Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers

Teck Bok Tok[1], Samuel Z. Guyer[2], and Calvin Lin[1]

[1] Department of Computer Sciences,
The University of Texas at Austin, Austin, TX 78712, USA
[2] Department of Computer Science,
Tufts University, Medford, MA 02155, USA

Abstract. This paper presents a new worklist algorithm that significantly speeds up a large class of flow-sensitive data-flow analyses, including typestate error checking and pointer analysis. Our algorithm works particularly well for interprocedural analyses. By contrast, traditional algorithms work well for individual procedures but do not scale well to interprocedural analysis because they spend too much time unnecessarily re-analyzing large parts of the program. Our algorithm solves this problem by exploiting the sparse nature of many analyses. The key to our approach is the use of interprocedural def-use chains, which allows our algorithm to re-analyze only those parts of the program that are affected by changes in the flow values. Unlike other techniques for sparse analysis, our algorithm does not rely on precomputed def-use chains, since this computation can itself require costly analysis, particularly in the presence of pointers. Instead, we compute def-use chains on the fly during the analysis, along with precise pointer information. When applied to large programs such as `nn`, our techniques improve analysis time by up to 90%—from 1974s to 190s—over a state of the art algorithm.

1 Introduction

Flow-sensitive analysis is important for problems such as program slicing [22] and error checking [6, 7]. While recent work with BDD's has produced efficient algorithms for solving a variety of flow-insensitive analyses [24, 25], these techniques have not translated to flow-sensitive problems. Other techniques, such as demand interprocedural analysis [11], do not apply to pointer analysis. Thus, the most general technique for solving flow-sensitive problems continues to be iterative data-flow analysis. Existing iterative data-flow analysis algorithms work well within a single procedure, but they scale poorly to interprocedural analysis because they spend too much time unnecessarily re-analyzing parts of the program.

At issue is the manner in which worklists are managed, which can greatly affect the amount of work performed during each iteration. The most basic algorithm maintains a worklist of basic blocks for each procedure. Basic blocks are repeatedly removed from the worklist and applied with the flow functions. If any changes to the flow values occur, all reachable blocks are added to the worklist. This basic algorithm becomes extremely inefficient when used for interprocedural analysis: when re-analyzing a block

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 17–31, 2006.

that contains procedure calls, the algorithm may revisit all of the called procedures, even though many of them may not require re-analysis. Extensions to this basic algorithm, such as Hind and Pioli's priority queue approach [10], which considers the structure of the control flow, also suffer from this problem of useless work. For example, when the Hind and Pioli algorithm is applied to the nn program (about 36K lines of C), we find that only 3% of the basic block visits are useful—the others do not update any flow values.

In this paper we present a new algorithm for interprocedural iterative data-flow analysis that is significantly more efficient than previous algorithms. The algorithm exploits data dependences to reduce the number of times that blocks are revisited. The algorithm builds on an insight from previous work on intraprocedural algorithms: def-use chains can be used to directly identify those blocks that are affected by flow value updates [23]. This goal, however, is complicated by the fact that the computation of def-use chains is itself an expensive flow-sensitive computation, particularly in the presence of pointers. The example in Fig.1 shows why: the first time through the loop "*p" refers to x and therefore implies a def-use chain to the statement above it. The second time through the loop, however, "*p" refers to z, which implies a def-use chain to the block following the loop.

```
p = &x;
while (cond) {
  y = x;
  *p = 7;
  p = &z;
}
y = z;
```

Fig. 1. A loop example

Our algorithm solves this problem by computing data dependences on the fly, along with precise pointer information, while solving the client data-flow analysis problem. The key to our approach is that as the pointer analysis computes the uses and defs of variables, it builds a network of use-def and def-use chains: the use-def chains enable fast lookup of flow values, while the def-use chains are used to narrow the scope of re-analysis when flow values change. Initially, the framework visits all basic blocks in a procedure to compute a first approximation of (1) the pointer information, (2) the data dependences, and (3) the client data-flow information. Subsequent changes in the flow values at a particular def only cause the corresponding uses to be re-analyzed. More importantly, our system incorporates new dependences into the analysis as the pointer analysis discovers them: changes in the points-to sets cause reevaluation of pointer expressions, which in turn may introduce new uses and defs and force reevaluation of the appropriate parts of the client analysis problem. Occasionally, we find pairs of basic blocks that are connected by large numbers of def-use chains. For these cases we have explored a technique called *bundling* which groups these def-use chains so that they can be efficiently treated as a single unit.

This paper makes the following contributions. First, we present a metric that allows us to compare the relative efficiency of different worklist algorithms. Second, we present a new worklist management algorithm, which significantly improves efficiency as measured by our metric. Third, we evaluate our algorithm by using it as the data-flow engine for an automated error checking tool [7]. We compare our algorithm against a state-of-the-art algorithm [10] on a large suite of open source programs. We show that improved efficiency translates into significant improvements in analysis time. For our set of 19 open source benchmarks, our algorithm improves efficiency by an average of 500% and improves analysis time by an average of 55.8% when compared with the Hind and Pioli algorithm. The benefits of our algorithm increase with larger and more complex benchmarks. For example, the `nn` benchmark sees an order of magnitude improvement in efficiency, which translates to a 90% improvement in analysis time.

This paper is organized as follows. We review related work in Section 2. Section 3 briefly describes the analysis framework. Section 4 presents our worklist algorithm, $\mathcal{DU}$ that enables sparse analysis, and a variant, $\mathcal{DU}_{loop}$ that exploits loop structures. Section 5 presents our empirical setup and results. We conclude in Section 6.

2 Related Work

There are two families of data-flow algorithms: elimination methods [21] and iterative algorithms. Elimination methods, such as interval analysis, solve systems of equations and do not work well in the presence of pointers. The class of iterative algorithms include worklists, round robin, and node listing algorithms [13, 1]. Both the round-robin and node listing approaches are dense analyses in the sense that blocks are re-analyzed needlessly.

Previous work on comparing worklist algorithms includes Atkinson and Griswold's work [2], which shows that the performance difference between a round-robin algorithm and a worklist algorithm can be huge. They propose a hybrid algorithm that combines the benefits of the two. In separate work, Hind and Pioli [10] exploit loop structure by using a priority queue. We find that Atkinson and Griswold's hybrid algorithm can sometimes be better and sometimes worse than Hind and Pioli's algorithm. To provide a basis for comparison with our new algorithm, we use as our baseline a version of the priority-queue approach that does not use the identity transfer function or *IN/OUT* sets.

Wegman and Zadeck pioneered the notion of sparse analysis in their sparse constant propagation algorithm [23]. We extend their approach to handle pointers, and we address the need to discover def-use chains on the fly as the analysis progresses.

Another possible method of exploiting sparsity is to use a sparse evaluation graph (SEG) or its variants [4, 17], which are refinements of CFGs that eliminate irrelevant statements. Hind and Pioli report improvement with pointer analysis when SEG is used [10], but because their use of *IN/OUT* sets does not fully exploit sparsity. It is unclear how much our sparse analysis can benefit from an SEG, and we leave this study as future work.

For some classes of data-flow analysis problems, there exist techniques for efficient analysis. For example, demand interprocedural data-flow analysis [11] can produce precise results in polynomial time for interprocedural, finite, distributive, subset problems (IFDS). Unfortunately, this class excludes pointer analysis, and a separate pointer analysis phase may be required.

In the context of pointer analysis itself, previous work on flow-sensitive pointer analysis algorithms that makes use of worklists [18, 3] do not attempt to tune the worklist, so our worklist algorithm can be applied to such work to improve their performance. Other pointer analysis algorithms sometimes tradeoff precision for scalability [9]. Our algorithm improves the efficiency of the worklist component that drives the analysis, without affecting the precision of the analysis.

Worklist algorithms have also been studied from other perspectives. For example, Cobleigh et at. [5] study the effects of worklist algorithms in model checking. They identify a few dimensions along which an algorithm can be varied. Their main result is that different algorithms perform best during different phases of analysis. We do not attempt to partition an analysis into phases. Similarly we do not address the issue of partitioning the problem into subproblems [20], nor do we divide a large program into manageable modules [19, 15].

3 Analysis Framework

This section provides background about our data-flow analysis framework, including details about how we efficiently compute reaching definitions using dominance information.

We assume an iterative-based whole-program flow-sensitive pointer analysis that uses a worklist for each procedure, where each worklist maintains a list of unique CFG blocks. An alternative is a single worklist of nodes from a supergraph [16], eliminating procedure boundaries, but we believe that such a large worklist would be too expensive.

Our algorithm requires accurate def-use chains. Since definitions are created on the fly during pointer analysis, we need to update chains whenever a new definition is discovered. To perform such updates efficiently, we assume SSA form for all variables, including heap objects. SSA has well-understood properties: every use u has a unique reaching definition d, and d must dominate u if u is not a phi-use. These properties, together with dominance relations (described below), allow us to quickly determine if a newly-discovered definition invalidates any existing def-use pairs. Finally, to merge flow values at different call sites, the system uses interprocedural ϕ-functions at procedure entries.

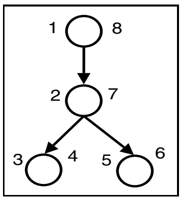


Fig. 2. Node numbering on an expanded dominator tree

Our system does not use *IN/OUT* sets to propagate flow values [3] because their use would mandate a dense analysis: any update on a node would force all of its successors to be revisited. Our sparse analysis instead uses dominance information to efficiently retrieve flow values across use-def chains. To obtain the nearest reaching definition for a given use, we build from the CFG an expanded dominator tree where each node represents a statement. We assign to each node n a preorder number and a postorder number, denoted $min(n)$ and $max(n)$, respectively, so that given two distinct statements m and n, m dominates n, denoted by $DOM(m, n)$, if and only if $min(m) < min(n) \ \wedge max(m) > max(n)$. These numbers are assigned by performing a depth-first traversal and incrementing a counter each time we move either up or down the tree. Fig.2 shows the numbers assigned to an example expanded dominator tree. The number to the left of each node is its min number and the number to the right of each node is its max number.

To use this expanded dominator tree, each definition of a variable is associated with a unique statement, and we store all definitions of a variable in a list that satisfies the following invariant: let d_i be the i^{th} definition in the list, and let n_i be the statement associated with d_i, then the min of the statements are stored in decreasing order:

$$\begin{aligned} & \forall i < j : min(n_i) > min(n_j) \\ \Rightarrow \quad & \forall i < j : \neg DOM(n_i, n_j) \end{aligned} \tag{1}$$

To find the nearest reaching definition that strictly dominates a statement m, we (1) perform a binary search to obtain the minimum i such that d_i in the list satisfies $min(n_i) < min(m)$; and (2) perform a linear scan from i to the end of the list to find the first d_i such that $max(n_i) > max(m)$. For the resulting d_i, $DOM(n_i, m)$. Without the binary search, a linear search alone (starting from $i = 1$) can still find the correct result if the $DOM(n_i, m)$ test is used, because by Invariant (1) the first n_i that dominates m must also be the nearest.

4 $\mathcal{DU}$: Worklist Management

Our algorithm is based on a well-known idea: use def-use chains to identify those blocks that may be affected by the most recent updates, thereby exploiting the sparsity of the analysis. To compute def-use chains in the presence of pointers, we present $\mathcal{DU}$, a worklist algorithm that is coupled with pointer analysis. This algorithm can exploit both intra- and inter-procedural def-use chains.

To simplify our presentation, we start off with a naive, inefficient version and gradually add details to build our full version at the end of this section. We will use Fig.3 as a running example.

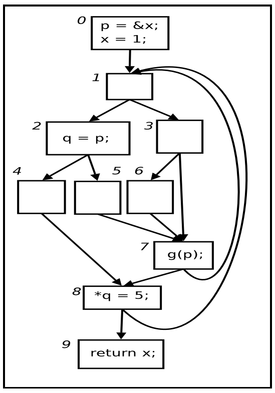


Fig. 3. An example CFG

Structure of a Worklist Algorithm

The left box of Fig.4 gives a high-level description of a generic worklist algorithm. It maintains a queue of CFG blocks, initially set to include all blocks in reverse post-order. The pointer analysis retrieves and analyzes one block from the worklist. The pointer analysis then identifies the set of *changes*, which is the set of variables whose flow values have been updated. The algorithm then uses a function R to compute and add to the worklist the blocks that will be revisited. The worklist may then be reordered, as we discuss in Section 4.1. The entire process is repeated until the worklist becomes empty. Different implementations differ in the computation of R and in the worklist reordering.

Naive Worklist Algorithms

The behavior of the function R is crucial to the worklist efficiency. If we do not know which blocks are affected by the changes in the last block visit, then we must conservatively return all the reachable blocks of the given block n. We refer to this version as R_{reach}, shown in the right box of Fig. 4. Considering the example in Fig.3, suppose we have just revisited the loop header (block 1), where a new ϕ-function for variable x is created. R_{reach} will return blocks 2–9, a total of 8 blocks.

Worklist Algorithm Using Intraprocedural Def-Use Chains

R_{reach} is easy and cheap to compute, but it adds too many blocks. We introduce R_{DU}, shown in the right of Fig.4. This function iterates over the set of variable changes, retrieves their last definitions in the block, and obtains their use sites in the procedure. The blocks containing these use sites are returned and added to the worklist. For now, assume that only intraprocedural def-use chains are used. In the example of Fig.3, only two blocks (7 and 8) are returned by R_{DU}, so R_{DU} is more efficient than R_{reach}.

```
Initially:
$WL$ = reverse_post_order($CFG$)

Main loop:
while $WL \neq \emptyset$ do
  block $n$ = remove_front($WL$);
  $var\_changes$ = visit_block($n$);
  if $var\_changes \neq \emptyset$ then
    $more$ = $R(n, var\_changes)$;
    merge($more$, $WL$);
```

```
$R_{reach}(n, var\_changes)$ {
  return reachable_blocks($n$);
}
$R_{DU}(n, var\_changes)$ {
  for $v \in var\_changes$ do
    $d$ = last_def_of($v$,$n$);
    for $u \in$ uses($d$) do
      add(block_of($u$), $result$);
  return $result$;
}
```

Fig. 4. Initial version of algorithm $\mathcal{DU}$. The function R computes what blocks need to be added to the worklist. The first version, R_{reach}, simply returns reachable blocks from block n. R_{DU} uses def-use chains to compute the blocks affected by the latest variable updates during the last block visit.

Dynamic Def-Use Computation

Def-use chains are computed on the fly as new pointer information is discovered, so the worklist algorithm needs to be aware that some defs may temporarily have no uses. As we shall see, the solution requires a new form of communication between the pointer analysis and the worklist algorithm.

New definitions are created at indirect assignments, function calls, and ϕ-functions. There are three cases to consider: (i) a new def leads to a new ϕ-function; (ii) a new def resides between an existing def-use pair; (iii) a new def temporarily has no reaching definition.

Consider case (i). SSA form requires that whenever a new definition d is created, a ϕ-function is also created at dominance frontiers. Because pointer information is not yet available, many ϕ-functions cannot be computed in advance.[1] Therefore after d is created, the algorithm must make sure that the dominance frontiers are eventually revisited, so that the ϕ-functions can be created.

Cases (ii) and (iii) are similar because any existing use below the new def d may need to update its reaching definition. Such situations often occur in the presence of loops when a use is visited before its reaching definition is created. In the example of Fig.3, if a new ϕ-function for p is created at the loop header, we need to make sure that block 7 is revisited, even if the new def has no known use yet.

[1] Short of exhaustive up-front creation.

There are two possible solutions. The first method identifies those uses that need to be revisited by simply searching through existing def-use chains and through existing uses without defs. (It only needs to inspect those chains whose def is the nearest definition above d.) The second solution handles (iii) as follows: whenever a use u without a reaching definition is discovered, statements above u are marked if they are merge points or if they contain indirect assignments or function calls. Later when d is discovered at one of these statements, u is revisited.

The first method can be quite expensive, while the second method does not handle case (ii). We have found that combining the two is cost effective. We use the second method on case (iii) by marking only loop headers, and we use the first method otherwise. This combination works well in practice, most likely because uses that initially have no reaching definition typically occur in loops, so marking and inspecting loop headers is sufficient. Because in practice there is usually a small, fixed number of loop headers in any procedure, the overhead due to the markings is small.

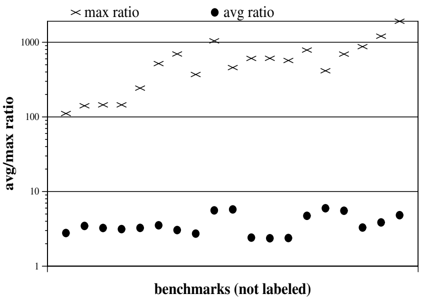


Fig. 5. Maximum and average ratio $r = C/(B+1)$, in log scale. The set of benchmarks is explained in Section 5. The high ratios indicate potential for high overhead due to def-use chains.

Bundles

One problem with R_{DU} is that it can be expensive to follow du-chains if there are many du-chains that connect the same two basic blocks. We can measure the extent of this problem as follows. Define C to be the number of variables whose flow values change after analyzing a given basic block. Define B to be the number of unique basic blocks that contain uses of these C variables. If the ratio $r = C/(B+1)$ is large, then there is a large amount of redundancy in the dependence information represented by the du-chains. (The $+1$ term prevents division by zero.) Fig.5 shows the maximum and average values of this ratio for the benchmarks that we use in our later experiments. We omit the minimums, which are all close to zero. We see that the average ratios hover between two and ten, while the maximums are two orders of magnitude larger. One reason for the large maximums is the large number of global and heap variables defined at merge points near the end of procedures, which leads to large values of C with no further uses in the procedure ($B = 0$).

To handle the cases where the value of r is large, we define a *bundle* $\langle D, U\rangle$ to be the set of all def-use chains whose definitions and uses share the blocks D and U, respectively. A bundle is used as follows (see Fig.6). After analyzing a block n, all bundles of the form $\langle n, u\rangle$ are retrieved. R_{bundle} then iterates through these bundles: for

```
$R_{bundle}(n, changes)$ {
  $bundles$ = set of bundles $\{\langle n, *\rangle\}$;
  for $b \in bundles$ do
    if $b$ contains var$\in changes$ then
      let $b$ = $\langle n, u\rangle$;
      add $u$ to $result$;
  return $result$;
}
```

Fig. 6. Efficient R_{bundle} that uses bundles

each bundle that contains a variable in the $changes$ set, the u stored in the bundle is added to the worklist. When there is no bundle ($B = 0$), no overhead will be incurred even if there is a large number of changes.

Our experimental results with an earlier implementation of our $\mathcal{DU}$ algorithm shows that bundles are quite effective for reducing analysis time. Our results also show that bundles can consume considerable space. Given the space overhead of bundles and the bi-modal distribution of r values, we use a simple heuristic to apply bundles selectively. This heuristic compares C to a threshold that is defined as some factor of the size of the basic block in question (as defined by the number of statements in the block).

Because we have not yet tuned the selective use of bundles for the current implementation of our worklist algorithms, the results shown later in this paper do *not* use bundles. We expect to see improved results once this tuning has been completed.

Handling Interprocedural Def-Use Chains

Our system allows def-use chains to cross procedure boundaries, which typically occurs when a procedure accesses global variables or accesses variables indirectly through pointers. The framework treats these variables as if they were inputs or outputs to the procedure but not explicitly mentioned in the formal parameters. During interprocedural analysis, these def-use chains can be used to further improve worklist efficiency.

A procedure input is a variable that has a use inside a procedure and a reaching definition inside a caller. When re-analyzing a procedure due to changes to procedure inputs, we revisit only the affected use sites—which are often a subset of the procedure's blocks—because we know which inputs' flow values have changed. To identify these changed flow values, we use interprocedural ϕ-functions, which merge flow values at procedure entries. As before, these ϕ-functions are created on the fly.

A procedure output has a definition inside the procedure with some use inside a caller. The output can export a new variable, for example, a heap allocated object, or it can export a side effect on an input. We use information about the procedure output to help manage the worklists of the callers: if there is change in flow value in an output variable, the worklist of each caller marks the sites that need to be revisited. For this idea to work, we require a departure from the usual way worklists are used.

In many existing algorithms, analysis is performed one procedure at a time: analysis of a procedure P is started by placing all of its blocks on its worklist. To exploit interprocedural def-use chains, we no longer initialize the worklist to all blocks, except when the procedure is analyzed for the first time. Instead, a procedure P's blocks are marked to identify callers of P that change P's inputs and to identify callers of P that are affected by P's outputs.

In conjunction with a call graph worklist, this strategy allows us to exploit sparsity at the granularity of the procedure level. Thus, a procedure need not appear in the call graph worklist if its corresponding worklist is empty.

Full Version of Algorithm $\mathcal{DU}$

Fig.7 presents our full algorithm. It first computes the reverse post-order, rpo, of the procedure, which is used as the worklist if the procedure is analyzed for the first time. Otherwise, the inputs are processed, searching for those with new flow values, so that

Initially:

```
if analyze proc for 1st time then
  $rpo$ = reverse_post_order($CFG$);
  $WL$ = $rpo$;
  $marked$ = $\emptyset$;
else
  $WL$ = process_proc_inputs();
  merge($marked$, $WL$);
  sort $WL$ according to $rpo$;
```

Main loop:

```
while $WL \neq \emptyset$ do
  $n$ = remove_front($WL$);
  $changes$ = visit_block($n$);
  if $changes \neq \emptyset$ then
    $more$ = $R_{bundle}(n, changes)$;
    merge($more$, $WL$);
```

Finally:

```
// worklist done; export variables.
$outputs$ = vars to export;
if $outputs \neq \emptyset$ then
  add(exit_block(), $outputs$);
```

Addition interface:

```
add($e$, $changes$) {
  $bundles$ = set of $\{\langle e, *\rangle\}$;
  for $b \in bundles$ do
    if $b$ contains var$\in$
    $changes$ then
      let $b$=$\langle e, u\rangle$;
      $p$=proc_of($u$);
      $p\_marked$=marked_set($p$);
      add $u$ to $p\_marked$;
}
```

Fig. 7. Full version of algorithm $\mathcal{DU}$, when it considers both intra- and interprocedural def-use chains. Note that we also use bundles to export variables.

their use sites are put in the worklist. Those blocks marked for re-analysis are placed on the worklist, which is then sorted according to rpo.

The main loop is the same as that of Fig.4. After the loop, all outputs with changed flow values are gathered, and the callers' callsites are processed. During this final stage, bundles can again be used in the add routine to avoid looping through all the variables in $changes$. We assume that there is a definition for each output variable at the callee's exit block e. Each bundle of the form $\langle e, u\rangle$ therefore has a use site in a caller. We can then mark this use site in the caller's $marked$ set, enabling the caller to re-analyze it later.

4.1 Exploiting Loop Structure

By always adding blocks to the rear of the worklist, our $\mathcal{DU}$ algorithm ignores loop structure, which would seem to be a mistake because CFG structure seems to be closely related to convergence. For example, Kam and Ullman [14] show that for certain types of data-flow analyses, convergence requires at most $d + 3$ iterations, where d is the largest number of back edges found in any cycle-free path of the CFG. Thus, it seems desirable to exploit knowledge of CFG structure when ordering the worklist, which is precisely what Hind and Pioli's algorithm does [10], although their algorithm does not distinguish different types of loops.

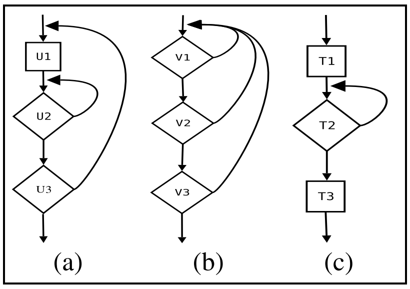


Fig. 8. Three loop examples: s simple loop, a nested loop, and a loop with multiple back-edges

To understand the complexities that arise from handling different types of loops, consider two types of loops. First, in a nested loop

(Fig.8(a)), which loop should we converge first? Second, in a loop with multiple back-edges (Fig.8(b)), which back edge should get priority, ie, after visiting V2 in the figure, should V1 be revisited before or after V3? After exploring many different heuristics, we evaluate a minor variant of our $\mathcal{DU}$ algorithm that ignores inner loops and uses a round-robin schedule for each loop. This algorithm, $\mathcal{DU}_{loop}$ does not try to converge an inner loop because the loop will be revisited when trying to converge the outer loop. The round-robin schedule ensures that all blocks in a loop are visited before any block is revisited.

In general, we believe that exploiting loop structure alone is not enough to yield significant improvement—we need to also account for data dependences in loops. Unfortunately, these dependences can be indirect. For example, in Fig.8(c) we have implicitly assumed that there is only a forward dependence from block T2 to T3. However, a backward, indirect dependence from block T3 to T2 can exist via a sequence of interprocedural def-use chains, so that a change in T3 could force T2 to be revisited. This phenomenon reduces the effectiveness of any techniques that try to exploit loop structures.

5 Experiments

5.1 Benchmarks and Metrics

Our experiments use 19 open source C programs (see Table 1), which—except for `sendmail`—were used in previous work [7]. In addition to measuring analysis time, we define metrics to evaluate the efficiency of worklist algorithms.

Table 1. Properties of the benchmarks. Lines of code (LOC) are given before preprocessing.

Program	Description	LOC	Procs	Stmts	CFG nodes	Call sites
stunnel 3.8	Secure TCP wrapper	2K	42	2,067	511	417
pfingerd 0.7.8	Finger daemon	5K	47	3,593	899	545
muh 2.05c	IRC proxy	5K	84	4,711	1,173	666
muh 2.05d	IRC proxy	5K	84	4,921	1,245	669
pure-ftpd 1.0.15	FTP server	13K	116	10,772	2,537	1,180
crond (fcron-2.9.3)	cron daemon	9K	100	11,252	2,426	1,249
apache 1.3.12 (core only)	Web server	30K	313	16,717	3,933	1,727
make 3.75	make	21K	167	18,787	4,629	1,855
BlackHole 1.0.9	E-mail filter	12K	71	20,227	4,910	2,850
openssh client 3.5p1	Secure shell client	38K	441	21,601	5,084	4,504
wu-ftpd 2.6.0	FTP server	21K	183	22,185	5,377	2,869
wu-ftpd 2.6.2	FTP server	22K	205	23,130	5,629	2,946
named (BIND 4.9.4)	DNS server	26K	210	23,405	5,741	2,194
privoxy 3.0.0	Web server proxy	27K	223	23,615	5,765	3,364
openssh daemon 3.5p1	Secure shell server	50K	601	28,877	6,993	5,415
cfengine 1.5.4	System admin tool	34K	421	38,232	10,201	6,235
sqlite 2.7.6	SQL database	36K	386	43,489	10,529	3,787
nn 6.5.6	News reader	36K	493	47,058	11,739	4,104
sendmail 8.11.6	Mail server	69K	416	67,773	15,153	7,573

1. *Basic block visitation*, or *BB-visit*, is the number of times blocks are retrieved from the worklists and analyzed.
2. *Basic block changes*, or *BB-change*, is the number of basic block visitations that update some data-flow information. *BB-change* is a measure of useful work.
3. *Efficiency*, $\mathcal{E}$, is the percentage of basic block visitations that are useful, i.e. the ratio *BB-change/BB-visit*.

5.2 Setup

We implement our worklist algorithms using the Broadway compiler system [8], which employs an interprocedural pointer analysis that computes points-to sets for all variables. The system supports flexible precision policies, such as fixed-modes context sensitive (CS) and insensitive modes (CI), and Client Driven (CD) mode [7]. CD allows a subset of procedures to be analyzed context sensitively, according to the needs of the client analysis. To handle context sensitivity correctly, the $\mathcal{DU}$ algorithm is modified to mark a block for re-analysis under specific contexts. Broadway also supports flexible heap models; in this paper we use one abstract heap object per allocation site in CI mode, and one object per allocation context in CS mode.

To evaluate our worklist algorithm, we need to choose a pointer analysis algorithm. Because the characteristics of the pointer analysis will affect the performance of our worklist algorithm, we present results for pointer analysis algorithms that represent two extreme points, CI and CS.

All experiments are performed on a 1.7GHz Pentium 4 with 2GB of main memory, running Linux 2.4.29. We compare our algorithms against a priority-queue worklist. This algorithm assigns a unique priority to each block in a CFG, and uses R_{reach}. Procedure exits always have lowest priority, so loops are always converged first. This algorithm is similar to that used by Hind and Pioli [10], except we don't use *IN/OUT* sets. When we tried using *IN/OUT* sets, the compiler ran out of memory for many of the larger benchmarks.

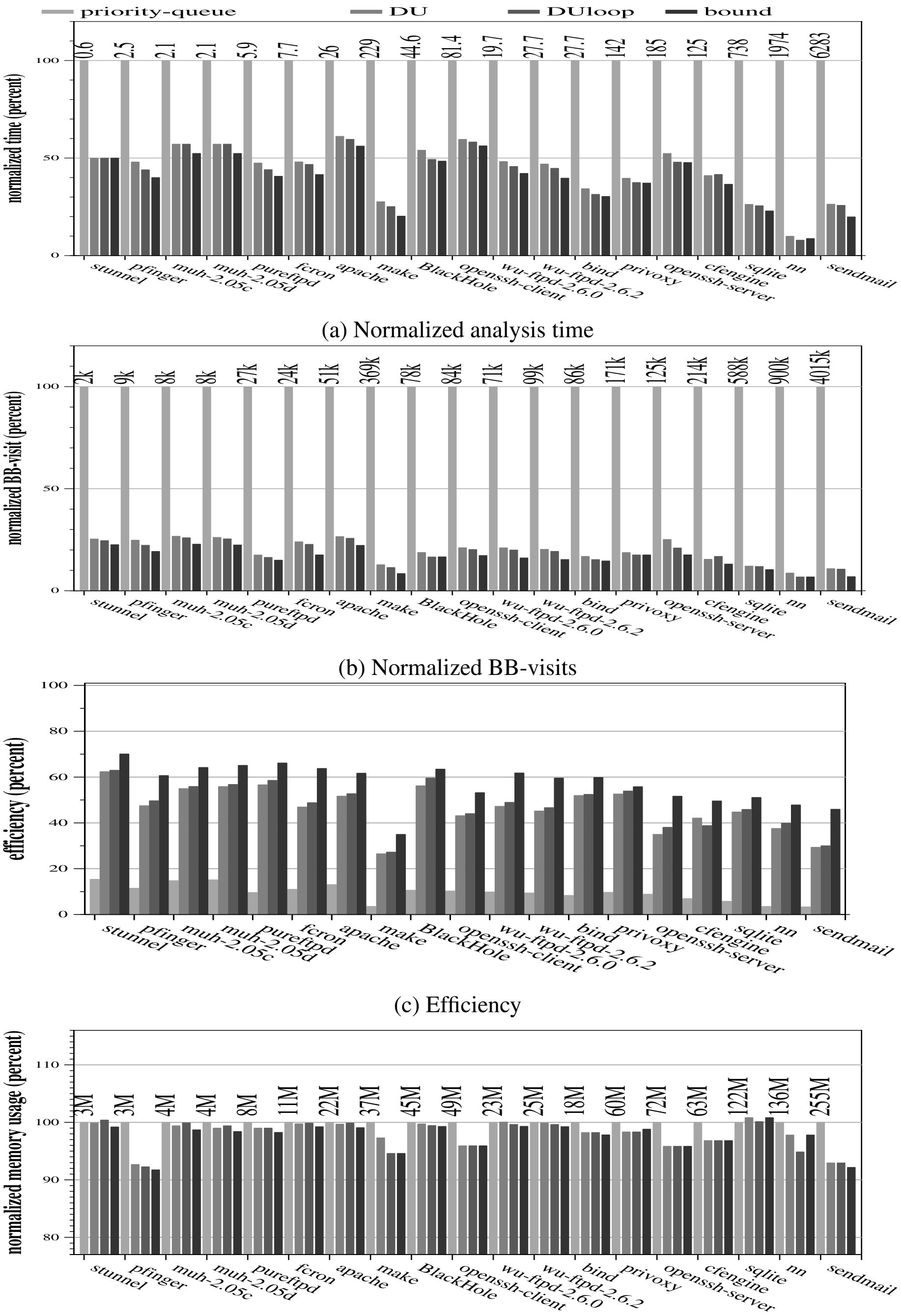
5.3 Empirical Lower Bound Analysis

To see how much room there is for further improvement, we empirically estimate a lower bound as follows. First, we execute $\mathcal{DU}$ to produce a trace of block visitations where data-flow information is updated, so the length of the trace is *BB-change*. We then re-execute the analysis, visiting blocks using the trace. In theory, this second execution should yield 100% efficiency. In practice we do not get 100% efficiency because, due to implementation details, the compiler has to visit additional blocks to ensure state consistency between useful visits. We measure this second execution to approximate a lower bound,[2] which on subsequent graphs is labeled as 'bound'.

5.4 Results

We first consider the behavior of our worklist algorithms in conjunction with CI pointer analysis. Each graph in Fig.9 shows the performance of $\mathcal{DU}$, $\mathcal{DU}_{loop}$ and our

[2] Note that a better ordering of the visits in the first execution may lead to an even lower bound.



(a) Normalized analysis time

(b) Normalized BB-visits

(c) Efficiency

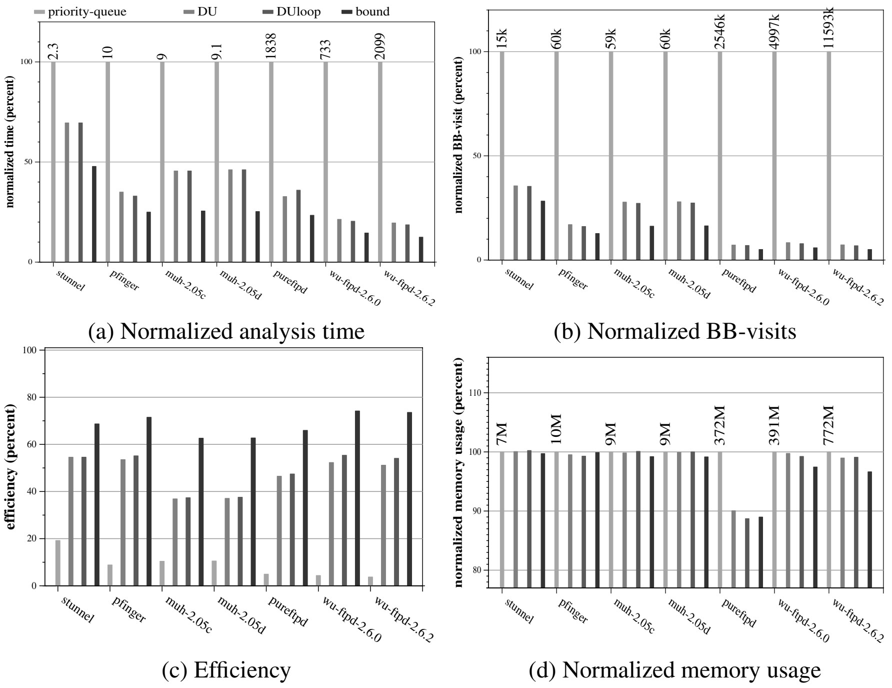
(d) Normalized memory usage

Fig. 9. Performance results of $\mathcal{DU}$ and its variant, on CI pointer analysis

empirical lower bound normalized against our baseline, which uses Hind and Pioli's priority queue. The benchmarks are listed in order of increasing size, so we see that $\mathcal{DU}$ significantly reduces analysis time, with an average reduction of 56%, and that larger benchmarks tend to benefit the most. For example, $\mathcal{DU}$ analyzes `sendmail` 74% faster than the baseline. We also see that $\mathcal{DU}_{loop}$ only improves upon $\mathcal{DU}$ by a few percentage points and that the main source of improvement is the increased efficiency. For example, for the large benchmarks, the efficiency of the baseline is just a few percent, but for $\mathcal{DU}$ it is in the 30-60% range. The cost of this reduced analysis time is a modest increase in memory usage. Finally, we see that there theoretically is still room for increased efficiency.

Fig.10 shows similar results for context sensitive pointer analysis. Results are only shown for benchmarks that complete under the baseline. The benefit of $\mathcal{DU}$ is larger for CS mode than CI mode because the number of large number of contexts exacerbates any inefficiencies in the worklist. For example, $\mathcal{DU}$ improves the analysis time of `wu-ftpd-2.6.2` by about 80%, while in CI mode its improvement is only about 53%. These results are encouraging, and currently we are extending our algorithm to Client Driven mode. We also see that the memory overhead of our algorithms increases under CS mode.

We have repeated our experiments with five different error-checking clients [7]. These are interprocedural analyses that generally yield better precision with



(a) Normalized analysis time

(b) Normalized BB-visits

(c) Efficiency

(d) Normalized memory usage

Fig. 10. Performance results of $\mathcal{DU}$ and its variant, on CS pointer analysis

flow-sensitivity. We run each client concurrently with the pointer analysis, and the results generally follow the same pattern as those in Figures 10 and 9, so we omit these to conserve space.

6 Conclusion

The ability to accurately analyze large programs is becoming increasingly important, particularly for software engineering problems such as error checking and program understanding, which often require high precision interprocedural analysis. This paper shows that by tuning the worklist, data-flow analysis can be made much more efficient without sacrificing precision.

We have implemented and evaluated a worklist algorithm that utilizes def-use chains. When compared with previous work, our $\mathcal{DU}$ algorithm shows substantial improvement, reducing analysis time for large programs by up to 90% for a context-insensitive analysis and by up to 80% for a context-sensitive analysis. The $\mathcal{DU}$ algorithm works well because it avoids a huge amount of unnecessary work, eliminating 65% to 90% of basic block visitations. We have also explored methods of exploiting CFG structure, and we have found that exploiting loop structure provides a small benefit for most of our benchmarks.

An empirical lower bound analysis reveals that there is room for further improvement. More study is required to determine whether some technique that considers both CFG structure and its interaction with data dependences can lead to further improvement.

Acknowledgments. We thank Ben Hardekopf and Kathryn McKinley for their valuable comments on early drafts of this paper. This work is supported by NSF grant ACI-0313263, DARPA Contract #F30602-97-1-0150, and an IBM Faculty Partnership Award.

References

1. A. V. Aho and J. D. Ullman. Node listings for reducible flow graphs. In *Proc. 7th Annual ACM Symp. on Theory of Computing*, pages 177–185, 1975.
2. Darren C. Atkinson and William G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'01)*, pages 52–61, November 2001.
3. J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.
4. J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66, 1991.
5. Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. The right algorithm at the right time: comparing data flow analysis algorithms for finite state verification. In *Int'l Conf. on Software Engineering*, pages 37–46, May 2001.
6. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *ACM SIGPLAN'02 Proc. 2002 PLDI*, pages 1–12, June 2002.

7. Samuel Z. Guyer and Calvin Lin. Client driven pointer analysis. In Radhia Cousot, editor, *10th Annual Int'l Static Analysis Symp. (SAS'03), volume 2694 of Lecture Notes on Computer Science*, pages 214–236, June 2003.
8. Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *2nd Conf. on Domain Specific Languages*, pages 39–53, October 1999.
9. Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2000)*, pages 113–123, August 2000.
10. Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analysis. In *5th Annual Int'l Static Analysis Symp. (SAS'98), volume 1503 of Lecture Notes on Computer Science*, pages 57–81, September 1998.
11. Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *ACM 3rd Symp. on the Foundations of Software Engineering*, pages 104–115, 1995.
12. Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *POPL*, pages 207–217, 1973.
13. K. W. Kennedy. Node listings applied to data flow analysis. In *Proc. 2th ACM Symp. on Principles of Programming Languages*, pages 10–21, 1975.
14. John B. Kam and Jeffrey D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of ACM*, 23(1):158–171, 1976.
15. Sungdo Moon et al. SYZYGY — a framework for scalable cross-module IPO. In *2004 Int'l Symp. on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, pages 65–74, March 2004.
16. Eugene M. Myers. A precise inter-procedural data flow algorithm. In *Proc. 8th ACM Symp. on Principles of Programming Languages*, pages 219–230, January 1981.
17. G. Ramalingam. On sparse evaluation representations. Research Report RC 21245(94831), IBM Research, July 1998.
18. 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", In *ACM Transactions on Programming Languages and Systems*, 23(1):105–186, March 2001.
19. Atanas Rountev, Barbara G. Ryder, and William A. Landi. Data-flow Analysis of Program Fragments. In *Proc. 7th Symposium on the Foundations of Software Engineering*, pages 235–253, September 1999.
20. Erik Ruf. Partitioning dataflow analyses using types. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 15–26, January 1997.
21. Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys (CSUR)*, 18(3):277–316, September 1986.
22. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3, 1995.
23. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
24. John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analyses Using Binary Decision Diagrams. In *ACM SIGPLAN'04 Proc. 2004 PLDI*, pages 131–144, June 2004.
25. Jianwen Zhu and Silvian Calman. Symbolic Pointer Analysis Revisited. In *ACM SIGPLAN'04 Proc. 2004 PLDI*, pages 145–157, June 2004.