each such path satisfies three properties.[5]

1. **Decomposability** : P can be represented as $P = p_0..., e_0 \| f_1...e_1 \| f_2...e_2 \| ... \| f_j...e$ where $f_1, f_2, ..., f_j$ are the *lp*-forks that are encountered along the path and each e_i is an exit node for the *lp*-region induced by the fork f_i (i.e. $e_i \in G_i$) and the cross-region edges are valid (i.e $(e_i, f_{i+1}) \in E$).

2. **Existence :** The converse is also true — any sequence of segments from different regions connected by valid cross-region edges constitutes a valid path (i.e. it is in $paths(p_0, e)$).

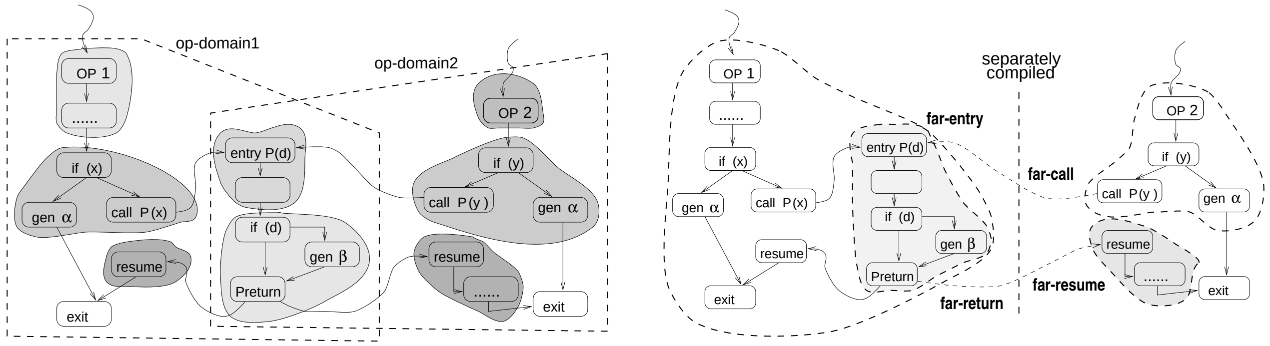
3. **Uniqueness :** Each segment of the decomposed path (except the first segment $(p_0, ..., e_0)$ belongs to one and only one region. This is simply because each segment, except the first, begins with a *lp*-fork node (f_i) and each such fork-node can belong only to its own region. We can remove the qualification for the first segment when p_0 is also a *lp*-fork or the op-node.

We are now ready to prove the other two results.

DDFA is safe: For any feasible execution path P, from *op* to an exit node $e \in exits(G_v)$, 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 is $\{ op..., e_0 \| f_1...e_1 \| f_2...e_2 \| ... \| f_j...exit \}$ then $FLOW(P) = f_{op} \circ ... \circ f_1 \circ ...f_j ... \circ f_{exit}(\perp)$. We need to show that the $OUT(op)$ computed by DDFA will be a conservative approximation of $FLOW(P)$ (i.e. $OUT(op) \leq FLOW(P)$ for all $P \in paths(op, e)$). The proof rests on the the fact that every path is decomposable. It is easy to show that for every segment (f_i, e_i) enumerated in the decomposition of path P, the *builder* will compute a summary transfer function $(\phi_i(f_i, e_i))$ which will be safe approximation for the composition of transfer functions of nodes that lie in the i-th segment of path P (inductive proof on path-lengths based on the number of nodes, and using the safety of functional-meets and functional composition). Having shown that the builder will create safe summary functions corresponding to each segment of P's decomposition, we must now show that the stitcher will combine these summary functions safely. This requires showing two properties. First, for every cross-segment edge (e_i, f_{i+1}) in the decomposition of P, we need to show that there is a corresponding cross-region edge included in the stitcher stage (i.e. region $r_i \in pred_{r_{i+1}}$ and $e_i \in pred_{r_{i+1}}(r_i)$))). This essentially follows from our definition of $pred$. As path P has a segment $(f_i, ...e_i)$ that lies completely within region r_i, it follows that e_i will be marked as a reachable exit from f_i. This will mark region r_{i+1} as a reachable successor of region r_i via exit e_i. Since $pred$ is obtained by transposing the $succ$ set, it follows that the stitcher will properly account for all cross-segment edges in P. Second, we need to show that the result obtained by iteratively applying the region summary functions until convergence safely approximates $FLOW(P)$. This requires another proof by induction over path-lengths (the induction is over the number of segments, instead of number of nodes).

DDFA is at least as good as the compile-time meet-over-all-paths solution: The idea here is to show that if we did not use any runtime information (knowledge of control-flow decisions), the results of DDFA's two-phase analysis

[5]There are subtle differences between our path decomposition lemma and a similar lemma used in [17]. 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 a node in our region-segments can belong to multiple segments. (except for the *lp*-fork nodes and the op-node).



(a) (b)

Figure 5: (a) 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 region data-structures. (b) 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 `far-call`.

would still be just as good as a meet-over-all-paths (MOP) analysis. Consider a crippled version of DDFA (called pDDFA, for poor-man's DDFA) that is identical to DDFA except that the stitcher does a poorer job of selecting the summary functions to apply. Unlike DDFA which selects functions based on fork-direction as well as the exit node, pDDFA selects based only on the exit node. This activates a larger number of summary functions all of which are applied to the incoming flow attributes and the result combined with a meet ($\sqcap$). We show that pDDFA can perform as well as a MOP analysis. The proof proceeds in three steps, of which only Step 1 is new. **Step 1 :** We show that pDDFA is at least as good as a approach that we call BOA (builder-only approach). The BOA approach considers the entire op-domain as a single region and computes a summary function (multiple functions if there are multiple exits) for the entire op-domain. The stitcher for BOA is trivial - it merely applies each of these functions to $\perp$ and performs a meet on the results. The proof uses the existence property of path-decomposition lemma. Any sequence of segments used by pDDFA's stitcher is a valid path and must be used by BOA's builder to compute the global summary function. We can show by induction on path-length that applying the big summary-function computed by BOA will be no better than pDDFA's approach of applying each region's transfer function. **Step 2 :** Note than BOA is computing the big 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. 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 [11]. **Step 3 :** This step shows that operating on the functional domain does not hurt the quality of the solution. In other words the FMOP solution is as good as the MOP solution. For this we just cite previous work on summary functions, in particular Theorem 7-3.4 in [17].

4 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.

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 [10] and suggest a similar solution. For each op-domain that extend 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.[6] Figure 5(a) illustrates the interprocedural representation and sharing of regions. In the figure, the regions of procedure P are shared by both op-domains.

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. We make an conservative approximation which loses precision across the call-sites but preserves it on each side of the boundary.

We refer to calls to separately compiled code and to unknown functions as *far-calls*. The corresponding call- and resume-nodes are marked to reflect this. Similarly, we mark the entry- and return-nodes of *escaping* functions (functions that can be called from unknown sites). 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 op-node at its entry-node. We also place an op-node at the 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. See Figure 5(b) for an example of separately-compiled code. The shaded regions show the new op-domains that would be created to handle the *far-call*.

5 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. [8] allow programmers to construct and manipulate templates explicitly. Others, [5, 9, 13], 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. Similarly, DDFA generates information at compile-time for the use of a runtime-stitcher. However, as our goal is to optimize heavy-weight operations we focus on improving the precision of the *dataflow* analysis rather than generating optimized code. DDFA does not modify code, except to insert a call to the runtime stitcher at each operation.

Our use of summary functions have been adapted from previous research on interprocedural analysis, in particular Sharir et. al. [17] and Duesterwald et. al. [6]. An important distinction is that we apply these summary function at runtime. Another important distinction is that we use summary functions for regions instead of procedures this introduces an additional complications as regions can overlap and procedures cannot.

DDFA needs to know about future control flow decisions in order to be effective. There has been considerable research recently in branch prediction [12, 19], profiling [3] and elimination of conditional branches. [14] has

[6] 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.

reported the existence of significant amounts of correlation among conditional branches. Bodik et. al. [4] present techniques to statically detect these correlations. Their goal is to eliminate conditionals; however the knowledge of these correlations could also be used to predict future control-flow decisions.

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

References

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

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

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

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

[5] C. Consel and F. Noel. A general approach to run-time specialization an its application to c. In *POPL'96*, pages 145–56, 1996.

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

[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] 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.

[9] B. Grant, M. Mock, M. Phillpose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in c. In *PEPM'97*, 1997.

[10] L. Hornof and J. Noye. Accurate binding-time analysis for imperative languages: Flow, context and return sensitivity. In *PEPM'97*, 1997.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] Barry Rosen. Monoids for rapid data-flow analysis. In *POPL*, 1978.

[17] M. Sharir and A. Pnueli. *Two approaches for interprocedural data flow analysis*, chapter Program Flow Analysis : theory and applications. Prentice-Hall, Edited : S. Muchnik and N.D. Jones, 1981.

[18] Shamik Sharma, A. Acharya, and J. Saltz. Deferred data-flow analysis : Algorithms, proofs and applications. Technical report, University of Maryland, 1997.

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