The Design of Whole-Program Analysis Tools*

Darren C. Atkinson and William G. Griswold
Department of Computer Science & Engineering, 0114
University of California, San Diego
San Diego, CA 92093-0114 USA
{atkinson,wgg}@cs.ucsd.edu

Abstract

Building efficient tools for understanding large software systems is difficult. Many existing program understanding tools build control-flow and data-flow representations of the program a priori, *and therefore may require prohibitive space and time when analyzing large systems. Since much of these representations may be unused during an analysis, we construct representations on demand, not in advance. Furthermore, some representations, such as the abstract syntax tree, may be used infrequently during an analysis. We discard these representations and recompute them as needed, reducing the overall space required. Finally, we permit the user to selectively trade-off time for precision and to customize the termination of these costly analyses in order to provide finer user control. We revised the traditional software architecture for compilers to provide these features without unnecessarily complicating the analyses themselves.*

*These techniques have been successfully applied in the design of a program slicer for the Comprehensive Health Care System (*CHCS*), a million-line hospital management system written in the* MUMPS *programming language.*

Keywords: interprocedural analysis, software architecture, demand-driven, context-sensitive, compiler, program understanding, program slicing.

1 Introduction

1.1 Motivation

Software designers and maintainers need to understand their systems if their successful development and maintenance is to continue. Unfortunately, large software systems are difficult to understand, in part because of their age. Some were not implemented using modern programming techniques such as information hiding [19], which can help reduce the complexity of a system. Additionally, many modifications were not anticipated in the original design, resulting in global modifications to incorporate the change. A global change distributes design information that is preferably hidden within a single module to ease future changes dependent on that information. Finally, these large systems have been evolved in this fashion over several years, with modification after modification being layered upon the original implementation by several generations of programmers. The resulting complexity may be exponential in the age of the system [17]. Many of these systems are still in use today, such as the Comprehensive Health Care System (CHCS), a 1,000,000 line hospital management system that we are analyzing in collaboration with Science Applications International Corporation (SAIC).

Because of their complexity, large systems can greatly benefit from automated support for program understanding. Several automated semantic techniques have been developed for understanding software. For instance, a program slicer computes the set of statements in a program that may affect the value of a programmer-specified variable [26]. A static assertion checker such as ASPECT checks the consistency of a program's data-flow and control-flow characteristics against declared computational dependencies [13].

In order to analyze a program, such tools construct control-flow and data-flow representations of the program, similar to those used by an optimizing compiler. One such representation is the program dependence graph (PDG) [7], in which nodes denote operations and edges denote dependencies between operations. Program slicing using the PDG is simple and algorithmically efficient, once the PDG has been constructed [18]. More traditional representations include the control-flow graph (CFG) and dominator trees [1].

However, program understanding tasks are interactive, unlike compilation, and an analysis such as slicing is often applied iteratively to answer a programmer's question about the program. Thus, a whole-program analysis tool must perform analyses quickly in order to effectively answer many of the questions posed by programmers and designers. Two problems arise when applying traditional compiler techniques to the construction of whole-program analysis tools.

*This work is supported in part by NSF Grant CCR-9211002, a Hellman Fellowship, and UC MICRO Grant 94-053 in conjunction with SAIC.

representation	time (min)	space (MB)	comment
1. source code	N/A	40	1M lines
2. AST (construction for the entire program)	13.1	414	18M nodes
3. AST (construction on a per-routine basis)	6.9	30	25% > CPU util. than above
4. CFG with symbolic labels (preserving routine's AST after use)	unknown	≈800	exceeds virtual memory
5. CFG with symbolic labels (discarding routine's AST after use)	17.9	397	6.5M 3-addr. statements
6. complete CFG (symbolic labels replaced by graph edges)	27.5	397	39% CPU utilization

Table 1: Statistics for constructing various representations of CHCS.

First, unlike an optimizing compiler, which typically analyzes a few procedures at a time, these tools analyze the entire program in order to provide the programmer or designer with a global view of the system. Consequently, both the running time and space required for many traditional interprocedural compiler algorithms may be prohibitive for a large program, especially in an interactive context. For example, the size of a PDG can be quadratic or greater in the size of the program (depending on the handling of aliasing), and therefore a PDG for a large system may exceed the virtual memory capacity of the machine. Even the cost of constructing simple linear-space representations such as the abstract syntax tree (AST) can be prohibitive. As the second item in Table 1 illustrates,[1] although the size of an AST is linear in the size of the program, the space consumed by an AST constructed for CHCS, 414 MB, exceeds the capacity of the main memory of the machine, even though care was exercised in its design [8]. In such cases, the time required to first construct the representation and then later retrieve it from the slower portions of the memory hierarchy for actual use may be unacceptable. As shown by the fourth item in Table 1, the space required for both an AST and a CFG can exceed the virtual memory capacity of the machine. Furthermore, the additional iteration over the three-address statements of the CFG that resolves symbolic references to labels into graph edges (Table 1, item 6) requires an additional 9.6 minutes, illustrating poor performance due to heavy use of the slower portions of the memory hierarchy. Based on the per-routine cost of constructing the AST (Table 1, item 3), we estimate that the actual cost of constructing the CFG is 11 minutes, only 1.4 minutes longer than this additional iteration.

The second problem is that a program understanding tool must be able to answer a wide variety of questions about a program. However, because program analysis algorithms are complex, it is not always feasible for the tool user to program a new algorithm to answer a specific question. Consequently, program understanding tools tend to provide a small set of general analysis algorithms that together can answer a wide variety of questions. Unfortunately, although these algorithms are sufficient to answer most queries, their generality can result in an unacceptably long running time and extraneous information. For example, if a tool user desires to know only if a procedure *P* is included in a forward slice (e.g., if the procedure may be affected by a proposed change) then the entire slice may not need to be computed. In particular, a statement in *P* may appear in the slice during the first few iterations of the analysis. If so, then computing the entire slice is unnecessary.

[1] All statistics in this paper were gathered on a Sparcstation 10 Model 61 with 160 MB of main memory and 450 MB of swap space using the `mallinfo()` and `getrusage()` functions available in SunOS, also used by the UNIX `time` command.

1.2 Approach

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 1,000,000 lines and is always growing. A project may also have dozens of programmers requiring such resources to perform analyses. The real problem is waste of computational resources, not lack of resources.

For instance, the per-routine construction of representations shown in Table 1—which require much less space and time, and exhibit much better CPU utilization—suggest that the prohibitive computation costs are largely due to the computation and movement of representations for the entire program, regardless of the analysis algorithm to be subsequently applied to the representations. The underlying tool infrastructure *fails to adapt* to the nature of the analysis being performed and the program being analyzed. Our first goal, then, is that the cost of an analysis be a function of the size of the relevant portions of the program, rather than of the size of the entire program. For example, the cost of computing a program slice should be a function of the number of statements in the slice. To meet this goal, the execution of the analysis algorithm needs to drive the construction of the representations that it accesses. In particular, we propose that a whole-program analysis tool:

- Construct *all* program representations on demand, rather than *a priori*: Demand-driven construction reduces the space and time required since portions of the program that are irrelevant to the analysis are ignored.

- Discard and recompute infrequently used representations that are large but relatively inexpensive to com-

pute: Many representations such as the AST are infrequently used but can exhaust virtual memory. The recomputation cost for these representations may be no worse than the cost of moving them to the slower portions of the memory hierarchy and later retrieving them.

- Persistently cache frequently used representations that are small but relatively expensive to compute: Resolving interprocedural labels in the CFG is expensive and impractical to demand incrementally, but requires little space. Time can be saved by saving this information on disk and only recomputing it when the analyzed software is changed.

In general, the properties of a representation such as space occupied, cost to construct in its entirety, cost to demand in parts, and frequency of access determine whether it should be discarded, retained during execution, or retained persistently across uses of the tool.

The second source of waste is performing an analysis that is more general than the tool user requires because of *lack of flexibility* in these tools. Our second goal, then, is that the tool user should be able to customize the parameters of the analysis—possibly saving computation time—to better match the tool user's needs. For example, if the tool user only wishes to know whether a certain procedure is in a slice, then the analysis should terminate when this fact becomes known. In particular, we propose that a whole-program analysis tool:

- Allow the user to control the precision of the analysis algorithm: The user can provide additional information based on external factors such as the desired precision of the result, urgency, and system load. For example, by reducing precision the tool user can reduce the time of an iteration of an iterative analysis and thus receive an answer more quickly.

- Allow the user to customize the termination criterion for a particular analysis: For example, the number of iterations required can be substantially reduced because iterative algorithms tend to have an initial rapid convergence and so might discover the needed information quickly.

Unfortunately, these new features risk complicating analysis algorithms that are already complicated. Consequently, we have a designed a software architecture that is event-based and exploits the structure of interprocedural analysis to support demand-deriving and discarding data without complicating these algorithms. With careful choice of abstractions, precision control and customized termination of analyses can be accommodated within this framework. The result is a system that can perform analyses independent of system size and provide flexible control of the analysis time.

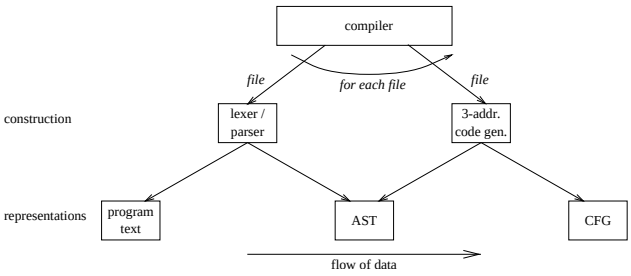


Figure 1: Typical front-end compiler architecture, showing iteration over each file of the program. Boxes indicate modules and arrows indicate calls. Italicized items designate program components being accessed.

In the following sections we discuss our design for achieving good performance in a whole-program analysis tool. To evaluate our design, we discuss the application of our design choices to the construction of a program slicer for CHCS and present statistics gathered from several program slices.

2 Design

Since program understanding tools extract detailed information from the program source, their designs have tended to borrow heavily from optimizing compilers. However, the added requirements of full interprocedural analysis and the wide range of user queries stress traditional compiler designs. Demand-driven computation, discarding, customizable precision, and user-controlled termination can improve performance substantially, but these are not accommodated by standard compiler practice. Because the algorithms used in compilers are quite complicated, our goal is to introduce techniques that minimally perturb these algorithms, while also giving us the performance we desire.

2.1 Improving adaptability

Figure 1 presents a typical software architecture for the front-end of a compiler, which iterates over each file in the program. The general flow of control is from top-to-bottom and left-to-right. The flow of data is left-to-right. The space required in a typical optimizing compiler is not prohibitive, since the program representations for one file are discarded before processing the next file, as they are no longer needed. However, if the representations were to be retained for later use, as required by a whole-program analysis tool, then the resulting space could be prohibitive.

A demand-driven algorithm [3, 4, 12] can reduce the space (and time) requirements of an analysis by ignoring portions of the program that are irrelevant to the analysis. However, we have found that demand-driven construction

of a single program representation does not sufficiently reduce the space requirements, since many program representations are derived from other representations. Although in some analyses the intermediate representations may be *a priori* discarded, in others many must be retained [9]. Our slicer, for example, depends upon a CFG, dominance frontiers [5], and an AST for display. Thus, there are many representations for each procedure in the program, but large portions of some of these representations are used infrequently or not at all. In computing a backward slice, for example, the only needed portions of the program representations are those that are on the control-flow path from the beginning of the program to the slicing criterion.

Basic demand-driven computation does not provide all the savings possible. In particular, our analysis in Table 1 shows that retaining an infrequently used representation can exhaust the main memory or even the virtual memory resources of the computer. Thus, we choose to discard such representations—in our case the AST—and recompute them when they are required. Although this adds time to recompute any discarded data that is later needed, we can still achieve savings by avoiding the cost of (1) moving out retained data to the slower parts of the memory hierarchy, and (2) retrieving it later when needed.

Other representations are expensive to compute and are used frequently, but require little space. For instance, our slicer needs to compute the callers of a procedure, which would normally be resolved by the second pass over the CFG discussed in Section 1.1. Although this information is demanded like other representations, it is stored on disk rather than discarded. Subsequent runs of the slicer on the same program can reuse this information as long as the program has not changed.

Many control-flow and data-flow analyses—such as interval analysis [1] or alias analysis [16, 2]—are sufficiently complicated without the additional burden of requiring the algorithm to demand-derive additional data structures. It is desirable to make minimal changes to these algorithms when addressing the problems encountered when analyzing large systems. We have revised the standard software architecture for compilers to allow us to make only small changes to existing analysis algorithms and yet support the demand-driven construction and subsequent discarding of the program representations.

The primary problem with the standard architecture is that the flow of control largely follows the flow of data from source to sink. This flow is controlled from the top-level analysis algorithm. However, demand-driven computation requires that the sink must be able to "demand" data from the source, reversing the control-flow relation to be not only right-to-left, but also coming from the bottom of the hierarchy, not the top. One solution to this problem is to have the CFG module directly call the AST module, and so

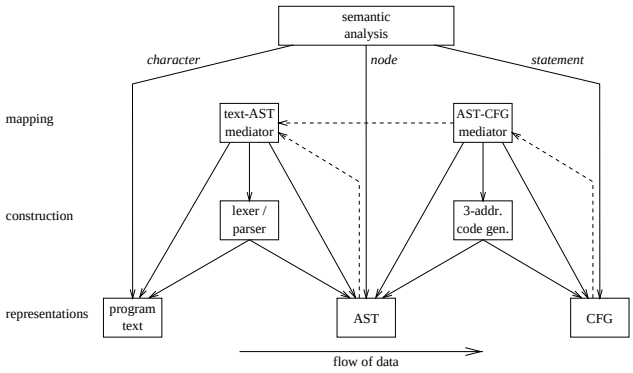


Figure 2: Software architecture for a whole-program analysis tool. Boxes indicate modules, solid arrows indicate calls, and dashed arrows indicate events. Italicized items designate program components being accessed.

forth. However, this solution significantly reduces the independence of the CFG module. For instance, it no longer could be constructed easily from representations other than the AST. A solution that instead modifies the analysis algorithm would further complicate an already complicated algorithm. Additionally, each new algorithm would require essentially the same (complex) modifications. The redundancy distributes the design decisions regarding demand-driven computation across several system components, potentially complicating future changes related to those decisions.

To accommodate the needed changes in control-flow without compromising independence, our solution is to modify the existing architecture to use events and mappings. This architecture borrows from our previous experience with layered and event-based architectures [10, 8], but these architectures do not accommodate demand-driven computation or discarding. Figure 2 presents an example of our architecture containing three program representations—the program text, the AST, and the CFG—with each representation fully encapsulated inside a module. Accessed data structures, shown italicized, are program components, rather than the entire program or whole files as in Figure 1. Unlike in the compiler architecture of Figure 1, the analysis algorithm does not call the construction modules directly, since the program representations are demand-derived as they are accessed through their module abstractions. The architecture's underpinnings, described below, take care of computing the required structures.

Mediator modules: Many semantic tools must maintain mappings between the various program representations. For example, in a program slicer, when the user selects a variable in the program text represented by the AST, the corresponding AST node must be mapped to a three-address statement in the CFG to begin slicing. When the result-

ing slice is displayed, the CFG statements in the slice must be mapped back to their corresponding AST nodes. These mappings could be maintained explicitly within each module (e.g., by having each AST node contain a pointer to its corresponding CFG statement), but this would reduce the independence of the individual modules [25]. Instead, we use separate *mediators* to maintain mappings between the modules [24, 25]. However, since the representations need to be constructed on demand, each mediator may call the required construction module for the representation, as shown in Figure 2. For example, the AST-CFG mediator module, which maintains a mapping between AST nodes and three-address statements in the CFG, calls the code generator if there is a need to map an AST node to its corresponding CFG statement, but that statement has either never been constructed or has been discarded.

Events: Giving mediators the ability to construct representations on demand does not allow the program representations to demand-derive *each other*. For example, the `called_routine` operation on a CFG `call` statement may need to access a three-address statement that has not been constructed yet, which may in turn require construction of the AST nodes from which it is to be derived. Rather than have the CFG module call the AST-CFG mediator, which would require a modification to the CFG module and consequently reduce its independence, our solution is to use events [25], shown in Figure 2 as dashes. The CFG module can send an event "announcing" that it is about to execute the `called_routine` operation. The mediator module "hears" this announcement, and thus responds to the event by calling the code generator, if necessary. For the mediator to hear the announcement, the event handler of the mediator module must be registered with the CFG module by an initialization module (not shown).

If an event were announced for every exported CFG operation, the resulting overhead could be prohibitive. This cost can be reduced by having a high *granularity* for event announcements: the CFG module announces an event for accesses to major program components such as a procedure, and as a result the CFG for an entire procedure may be constructed. This concept of processing granularity for events and the construction of program representations unifies the entire architecture, since it naturally exploits the structure of the problem [15], namely interprocedural analysis. The intraprocedural algorithms are unaffected. If the CFG was constructed incrementally for each statement and the AST constructed incrementally for each file, the resulting architecture would be more complicated.

Address-independent mappings: If a representation may be discarded, the mapping module must support address-independent mappings. These are in essence a pointer abstraction similar to that provided by virtual memory, but resulting in the rederivation of data, rather than the movement of data. Since the AST may be discarded, the AST-CFG mediator must support this type of mapping. Address-independent mappings can be implemented, for example, by assigning each AST node a unique index number that can be reassigned to a reconstructed node, or by using the file name and character position as a key for each AST node.

2.2 Improving flexibility

Reducing the space requirements of an algorithm will also reduce its running time by avoiding the slower portions of the memory hierarchy. However, because an algorithm can require between polynomial and exponential time, depending on its precision, it is also necessary to control the time complexity of the algorithm itself in order to obtain acceptable performance of a whole-program analysis tool. Since many data-flow analyses are iterative [1], significant improvements can be achieved by either reducing the running time of each iteration or by reducing the number of iterations performed.

Controlling precision: As with demand-driven computation, providing the tool user with control over the precision of an analysis should require only minor modifications to the original iterative algorithm. One approach is to allow the tool user to specify the interprocedural *context-sensitivity* of the algorithm. Before we describe our support for this feature, we present some background on precision and context-sensitivity.

Recent work has focused on the trade-offs between context-sensitive and context-insensitive analyses [27, 21]. Many approaches such as the slicing algorithm of Weiser [26] use only a single calling context for each procedure—that is, there is no accounting of the calling sequence that led to the call in order to precisely estimate the calling sequence's influence—and therefore are *context-insensitive*. In contrast, the invocation graph approach [6] is fully *context-sensitive* since each procedure has a distinct calling context for each possible call sequence starting from the main procedure of the program.[2] The invocation graph can be understood as treating all the procedures as inlined at their call sites. Figure 3 presents the call graph of a simple program, with *M* representing the main procedure, along with various *context-graphs*.

The nodes of a context-graph represent a calling context of a procedure and the edges represent procedure calls. Each context-graph has an associated *context-depth*. Figure 3b shows the context-graph using Weiser's approach. Since each procedure has only a single context, this graph is identical to the call graph of Figure 3a. This is the *depth-1*

[2]Recursion is handled by following the recursive call once and then using the resulting data-flow set of the recursive call as an approximation for subsequent calls.

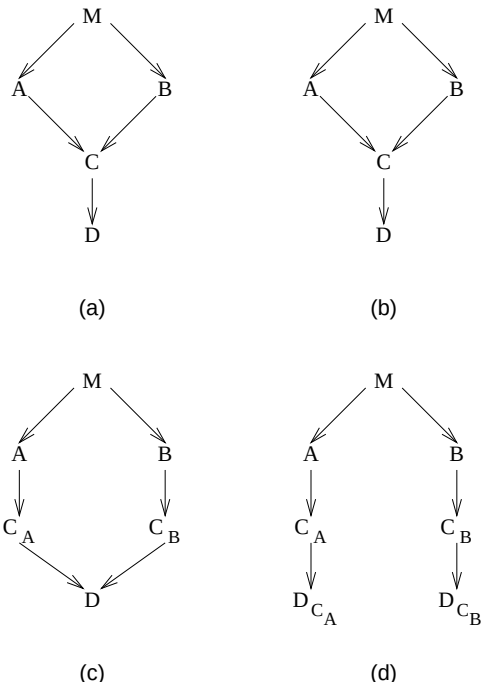


Figure 3: A sample program with (a) its call graph, (b) its depth-1 context-graph, (c) its depth-2 context-graph, and (d) its unbounded-depth context-graph. A procedure with multiple contexts is annotated with its call path.

context-graph, since the context of a procedure is determined by searching a *single* procedure down the call stack. For example, the call stacks *M A C* and *M B C* (shown growing from left to right) are equivalent since only the topmost procedure, *C*, is examined in tracing the call stack. Figure 3d shows a context-graph equivalent to the invocation graph for the program, with procedures having multiple contexts annotated by their call path. This graph has effective *unbounded-depth*, since the context of a procedure is determined by searching back through the call stack as many procedures as necessary to reach the main procedure.

To control precision, our approach allows a variable degree of context-sensitivity. For example, Figure 3c shows the *depth-2* context-graph for the program. The call stacks *M A C* and *M B C* are not defined to be equivalent since the depth-2 call stacks *A C* and *B C* are unequal, resulting in two contexts for *C*. However, *D* still has only a single calling context since the call stacks *M A C D* and *M B C D* are equivalent, as both have a depth-2 call stack of *C D*. This approach is similar to the approach of Shivers [22] for analyzing control-flow in languages with functions as first-class objects.

A depth-1 context-graph has an equal number of procedures and contexts, resulting in a high degree of imprecision but an efficient analysis. An iterative algorithm using a depth-1 context-graph with n procedures and m data-flow facts will require $O(n)$ space and $O(mn)$ time in the worst case. An iterative algorithm using an unbounded-depth context-graph will produce a precise result but will require exponential space and time in the worst case. As the context-depth increases the analysis becomes more precise, but requires more time and space. The tool user may first perform the analysis at a low context-depth and examine the results. If the user's query has not been satisfactorily answered then the context-depth is increased until either a satisfactory answer is produced or the running time of the analysis becomes unacceptable.

The context-graph approach integrates easily with our demand-driven software architecture. In order to isolate the analysis algorithm from our additions, we introduce a context module that encapsulates the control of context sensitivity. When a data-flow algorithm traverses a call edge of the CFG, a new context is demanded for the called procedure. The context module (not shown in Figure 2) either creates a new context for the procedure or returns a pre-existing context. As a consequence, the analysis algorithm is impervious to the changes in context-sensitivity. The only real difference is the context-graph that is implicitly traversed. Contexts are demanded with a standard procedure call to the context module, not an event, since an analysis algorithm is not logically independent of precision.

Customizable termination: If an iterative analysis initially converges towards the ultimate answer quickly, but does not complete for some time, then customizable termination can substantially reduce the analysis time required. One way to provide user controlled termination of an analysis is to permit the user to suspend an analysis, examine the intermediate results, and decide if the analysis has sufficiently answered the tool user's question. Another way is to allow the user to provide a termination test procedure that is periodically applied to the current result of the analysis. A simpler but less flexible approach is for the tool to provide a fixed set of parameterized termination tests.

Supporting customized termination requires a minor modification to the analysis algorithm. Events can be used to announce that a certain slicing milestone is met—such as the end of an iteration—giving the tool's user interface an opportunity to update the display and apply the user's termination test to the current results of the analysis.

3 Implementation

We have implemented a program slicer for CHCS using our techniques. CHCS is implemented in the MUMPS programming language, which is an interpreted programming language with a BASIC-like syntax, reference parameters, and dynamic scoping. Our slicing algorithm correctly handles dynamic scoping by treating each variable reference as

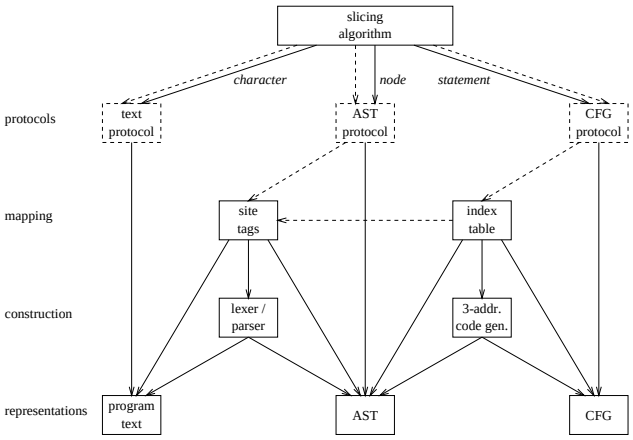


Figure 4: Implementation of the program slicer for CHCS, adapted from Figure 2 with the addition of a protocol layer. Boxes indicate modules, solid arrows indicate calls, and dashed arrows indicate protocols. Italicized items designate program components being accessed.

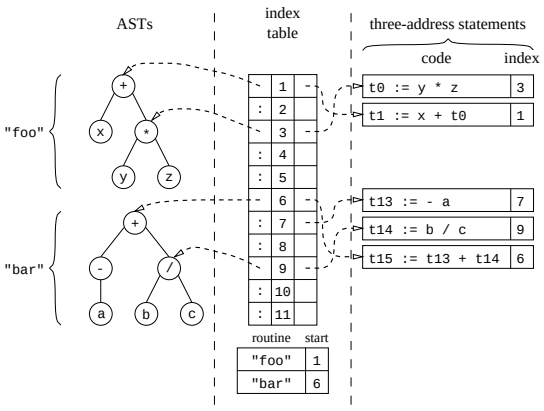


Figure 5: The index-table module maintaining mappings between the ASTs for two routines and the associated three-address statements.

a pointer dereference to any of the reaching variable declarations.

Our program slicer, shown in Figure 4, uses three primary representations—the program text, the AST, and the CFG—and is written in the C programming language. The *site-tags* module maintains mappings between the program text and the AST. Efficient demand-driven computation of the AST depends upon the (possibly cached) caller information, which maps a routine to a file, line number, and list of callers. The *index-table* module maintains mappings between the AST and the CFG.

3.1 Architectural modifications

Because C does not directly support events, we modified the event-based software architecture described in Section 2 in order to simulate events. One possible approach is to implement an event mechanism, instrument the AST and CFG module operations with event announcements, and register the mediators with the appropriate operations. To avoid this effort and minimally perturb the underlying tool infrastructure, we added a *protocol layer*, as shown in Figure 4. The modules of the protocol layer are virtual in that they do not manifest themselves as functions, but rather as requirements. If the slicer wishes to call the `called_routine` function of the CFG module, it must first obey the protocol that requires it to first "announce" an event to the index module. Rather than events flowing from a lower layer to a higher layer, the protocol requests flow from a higher layer to a lower layer. The protocol layer requires no modifications in the lower layers, but instead places a burden on the client (i.e., the slicer) of these layers. However, to both minimize additions to the client and increase performance, the protocol uses the same high level of granularity between requests as proposed for events (i.e., procedure granularity).

3.2 Address-independent mappings

The index-table module functions as a mediator, maintaining mappings between AST nodes and three-address statements in the CFG, as shown in Figure 5. The address-independent mappings are maintained using *index numbers*. When the AST for a routine is constructed, each node is assigned an increasing index number during a preorder traversal of the AST. The index numbers of each root node are also stored in a small auxiliary table for later use. The index-table itself contains bidirectional mappings from AST nodes to three-address statements using hash-tables. Each three-address statement also contains an index number representing its associated AST node.[3] If the AST for a routine is destroyed, its corresponding entries in the index-table are removed. If the AST needs to be reconstructed for a given three-address statement (i.e., a miss occurs in accessing the hash-table), the AST for the entire routine containing the statement is reconstructed. The routine name and its starting index number are determined by searching the auxiliary table using the index number of the three-address statement, and a preorder traversal of the AST is performed to update the index-table.

3.3 Context creation

As discussed in Section 2.2, when a data-flow algorithm traverses a call edge of the CFG, a new calling context must be demanded for the called procedure. The context module either creates a new context for the procedure or returns a pre-existing context. Our backward slicing algorithm performs the following operations when a `call` statement is encountered:

[3] An index number is explicitly stored within a three-address statement for simplicity. A design with more separation would be to use another hash-table for mapping statements to index numbers.

routine	line no.	variable	contexts	time (min)	space (MB)	size of slice (stmts)
NSUN	26	NSORD	253	6.1	17.3	14660
FHDRSTR	52	FHQUIT	256	5.0	15.0	14700
DGBED	84	DGW	254	5.3	18.1	14829
MSAKP	62	MSAEFFDT	267	4.6	15.3	15645
CHPLI	33	LRPTN	394	9.0	17.3	20852
PSNST	25	IN	596	22.9	22.3	32351
LRPRACT	10	LRALL	611	24.1	22.6	33156
CPENRFRM	21	CPRBUF	647	29.8	23.8	33935
ORSETN	42	ORACTIN	1255	141.9	60.5	73707
ORSIGN	32	ORLPKFG	1321	154.9	71.8	73907
ORENTRY	68	LRORPTNZ	1432	179.9	74.1	79319

Table 2: Statistics for different slices of CHCS with a single context per procedure.

1. Demand-derive a context for the called procedure, *Q*, using the context of the calling procedure *P*.

2. Using the current slicing criterion, create new slicing criteria at the `return` statements of the context for *Q*. The new slicing criteria are merged with any already existing criteria using a `union` operation.

3. Compute the slice of *Q* during a backward depth-first search of *Q* from each `return` statement.

4. Use the updated criterion at the first statement of *Q* as the new slicing criterion for the `call` statement and resume slicing *P*.

In the depth-1 context-graph of Figure 3b, first *M* slices into *A*, and *A* calls *C* by demand-deriving a context for *C* and updating the slicing criteria at the `return` statements of *C*. After a depth-first search of *C*, the criteria at the first statement of *C* is used to continue slicing *A*. Next, *M* calls *B*, and *B* follows the same steps as *A*. However, since *C* has only one context, the criteria from *A* and *B* are merged in *C*. The depth-first search returns immediately, since all blocks in *C* have been marked as visited by *A*, and *B* uses the (approximate) criterion at the first statement of *C* to continue slicing. Thus, data placed in *C* by *A* flows back into *B*, and on the next iteration the data from *B* will flow back into *A*, resulting in imprecision. If the depth-2 context-graph of Figure 3c is used, this imprecision will not occur; however, some imprecision may still occur since *D* has only a single context. Using a depth-3 context-graph, which is equivalent to the unbounded-depth context-graph for our sample program, will result in a precise analysis.

Unless an unbounded-depth context-graph is used, data may be propagated along *unrealizable paths* [12, 11]. For example, data merged at the `return` statements of *C*, which is the source of the imprecision, is propagated along unrealizable paths (e.g., the data of *A* is propagated through *C* to *B*). Our slicing algorithm tries to avoid unrealizable paths. Since the call to *C* from *A* returns to *A* and not to *B*, should the slicing algorithm terminate before *B* is called from *M* then no imprecision will result.[4]

3.4 Customizable termination

Currently the tool provides suspension of an analysis for inspection of the current results. The user can unintrusively monitor the progress of the analysis by means of an on-the-fly display. Our program slicer for CHCS allows viewing the number of statements analyzed, size of the slice, and other criteria interactively.

4 Performance results

To test our claims about the value of demand-driven computation, discarding, precision control, and customizable termination, we performed several backward slices of CHCS using our program slicer.

When examining these preliminary results, conclusions should not be drawn about slicing itself, but only about our techniques. For one, only one large program was sliced. Two, the slicing criteria were chosen to produce a reasonable distribution of slice sizes, not to be representative of typical slices. Also, the slices computed do not slice into the callers of the routine in which the slice is initiated; consequently the slices are akin to slicing on statements in the main procedure, which has no callers.

The times reported below do not include the time required to compute and write the caller information for each routine, since it is only recomputed when a file of the program to be sliced is changed. For CHCS this information is computed in 8.0 minutes and occupies 1.1 MB of disk space.

4.1 Demand-driven computation and discarding

Table 2 presents the statistics for a range of slices of CHCS. Our analysis of the times and sizes of the slices suggests that they are linearly related. The space required also

[4] The algorithm may terminate if the slicing criterion becomes empty or if the algorithm is interrupted by the tool user. Additionally, *B* may not be called from *M* if the call paths are constrained, as in chopping. [14, 20]

Correction: At the beginning of Section 4.1, the paper states "Our analysis of the times and sizes of the slices suggests that they are linearly related." However, Table 2 on the the same page clearly shows a quadratic relationship; "linearly" should be replaced by "quadratic" in the above statement.

routine	line no.	variable	procedures	depth	contexts	time (min)	space (MB)	size of slice (stmts)
COMARG	15	ERRTYP	26	1	26	0.017	0.29	488
				2	49	0.033	0.37	483
				unbounded	363	0.217	0.97	483
COMBLK	14	ERRTYP	36	1	36	0.030	0.37	780
				2	74	0.058	0.43	774
				3	121	0.092	0.55	769
				unbounded	602	0.484	1.42	769

Table 3: Statistics at different context-depths for a 1,000 line compliance checker for CHCS.

routine	line no.	variable	procedures	depth	contexts	time (min)	space (MB)	size of slice (stmts)
PSPA	46	PSDT	248	1	248	4.9	14.9	14630
				2	488	11.1	18.5	14448
				3	2586	127.6	49.1	14446
DIC	38	DUOUT	248	1	248	5.3	19.6	14711
				2	488	10.3	23.1	14532
				3	2553	121.0	53.6	14530

Table 4: Statistics at different context-depths for two slices of CHCS.

appears to be linearly related to the slice size. These results indicate that we have met our goal of having the cost of the analysis be a function of the result's size, rather than the size of the entire program.

Because the slices did not exhaust real memory—much less virtual memory—the role of discarding did not come into play. However, a separate set of measurements of slicing without discarding indicated that the cost of discarding the AST was insignificant.

4.2 Context-depth sensitivity

Tables 3 and 4 present statistics for four slices at varying context-depths. Table 3 presents two slices from a 1,000 line compliance checker for CHCS, and Table 4 presents two slices from CHCS itself. In each slice, the number of calling contexts increases rapidly with the context-depth, significantly impacting the time and space required. However, in the two slices from the compliance checker, a low context-depth yields a program slice equivalent to a program slice obtained at an unbounded context-depth. Although the resulting context-graphs differ, unbounded context-depth is unnecessary to obtain a precise slice. The two larger slices from CHCS show that as the context-depth increases, at first there is a considerable decrease in the number of statements in the slice. However, an additional increase of the context-depth yields little improvement.[5] The four slices together suggest that a high context-depth may be unnecessary to obtain a precise slice. They also support our hypothesis that a low context-depth slice is usually several times less costly than a higher one, suggesting that there is little extra cost to the tool user in performing a low context-depth slice first, on the hope that the result will adequately answer the tool user's question.

4.3 Algorithmic convergence

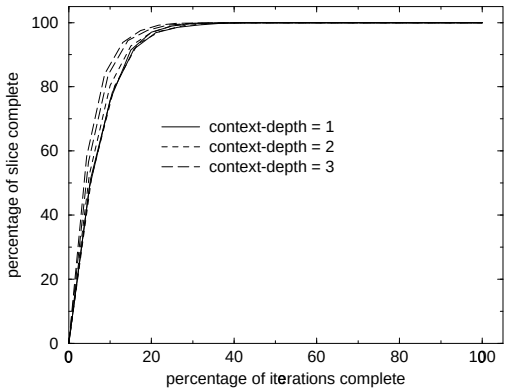
Some program slices in Table 2 require more than two hours to complete, which for some uses may be unacceptable. However, our results show that the majority of statements in a slice are obtained after the first few iterations. Figures 6a and 6b depict the convergence properties of our program slicing algorithm by plotting the size of the slice at each iteration.[6] These figures show that 90% of the statements in the slice are obtained within the first 20% of the iterations. Figure 6a presents data for two slices of CHCS at different context-depths, illustrating that the rate of convergence appears to be independent of the context-depth. Figure 6b presents data for several sizes of slices of CHCS, illustrating that the convergence rate also appears to be independent of the slice size. This data seems to confirm our belief that the tool user can use customized termination of the slicer to substantially reduce the number of iterations of an analysis, independent of the context-depth and slice size.

5 Conclusion

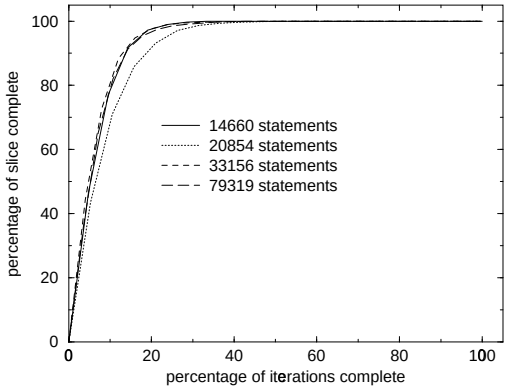
Because a large software system is difficult for its programmers and designers to understand, they could greatly benefit from automated support for program understanding. Tools supporting such analyses need to construct representations similar to those used by an optimizing compiler.

[5] We were unable to determine the context-depth that would result in a program slice equivalent to that obtained using an unbounded context-depth, since the space required exceeds the virtual memory capacity of the machine.

[6] The data are normalized to percentages so that slices of different sizes and iterations can be compared.



(a) varying context-depth



(b) varying size of slice

Figure 6: Algorithmic convergence.

However, unlike an optimizing compiler, program understanding may require analyzing large portions of the program simultaneously for an arbitrary task. Consequently, to minimize time and space requirements, the tool infrastructure must be able to adapt to the requirements of the analysis being performed, and the tool provide flexible control of the analysis to the user.

Although it may seem overly aggressive to simultaneously apply the demanding of all data, the discarding of infrequently used data, the retention of costly data, tunable precision, and customizable termination, omitting any technique can unacceptably compromise performance:

- Without demand-driven computation, the space and time required to perform analyses is necessarily a function of the size of the overall system, and is likely to exhaust the memory resources of most computers.

- Without discarding, virtual memory can be exhausted by sizable representations that are not currently involved in the computation. Additional time is also expended in moving these representations out to disk.

- Without persistent storage, deriving data that is costly to construct, albeit compact, can increase the start-up time of an analysis substantially.

- Without providing control of precision, an analysis can take unnecessarily long if a high degree of precision is not required. On the other hand, providing only a low degree of precision may be ineffective in answering sensitive queries.

- Finally, without the ability to control the termination of an analysis, it may run unnecessarily long to answer the question at hand.

In addition, to support these features without unnecessarily complicating the analysis algorithms, the architecture traditionally supporting these algorithms must be adapted to include events and mappings.

Work remains to be done to determine the feasibility of economical whole-program analysis tools. These results are for only a single analysis algorithm on a single system. Also, despite the positive implications of Section 4, we are disappointed by the time required to compute the larger slices of CHCS, interruptability notwithstanding. On these larger slices, the slicer examined approximately 5% of the statements in CHCS indicating that despite our aggressive design, program slicing of large systems may still be very costly. We need to look more closely at our design, implementation, and the slices computed to better understand what is determining the overall performance. Our initial experiments suggest several interesting questions:

- Is there a practical framework for deciding how a representation should be treated with respect to construction on demand, discarding, persistent storage, etc.? We have informally identified the following properties as influential: relative sizes of the source and target data structures, the time required to construct the target, the cost of constructing portions of the target on demand, and the frequency of access of the target.

- Can it be cost effective to share program representations among multiple tool users? How would sharing affect the way representations should be managed?

- Is it possible to design a heuristic for determining the progress of an analysis with respect to convergence? Since all of the convergence curves in Figure 6 are very similar, a curve-fitting heuristic might allow predicting the progress of a slice.

- Can the optimal context-depth be heuristically determined? Can the context-depth be increased or the context-graph modified during slicing to provide better precision where needed?

- How does context-depth affect iteration time? Does added precision lead to convergence in fewer iterations?

- How does the structure of the program being sliced affect the slicing algorithm? Do slices over well-structured modules of the system require fewer iterations and converge faster? Does the structure impact the recomputation time or the performance of the memory hierarchy?

- Is it beneficial to discard procedures of the CFG that have not yet contributed to the analysis? Is there a way to conservatively cache information about a procedure to avoid re-examining it?

As part of our attempt to answer these and other questions about whole-program analysis, we are planning to use our techniques in the design of a slicer for C programs. Although C has aliasing through pointers, our implementation for MUMPS handles dynamic scoping as pointer aliasing. Also, some of the newer results in points-to analysis [23] should make it possible to inexpensively compute alias relations on a demand basis. Our intuition is that the performance results for analyzing C programs should be similar.

Acknowledgments: Thanks to David Notkin and Jeanne Ferrante for their help on improving the organization of this paper. Mark Wegman of IBM Research suggested the question regarding modifying the context-graph during slicing. We thank the anonymous reviewers for their helpful comments.

References

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

[2] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC, Jan. 1993.

[3] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, FL, Jan. 1991.

[4] J.-D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Trans. Softw. Eng.*, 20(2):105–114, Feb. 1994.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[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 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] W. G. Griswold and D. C. Atkinson. Managing the design trade-offs for a program understanding and transformation tool. *J. Syst. Softw.*, 30(1–2):99–116, July–Aug. 1995.

[9] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Meth.*, 2(3):228–269, July 1993.

[10] W. G. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Trans. Softw. Eng.*, 21(4):275–287, Apr. 1995.

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

[12] 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.

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

[14] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the 2nd ACM Symposium on the Foundations of Software Engineering*, pages 2–10, New Orleans, LA, Dec. 1994.

[15] S. C. Johnson. A portable compiler: Theory and practice. In *Proceedings of the 5th ACM Symposium on Principles of*

Programming Languages, pages 97–104, Tucson, AZ, Jan. 1978.

[16] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM '92 SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA, June 1992.

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

[18] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 23–25, Pittsburgh, PA, Apr. 1984.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[20] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 41–52, Washington, DC, Oct. 1995.

[21] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA, June 1995.

[22] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, May 1991.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, Jan. 1996.

[24] K. J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems*. Ph.D. dissertation, University of Washington, Department of Computer Science & Engineering, Aug. 1994.

[25] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Trans. Softw. Eng. Meth.*, 1(3):229–268, July 1992.

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

[27] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.