Implementation Techniques for Efficient Data-Flow Analysis of Large Programs

Darren C. Atkinson
Dept. of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566 USA
atkinson@engr.scu.edu

William G. Griswold
Dept. of Computer Science and Engineering
University of California San Diego
La Jolla, CA 92093-0114 USA
wgg@cs.ucsd.edu

Abstract

Many software engineering tools such as program slicers must perform data-flow analysis in order to extract necessary information from the program source. These tools typically borrow much of their implementation from optimizing compilers. However, since these tools are expected to analyze programs in their entirety, rather than functions in isolation, the time and space performance of the data-flow analyses are of major concern. We present techniques that reduce the time and space required to perform data-flow analysis of large programs. We have used these techniques to implement an efficient program slicing tool for C programs and have computed slices of programs with more than 100,000 lines of code.

1. Introduction

1.1. Motivation

A variety of semantic tools have been proposed to assist the software engineer in understanding a system. Example tools include program slicing tools [3, 10, 18], invariant checkers [16], and static assertion checkers [12]. For example, a (forward) program slicer helps determine the effects of a proposed change by computing the set of statements that might be affected by the value of a given variable. By definition, these tools need to understand the semantics of the system in order to determine the possible values and states of program variables that are of interest to the software engineer. Consequently, most program understanding tools borrow insights, algorithms, and techniques from compiler technology and in particular from data-flow analysis [1].

However, the scope, extent, and context of program understanding tools differs greatly from those of a compiler. A compiler need only examine the functions contained within a single file and only then examine multiple functions simultaneously when advanced optimizations such as inlining are desired. In contrast, a program understanding tool needs to perform a global, interprocedural (whole-program) analysis of the system, for it is understanding the interprocedural flow of data that is most difficult for an engineer.

Unlike compilation, program understanding tasks are interactive, and an analysis such as slicing is often applied iteratively to answer a programmer's question about the program. For example, a programmer may need to perform several slices with different slicing criteria, incorporating the knowledge gained from previous slices (e.g., which functions were included in the slice). Thus, a whole-program analysis tool must perform analyses quickly in order to answer effectively many of the questions posed by programmers and designers.

These differences become apparent when whole-program analysis tools are used on large systems (i.e., systems with at least 10,000 lines and typically more than 100,000 lines of code). The running time and space required for many traditional interprocedural compiler algorithms may be prohibitive for a large program, especially in an interactive context such as software maintenance. For example, the size of the program dependence graph (PDG) [7], a common program data-flow representation, can be quadratic or greater in the size of the program (depending on the handling of pointers). The cost of constructing the PDG for a moderately sized program such as the 100,000-line GNU C compiler (GCC) will exceed the main memory and often the virtual memory of most desktop machines.

Unfortunately, large systems such as GCC are precisely the systems that could benefit the most from these automated program understanding tools. As a system increases in size, engineers struggle to maintain and improve the modularity of the system and the cohesiveness of the individual modules. In the end, they invariably fail because of the complexity of the task, and the structure of the system degrades until design decisions are no longer localized to a single module. Consequently, global modifications must now be made to incorporate a single change and the engineer requires global, rather than local, knowledge to successfully implement a desired change. The complexity of future changes may be exponential in the number of past changes made to the system [14]. Such complexity necessitates the use of automated tools to combat it. Finally, large systems are also often written in an aggressive programming style and use sophisticated language constructs such as function pointers. The use of these constructs is typically necessary to achieve good performance or to ease implementation. However, their use hinders program understanding. For example, function pointers are commonly used to implement efficient and simple dispatch tables. Large systems such as GCC and EMACS [17], as well as smaller systems like BURLAP [8], make extensive use of function pointers.

1.2. Prior solutions

One might argue that simply buying more memory, disk, and a faster processor could solve these problems, but this solution is not cost effective. The size of many modern systems is several times greater than that of GCC and is always growing. A project may also have many programmers requiring such resources to perform analyses.

Demand-driven techniques [2, 5, 11] have been proposed as a solution to the problems of prohibitive time and space. Unfortunately, these techniques cannot be applied in many cases. For example, pointer information is inherently global and cannot be computed easily on demand. In the C programming language [13], for example, all files must be analyzed to account for the use of pointers in initializers for static variables, regardless of whether a file contains a function that might be reachable during subsequent data-flow analysis. Data-flow analyses such as program slicing are not locally separable and therefore cannot be precomputed and stored efficiently.

2. Approach

Although demand-driven and memoization techniques can be adapted to work on large systems, the effort is often insufficient and difficult to implement. Consequently, a multifaceted approach that aims at improving performance at all levels of the analysis, both "macro" and "micro", is required. Indeed, small improvements at lower levels are often magnified by an order of magnitude or more when the total running time of the analysis is considered.

Examining well-known, lower-level techniques such as data-flow analysis, we find that subtle adjustments can significantly improve performance of whole-program analysis tools. For example, by factoring the data-flow sets according to storage class, significant space can be saved, and thereby significant time can be saved by avoiding use of the slower portions of the virtual memory hierarchy. As a case in point, GCC could not be analyzed without such factorization because the data-flow sets alone would require over 10 GB of space. In this paper, we examine several implementation techniques designed to improve performance without unduly increasing the engineering complexity of the data-flow analysis:

1. Factoring of the data-flow sets themselves to match the structure of the analysis can reduce time and space and improve the understandability of the data-flow algorithm itself.

2. The visitation order of statements during analysis can significantly impact the running time of the analysis. In particular, a "for each basic block" approach to traversing the control-flow graph (CFG) is generally faster than using a worklist. In addition, we have developed a hybridized iterative-worklist algorithm that processes fewer blocks than the traditional algorithm.

3. Selective reclamation of the data-flow sets during the analysis can dramatically save space without disturbing its correctness or running time. This technique is dependent only on the properties of the control-flow graph and not on the particular data-flow analysis being performed, making it suitable for use in many tools.

We have used our techniques to implement a slicing tool for C programs called Sprite, which is part of the ICARIA and PONDER [9] packages.[1] We evaluated our techniques by using Sprite to compute backward program slices of three large programs. Program slicing was chosen as the data-flow analysis because it is nontrivial and widely known. It has become an archetype of program analysis for software engineering. Because of factorization, slicing large programs is now possible. Using our other techniques, program slices of large programs such as GCC can be performed orders of magnitude faster and require 2–3 times less space. In the following sections, we present a brief background on the data-flow analysis necessary for backward program slicing, describe each of our implementation techniques in detail, present our results, and conclude with ideas for future work.

[1]Both of these packages are available for download via the Internet at `http://www.cse.scu.edu/~atkinson`.

`return from f()`

$$P_{exit} = P_{exit} \cup P_i \quad (1a)$$
$$D_{exit} = D_{exit} \cup (D_i \cap S) \quad (1b)$$
$$N_{exit} = N_{exit} \cup N_i \cup (D_i - S) \quad (1c)$$

`call to f()`

$$P_i = P_i \cup P_{entry} \quad (2a)$$
$$D_i = (D_i - S) \cup (D_{entry} \cap S) \quad (2b)$$

`x := y`

```
if $x \in P_i$ then
    $D_i = D_i \cup \{y\}$                        (3a)
else if $x \in D_i$ then
    $D_i = D_i - \{x\} \cup \{y\}$                (3b)
```

`x := *p`

```
if $x \in P_i$ then
    $P_i = P_i \cup (*p - S)$                     (4a)
    $D_i = D_i \cup (*p \cap S) \cup \{p\}$       (4b)
else if $x \in D_i$ then
    $P_i = P_i \cup (*p - S)$                     (4c)
    $D_i = D_i - \{x\} \cup (*p \cap S) \cup \{p\}$   (4d)
```

`*p := x`

```
if $*p \cap (D_i \cup P_i \cup N_i) \neq \phi$ then
    $D_i = D_i \cup \{p, x\}$                     (5)
```

N = nonlocal local (automatic) variables of interest
D = variables of interest with killing definitions
P = local variables of interest with preserving definitions
S = all global (static) variables
$*p$ = points-to set of variable p

Figure 1. Data-flow equations for slicing in the presence of recursion and pointers to local variables. Sets are subscripted with the program point to which they refer. Sets that are not subscripted are the same for all program points. The current statement has program point *i*. Unless otherwise noted, a set passes through a program point unchanged.

3. Background

In backward program slicing, a data-flow set represents the set of variables of interest (i.e., those variables for which we wish to find definitions). At each program point, D represents a data-flow set. At each assignment statement, some set of variables, *defs*, are defined and another set of variables, *uses*, are used. If some variable in *defs* is also in D (i.e., of interest), then the killing definitions of *defs* are removed from D and *uses* are added to D (and the statement is included in the slice). Otherwise, D remains unchanged.

In the absence of either recursion or pointers to local variables, interprocedural slicing is simple and well-understood. However, for a language such as C that provides both of these features, traditional data-flow analyses may yield unsafe results. To overcome these problems, our analysis treats definitions of local variables that are referenced through a pointer as preserving definitions [3]. Equations for the data-flow analysis are shown in Figure 1. Such local variables are "demoted" and a separate data-flow set is required to maintain them. For example, a simple assignment statement is handled using two cases, Equations 3a and 3b. The former handles the case in which `x` has been demoted (because it was referenced previously through a pointer), and the latter handles the more traditional case.

Furthermore, an additional set, N, is required to model the transitive closure of the program stack, but only for local variables of interest. This set must be checked upon an assignment through a pointer deference (Equation 5), since the pointer may refer to a local variable that is currently out of scope but is on the stack. Therefore, three data-flow sets per statement (or basic block) are required. Finally, the S set contains all static (global) variables in the program and is used to partition a set into its local and global variables.[2] This set is used to remove local variables from the D set across function calls and returns.

4. Data-flow set factorization

If implemented naively, our data-flow equations require three data-flow sets per basic block (i.e., there is one D, P, and N set per block). Given that GCC has 238,000 symbols and 120,000 blocks, a bit-set implementation of data-flow sets would require over 10 GB (238,000 *symbols* $\times$ 120,000 *blocks* $\times$ 3 *sets / block* $\times$ 1 *bit / symbol* $\div$ 8 *bits / byte*) of space. A bit-set is implemented by consecutively mapping the elements of the input set onto the natural numbers. Each number represents the bit position in a bit-vector, which is only long enough to contain its highest-numbered bit. An element is a member of the set if and only if its corresponding bit in the bit-vector is set. A bit-vector representation allows set operations such as union and intersection to be implemented efficiently using logical bit-wise operations. However, such operations can only be performed across bit-sets with identical mappings (bit-numberings).

Since N does not change while slicing a function, a single N set can be used for all blocks of the function. Using

[2]We use the term *local variable* to mean an *automatic variable* in C. Similarly, the term *global variable* should be read as *static variable*. Since C overloads the use of the "static" keyword, we use the terms local and global variable instead.

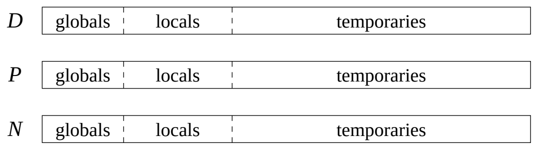

Figure 2. A simple implementation of the data-flow sets for our equations. The space required is approximately 6.7 GB for GCC even if the N set is flow-insensitive.

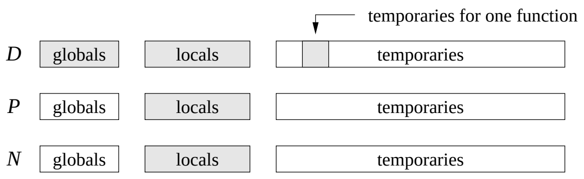

Figure 3. A better implementation of the data-flow sets for our data-flow equations. The sets are partitioned into their three distinct classes. Only the shaded areas of a set are actually used at any given time.

this implementation, the space requirements for GCC are reduced to approximately 6.7 GB (10 GB $\times \frac{2}{3}$), still an unacceptable amount of space. This simple implementation is shown in Figure 2.

An analysis of the bit-sets revealed that they are typically very sparse. Examining our equations, we see that P and N contain only local variables, while D contains local variables, global variables, and generated temporaries. Also, temporaries cannot usually be the target of a pointer and therefore cannot be referenced out of scope. There are a few cases where temporaries can be the target of a pointer, such as when a structure is returned from a function. For these cases, we introduce a new type of temporary variable called a *special*, which is treated as a local variable. The introduction of specials allows us to treat the vast majority of temporaries as though they could not be the target of a pointer. We therefore decided to partition the bit-sets into three distinct classes: global variables, local variables, and temporaries for each function, as shown in Figure 3. The D set now consists of three bit-sets, but requires space to store only the global variables, local variables, and the *maximum* number of temporaries *per function*. If we assume for simplicity that the 220,000 temporaries are evenly distributed among GCC's 2,300 functions, the space requirements are reduced to approximately 60 MB, which is acceptable.

The set partitioning also improves algorithmic performance and eases implementation. The data-flow equations of Figure 1 require that the D and P sets be partitioned into their local and global variables components. Logically, this partitioning is done using set intersections and differences. With these components maintained as separate sets, the partitioning operations are trivial. For example, rather than computing $D \cap S$ to retrieve the global variables of D, only the set of global variables of D (symbolically `D.globals`) need be retrieved, thereby changing an $O(n)$ operation into an $O(1)$ operation.

Examining the P and N sets in even greater detail, we see that they can contain only local variables that are pointed to by some pointer variable. Since there are few of these *target locals*, the locals of the P and N sets can be further partitioned into two classes to save space, as shown in Figure 4. However, examining our data-flow equations, we see that the local variables of the P and N sets must be operated on in conjunction with those of the D sets (Equations 1c and 5 of Figure 1). Therefore, *explicitly* partitioning the local variables into two classes would complicate these operations. Such an implementation of the data-flow equations would be complicated by the need to combine the sets of target and nontarget locals into one set for any operation involving the locals from the D set. Given that all data-flow sets are implemented using bit-sets, this process could be complicated if the various sets have different bit-numberings.

Rather than partitioning the local variables into two distinct sets, we elected to keep them as one set. However, since the points-to analysis must be performed prior to data-flow analysis, we know which local variables are target locals and which are nontarget locals. We can therefore easily ensure that the target locals are assigned lower bit-numbers than the nontarget locals. This numbering ensures that the target locals are "packed" at the start of the bit-sets. Since a bit-vector is only long enough to hold its highest-numbered bit, this implementation gives us the space savings we desire without the implementation complexities of splitting the locals into two bit-sets. Because programs often contain few target locals, the space allocated for the P and N sets is negligible as a result of our aggressive implementation.

Finally, although this discussion has focused on factorization of the data-flow sets for backward program slicing, the principle of factorization applies to other data-flow analyses as well. For example, live-variable analysis, which is used by optimizing compilers to determine the lifetime of variables, has data-flow equations very similar to those shown in Figure 1. Only the conditional checks of the D set in the else clauses need be eliminated. Alternative implementations of our equations are also possible. For example, reusing temporaries across functions or within a function would dramatically decrease the number of temporaries and would therefore change the data-flow set implementation. However, reuse of temporaries would also increase the size

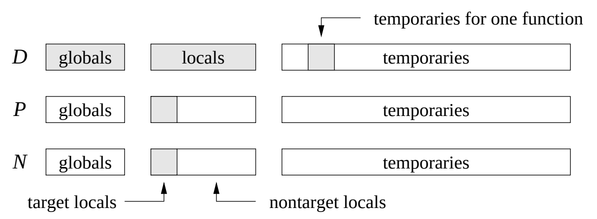

Figure 4. Our final implementation of the data-flow sets for our data-flow equations. The local variables of the P and N sets are further separated into target and nontarget locals. The slicing tool assigns the target locals lower bit-numbers than the nontarget locals to ensure "packing" of the bit-sets.

```
changed := true

while changed do
    changed := false
    for each block B do
        old := out [B]
        process (B)
        if old ≠ out [B] then
            changed := true
        end if
    end for
end while
```

(a)

```
worklist := {start}

while worklist ≠ ϕ do
    worklist := worklist - {B}
    old := out [B]
    process (B)
    if old ≠ out [B] then
        for P in pred [B] do
            worklist := worklist ∪ {P}
        end for
    end if
end while
```

(b)

Figure 5. Example pseudocode for visitation algorithms: (a) the iterative search algorithm, and (b) the worklist algorithm. The notation *out* [*B*] refers to the output data-flow set of *B*.

of the points-to sets since temporaries are often used to hold the results of pointer arithmetic.

5. Block visitation order

5.1. Traditional algorithms

To perform data-flow analysis (e.g., reaching definitions, program slicing, live-variable analysis), a compiler or program understanding tool propagates the computed data-flow information along the edges of the constructed CFG. The data-flow facts along the incoming edges of a node are combined into a single set that is then transformed according to the data-flow properties of the node. The resulting set is then propagated along all output edges of the node. If the CFG is reducible (e.g., the program does not contain any unstructured jump statements) and the data-flow analysis simple enough (i.e., locally separable), then the data-flow information can typically be propagated fully in a single pass over the CFG [1]. Otherwise, an iterative algorithm must be used that propagates the data-flow information until no further changes to the data-flow sets occur.

Since the C language allows unstructured control-flow and slicing is a nontrivial analysis, an iterative algorithm is required for data-flow analysis. The visitation order of the nodes does not affect the correctness of the algorithm, so long as the data-flow information is fully propagated along all edges until no more changes occur to the data-flow sets. However, the visitation order can greatly impact the performance of a specific data-flow algorithm. Two common visitation algorithms are used to perform data-flow analysis, as shown in Figure 5.

Iterative search algorithm: In the iterative search (i.e., "for each basic block") algorithm (Figure 5a), each block (i.e., CFG node) is visited once. If any changes have occurred, then each block is visited once again. This process repeats until no further changes occur to the data-flow sets. Typically, a depth-first or breadth-first search of the CFG is used to visit all blocks exactly once in an iteration, with depth-first search usually resulting in fewer iterations [1].

Worklist algorithm: In the worklist algorithm (Figure 5b), the blocks (i.e., nodes) to be visited are placed on a worklist, which is typically implemented using a stack or queue. A block is removed from the worklist and visited. If any changes occur to the data-flow sets of the block, then all predecessors of the block (successors for a forward data-flow analysis) are placed on the worklist. The algorithm repeats until the worklist is empty.

Figure 6 shows a program fragment and its associated CFG, annotated with block numbers. Consider starting a backward data-flow analysis at the return statement located at block *B6*. An example visitation order for *one iteration* of the iterative search algorithm would be *B6*, *B2*, *B1*, *B5*, *B4*, and *B3*. For the worklist algorithm, a *complete visitation* order might be *B6*, *B2*, *B5*, *B4*, *B3*, *B2*, *B5*, and *B1*. Unlike the iterative search algorithm, the worklist algorithm can visit a block many times before visiting other blocks. For example, blocks *B2* and *B5* are visited twice before block *B1* is ever visited.

The worklist algorithm tends to propagate or to "push" the changed data-flow information immediately to those blocks that require that changed information. In contrast, the iterative search algorithm waits until the next iteration over *the entire program* to push the information. Therefore, we would expect the worklist algorithm to require less block visits and consequently require less time than the iterative search algorithm.

```
i = 0;

while (i < n) {
    if (sum > 20)
        j = j + 1;

    sum = sum + i;
    i = i + 1;
}

return sum;
```

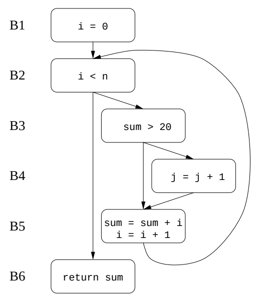

Figure 6. A program fragment and its annotated CFG.

However, the eagerness of the worklist algorithm may in fact yield poorer performance for interprocedural analyses. If the analysis is not fully context-sensitive [6], then the nodes of a function are shared among the different calls to the function. The worklist algorithm will converge the function for each call (depending upon its implementation) resulting in the nodes of a function being processed multiple times and an increased running time. Although the worklist algorithm may be a good choice for context-sensitive analyses, for large programs context-sensitive analyses are not generally practical, which makes the worklist algorithm a poor choice in our implementation in which each function has its own worklist.[3]

5.2. The hybrid search algorithm

The iterative search algorithm computes data-flow information globally on each iteration. The global nature of the algorithm makes it a good choice when the data-flow information is changing over a large portion of the program. However, on each iteration the entire program is analyzed even if the data-flow changes from the previous iteration are localized. In contrast, the worklist algorithm computes the data-flow information locally and does not process a node unless required.

Ideally, we could like the iterative search algorithm to process only those nodes that are required to be processed on each iteration, rather than processing all nodes. To do so, we need to determine the necessary condition for processing a node. Fortunately, the worklist algorithm provides such a condition. If the output set of a node has changed, then the predecessors of that node need to be processed. We can therefore easily modify the iterative search algorithm so that a node is processed only when necessary. Figure 7 shows

```
function dfs (B)
  visited := visited ∪ {B}

  if B ∈ pending then
    old := out [B]
    process (B)
    pending := pending - {B}

    if old ≠ out [B] then
      changed := true
      for P in pred [B] do
        pending := pending ∪ {P}
      end for
    end if
  end if

  for P in pred [B] do
    if P ∉ visited then
      dfs (P)
    end if
  end for
end function
```

Figure 7. Example pseudocode for the hybrid search algorithm. A block is processed only if it is pending.

the resulting hybrid search algorithm implemented using a depth-first search.

In the hybrid search algorithm, a node is *processed* (i.e., the data-flow information computed) only if the data-flow information of any of its successors (predecessors in a forward analysis) has changed, as indicated by the *pending* set.[4] However, each node is still *visited* exactly once on each iteration. Therefore, the hybrid algorithm retains the "fairness" of the original iterative search algorithm. That is, a block is not processed again before any other pending block is processed eliminating the problem of multiple convergence in the worklist algorithm. Although still requiring the same number of iterations and block visits as the iterative search algorithm, the hybrid search algorithm requires that less blocks be processed. Since processing the blocks (i.e., processing the statements within the blocks) consumes the majority of the computation time, the hybrid algorithm runs faster than the traditional algorithm.

6. Data-flow set reclamation

The iterative search algorithm guarantees that each block in the program is visited exactly once before any block is revisited. Recall that the input data-flow set of a block is the confluence of the data-flow sets on all incoming edges. In

[3]Such an implementation matches the natural call-and-return structure of the program and eases implementation. One alternative implementation is to have a single worklist for all nodes, regardless of function.

[4]In practice, it may be necessary always to process certain nodes such as function calls or entry and exit nodes, or to process all nodes on the initial iteration.

a backward data-flow analysis, once all predecessors (successors for a forward analysis) of a block have been visited, then the data-flow set for the block itself will no longer be needed by other blocks. Consequently, the data-flow set can be deallocated.

However, the data-flow sets are typically used to determine whether the data-flow analysis should continue. For example, in Figure 5a, the output set is used to determine if another iteration of the iterative search algorithm is necessary. We would therefore expect that we would be required to save the data-flow set. However, most data-flow analyses are monotonic: the size of the data-flow sets is either monotonically nondecreasing or nonincreasing. For example, program slicing and reaching definitions are nondecreasing analyses (the data-flow sets never decrease in size); computation of available expressions is a nonincreasing analysis. Therefore, it is sufficient to compare the *size* of the data-flow sets rather than the actual sets themselves. The size of a data-flow set can be preserved even if the set is reclaimed.

If a depth-first search is used to visit all blocks, then the maximum number of data-flow sets that need to be allocated at any one time is proportional to the width of the CFG, resulting in substantial savings in space.[5] By saving space, the time required to perform the analysis will also be reduced by avoiding the slower portions of the virtual memory hierarchy.

Iterative search: In our first implementation of this reclamation approach, we found that it was cumbersome to keep a reference count on each block to keep track of the number of its predecessors that had been visited. Consequently, we chose a simpler implementation without reference counts: once the depth-first search of a called function is complete and all blocks have been visited, the data-flow set of each block (other than the entry and exit blocks) is deallocated if all predecessors of the block were visited after the block itself was visited. Therefore, data-flow analysis with reclamation can easily be done as a two-step process, in which the first step visits each block and also stores the visitation order (Figure 8a), and the second step reclaims the data-flow sets of those blocks that meet the given criteria (Figure 8b). Slightly more data-flow sets remain active at any time than are minimally needed, but the implementation is much simpler since reference counting is not needed.

Hybrid search: Performing reclamation with the hybrid search algorithm is slightly more difficult. Reclamation is based on the knowledge of which blocks will be *processed* before other blocks. In the iterative search algorithm, all blocks are processed on each iteration as they are visited.

[5]Although worst-case flowgraphs (e.g., grids) exist in which reclamation is impossible, they do not occur in practice.

```
function dfs (B)
  ordered := ordered · [B]
  visited := visited ∪ {B}
  process (B)

  if size [B] ≠ |out [B]| then
    size [B] := |out [B]|
    changed := true
  end if

  for P in pred [B] do
    if P ∉ visited then
      dfs (P)
    end if
  end for
end function
```

(a)

```
for B in ordered do
  before := before ∪ {B}
  if B ≠ entry and B ≠ exit then
    reclaimable := true

    for P in pred [B] do
      if P ∈ before then
        reclaimable := false
      end if
    end for

    if reclaimable then
      delete (out [B])
    end if
  end if
end for
```

(b)

Figure 8. Example of data-flow set reclamation: (a) a depth-first search algorithm that also stores the visitation order, and (b) a reclamation algorithm that uses the visitation order to safely reclaim unneeded data-flow sets. The notation $a \cdot [x]$ indicates the concatenation of the list a with the single-element list containing x.

The hybrid search algorithm reduces the number of blocks that need to be processed during an iteration. Therefore, we would expect that the number of blocks that can be safely reclaimed must also be reduced. A block can be reclaimed during the hybrid search algorithm if, in addition to being reclaimable during the traditional iterative search algorithm, the block will be processed on the next iteration. Requiring that the block be processed on the next iteration ensures that the data-flow set will be available (i.e., computed) when needed.

Worklist: Whereas the iterative and hybrid algorithms are driven strictly by the control-flow properties of the program and have the inherent property that all blocks will be visited exactly once during each iteration, the worklist algorithm is driven more by the data-flow properties of the program. Therefore, it can be difficult to predict when a data-flow set will no longer be needed. In order to safely reclaim the data-flow set of a given block, we need to ensure that the block will always be visited (and processed) before any of its predecessors. This would guarantee that the data-flow set of the block would always be available when it is required by a predecessor. Therefore, we require that a block (reverse) dominate its predecessors in order to be reclaimed. Although not optimal (i.e., some data-flow sets will not be reclaimed when in fact they can be), this solution is simple, and efficient algorithms for computing dominators in a flowgraph are widely available [15].

	Lines of code		CFG	
	Before CPP	After CPP	Time	Space
WC	2,692	6,756	1.0	1.3
DIFF	9,836	21,844	2.1	4.2
BURLAP	40,363	112,301	11.1	22.2
EMACS	111,714	244,328	30.2	47.1
GCC	189,043	244,723	36.5	69.0

Table 1. Statistics for the programs used in the experiments. Time is given in seconds and space in megabytes. All blank lines of code have been removed. The measurements for the CFG include the symbol table, pointer information, and other necessary data structures.

As discussed, the algorithm for data-flow set reclamation is based almost entirely on the visitation order of nodes in the graph. The only requirement on the data-flow analysis itself is that it be monotonic. Therefore, reclamation should be possible for any monotonic data-flow algorithm (e.g., reaching definitions) that traverses the CFG in order to compute a maximum fixed point solution.

7. Results

We expect that reclamation of the data-flow sets would result in a significant reduction in the amount of space required to perform an analysis. We also expect that the hybrid search algorithm would be faster than the traditional iterative search algorithm and that the hybrid algorithm would therefore be the algorithm of choice. The worklist algorithm would most likely not be practical on large programs using our implementation, as discussed in Section 5.

To validate our hypotheses, we implemented a slicing tool for C programs called Sprite based on our ideas. We chose five readily available, mature programs of varying sizes and complexities, whose sizes are given in Table 1. In particular, WC refers to the `wc` program from the GNU `textutils` package, version 2.0; DIFF refers to the `diff` program from the GNU `diffutils` package, version 2.0; BURLAP refers to the `burlap` program from the FElt finite element analysis system, version 3.05; EMACS refers to the `temacs` program of the GNU Emacs editor, version 20.7; GCC refers to the `cc1` program of the GNU C compiler, version 2.7.2 for Solaris 2.8.

Sprite correctly handles functions with a variable number of arguments and the effects due to library functions. Library functions are handled by providing a skeleton for each function that correctly summarizes its effects. Signal handlers and the `longjmp` and `setjmp` functions are not handled. All experiments were performed on an idle 440 MHz Sun UltraSparc 10 running Solaris 2.8 with 256 MB of physical memory and 1.6 GB of swap space. All slices were computed using data-flow set factorization and with strong prototype filtering enabled. Section 4 presents statistics supporting the use of factorization. Although Sprite supports computing context-sensitive slices, all slices performed were context-insensitive for maximum performance and since context-sensitivity has not been found to yield substantially better slices [2, 4].

Table 2 presents statistics for performing several slices of our test programs. The slicing criteria are based on those used in other papers [3, 4] and to yield a variety of different sized slices. For all slices, we tried to choose variables that might be selected by a programmer during debugging. As expected, flow-set reclamation reduces the amount of space required to perform the data-flow analysis. On average, reclamation reduces the space by 40%. The averages for the iterative, hybrid, and worklist algorithms are approximately 60%, 17%, and 30%, respectively. The time overhead due to reclamation is small for the iterative and hybrid searches, but can be much greater for the worklist algorithm, probably because of the need to compute and store dominator information. For example, the worklist algorithm runs approximately 33% slower using reclamation for Slice 5. However, the space savings can result in substantial net time savings of an order of magnitude or more for large slices. For example, using reclamation, Slice 7 is computed sixteen times faster using the iterative search algorithm because the data-flow analysis now fits in main memory, eliminating paging overhead. Unfortunately, because fewer blocks can be reclaimed using the hybrid search algorithm, the same analysis performed using the hybrid algorithm does not fit in main memory. As a result, the iterative search algorithm with reclamation outperforms the hybrid search algorithm with reclamation for this slice. In fact, the hybrid algorithm actually runs much slower with reclamation. We suspect the additional memory references used in determining if a block should be processed result in a greater number of page faults. Even Slice 6, which is much smaller in size, is computed faster with the iterative search algorithm with reclamation, although the hybrid algorithm does not perform as poorly in this case.

The hybrid algorithm results in approximately 20% fewer block visits than the traditional iterative algorithm. Unfortunately, on large slices where the savings would be most beneficial, the smaller number of reclaimable blocks is a serious disadvantage. Finally, the worklist algorithm performs extremely poorly on large slices and is therefore not suitable except on small programs. The (possibly) nested convergence of the algorithm results in a substantially increased running time and number of block visits. For example, the worklist algorithm requires nine times as many block visits as the iterative search algorithm for Slice 7. In-

	Program and	Slice size	Reclaim	Iterative Search			Hybrid Search			Worklist		
	Slicing criterion		sets?	Blocks	Time	Space	Blocks	Time	Space	Blocks	Time	Space
1.	WC	437 lines	no	3,576	0.22	0.34	2,963	0.19	0.34	3,736	0.21	0.31
	`wc.c:364:total_lines`		yes	3,576	0.27	0.25	2,963	0.22	0.30	3,736	0.33	0.29
2.	DIFF	1,976 lines	no	31,740	2.17	2.54	25,199	1.82	2.54	52,326	3.46	2.45
	`diff.c:1071:val`		yes	31,740	2.68	1.55	25,199	2.09	2.29	52,326	4.93	1.92
3.	GCC	2,617 lines	no	30,821	11.13	8.33	24,285	10.36	8.33	28,576	17.76	8.12
	`unroll.c:3085:const0_rtx`		yes	30,821	11.18	2.88	24,285	10.40	6.87	28,576	19.33	5.31
4.	BURLAP	2,709 lines	no	43,038	6.48	6.04	31,301	5.58	6.02	185,451	18.33	5.92
	`matrixfunc.c:767:status`		yes	43,038	6.70	2.94	31,301	6.13	5.11	185,451	23.93	4.40
5.	BURLAP	12,336 lines	no	162,070	34.83	23.78	135,729	32.47	23.78	629,063	67.96	22.80
	`apply.c:243:result`		yes	162,070	30.71	10.40	135,729	36.75	18.85	629,063	90.12	14.73
6.	EMACS	34,386 lines	no	661,640	1793.68	208.42	564,149	1703.42	208.42	7,657,037	2596.02	188.71
	`alloc.c:1936:gc_cons_threshold`		yes	661,640	347.39	64.69	564,149	815.66	163.37	7,657,037	1886.61	115.89
7.	GCC	57,004 lines	no	1,155,077	10103.60	316.04	870.448	6657.77	316.04	9,438,892	10625.77	299.40
	`sched.c:4964:reg_n_calls_crossed`		yes	1,155,077	622.50	84.70	870,448	8679.52	253.68	9,438,892	14933.10	181.25

Table 2. Statistics for various slices of the example programs using the different algorithms. The number of blocks given refers to the number of blocks processed. Time is given in seconds and space in megabytes, both of which are for the computation of the slice itself. Build time for the CFG is given in Table 1.

terestingly, the time for the slice does not increase substantially, which we believe may be due to the worklist algorithm exhibiting better reference locality since it converges locally, rather than globally.

An unexpected result, not shown in Table 2, is that the worklist algorithm computed slightly more precise (i.e., smaller) slices in six of the seven slices shown. (The slice size reported in Table 2 is for the iterative and hybrid algorithms, which always compute identical slices.) On average, the worklist algorithm computed slices that contained 1% fewer lines. We attribute this decrease to the fact that the worklist algorithm converges locally, rather than globally, and hence can be slightly more precise with regard to interprocedural calling context.

For example, consider a CFG in which two functions, `A()` and `B()`, call function `C()`. During analysis, `A()` will place data-flow information at the exit statement of `C()` and will collect information from the entry statement of `C()`. Later, `B()` will perform a similar set of operations and, in the absence of context-sensitivity, will therefore collect information relevant from not only its own call to `C()` but from `A()`'s call to `C()` as well. On the next iteration of the iterative search algorithm, `A()` will collect information relevant to its own call to `C()`, but also from `B()`'s call to `C()` on the *previous iteration*. However, using the worklist algorithm, the data-flow sets of `C()` are fully converged upon `A()`'s initial call. Consequently, `A()` will not need to collect information from `C()` on the "next iteration" (in fact, the worklist algorithm has no explicit iterations) and will therefore not see the effects of `B()`'s call to `C()`. Therefore, an increase in precision may result.

8. Conclusions and future work

Although a variety of semantic tools have been proposed to aid the software engineer in understanding a system, few are practical to use on large systems, especially in an interactive context such as software maintenance. Demand-driven techniques solve some, but not all, of the efficiency concerns and can be difficult to implement, possibly requiring too many changes to the tool to be worthwhile.

We proposed modifications to the underlying data-flow analyses that are easy to implement and can yield a substantial improvement in performance. In particular, reclamation of the data-flow sets during data-flow analysis and factorization of the data-flow sets themselves can result in substantial savings in space and therefore also result in substantial savings in time by avoiding the slower portions of the memory hierarchy.

To validate our hypotheses, we constructed a program slicing tool for C programs and performed several slices of well-known programs. Factorization of the data-flow sets makes it possible to compute slices of large programs. Our results indicate that flow-set reclamation can yield a significant reduction in the space required to perform the data-flow analysis. As a result, many analyses now can be contained in main memory, thus avoiding use of the virtual memory system. Consequently, these analyses run an order of magnitude or more faster than analyses ran without flow-set reclamation.

We also presented a new algorithm for visiting the nodes of a flowgraph, called the hybrid search algorithm. Although the hybrid algorithm results in 20% fewer block visits, much less reclamation can be performed during the analysis. Consequently, the traditional iterative algorithm with reclamation is a better choice than the hybrid algorithm with reclamation for large programs. More investigation is needed to determine why the worklist algorithm performed poorly. One possible area would be to change the implementation of the worklist algorithm so a single worklist over all nodes was used, rather than one worklist per function, thereby possibly eliminating the nested convergence behavior.

An interesting direction for future work would be for the slicing tool to heuristically determine the best algorithm to use. The tool could use the size (e.g., number of symbols and functions) and structure (e.g., nesting depth of function calls) of a program in such a determination. Extending this idea, the slicing tool could dynamically switch algorithms based on the space characteristics of the machine and the analysis. For example, the tool could initially use the hybrid search algorithm in order to visit fewer blocks, but could switch to the traditional iterative algorithm if too much space is required in order to reclaim more data-flow sets. Given the different natures of the search and worklist algorithms, switching between them would most likely be problematic however.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany, Mar. 1996.

[3] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the 6th ACM International Symposium on the Foundations of Software Engineering*, pages 46–55, Lake Buena Vista, FL, Nov. 1998.

[4] L. Bent, D. C. Atkinson, and W. G. Griswold. A comparative study of two whole programs slicers for C. Computer Science Technical Report CS2000-0643, University of California, San Diego, Department of Computer Science & Engineering, 2000.

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, CA, Jan. 1995.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM '94 SIGPLAN ACM Conference on Programming Language Design and Implementation*, pages 20–24, Orlando, FL, June 1994.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[8] J. I. Gobat and D. C. Atkinson. The FElt system: User's guide and reference manual. Computer Science Technical Report CS94-376, University of California, San Diego, Department of Computer Science & Engineering, 1994.

[9] W. G. Griswold and D. C. Atkinson. A syntax-directed tool for program understanding and transformation. In *Proceedings of the 4th Systems Reengineering Technology Workshop*, pages 274–282, Monterey, CA, Feb. 1994.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[11] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 104–115, Washington, DC, Oct. 1995.

[12] D. Jackson. ASPECT: An economical bug-detector. In *Proceedings of the 13th International Conference on Software Engineering*, pages 13–22, Austin, TX, May 1991.

[13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[14] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press, Orlando, FL, 1985.

[15] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Prog. Lang. Syst.*, 1(1):121–141, July 1979.

[16] G. N. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*, pages 93–105, San Francisco, CA, Nov. 1996.

[17] R. M. Stallman. *GNU EMACS Manual*. Free Software Foundation, Cambridge, MA, 1993.

[18] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4):352–357, July 1984.