Iterative Data-flow Analysis, Revisited

Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy

Department of Computer Science
Rice University
Houston, Texas, USA

ABSTRACT

The iterative algorithm is widely used to solve instances of data-flow analysis problems. The algorithm is attractive because it is easy to implement and robust in its behavior. The theory behind the algorithm shows that, for a broad class of problems, it terminates and produces correct results. The theory also establishes a set of conditions where the algorithm runs in at most $\mathsf{d}(G) + 3$ passes over the graph — a round-robin algorithm, running a "rapid" framework, on a reducible graph [25]. Fortunately, these restrictions encompass many practical analyses used in code optimization.

In practice, compilers encounter situations that lie outside this carefully described region. Compilers encounter irreducible graphs — probably more often than the early studies suggest. They use variations of the algorithm other than the round-robin form. They run on problems that are not rapid.

This paper explores both the theory and practice of iterative data-flow analysis. It explains the role of reducibility in the classic Kam-Ullman time bound. It presents experimental data to show that different versions of the iterative algorithm have distinctly different behavior. It gives practical advice that can improve the performance of iterative solvers on both reducible and irreducible graphs.

1. INTRODUCTION

Iterative analysis is one of the most widely used techniques for data-flow analysis. The algorithm, itself, has a simple structure that makes it easy to understand and to implement. In practice, it solves a wide range of problems efficiently. For an even larger set of problems, it solves them less efficiently, but still correctly. Its behavior is supported and explained by a well-developed theory [29, 22, 25, 26, 27]. That theory ensures correctness and termination, even on "pathological" graphs and hard problems. It shows that the algorithm's running time depends on both the structure of the problem and the structure of the underlying graph. It identifies a restricted category of problems, the *rapid* frameworks, where the algorithm is guaranteed to run quickly.

Modern practice applies the iterative algorithm in contexts where some of these restrictions may not apply. Compilers routinely use the iterative algorithm to solve problems that are not rapid [33, 7, 13, 17, 24, 31]. These problems operate over graphs that may not resemble the single-procedure control-flow graphs (CFGs) studied in the classic work on control-flow analysis [2, 30, 28]. Two areas, in particular, where such problems arise are interprocedural analysis and binary-level optimization.

In other contexts, such as just-in-time compilers, the speed of compilation is critical. Time constraints limit the amount of analysis that these compilers can perform. The techniques in this paper can reduce the cost of iterative data-flow analysis by 25 to 40%. In speed-sensitive compilers, such savings are important.

A generation of practice with the iterative algorithm shows that it works. It runs quickly. It produces correct answers across a broad range of problems [26]. However, the version of the algorithm described in the classic literature — the round-robin iterative algorithm — may not be best, in practice. This paper examines the theory and practice of iterative data-flow analysis. It explains the relationship between reducibility and the classic time bound. It examines the behavior of iterative solvers on irreducible graphs and non-rapid problems. It presents practical ways of improving both the worst case and average case behavior.

2. BACKGROUND

The iterative algorithm is one of the oldest and most heavily studied algorithms for performing data-flow analysis [29, 25, 27, 26]. It is well known that, in practice, the algorithm works on all classes of graphs.

2.1 Time Bounds

The best known time bound for iterative data-flow analysis comes from a 1976 paper by Kam and Ullman [25]. They showed a class of problems that a *round-robin iterative algorithm*, visiting nodes in *reverse postorder*, can solve in $\mathsf{d}(G) + 3$ passes over a graph G. They present a lattice-

Submitted to PLDI 2003 November, 2002

theory formulation that characterizes the problems that lie inside their class, called the *rapid* problems.

The key term in the time bound is $\mathsf{d}(G)$, the *depth* or *loop-connectedness* of G [23, 25]. Loop connectedness is a structural property of a directed graph G and a depth-first spanning tree (DFST) of G. The DFST partitions the edges in G into three classes: edges used in the DFST, called tree edges or forward edges; edges that run from a node back to an ancestor in the DFST, called back edges; and edges that run between nodes in disjoint subtrees of the DFST, called cross edges. With this partitioning, $\mathsf{d}(G)$ is the maximum number of back edges that can occur on any acyclic path through G.

This bound reassures implementors. Since $\mathsf{d}(G)$ is almost always small, it suggests that the algorithm always makes only a small number of passes over the graph. However, it fails to address some of the important issues that arise in practice. For example:

1. Is round-robin reverse-postorder iteration, in general, the fastest form of the iterative algorithm?
2. Does the choice of DFST really matter?
3. How can the compiler pick good iteration orders?
4. Do these orders help on non-rapid problems?

In this paper, we address each of these issues.

2.2 Reducibility

Graph reducibility plays a key role in the theory of data-flow analysis. A graph is either reducible or not. Several well-known algorithms cannot handle irreducible graphs [1, 9, 21, 28]; later work extended interval analysis to work over irreducible graphs.

Graph reducibility is defined by a pair of transformations, called T_1 and T_2, shown below (Figure 1).

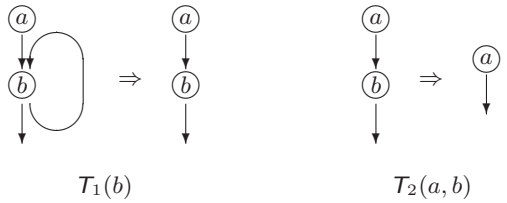


Figure 1: Transformations T_1 and T_2

T_1 removes a self-loop, while T_2 folds a node that has a single predecessor into that predecessor.

Applying T_1 and T_2 to applicable subgraphs, in any order, will either reduce the graph to a single node, or it will block. In the latter case, the graph is irreducible. Repeated application of T_1 and T_2 partitions the graph into two kinds of regions. Any region that reduces to a single node is reducible. Any multi-node region that cannot be further reduced is an irreducible kernel.

Figure 2 shows two irreducible graphs. The graph on the left shows the smallest irreducible region. In general, irreducible kernels arise from cyclic regions in the graph that have multiple entry-points.

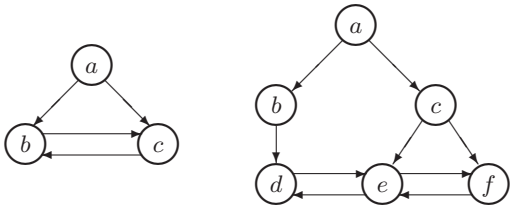


Figure 2: Irreducible graphs

Early studies of control-flow graphs in Fortran found that almost all of them are reducible [30, 2, 28]. This left the impression that data-flow analyzers could, essentially, ignore efficiency on irreducible graphs. However, modern compilers apply iterative analysis in many domains where irreducible graphs can and do occur.

- Interprocedural call graphs have different properties than CFGs; in fact, they are usually multigraphs. We know of no study that measures the reducibility of call graphs, for either imperative programs or object-oriented programs.
- Control-flow features of modern languages may create a higher incidence of irreducible graphs. For example, the `break` statement in C, C++, and Java branches out of an inner loop. Such a loop has two entries in its reverse CFG, so the reverse CFG is irreducible. Backward data-flow problems (*e.g.*, LIVE) use the reverse CFG.
- Iterative analysis is applied to graphical versions of SSA form [8]. Irreducible CFGs can beget irreducible SSA graphs. Backward problems solved on the SSA graph can also encounter irreducible regions.
- Binary-level and link-time optimizers operate on compiled, optimized code. Aggressive optimization may create unusual control flow [12]. A binary-to-binary optimizer needs to reconstruct assembly-level labels before it can build accurate graphs of control-flow. (This problem is similar to call-graph construction in the presence of function variables [6].) These systems use iterative analysis to solve these problems because of its robust behavior on arbitrary graphs.

Together, these facts suggest that compilers may encounter irreducible graphs more often than the early studies suggest.

The iterative algorithm functions correctly on any graph, reducible or not. However, reducibility plays a role in the behavior of iterative solvers. The key term in the Kam-Ullman time bound, $\mathsf{d}(G)$, depends on reducibility in a subtle way. If G is reducible, then every DFST of G has the same value

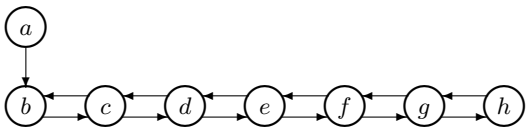
for $\mathsf{d}(G)$. If G is irreducible, $\mathsf{d}(G)$ can vary from one DFST to another. For these graphs, the order in which the DFST is constructed matters. Increasing $\mathsf{d}(G)$ has a practical impact on the running time of an iterative solver.

This makes iterative analysis more important, due to its robust behavior. It also increases the importance of understanding the issues that affect the performance of iterative analyzers on both reducible and irreducible graphs.

3. ITERATION ORDER MATTERS

As part of our investigation into a iterative solver for the dominance problem, we closely studied the performance of an iterative solver for Allen's dominance equations [1, 2] on large graphs — synthetic CFGs of 30,000 nodes. The graphs are constructed from a statistical model of CFG characteristics. Their sheer size exposes performance quirks of the iterative algorithm. The reverse graphs — considering the edges in reverse for a postdominance computation — are very often irreducible.

Studying the behavior of the iterative algorithm on these large irreducible graphs led us to some insights into its performance. For the round-robin iterative algorithm, the worst case behavior occurs when $\mathsf{d}(G)$ is large. For example, the following seven-node graph has a loop connectedness of six, based on a DFST of $a{\rightarrow}b{\rightarrow}c{\rightarrow}d{\rightarrow}e{\rightarrow}f{\rightarrow}g{\rightarrow}h$. We see that the path $h{\rightarrow}g{\rightarrow}f{\rightarrow}e{\rightarrow}d{\rightarrow}c{\rightarrow}b$ contains six back edges.



To understand the time bound on the round-robin iterative algorithm (shown in Figure 3 below), consider what happens when the nodes are visited in a reverse postorder ($\mathcal{RPO}$) of a, b, c, d, e, f, g, h.

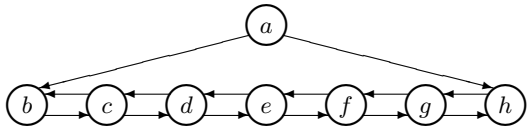
```
for i ← 1 to N
    initialize the sets at node i

while (sets are still changing)
    for i ← 1 to N
        recompute set at node RPO(i)
```

Figure 3: The round-robin iterative algorithm

Consider a fact that occurs in h. In the first iteration, it propagates to g. The next iteration takes it to f. In the sixth iteration, it reaches c. The seventh iteration recomputes the value at b to include our fact and tries to propagate it forward along the path from $b{\rightarrow}c{\rightarrow}\cdots{\rightarrow}h$. Since the fact is already established at each of those nodes, nothing changes and the iteration halts with a stable fixed point.

This is the worst-case propagation. To it, we must add one pass to initialize all the sets. This works out to $\mathsf{d}(G)+2$ passes. For some problems, one more pass is required.[1]

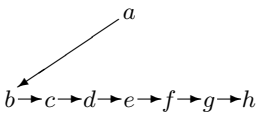
What happens on an irreducible graph? To make this graph irreducible, we can add an edge (a, h).



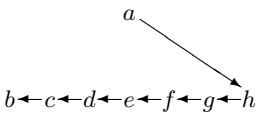
Adding this edge does not change the algorithm's behavior. The worst case propagation remains the same. The edge (a, h) does create the possibility for a second DFST, $a{\rightarrow}h{\rightarrow}g{\rightarrow}f{\rightarrow}e{\rightarrow}d{\rightarrow}c{\rightarrow}b$, with another reverse postorder for the iteration. An equivalent worst case scenario exists for this second DFST, starting with a fact in b.

Why then, does the time bound carefully restrict itself to reducible graphs? The issue is efficiency, not correctness. (The algorithm computes a fixed point on any graph; the uniqueness of that fixed point is a function of the data-flow framework, or problem, rather than of a specific graph, or problem instance.) On a reducible graph, $\mathsf{d}(G)$ does not depend on the specific DFST — that is, any reverse postorder produces the same $\mathsf{d}(G)$. Several papers have made this point.

However, an irreducible graph can produce multiple DFSTs, with different values for $\mathsf{d}(G)$. Our previous example does not have this property. It has two DFSTs, shown below.



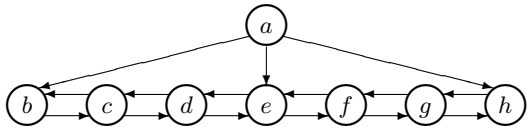
Leftmost DFST



Rightmost DFST

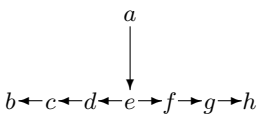
Both of these produce the same value for $\mathsf{d}(G)$, namely 6.

Adding the edge (a, e) to the example, however, lets us find a DFST with a different value for $\mathsf{d}(G)$.



This new edge creates one more DFST.

[1]Kam and Ullman specify that $\mathrm{D}(G)+3$ passes may be required if the data-flow framework does not have a $\top$ element. In practice, the classical problems in single-procedure data-flow analysis can all be formulated with a $\top$ element, to produce a bound of $\mathsf{d}(G)+2$.



Centermost DFST

This "centermost" DFST has a $\mathsf{d}(G)$ equal to three, where the leftmost and rightmost DFSTs have $\mathsf{d}(G)$ equal to six. The centermost DFST induces two new $\mathcal{RPO}$s: $\{a, e, d, c, b, f, g, h\}$ and $\{a, e, f, g, h, d, c, b\}$. With either of these orders, the round-robin algorithm should halt in $\mathsf{d}(G) + 3$, or 6, passes. Thus, the choice of DFST has a material affect on the running time of the algorithm. We believe that this example is the first published constructive proof that irreducible graphs can produce different values for $\mathsf{d}(G)$, depending on the choice of DFST.

Can the compiler choose the DFST that minimizes $\mathsf{d}(G)$? We know of no algorithm for finding this DFST, except for enumerating all the possible DFSTs and testing them. In any case, the iterative algorithm runs quickly in practice, so the compiler cannot afford to spend much time exploring alternative DFSTs. However, the example shows that computation order really does matter and suggests that we explore other iteration orders as well.

4. THE WORKLIST ALGORITHM

The classic time bound for the iterative algorithm applies to the round-robin iterative algorithm, which visits every node in a fixed order. The rigid structure of the algorithm makes it easy to analyze. It also provides an upper bound on the number of recomputations that would be required by any reasonably engineered scheme. In practice, however, other formulations of the iterative algorithm have better performance. We will focus on the worklist iterative algorithm (shown in Figure 4) because, as this paper will show, simple data-structure modifications can make it exhibit a variety of behaviors.

```
Worklist ← ∅
for i ← 1 to N
    initialize the sets at node i
    add i to the Worklist
while (Worklist ≠ ∅)
    remove a node i from the Worklist
    recompute set at node i
    if new set ≠ old set for i then
        add each successor of i to Worklist, uniquely
```

Figure 4: A worklist iterative algorithm

The worklist iterative algorithm improves on the round-robin algorithm by focusing the iteration on regions in the graph where information is changing. The algorithm begins by initializing the sets at all nodes and constructing an initial worklist. It then repeats the process of removing a node from the worklist and updating its data-flow information. If the update changes the data-flow information at the node, then all of the nodes that depend on the changed information are added to the worklist.[2]

When it starts, the worklist contains all the nodes in the graph. If the worklist operates in a first-in, first-out order, then the algorithm visits the nodes in lexicographic order, which corresponds to $\mathcal{RPO}$. When the data-flow set for a node changes, the algorithm adds any of its successors that are not already on the worklist to the worklist. In this way, the algorithm avoids recomputing the set at a node where none of the facts have changed. (The result at a particular successor node may not change when it gets recomputed, but it will be true that at each recomputation of a node's data-flow set, at least one of its inputs has changed.)

In a long chain of back edges, like the one shown in the earlier example, this lets a fact propagate from node to node with a minimal amount of work. Since it only adds a node to the worklist when its input sets have changed, the worklist algorithm only recomputes sets when necessary. This avoids many non-productive recomputations, reducing the cost of the analysis.

The algorithm in Figure 4 does not specify an organization for the worklist. This unspecified detail plays a critical role in the performance of the algorithm. Just as $\mathcal{RPO}$ iteration creates the behavior of the round-robin algorithm, so too does the worklist organization govern the behavior of the worklist algorithm As part of this study, we investigated four distinct ways of organizing the worklist: a stack, a pair of stacks, a priority queue, and a simple first-in, first-out queue. Each has a distinct run-time cost. Each required some tuning to find the most effective configuration.

4.1 Single Stack Worklist

The simplest data structure we considered is a single stack. To avoid duplicate entries in the stack, we use a *SparseSet* [4]. Without duplicates, the implementation can allocate one array, sized to the number of blocks in the control-flow graph, as the stack. This data structure is trivial to implement. It has extremely low run-time overhead. It proved somewhat difficult to debug and to tune for performance.

The naive implementation of the stack performed worse than the round-robin algorithm. We discovered that the primary reason for this was our initialization. In the first pass over the code, we put all of the blocks onto the worklist, but because we walked the DFST in reverse postorder, the worklist starts with the nodes in the worst-case order for propagation. The solution to this is obvious; we mention it for two reasons. It is an excellent example of how a simple mistake can significantly hamper performance. Equally important, it led to another observation that had an impact on the performance of all four data structures.

[2]The code in Figure 4 uses the term "successors." When we discuss backward problems, we assume that the graph is reversed. Thus, successors is still the appropriate term, since the analysis operates on the DFST built from the reverse control-flow graph.

As the algorithm adds successor nodes to the worklist, the order in which they appear can affect performance in the same way that initializing the worklist in the wrong order did. To test the impact of putting nodes onto the worklist, we added an option to sort each group of successors that are added to the stack. The sorted successors more closely approximate the $\mathcal{RPO}$ ordering of the round-robin method.

Finally, we changed the mechanism that initializes the stack. The round-robin time bound includes a pass to initialize sets from local computations before beginning the iteration. Initializing the stack so that the nodes leave it in $\mathcal{RPO}$ avoids the startup problems. We can predict which blocks will change on the first pass after initialization by careful observation. The $\mathcal{RPO}$ ordering ensures that we process a node before any of its children, except for backedges. Thus, after initializing a node, the algorithm runs through its list of successors. If any of them have been initialized, then that initialized node is probably the target of a backedge.[3] Our improved scheme only adds these backedge targets to the worklist.

Notice, also, that the worklist algorithm avoids the final pass of the round-robin algorithm. The round-robin version requires one final pass to observe that the sets have reached a fixed point (*i.e.*, no set changes). The worklist algorithm detects this situation when the worklist becomes empty.

4.2 Alternating Stacks

The behavior of the stack is hard to analyze because the blocks come off the stack in an order that seems to bear little relation to the order used in the round-robin algorithm. This is antithetical to the goal of the worklist algorithm, which aims to borrow from the insight of the round-robin algorithm that information flows up a back edge and then flows forward again. If the worklist algorithm can capture this style of propagation, it can avoid the pitfall of doing unnecessary work.

To produce more predictable behavior and, we hope, to improve performance, we considered an implementation that uses two stacks. This version of the algorithm initializes one of the stacks as usual, either with every block or only those that are the target of a potential backedge. The iterative algorithm runs as in the single-stack implementation, except that successors of changed blocks are pushed onto the second stack. When the first stack empties, the algorithm swaps the sense of the two stacks (*i.e.*, the second stack has its members removed, and the first stack now has blocks pushed onto it), and it continues. The termination condition then requires that both stacks be empty.

This modification makes the iteration order more closely resemble a pass over the CFG. It still avoids some amount of unneeded work. The alternating stack method follows different paths of analysis simultaneously in a lock-step fashion.

Our implementation of two stacks is extremely efficient and the code is very straightforward. Because the implementation is in C, swapping the stack involves swapping two pointers. This worklist data structure exhibited sensitivity to the same details as the single stack, namely, that we had to be careful with initialization and with sorting values that get put onto the worklist.

[3]The other possibility is that we have found a cross edge. We do not differentiate between these cases.

4.3 A Queue

The intuition behind the alternating stack scheme is to mimic more closely the sweep behavior of the round-robin algorithm. Another way to accomplish this goal is to use a queue. The queue implementation has almost the same behavior as the alternating stack implementation. The difference is that the queue lacks the well defined delineation between sweeps (major iterations in the round-robin scheme). As with all of these data structures, the fact that no node can appear twice on the worklist changes subtly the iteration order. With two stacks, paths that converge on the same step will result in the join point appearing on the stack only once. With the queue, it is possible that the join point could go onto the worklist twice if the paths converge close together (in the sense of when they are encountered in the worklist), but only once if their convergence is farther apart.

The queue was less sensitive than the stack schemes to the initialization and sorting details.

4.4 A priority queue

The final data structure that we used used to implement the worklist was a priority queue. We had high hopes for this data structure, because it is easy to reason about, and it is easy to rationalize arguments that it should be the quickest to converge. Actual experimentation shows that it suffers from the same types of sensitivity as the other data structures. This organization touched the fewest blocks of any that we tested. For some problems this translates into faster running times. On others, unfortunately, it does not. We believe that the priority queue's benefits will rise with the cost of evaluating the data-flow equations at each block.

5. EXPERIMENTAL RESULTS

To evaluate the tradeoffs between these different schemes, we ran all five schemes on a series of problems. (The five schemes are: the round-robin algorithm, the worklist single-stack algorithm, the worklist alternating-stack algorithm, the worklist queue algorithm, and the worklist priority queue algorithm.) For each of the worklist schemes, we investigated the tradeoff between the naive initialization and the careful initialization. (Careful initialization almost always wins.) We also measured the impact of sorting the successors of a changed node before adding them to the worklist.

To measure the algorithms' performance, we watched two metrics. We measured the number of blocks that each run touched—that is, the number of blocks where the algorithm evaluated the data-flow equations. We also measured the running time of the iterative solver.

Among the anecdotal results of the experiment, two effects stand out. First, it is easy to drive the block counts for the worklist algorithm above those for the round-robin algorithm. Second, careful attention to initialization helps with this effect, as does sorting the nodes added to the worklist (so that they come off the list in $\mathcal{RPO}$).

5.1 Methodology

In the experiments presented in this section, we used two different analyses: liveness analysis and dominance analysis. The LIVE calculation follows the classic equations. The dominance calculation uses the simple data-flow equation described in [11]. We chose these two problems for several reasons. They are among the most common analyses used in an optimizing compiler. Both analyses fit the Kam-Ullman rapid condition. Dominance is a forward problem, while LIVE is a backward problem. Finally, space considerations restrict our ability to fully explore all of the many possibilities; because these analyses are so well known, readers will feel confident that they can extrapolate our results to other equations, such as those for non-rapid problems.

We used two different test suites for our experiments. Our first test suite is made up of 169 real-world Fortran routines taken from Forsythe, Malcolm, and Moler's book on numerical methods[20] along with a subset of the Spec and Spec '95 benchmarks. These codes are relatively complicated and have proven over the years to be a reasonable test of performance for our experiments; however, they are also small enough that, given today's computers' speeds, it is difficult to get accurate timings when running them. We address this problem in three ways.

1. We ameliorate the problem of preemptive multitasking by averaging ten different runs of the entire set of codes.

2. We ameliorate the problem of too little granularity in the timing mechanism by running the analysis multiple times for each routine. We adjust the number of times to match the size of the CFG: for graphs with fewer than fifty nodes, we run the analysis $10,000$ times; for graphs with fewer than 200 nodes, we run the analysis 1000 times; for graphs with fewer than 1000 nodes, we run the analysis 100 times, and for graphs of more than 1000 nodes we run the analysis 10 times. This has a certain amount of inherent inaccuracy, because the subtleties of cache performance may skew the behavior of the analysis and misrepresent the actual running time.

3. We address the generally small size of the procedures by using a second test suite of extremely large graphs generated in a way that makes their characteristics mimic those of the test suite, as described in [11]. These generated graphs come in four different sizes: $10,000$ nodes, $15,000$ nodes, $20,000$ nodes, and $25,000$ nodes. For each size, we generated 100 versions, giving us a test suite of 400 generated graphs.

   Since these graphs consist solely of control flow, we simulated data flow for the liveness calculation by randomly initializing the `USES` and `KILLED` sets for each block. Statistical analysis of our real-world test suite showed that the average size of the `KILLED` sets is 8.95 names, and the average `USES` set is 5.08. Further, we found that the overall namespace could be tied to the size of the CFG – there were an average of 23.05 names per basic block, although this gave us untenably large namespaces for our large graphs. Instead, we chose a namespace of 100 names, regardless of the size of the graph. For each block we used a random number generator to get set sizes between zero and twice the above statistics. These settings proved sufficient to get behavior that is similar to the real-world test suites.

   Thus, we argue that these random graphs are statistically similar to the real-world codes, and they give us an idea not only of how the various schemes scale, but, also, they remove some of the uncertainty related to trying to time multiple runnings of the same analysis.

Our timings, then, are taken from real-world codes and graphs that we generate to mimic the real-world codes. The value of these timings, however, are limited until we add another measurement; namely, the actual number of blocks touched by each algorithm and data structure. If we compare the number of blocks touched by the versions of the worklist iterative algorithm against the number touched by the round-robin algorithm, we can see the cost of each scheme versus its impact on performance.

Technically, we could make the

5.2 Formulation for Live

To compute live variables, we solve a backward data-flow problem. The equations annotate each block, b, with two sets LIVEIN(b) and LIVEOUT(b). The former holds all the variables that are live on entry to b, while the latter contains all the variables that are live on exit from b. The equations that define these sets are:

$$\text{LIVEIN}(b) = \text{LOCALLIVE}(b) \cup (\text{LIVEOUT}(b) \cap \overline{\text{KILLED}(b)})$$

$$\text{LIVEOUT}(b) = \begin{cases} \emptyset, & \text{if } b \text{ is the exit block,} \\ \bigcup_{s \in succs(b)} \text{LIVEIN}(s), & \text{otherwise} \end{cases}$$

Here, LOCALLIVE(b) contains those variables that have upwards exposed uses in b and KILLED(b) contains every variable that is defined in b.

In solving these equations, we use the $\mathcal{RPO}$ of the reverse control-flow graph. This is distinct from the $\mathcal{PO}$ of the control-flow graph. While it is tempting to think of these orders as equivalent, they produce noticeably different

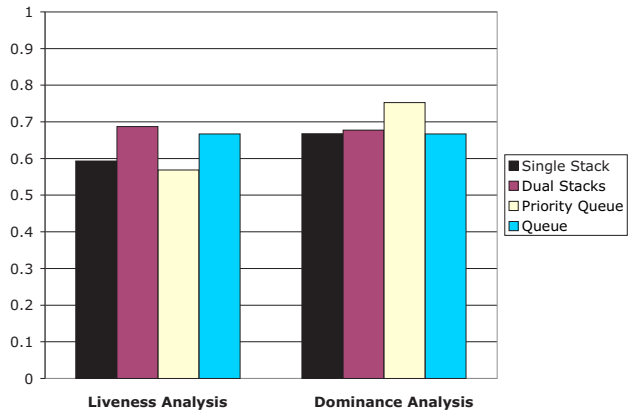


Figure 5: Normalized Times, Small Graphs

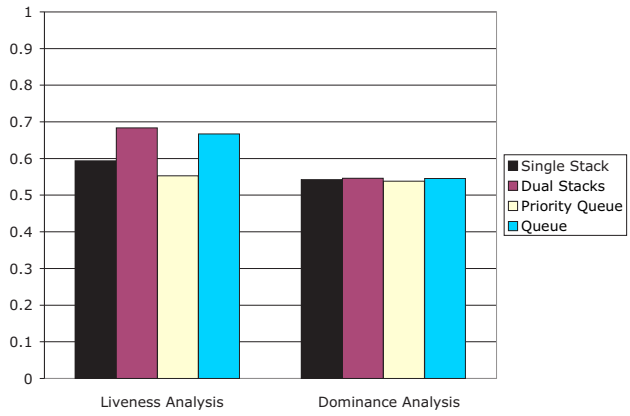


Figure 6: Normalized Block Counts, Small Graphs

behavior in the round-robin algorithm For a backward problem, the compiler should compute the $\mathcal{RPO}$ of the reverse control-flow graph.

5.3 Formulation for Dominance

To compute dominance in this information, we use the formulation due to Fran Allen, described in [1] and [11]. Each block is annotated with a set DOM that contains the names of the blocks that dominate it.

$$\text{DOM}(b) = \left(\bigcap_{p \in preds(b)} \text{DOM}(p) \right) \cup \{b\}$$

This equation is simple enough that any instance of the problem involving a reducible flow graph will halt in two iterations. On the codes from the benchmark suites, almost all the graphs are reducible. Thus, the numbers for dominance on the real world codes can also be viewed as proportional to the cost of a single pass over the graph.

5.4 Measurements

All of the measurements presented in this section were made on an unloaded Apple PowerBook computer running OS X (Jaguar) with a 500 MHZ PowerPC G4 processor, 1 MB of level 2 cache, and 256 MB of RAM.

Figure 5 shows the average running times of the four worklist schemes, normalized against the average running time for the round-robin algorithm. This graph only includes data from the real-world programs; it excludes the large synthetic graphs. All four techniques are substantially faster than the round-robin algorithm. The priority queue wins on liveness, while the simple stack does slightly better than the others on dominators.

Figure 6 shows the average number of blocks touched by the four worklist schemes, normalized against the average number of blocks touched by the round-robin scheme. These numbers give the reader some insight into the behavior that might be expected for problems where evaluating the data-flow equations at a block is more expensive. Notice that the priority queue touches the fewest blocks in both problems. For liveness, this translates into a faster running time. For dominance, on the small graphs, the additional overhead of the priority queue does not translate into running time.

Table 1 shows the raw data for running times, aggregated by graph size; the first column indicates the number of nodes considered in that row. The second column shows the number of graphs in that size range. The remaining columns show running times for the analysis, normalized to the running time of the round-robin algorithm.

All tests shown in the table were done using the $\mathcal{RPO}$ on the appropriate graph (the reverse control-flow graph for liveness and the control-flow graph for dominators). For the stack schemes and the simple queue, we tested the scheme with sorting on insertion into the worklist and without it. The table shows the results for the best overall scheme in each column. For the single stack, sorting on insertions won. With the double stack, liveness used unsorted insertions while dominators used sorted insertions. On the simple queue, unsorted insertions again won.

All of these schemes significantly outperformed the round-robin algorithm. However, no single structure is best in all situations. For liveness, the priority queue generally wins on the small graphs, while the single stack wins on the large graphs. For dominance, the priority queue wins on the small graphs. On the large graphs, the two queue implementations are essentially equivalent.

5.5 A Larger Experiment

The previous experiments isolated the data-flow analysis so that we could better describe the behavior of the worklist data structures. These results are instructive, but we also wanted to discover the impact on the overall running time of one of our optimization passes.

For this, we rewrote our implementation of Cooper and Simpson's strongly-connected-component value-numbering algorithm (SCC), which in turn uses the value-driven-code-motion algorithm (VDCM) [14, 15]. The VDCM algorithm relies on SSA form to let us manipulate the equations from

Number of Blocks	Number of Graphs	Round Robin	Single Stack	Double Stack	Priority Queue	Simple Queue
LIVENESS						
Under 50	108	1	0.7748	0.7609	0.7449	0.7449
$\geq$ 50 & $<$ 100	35	1	0.6889	0.6914	0.6512	0.6812
$\geq$ 100 & $<$ 200	13	1	0.6091	0.5893	0.4828	0.5743
$\geq$ 200 & $<$ 400	7	1	0.8939	0.9072	0.5076	0.8575
$\geq$ 400	6	1	0.4695	0.5124	0.4561	0.4773
10,000	1000	1	0.5863	0.5901	0.6034	0.5906
15,000	1000	1	0.5801	0.5845	0.5966	0.5859
20,000	1000	1	0.5636	0.5688	0.5799	0.5701
25,000	1000	1	0.5661	0.5718	0.5806	0.5736
DOMINATORS						
Under 50	108	1	0.7015	0.7222	0.7284	0.7068
$\geq$ 50 & $<$ 100	35	1	0.6277	0.6333	0.6928	0.6347
$\geq$ 100 & $<$ 200	13	1	0.6411	0.6449	0.7216	0.6457
$\geq$ 200 & $<$ 400	7	1	0.6743	0.6857	0.7657	0.6343
$\geq$ 400	6	1	0.6936	0.7003	0.8552	0.7138
10,000	1000	1	0.3890	0.3852	0.3913	0.3834
15,000	1000	1	0.3829	0.3773	0.3778	0.3746
20,000	1000	1	0.4036	0.3881	0.3831	0.3851
25,000	1000	1	0.4387	0.4049	0.3939	0.4018

Table 1: Comparisons Against Round-Robin

Drechsler and Stadel's lazy-code-motion algorithm [19] into simpler, more efficient versions. Specifically, we substituted our worklist data structures in the *availability*, *anticipability*, and *later* calculations. The results for the twelve largest routines in our test suite are shown in Table 2.

The first column of numbers shows the time, in seconds, used by the round-robin iterative algorithm to compute the data-flow analysis. The second column shows the total time used for the entire algorithm. These numbers suggest that as we begin processing larger files, the time spent on data-flow analysis increases as a ratio of the whole and dominates the total processing time for this optimization pass. The remaining four columns show the ratio of the time spent on the entire algorithm. These columns show that the worklist algorithm gets an average speedup of about seven percent overall.

These gains are relatively modest, but they were achieved with relatively simple engineering. Further, today's changing compiler technology argues strongly for every gain in compile time. Not only is this important in JITs, but other more active compilers such as those found in the GrADS Project will be required to be as fast as possible[3]. Cooper and Subramanian report on an artificial intelligence system aimed at improving optimization that invokes the compiler repeatedly[16]. In continuing experiments on that project, optimizations have been run upwards of *100 million times*. Clearly, all improvements in the speed of the compiler have impact in these environments.

6. NON-RAPID PROBLEMS

The classic time bound for the iterative algorithm also restricts its focus to a class of data-flow frameworks called the "rapid" frameworks. This class includes most of the classic problems in data-flow analysis, such as availability, dominance, and liveness.

To an increasing extent, however, modern compilers must solve problems that lie outside this class. Kildall's formulation of single-procedure constant propagation is not rapid [25]. The basic problems in interprocedural analysis, such as may-modifies information, parameter aliasing sets, and interprocedural constant propagation are not rapid [13, 10, 7, 5]. Pointer analysis, in both the alias formulation and the points-to formulation, is not rapid [31]. It is important that the compiler writer understand how the non-rapid nature of these problems affects the behavior of iterative solvers for them.

A problem can be non-rapid for any number of reasons. The equations for Kildall's formulation of constant propagation are not distributive; thus, an iterative solver will find a fixed point, but not a unique (and, therefore, maximal) fixed point [26]. The interprocedural summary prob-

File Name	Round Robin DFA Time	Round Robin *SCC-VDCM* Time	Ratio Versus *SCC-VDCM* Single Stack	Double Stack	Priority Queue	Simple Queue
deseco	0.10	0.25	0.92	0.92	0.92	0.88
inibnd	0.14	0.29	0.93	0.93	0.97	0.93
advbnd	0.15	0.30	0.90	0.90	0.93	0.90
iniset	0.17	0.31	0.94	0.94	0.90	0.94
radbg	0.24	0.42	0.98	0.95	0.95	0.93
radfg	0.25	0.46	0.91	0.93	0.93	0.93
init	0.33	0.74	0.93	0.91	0.92	0.91
twldrv	0.29	0.75	0.96	0.95	0.96	0.95
parmve	0.53	0.86	0.93	0.94	0.94	0.93
field	1.09	1.67	0.95	0.95	0.95	0.95
parmov	1.34	2.25	0.93	0.92	0.91	0.93
parmvr	1.53	2.47	0.94	0.94	0.93	0.92
		AVERAGE	0.93	0.93	0.93	0.92

Table 2: SCC-VDCM timing results

lems and pointer aliasing suffer from interactions between names; the facts for one name determine facts for another name. This can force the algorithm to propagate facts along binding chains that can wrap around cycles in the graph. Some problems have bidirectional data-flow equations; the complex patterns of flow in these problems slows convergence[18].

Fast algorithms exist for many of the non-rapid problems. For example, separating constant propagation into a name-by-name problem exposes a tight time bound based on the limited height of the constant propagation lattice [32]. The interprocedural may modifies problem can be solved in linear time by formulating it over an expanded analog of the call graph called the binding graph [13]. Both these problems are appropriately solved with a worklist iterative algorithm, even though the frameworks are not rapid.

The worklist iterative algorithm follows the data-dependences in the specific problem instance being analyzed. When the set associated with a node changes, all the sets that depend directly on that changed set are scheduled for re-evaluation (by adding their nodes to the worklist). Wegman and Zadeck's formulation of constant propagation uses a graphical form of the static single-assignment graph. Here, the natural representation has a single fact associated with each node — the lattice value associated with the name being defined. A worklist iterative scheme will follow the changing sets. Since each set can change no more than twice, the algorithm will converge quickly.

Similarly, the fast algorithm for may modifies information unrolls cycles in the graph to reflect the chains of parameter-to-parameter binding that occur. The worklist iterative algorithm, applied to the original call graph, will follow these chains. This may take it around a given cycle many times — something that cannot happen on a rapid problem. The number of iterations around that cycle will be bounded by the same constraints that bound the fast algorithm on the binding graph — the length of those parameter-binding chains. Again, the worklist iterative algorithm will converge quickly.

No general time bound applies to these non-rapid problems. In each case, the complexity depends on problem-specific details. However, the worklist iterative algorithm behaves appropriately. It follows the data dependences in the problem instance and evaluates the sets until it reaches a fixed point. In many cases, this is as fast (asymptotically) as we can solve the problem. Future work will explore the behavior of the worklist data structure on non-rapid problems.

7. SUMMARY AND CONCLUSIONS

Iterative data-flow analysis is widely used because it is robust, fast, and easy to implement. The round-robin algorithm has well understood behavior backed by solid theory. The worklist algorithms offer the compiler writer the promise of faster analysis; however, their behavior is harder to understand and their asymptotic time bounds are much less reassuring. This paper shows that data-structure choices in the implementation of worklist algorithms can have a significant impact on their actual run-time behavior. With careful engineering, all of our schemes outran the round-robin algorithm.

In practice, well-implemented worklist algorithms run quickly. The schemes presented here have the potential to speed up iterative analysis by up to 40%. This will allow compilers to tackle larger problems, such as whole-program binary-to-

binary translation. It will also allow speed-sensitive just-in-time compilers to perform more analysis. Finally, recent research has focussed on using sometimes massive numbers of recompilations to guide optimization; in these environments, even small improvements in throughput can have substantial effects on total compilation time.

8. ACKNOWLEDGEMENTS

Preston Briggs, Steve Reeves, Linda Torczon, and Todd Waterman have all contributed to this work by listening to our discussions, posing questions, and offering advice. All of the people who have contributed to the Massively Scalar Compiler Project deserve recognition for building the tools that we used in these experiments. This work was supported by the National Science Foundation, through the GrADS project, grant 9975020, and by the Los Alamos Computer Science Institute. We owe all these people and organizations a debt of thanks.

9. REFERENCES

[1] Frances E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[2] Frances E. Allen and John Cocke. Graph-theoretic constructs for program flow analysis. Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, July 1972.

[3] Francine Berman, Andrew Chien, Keith D. Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The grads project: software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[4] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, March–December 1993.

[5] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Notices*, 21(7):162–175, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[6] D. Callahan, A. Carle, Mary W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[7] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):152–161, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[8] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. *SIGPLAN Notices*, 32(6):273–286, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[9] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[10] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, New Orleans, Louisiana, January 1985.

[11] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *(in review)*, April 2003.

[12] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a control-flow graph from scheduled assembly code. Technical Report 02-399, Department of Computer Science, Rice University, June 2002.

[13] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Notices*, 23(7):57–66, July 1988. *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation.*

[14] Keith D. Cooper and L. Taylor Simpson. Scc-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, October 1995.

[15] Keith D. Cooper and L. Taylor Simpson. Value-driven code motion. Technical Report CRPC-TR95637-S, Center for Research on Parallel Computation, Rice University, October 1995.

[16] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, 2002.

[17] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *SIGPLAN Notices*, 30(6):93–102, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[18] Dhananjay M. Dhamdhere and Uday P. Khedker. Complexity of bidirectional data flow analysis. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Charleston, South Carolina, January 1993.

[19] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code

motion". *SIGPLAN Notices*, pages 29–38, May 1993.

[20] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[21] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23(1):172–202, 1976.

[22] Matthew S. Hecht and Jeffrey D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[23] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal of Computing*, 1(2):188–202, June 1976.

[24] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems* (TOPLAS), 21(4):848–894, July 1999.

[25] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[26] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[27] Ken Kennedy and Scott K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 32–49, Atlanta, Georgia, January 1976.

[28] Ken Kennedy and Linda Zucconi. Applications of graph grammar for program control flow analysis. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 72–85, Los Angeles, California, January 1977.

[29] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.

[30] Donald E. Knuth. An empirical study of Fortran programs. *Software – Practice and Experience*, 1:105–133, 1971.

[31] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, March 2001.

[32] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual Symposium on Principles of Programming Languages*. ACM, January 1985.

[33] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* (Toplas), 13(2):181–210, April 1991.